AN ABSTRACT OF THE DISSERTATION OF

Umme Ayda Mannan for the degree of Doctor of Philosophy in Computer Science presented on August 31, 2020.

Title: Investigating the Effects of Design Quality in OSS Projects

Abstract approved: _____

Carlos Jensen Anita Sarma

Software systems are becoming an essential part of the lives of both individuals and organizations, and as a consequence, these systems are getting bigger and more complex. Because of this, the tasks of maintaining the quality in these complex software systems are becoming increasingly difficult. Furthermore, these systems are subject to constant pressure to add modifications, implementing new features, or regular bug fixing. Due to an often-strict release schedule, developers may not get the chance to design and implement ideal solutions. This can lead to decay in software design and the growth of technical debt, which negatively impacts the overall quality of the software.

Design quality plays a vital role in maintaining the quality of software systems. In the past, several studies have shown the impact of design quality on different quality attributes such as bug density, productivity, and maintenance. Researchers also proposed several tools and techniques to improve design quality. Despite efforts by the research community in past years, there are still gaps in our understanding of how does design quality evolves in OSS projects, how projects manage design quality and how does design

quality impacts the long-term health of the projects. This thesis proposes a number of large-scale empirical investigations aiming to understand the gaps.

In the first part of this thesis, we investigate the evolution of design quality in OSS projects. Researchers used different metrics to measure the project's design quality. In this thesis, we investigated two different design quality metrics. The first one is code smells, which are the result of poor design or implementation choices. And the second one is entropy, provides an understanding of design quality in terms of flexibility during the enhancement of its functionality. Our results show that design issues build up and design quality (measured by code smells and entropy) degrades over time. As the project grows older and larger, design issues are fixed less and, as a consequence, build up.

In the second part, we investigate how OSS projects manage design quality. As design discussion is the primary mechanism for OSS projects to debate, make and document design decisions, we investigated how developers discuss design in the community, how they evolve along with design quality, and what effect they have on the design quality. Our results show that: I) Regardless of different communication channels used for discussion, 89.51% of all design discussions occur in the project mailing list, II) the average number of design discussions decrease; design quality degrade, as the project evolves, and III) the correlation between design discussions and design quality is small.

In the third part, we investigate how does design quality impacts day to day activity (such as code merging) as well as long term health of the projects. We find that code with design issues (code smells) are more likely to be involved in merge conflicts. Our results also show that both code smells and entropy have an impact on the long-term health of the project. In this thesis, we use bug-proneness of the codebase as the measurement of the project's long-term health.

Investigating the Effects of Design Quality in OSS Projects

by

Umme Ayda Mannan

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented August 31, 2020
Commencement June 2021

Doctor of Philosophy dissertation of Umme Ayda Mannan presented on August 31, 2020.

APPROVED:

Co-Major Professor, representing Computer Science

Co-Major Professor, representing Computer Science

Head of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Umme Ayda Mannan, Author

ACKNOWLEDGEMENTS

One of the greatest pleasures of research is to work with brilliant and thoughtful individuals from all over the world. I am grateful to my advisors for their encouragement and guidance, my collaborators for sharing their time and ideas, and the tremendous support of my family.

First and foremost, I would like to thank my advisor, Dr. Carlos Jensen, for the endless support and guidance throughout my Ph.D. and every aspect of my life around it. Carlos always pointed me in the right direction and helped me feel like I was making progress. I remember leaving our every meeting with a clear objective and tons of excitement to learn about a new topic or work toward a new result. I would also like to thank my co-advisor, Dr. Anita Sarma, for always being there to discuss my research. She always helped me clarify my thoughts, pointed out the inconsistencies or implicit assumptions where they existed, and helped me shape my research. Both of my advisors helped me gain confidence and played a crucial role in my growth as a researcher. I want to take this opportunity to express my sincere gratitude to them for being there and for being my advisors.

I would also like to thank the other professors at Oregon State University that have provided exceptional guidance and mentorship during my time here. Dr. Margaret M. Burnett, Dr. Danny Dig, and Dr. Arash Termehchy provided me advice on how to become a better researcher.

I would also like to thank my research group members and other students I have been fortunate enough to work with at Oregon State University: Iftekhar Ahmed, Jennifer Davidson, Rithika Naik, Rana Almurshed, Caius Brindescu, and Rahul Gopinath.

I am grateful to my parents, M. A. Mannan and Shahana Mannan, who love and support me from 0th grade to 23rd. From early on in life, you both encouraged and helped me asking questions and having fun with learning and life. Thank you for the many insightful conversations and experiences that have shaped nearly every aspect of my life. I am also grateful to my mother-in-law Begum Amena Ahmed for all her support and encouragement throughout my Ph.D. You are more of a mother than a mother-in-law to me.

Finally, and most importantly, I would like to thank my loving husband, Dr. Iftekhar Ahmed. This Ph.D. would not have been possible without you. I am forever grateful for all your love and support throughout the years. Thank you for listening to countless research ideas, inspiring me to take on new challenges, filling our days with epic adventures all the way from Sylhet, Bangladesh to Corvallis, USA, and always believing in me. The journey has been a blast so far, and I look forward to spending the rest of our life together!

There is freedom waiting for you,

On the breezes of the sky,

And you ask "What if I fall?"

Oh but my darling,

What if you fly?

— *Erin Hanson*

CONTRIBUTION OF AUTHORS

The following authors contributed to the manuscript: Umme Ayda Mannan, Iftekhar Ahmed, Rahul Gopinath, Caius Brindescu, Anita Sarma and Carlos Jensen. Umme Ayda is the first author for the three publications presented in chapter 3, chapter 4 and chapter 6 and did the data collection, analysis and vast majority of the writing for each of the publications. For the publications presented in chapter 2 and chapter 5, Umme Ayda contribute as second and third author. Carlos Jensen and Anita Sarma were involved in advising the design of these research studies, and edited all documents. In chapter 2, Umme Ayda collected data with Iftekhar and Rahul, did data analysis and writing for the first research question of the paper, which is used to partially answer one of the research questions of this dissertation. In chapter 5, Umme Ayda performed data analysis and writing for one research question and wrote the related work section. Umme Ayda's contribution in chapter 5 is used to answer one of the research question of this dissertation. Umme Ayda did data collection, data analysis and writing for chapter 3. In chapter 4, Umme Ayda did data collection and performed data analysis with Iftekhar and did the vast majority of writing. Iftekhar also edited the final draft of chapter 4 with Carlos and Anita. Caius and Iftekhar helped with data collection for Chapter 6. Iftekhar and Umme Ayda performed data analysis and writing for Chapter 6.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF TABLES

LIST OF TABLES (Continued)

Chapter 1: Introduction

Over the years, software has become an integral part of our life. From connecting friends from all over the world using social networks to our home refrigerators notifying us about spoiled food, software helps make our life convenient. Modern-day software systems are changing the lives of individuals and organizations by creating more advanced ways of doing business and involving clients. As a result, these software systems are getting bigger, complex, and challenging to maintain. Additionally, developers have to create software that meets performance, reliability, and maintainability requirements, in tight development schedules [218]. Failure to meet quality requirements can carry high costs, especially when these lead to software failures [43]. Therefore, ensuring the quality of these sophisticated software systems is very important.

Among different aspects of software quality, design quality has an important role in maintaining the quality of software systems [76, 206]. Good design is key in ensuring the performance, robustness, and maintainability of software [250]. For example, as per a study of five organizations, 64% of software defects could be traced back to software design issues [18] which are more expensive to fix [79, 230, 120]. Prior research has shown that design quality has impact on different quality attributes such as fault-proneness [30, 52], bug density [178], developer productivity [59], and the amount of maintenance modifications [155]. Therefore, it is essential to focus on design quality to ensure software quality.

However, ensuring design quality is far from trivial. Additionally, the design decisions made during the software lifecycle are typically not well documented, and so the rationale

for these choices is often lost [126]. For example, developers often discuss design-related decisions verbally [137] and may not always update the design documents. There are several reasons behind not documenting or updating the design document. One reason could be that fulfilling deadline commitment of delivering software seems more important at that moment than documenting the design decisions for use later [126]. Another reason could be the lack of coordination among team members on documenting the updated design regularly. As a result, developers who are unaware of the design decisions can make changes contrary to what was intended, setting others up for clashes, which can in turn further degrade the overall design quality of the system [249].

When the design quality of the software degrades it has ramifications on the software itself. In a typical scenario, the source code of a software experience constant modifications in the form of bug fixes and the addition of new features to satisfy end-user needs [152]. Due to time constraints and the constant pressure to keep the software updated to satisfy these end-user requirements, developers do not always have the time to design and implement corrective solutions. Such failure to maintain the design in response to the frequent software changes leads to decay in the software design and the growth of technical debt [91]. As a result, code becomes more difficult to understand [265]. This forms a vicious cycle of design degradation further negatively impacting the quality of the software. One of the symptoms of design degradation is that code structure drifts away from good object-oriented design principles.

Maintaining good and consistent design quality is an even more significant challenge for open source software (OSS) for several reasons. First, these projects may include 100's of geographically distributed developers, and the contribution are voluntary [50]. Second, because of this volunteering contribution model, OSS projects rarely produce updated design documents and, in some cases, lack formal design documentation altogether [46].

In OSS, design-related discussions among developers are the only form of design documentation that developers can use to coordinate and build together. Moreover, it is challenging for developers to find those design decisions from the archived discussions later and work accordingly.

1.1 Research Statement

Given the importance of design quality on software and the negative impacts of its degradation, this dissertation has the following high-level goal:

Goal: To investigate the impact of design quality on the long-term health of OSS projects, and how do projects manage this.

While there has been research on design quality, they still have gaps. Tahvildar et al. [241] proposed a framework that uses object-oriented metrics as indicators to detect situations for particular transformations to improve design quality automatically. Prabha et al. [206] used different Machine Learning Techniques to identify design patterns that could be investigated further to improve design quality. Sharma et al. [228] proposed a design quality assessment tool "Designite" that provides detailed metrics analysis at different granularity. Prior research has also investigated the relation between design quality and different project activities such as testing [182], code review [181, 201], development effort and governance [50]. Additionally, researchers have also looked into how design quality impact code quality [139, 104, 265, 166].

These works still leave gaps in our understanding of how does design quality evolves in OSS projects, how projects manage design quality and how does design quality impacts the long-term health of a (OSS) project.

To close these gaps in our understanding of design quality and achieve our goal, in

this dissertation we conducted a number of empirical studies focusing on the following four research questions:.

RQ1: How does design quality in OSS projects evolve over time?

Design quality could mean different things to different people. For example, it could mean poor feature selection, terrible UX design, or bad architecture design. While trying to identify a suitable metric for measuring design quality, researchers have investigated different metrics such as Coupling Between Object(CBO), Coupling Factor(COF), Weighted Method per Class(WMC), entropy and code smells etc. [50, 69, 156, 193, 191, 56, 167, 21]. To investigate our research question, we looked into two of the most frequently used metrics—code smell and entropy—for measuring design quality. The term code smell was introduced by Fowler [91], who gave an informal definition of 22 code smells focused on the maintainability of software systems. Code smells are poor design or implementation choices, a consequence of not having a well-designed source code of a system is from the beginning.

Another design quality indicator used in our work is *Entropy*. Entropy is used to measure the uncertainty associated with a random variable quantifying the information in a code [227]. This unit of code could be a source file, class, or method. Researchers used entropy to measure the naturalness of code [117]. According to Hindle et al. [117], entropy is the syntactic difference of an entity (e.g., method) and the rest of the codebase. A system's entropy provides an understanding of design quality in terms of flexibility during its functionality enhancement. Systems that have poor design may necessitate modifications to many classes, which increases the entropy in the system. On the flip side, systems with straightforward design can have changes confined to specific parts, keeping the system's entropy low. A large number of studies have used and validated

code smells and entropy for measuring design quality [181, 69, 156, 193, 167, 21, 181, 20, 163, 78, 237, 238, 191, 56].

RQ2: How do OSS projects manage design quality over time?

Managing design quality in OSS projects is challenging. A reason for this could be OSS projects rarely produce updated design documents and, in some cases, lack formal design documentation altogether [46]. Purposeful discussions over different communication channels are the only way developers discuss design-related decisions, which are the only form of design documents in OSS projects.

RQ3: How does design quality impact 'high-coordination need' work like merging changes?

In OSS projects, coordination among developers is a big challenge because of its distributed nature. To coordinate changes and work efficiently, developers use Version Control Systems (VCS). VCS has made parallel development easier by streamlining and coordinating code management, branching, and merging.

However, this process could get halted when isolated private development lines are synchronized. In some cases a developer may run into merge conflicts, which interrupts their workflow because they have to resolve the conflicts before they continue their work. Resolving the conflicts can be cognitively challenging as developers have to reason about the parallel (conflicting) changes and find an acceptable merge solution before they can move ahead with their implementation. A reasons behind merge conflicts could be design degradation as it makes the code more difficult to understand and change.

RQ4: How can design quality metrics be used to improve the long-term health of projects?

One way to ensure the software project's health is by finding and fixing bugs, which is effort-intensive and time consuming [141]. Another way is to reliably identify buggy

parts of the code to guide their bug-fixing efforts (bug prediction). In the last decade, researchers have investigated a wide range of bug prediction models [30, 175, 274, 275, 268, 107, 95, 144, 186, 207]. These models aim to isolate the parts of the code that are likely to be buggy in the current version or the immediate next version of the codebase (*bugginess at a given time*) to facilitate bug-fixing efforts..

While finding the likely location of bugs in the current codebase is useful, proactively identifying parts of the codebase that is bug-prone (*i.e., more likely to be buggy now and in the future*), can have a more significant impact on project's long-term health. Identifying bug-proneness of the codebase can help developers minimize the chances of the occurrences of future bugs, which can, in turn, reduce maintenance time and costs. In this thesis, we use the bug-proneness of the codebase to measure the project's long-term health.

1.2 Structure

This thesis is structured into seven chapters. Following are the summary of the chapters,

- Chapter 2 focuses on answering part of RQ 1 and was published as a paper in ESEM in 2015 titled *"An Empirical Study of Design Degradation How Software Projects Get Worse Over Time."*. The goal of this study was to shed light on how design degradation happens as project ages, and how traditional quality assurance (QA) activities contribute or fail to contribute towards improving design quality.

- Chapter 3 presents a technical report *"The Evolution of Software Entropy in Open Source projects: An Empirical Study"*. This chapter focuses on answering the second part of RQ 1 through an empirical study. The study's goal was to investigate the

evolution of entropy in OSS projects and its relation to project evolution.

- Chapter 4 presents our paper *"How Design Discussions and Design Quality Relate: A Case Study of Apache Projects"*. To appear in FSE 2020. This paper primarily investigated RQ 2. This study aimed to investigate how design quality evolves in OSS projects and the relationship between design discussion and design quality.

- Chapter 5 reports the design and the results of the experiment aimed to analyze the impact of code smells on merging changes in a distributed development environment. This chapter focuses on answering RQ 3 and was published in ESEM (2017), titled *"An Empirical Examination of the Relationship Between Code Smells and Merge Conflicts."*.

- Chapter 6 presents our paper *"Floss now or go to the dentist: An empirical assessment of bug-proneness prediction"*. Under review at the Information and Software Technology (IST). This paper investigates RQ 4. This study aimed to evaluate the design quality metrics as a predictor for bug-proneness in OSS projects.

- Chapter 7 concludes the thesis and discusses future directions on how to take our research forward.

Table 1.1 provides the mapping between research questions and the empirical studies used in this thesis. Table 1.1 also provides a summary of the author's contributions to these studies.

Table 1.1: Research questions mapped to chapters

Chapter	Research questions answered	Author's Contribution
2 & 3	RQ1. How does design quality in OSS projects evolve over time?	This research questioned is answered by two studies presented in chapter 2 and chapter 3. Among the four research questions from chapter 2, the first one is used to answer part of RQ (evolution of code smells) of this dissertation, which was part of my contribution to that paper. The second part of the RQ1 (evolution of entropy) is answered by the study presented in chapter 3.
4	RQ2. How do OSS projects manage design quality over time?	This research question is answered by the study presented in chapter 4. I am the first author and main contributor to this study.
5	RQ3. How does design quality impact 'high coordination need' work like merging changes?	This research question is answered by a study presented in chapter 5. RQ3 of this dissertation was answered by one of the research questions that I contributed to as an author.
6	RQ4. How can design quality metrics be used to improve the long-term health of projects?	The study presented in chapter 6 is used to answer RQ4. I am the first author and main contributor to this study.

1.3 Research Contributions

The contributions of this dissertation are:

1. Investigating the evolution of design quality in real-world OSS projects (Chapter 2 and 3).

2. Investigating the relationship between design discussions and design quality (Chapter 4).

3. Proposing a single Machine learning classifier to detect design discussion from different communication channels (Chapter 4).

4. Investigating the relation between design quality and code merging (Chapter 5).

5. Evidence that combining design quality metrics (i.e., entropy and code smell) with other code metrics and process metrics (i.e., number of authors) can significantly improve the bug-proneness prediction accuracy (Chapter 6).

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

Iftekhar Ahmed, **Umme Ayda Mannan**, Rahul Gopinath, Carlos Jensen

2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (pp. 1-10)

Chapter 2: An Empirical Study of Design Degradation: How Software Projects Get Worse Over Time

2.1 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 [41, 153]. 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 [265], and as a result the ability to evolve an application tends to decrease over time [169].

Design degradation leads to design debt, which contributes to technical debt [63] 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 [91]. This term was coined by Fowler [91], 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 [69, 156, 193].

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 [55, 138, 156, 192, 193, 222] 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.

2.2 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 [169]. 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 [69, 70, 156]. However, software repository mining provides a way to extract the historical evolution of a software system [129]. Researchers have also come up with techniques that rely on evaluation of successive versions of a software system [197, 130, 149] 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 [123]. Another facet of design degradation is design debt, which contributes to technical debt [63]. 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. [265] found that the God class smell is related to technical debt.

The concept of code smells was introduced by Fowler [91]. Code smells are symptoms of problems in source code, and indicators of where refactoring is needed [91]. 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 [156, 192] and code maintainability problems [91].

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 [149] and meta-model based [180]. Metric based measures show that code smells impact software quality [166]. Most of the code smell detection tools are based on metric analysis [149, 165]. This static analysis based approach has its drawbacks. Fowler and Beck claimed that "No set of metrics rivals informed human intuition" [91] 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" [164, 168].

Several studies have looked into the relationship between code smells and change-proneness. Olbrich et al. [192] report that classes infected with the "God class" and "Shotgun surgery" smells are more change prone. Contrary to their finding, Schumacher et al. [222] found that the "God class" is only more change prone if results are not normalized by LOC. However, Khomh et al. [138] found that classes infected with code smells are changed more often overall.

Identifying the impact of the code smells [192] and how these impact the understandability and maintainability of code [25, 75] has also been of interest to researchers. Smith et al. found that "God class" and "Switch statements" smells have an impact on software performance [234] . 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. [87] 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 [184]. Murphy et al. [184] 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 [124]. They also found that design pattern decay leads to testing anti-patterns such as "concurrentuser-relationship" and "self-use-relationship".

2.3 Methodology

2.3.1 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.

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

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 [5] 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 2.1 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.

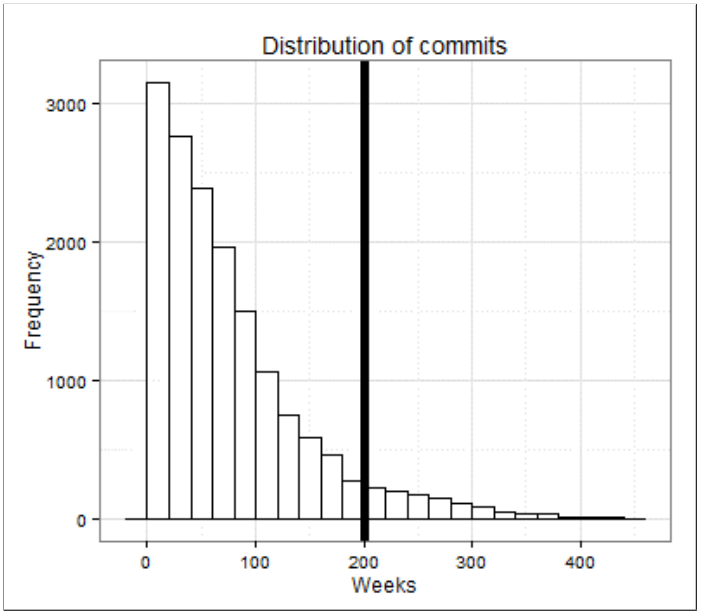

Figure 2.1: Distribution of commits over time (vertical line indicating cutoff at 200 weeks

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 [123, 124] 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 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.

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 2.1 provides a summary of features and other descriptive information about the projects that were part of our study.

Table 2.1: 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. [68]. Table 2.2 has the summary of the domains of the projects. In the next subsection, we discuss the code smell detection tools used.

Table 2.2: 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%

2.3.2 Tool selection

We chose to use InFusion [7] to identify code smells because it has been found to identify the broadest set of smells [87] (Table 2.3). Researchers have found that the metric-based approach identified by Marinescu [167] has the highest recall and precision (precision: 71%, recall: 100%) for finding most code smells [222]. 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 [197]. 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 (equation 2.1) and recall (equation 2.2) for identifying Blob and Feature Envy code smells from the oracle. We

Table 2.3: 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

also report the F-measure (equation 2.3), defined as the harmonic mean of precision and recall as an aggregate indicator of precision and recall [28]. Table 2.4 has the summary of the performance of InFusion.

$$\text{Recall} = \frac{|\ \text{True positive smells} \cap \text{Code smells detected by InFusion}\ |}{\text{True positive smells}}\% \tag{2.1}$$

$$\text{Precision} = \frac{|\text{ True positive smells} \cap \text{Code smells detected by InFusion }|}{\text{Code smells detected by InFusion}}\% \tag{2.2}$$

$$\text{F-measure} = \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2.3}$$

Table 2.4: InFusion Performance

Precision	Recall	F-measure
84%	100%	91.30%

2.3.3 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 [164]. 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 [257]. We gathered different coverage metrics, such as statement coverage from Emma [216], branch coverage from Cobertura [73], path coverage from JMockit [158], and mutation kills from PIT [62]. 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.

2.4 Results

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

2.4.1 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 (equation 2.4), which gives us a score between 0 and 1.

$$\text{Rescaled value} = \frac{x - min(x)}{max(x) - min(x)} \tag{2.4}$$

Where,

x = each data point.

min(x) = The minimum among all the data points.

max(x) = The maximum among all the data points.

Previous studies have shown that normalizing the smell count using the project size reduces the bias of larger projects on the overall smell count [222]. 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 2.5

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 2.5: 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 2.2 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 (Figure 2.3).

We found that code smells in the Dispensable category had a mixed tendency (Figure 2.4); 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.

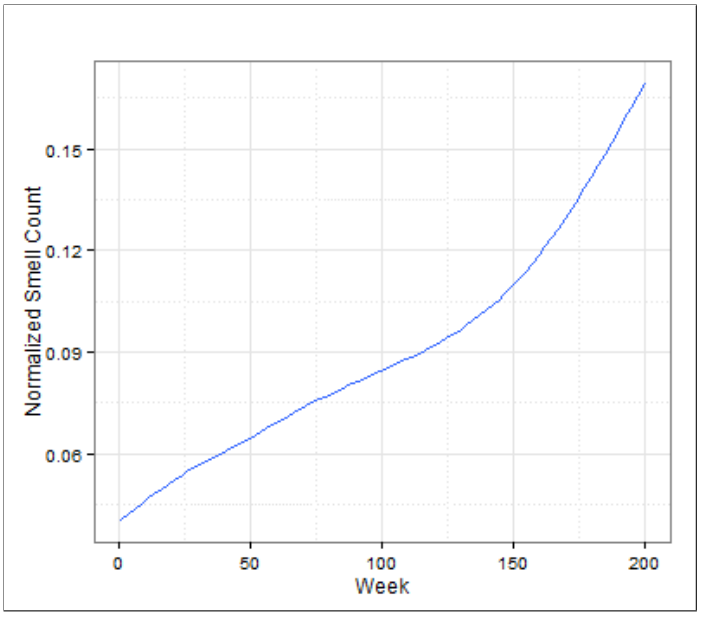

Figure 2.2: Week-wise average project smellines compared to the smelliest project

We found that code smells in the OO abusers category follow a mixed pattern also (Figure 2.5). 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.

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 (Figure 2.6).

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

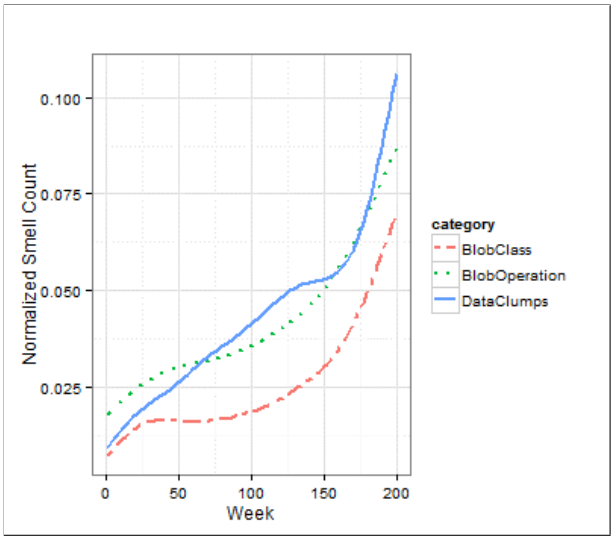

Figure 2.3: Week-wise average project smellines compared to the smelliest project of bloater category

the projects. Figure 2.7 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.

2.4.2 Who takes care of smelly code?

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

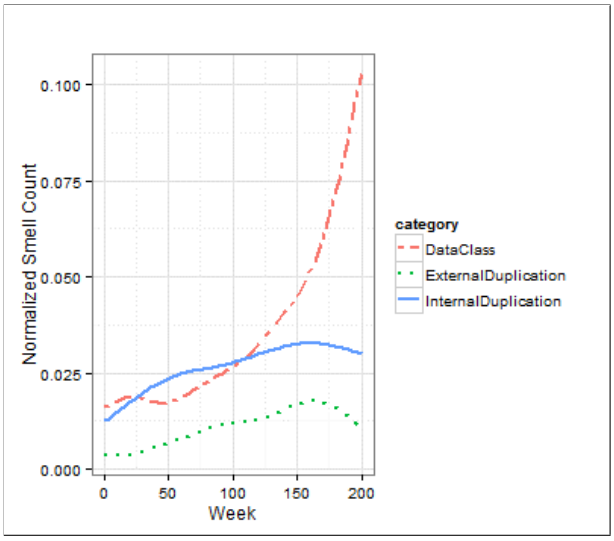

Figure 2.4: Week-wise average project smellines compared to the smelliest project of Dispensables category

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 (Figure 2.8). We do however see that core contributors appear to remove more smells than they introduce, whereas the inverse is true for non-core contributors (Figure 2.8). 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).

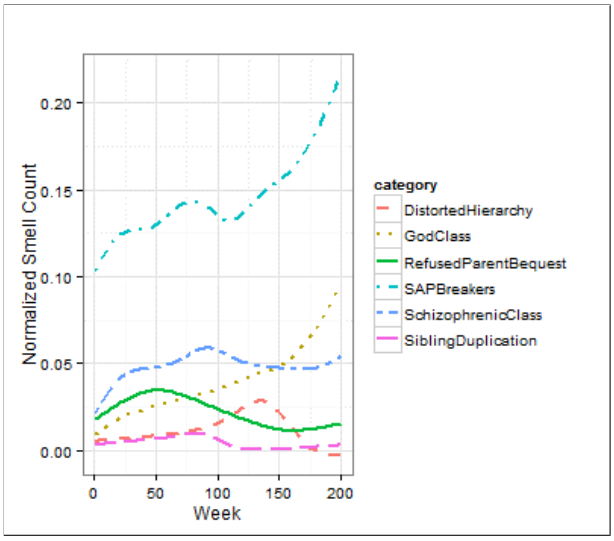

Figure 2.5: Week-wise average project smellines compared to the smelliest project of OO abusers category

2.4.3 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 2.9 shows the trend line found after plotting the normalized total count of test case and code smell count for each week, for all projects.

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 mu-

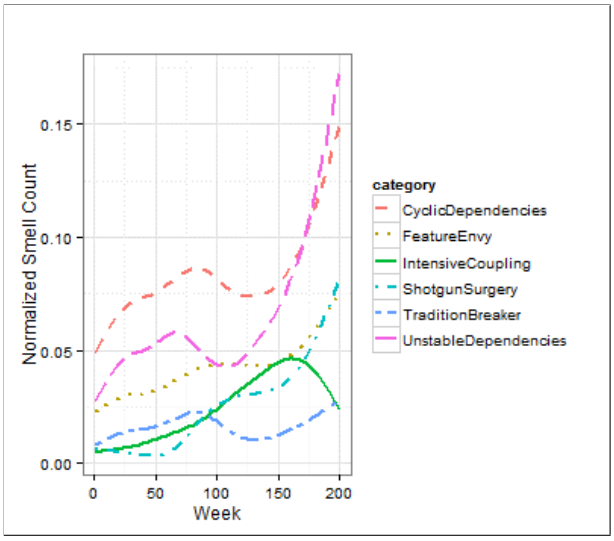

Figure 2.6: Week-wise average project smellines compared to the smelliest project of Other category

tation 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, pvalue = 0.01543, Statistically Significant), low and high coverage groups have statistically significant difference in number of code smells present in the project.

2.4.4 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 2.10 shows that, SAP Breakers, Data Class and Cyclic dependencies and Feature Envy were the most common smells, constituting almost 50% of the

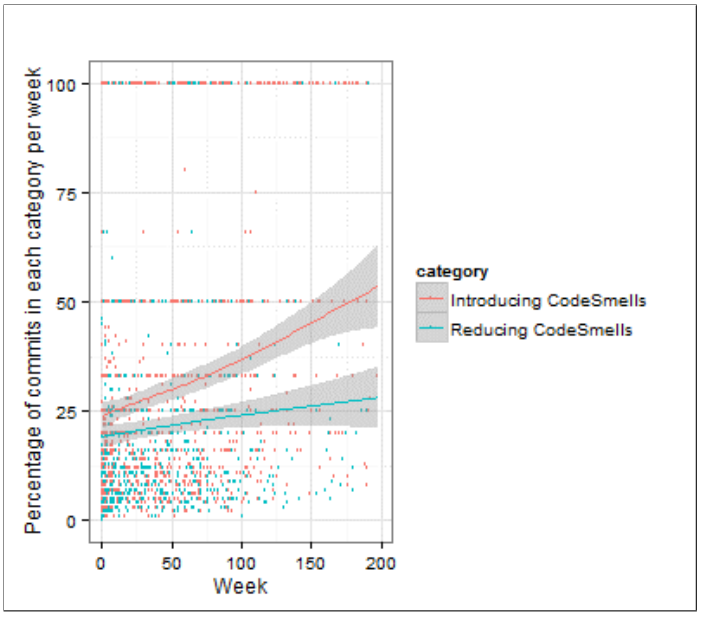

Figure 2.7: Breakdown of commits introducing(trend line on top) and removing(trend line on bottom) smells

code smells being 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. [269] and Sjoberg et al. [231]. 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 [269]. They identified 39 papers out of 319 papers that could answer the research question about which code smells receive most research. Sjoberg et al. [231] 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 2.6 we report the percent based ranking.

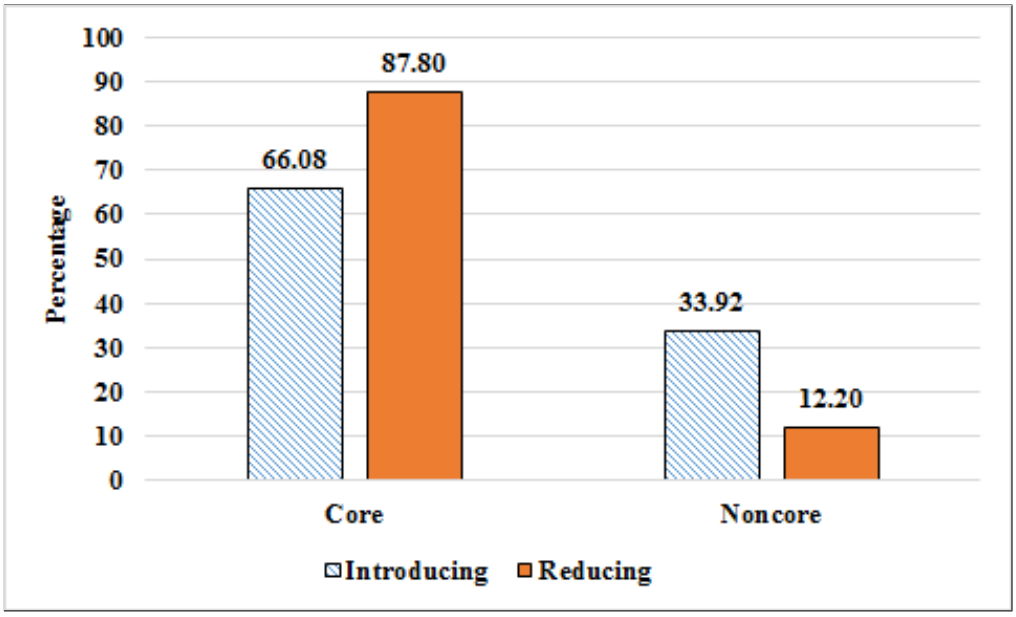

Figure 2.8: Origin of commits introducing or reducing code smells

2.5 Discussion

During our analysis we found that the overall number of smells increase over the life of open source projects, as shown in Figure 2.2. This is not to say that issues of design or technical debt are not addressed over the life of the project, but as Figure 2.8 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 2.7, 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

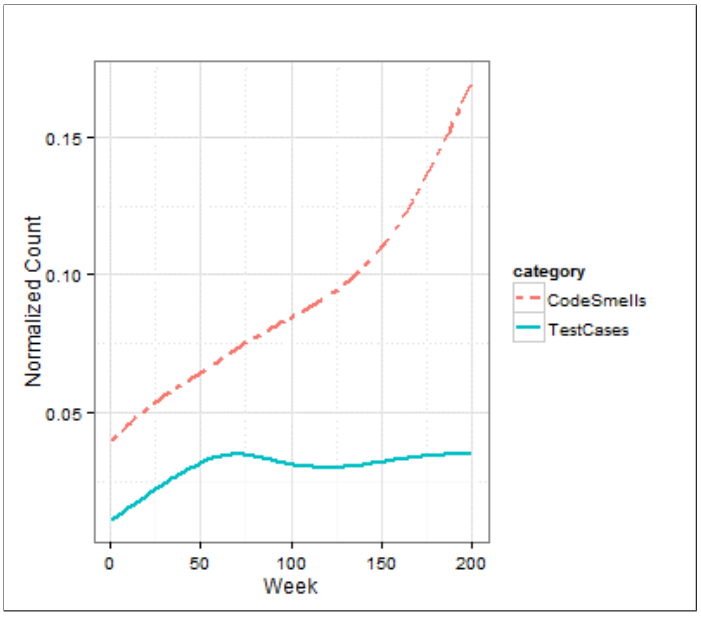

Figure 2.9: Comparison of Week-wise normalized total smell and test case count

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 2.3, 2.4, 2.5 and 2.6). 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 [91, 156, 192]. All code smells in the Bloating category show a growing tendency. This category is associated with centralized control structures in

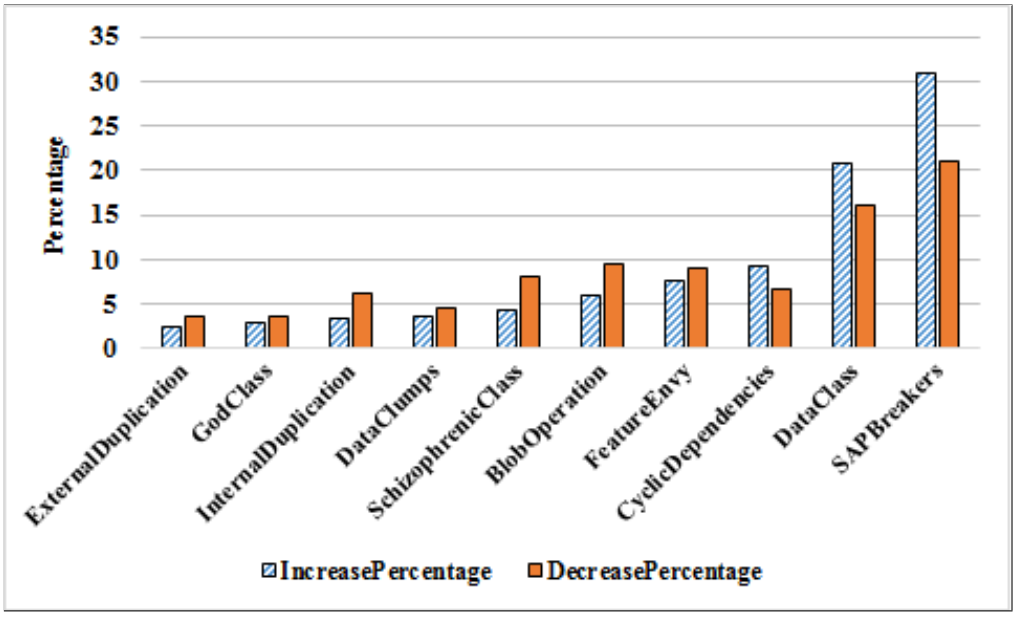

Figure 2.10: Breakdown of smells introduced and removed

object-oriented languages. Arisholm et al. found that novice developers perform better with centralized control styles [25], 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 [257] 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 2.5). 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

Table 2.6: 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

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 [257], 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 [257]. 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 [138]. 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 [257]. 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 [156, 192] and code maintainability problems [91] 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 [127]. 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. [87], 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 [127]. 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.

2.6 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 [197]. So the number occurrence of such "intrinsically historical" smells

should be different when historical information based smell detection technique is used.

2.7 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.

The Evolution of Software Entropy in Open Source projects: An Empirical Study

Umme Ayda Mannan, Iftekhar Ahmed, Carlos Jensen, Anita Sarma

Technical report 2020

Chapter 3: The Evolution of Software Entropy in Open Source projects: An Empirical Study

3.1 Introduction

During the lifetime, software systems change for various reasons, such as adding new features, fixing bugs, or refactoring. Due to time constraints, limited resources, and the lack of regular maintenance, these changes could deteriorate the structure of the software system from its original state, thus increase the source code complexity and, in general, make the system more challenging to understand and maintain in the future. Researchers have used entropy to quantify complexity and disorganization [35, 107]. As high entropy makes the code difficult to understand, it makes the code more bug-prone.

Hellendoorn et al. [114] observed that code quality is also associated with how developers write code. Programmers tend to be highly repetitive and predictable while developing real-world software [93]. Hindle et al. were the first to capture the repetitiveness in a statistical language model [117]. They called this property as the naturalness of code and measured it by entropy. If a code snippet has high entropy, that means the code is drifting from its natural state. Previous research shows that high entropic code is associated with future bugs [211] and design degradation [56]. Ray et al. [211] investigated if there is any correlation between the buggy code and entropy. They observed that buggy codes are, in general, less natural, i.e., they have higher entropy than non-buggy code. Chatzigeorgiou et al. showed that higher entropy also impacts the design quality

of a software system [56].

Previous research has investigated the impact of entropy on design quality as well as code quality. However, the overall situation of software entropy in real-life software has not been investigated yet. The goal of this study is to shed light on how entropy evolves as software ages and the relationship between entropy evolution and project evolution in terms of the number of contributors and number of methods and files. More specifically, we try to answer the following research questions:

RQ1: How entropy evolves over time in OSS projects?

RQ2: What is the correlation between entropy and project evolution in terms of project size and the number of contributors?

To answer our research questions, we conducted a large scale empirical study. We sampled 158 projects from GitHub [97] and calculated the week wise entropy for each project to investigate the evolution of entropy across OSS projects over 330 weeks. We also collected project metrics for each week for each project. Furthermore, we investigate the association between entropy and other project metrics by performing a cross-correlation as all of our data is time-series data.

The paper is structured as follows, Section 3.2 present our methodology, the demographics of our corpus, data collection, and analysis process. In Section 3.3, we present our findings. Section 3.4 discusses the results and outlines implications for developers and researchers. Section 3.6 provide a review of prior research efforts. Section 3.7 concludes with a summary of the key findings and future work.

3.2 Methodology

This empirical study aims to understand how entropy in OSS projects evolves and the relationship between entropy evolution and project evolution. For this study, we create a corpus of 158 open source projects. In the following subsections, we describe the pipeline we followed to collect, process, and analyze it.

3.2.1 Project Selection Criteria

To make sure our findings would represent the code developed in the real world, we selected active OSS Java projects from GitHub. We selected Java projects because Java is one of the most popular languages [17]. We adopted the initial project selection criteria from the study by Ahmed et al. [21]– searching and selecting initial project, project size, and the number of files in a project. Then we added additional criteria as discussed next for project selection.

We start by randomly selecting 900 projects by searching for java projects in the GitHub search engine. Next, to make sure the projects in our corpus represent real-world projects and not throw-away or class projects, we followed the guidelines by Kalliamvakou et al. [131]. Furthermore, following the guidelines, we eliminated the repositories that are not "actual" projects, have very few commits, are inactive, are not related to software development, or are personal projects. This left us with 500 projects. Then we selected only the projects that meet the following criteria,

- Project size (Must have more than ten files and 500 lines of code).
- Project age (More than ten weeks).

- Commit history (More than ten commits).

This resulted in our corpus containing 158 projects. Table 6.3 provides a summary of our selected projects.

Table 3.1: Project Statistics

Dimension	Max	Min	Average	Median	Std dev
Line count	804,658	686	34,582.51	11,188	83,536.64
File count	1,693	11	165.4845	87	253.5198
Total Commits	6,256	11	485.2981	183	847.7926
# Developers	157	1	19.37267	9	26.67907
Duration (weeks)	782	10	244	235	168.1799

3.2.1.1 Unit of measure selection

To investigate how entropy in OSS projects evolve, we could use different ways of partitioning time. Izurieta et al. [121, 122] have used releases as the unit of time, where others have used individual commits, or discrete-time units (years, months, weeks, days) [21, 170]. As individual commits would be too fine-grained for our purpose, we selected the week as our unit of measure similar to Ahmed et al. [21]. Because it gives us enough detailed insight into the evolution of projects. We then checked the distribution of commits across the projects' history and found that the majority (90%) of the projects had an active history of 330 weeks or less. We cut off our analysis at 330 weeks to prevent extremely long-lived projects from skewing the results.

3.2.2 Measuring Entropy

For our study we calculate week wise total entropy for each project. To get the total entropy, first we calculate the entropy for each java source file using Python's `Entropy` library [2]. The library uses Shannon's entropy [3]. Other researchers also used Shannon's entropy [48] $H(X)$, which is calculated using the following formula:

$$H(X) = \sum_{i=1}^{n} P(x_i) \log_2 P(x_i) \tag{3.1}$$

In Equation 3.1, X is a java method. And x_i is a term (in our case a token in a java method) in X. $P(x_i)$ is the probability of a change occurring in a method. n is the total number of tokens in each java method. We will get a maximum entropy when all tokens have the same probability of having a change: $P(x_i) = 1/, \forall i = 1, 2, \ldots, n$. We will get minimum entropy if for a token i, $P(x_i) = 1$, and for all other tokens, other than i, $P(x_j) = 0, \forall j \neq i$.

After collecting entropy for each java method, we sum up the scores to get entropy score for each java source file. Then for each week, we sum up the entropy scores of all java files in a project to get week wise total entropy for that project.

3.2.3 Collecting Project Metrics

To answer our second research question, we collected different project related metrics to investigate the correlation between project entropy and project evolution. Table 3.2 lists the metrics we collected for this purpose.

After collecting total entropy for each project, we collected the above-mentioned project metrics (Table 3.2) from our data set. We use Understand [4] to collect the

Table 3.2: Project metrics used for investigating project's evolution.

Metric	Description
Total methods per week	Week wise number of unique methods for each project.
Total contributors per week	Week wise number of unique contributors for each project.
Total files per week	Week wise number of unique files for each projects

number of files and number of methods for each week, for each project. We collect the number of unique contributors per week from the project repository by counting the number of developers who made commits in the codebase in a week.

3.2.4 Data Analysis

We calculated the total entropy and other project metrics (mentioned in table 3.2) for each project over 330 weeks. To compare these time series, first, we normalized the data since the number of metrics will vary according to the size of the development team and project. There are many ways of normalization, and the most commonly used one is dividing the data by the lines of code. Since the aim of or study is to identify general trends across projects, not to look at differences between them, we normalized all the week wise collected metrics using the feature scaling [40], which gives a score between 0 and 1 (Equation 3.2).

$$\text{Rescaled value} = \frac{x - min(x)}{max(x) - min(x)} \tag{3.2}$$

Where,

x = each data point.

min(x) = The minimum among all the data points.

max(x) = The maximum among all the data points.

One of our goals is to identify the correlation between total entropy per week and the other project-related metrics per week. We calculate the cross-correlations between entropy per week, and total methods per week, total contributors per week and total files per week individually for each project [177]. As the first step of time series analysis, we start by checking if there are any visible trends. If a time series exhibits a visible trend, we need to remove the trend before further analysis. This process is called detrending. We applied the first differencing method to detrend time series [177], and after that, we calculate the cross-correlation.

3.3 Results

In the following sections, We structure our study findings based on our research questions.

3.3.1 How entropy evolve over time

To answer our first research question, we start by looking into the general trend of average entropy across all projects. Figure 3.1 shows the *increasing* trend of the average entropy across all projects.

We also wanted to check whether this same trend holds for all projects. When we looked into each of the projects individually, four different trends emerged. The four categories are:

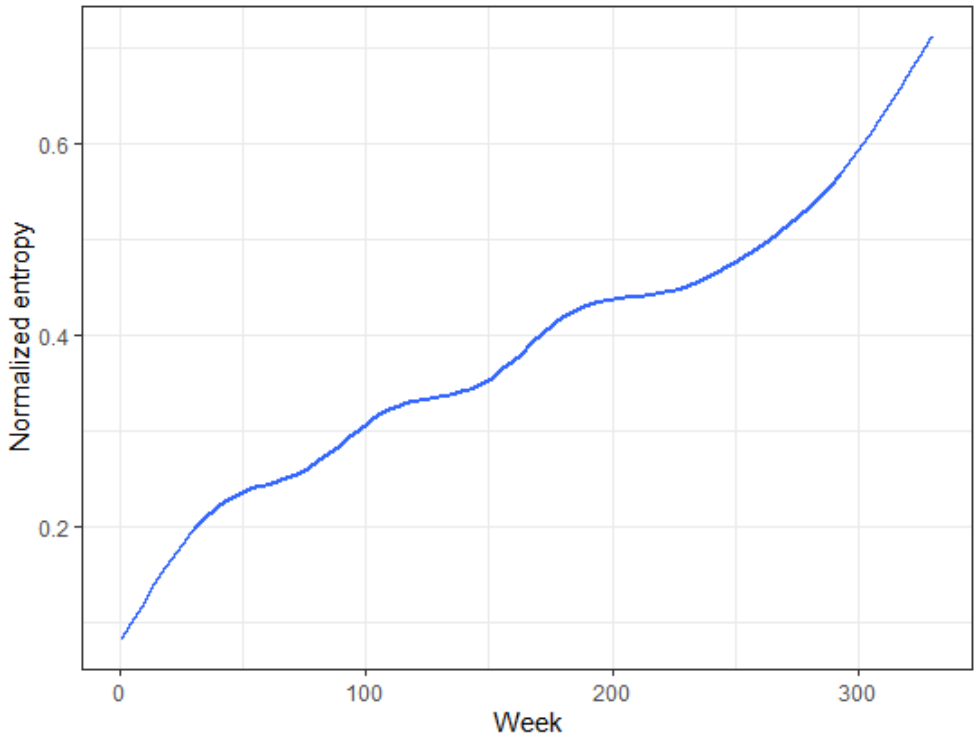

Figure 3.1: Week-wise average entropy for the projects.

Cat 1: Projects start with low entropy and increase over time.

Cat 2: Projects start with low entropy and remain low over time.

Cat 3: Projects start with high entropy and increase even more over time.

Cat 4: Projects start with high entropy and decrease over time.

Figure 3.2 presents examples of these trends using specific projects from our corpus. Category 1 includes projects that start with low entropy and increase as the project matures, this manifests in the "artifactory-client-java" project. Category 2 includes the projects with a constant low entropy trend over the time; "jitescript" is an exemplar of

this trend. Category 3 includes projects that start with high entropy and increase over time, as seen in the project "gelfj". Furthermore, category 4 includes the project that starts with a high entropy but decreases over time, as seen in the project "jmimemagic". Table 3.3 shows the percentage of projects in each entropy trend category in our corpus. We also check if these trends are statistically significant or not and found that 76% of all projects have statistically significant trends.

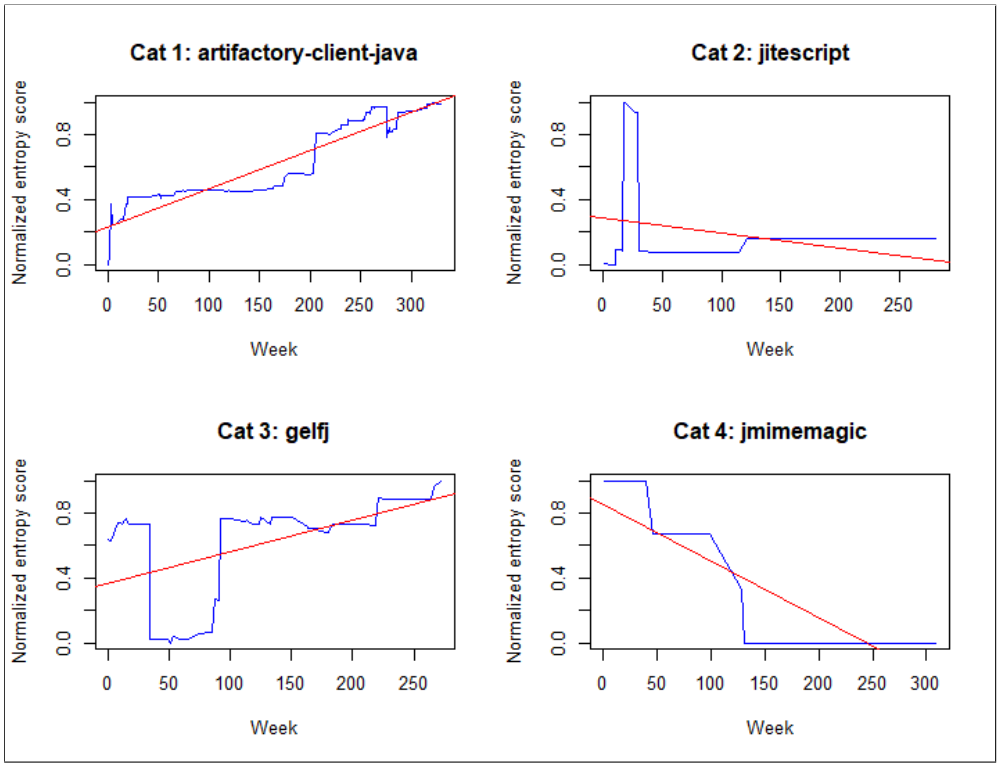

Figure 3.2: Week wise project's entropy trend.

Observation 1: Different types of entropy trends exist, however on an average entropy increases over time.

Table 3.3: Number of projects in each entropy trend category

General Trend	Number of projects(%)
Cat 1	69.62%
Cat 2	2.53%
Cat 3	15.82%
Cat 4	12.03%

3.3.2 Association between entropy and other project metrics

To answer our second research question, we investigate the correlation between entropy and other project metrics. As entropy is related to source code, we picked the project metrics that are also related to source code, total methods, total files, and total contributors.

We start by looking into the general trend of the average number of methods, files, and contributors across all projects. Figure 3.3 shows a *increasing* trend for each of the project metrics across all projects.

Since entropy per week, methods per week, contributors per week, and total files per week are time series data, a time series analysis is required to identify the correlation between them, called cross-correlation. We do time series analysis between entropy and each metrics(mentioned in table 3.2) individually for each project. Time series analysis requires a pre-processing step before doing the actual correlation analysis [177]. Due to space limitations, we report the results after each pre-processing step in the companion website[162].

Next, we calculate the correlation between each pair of the following time series data, entropy and total methods, entropy and total developers and entropy, and total files.

Figure 3.4 shows the result of this step for each pair as an example. From the figure, we

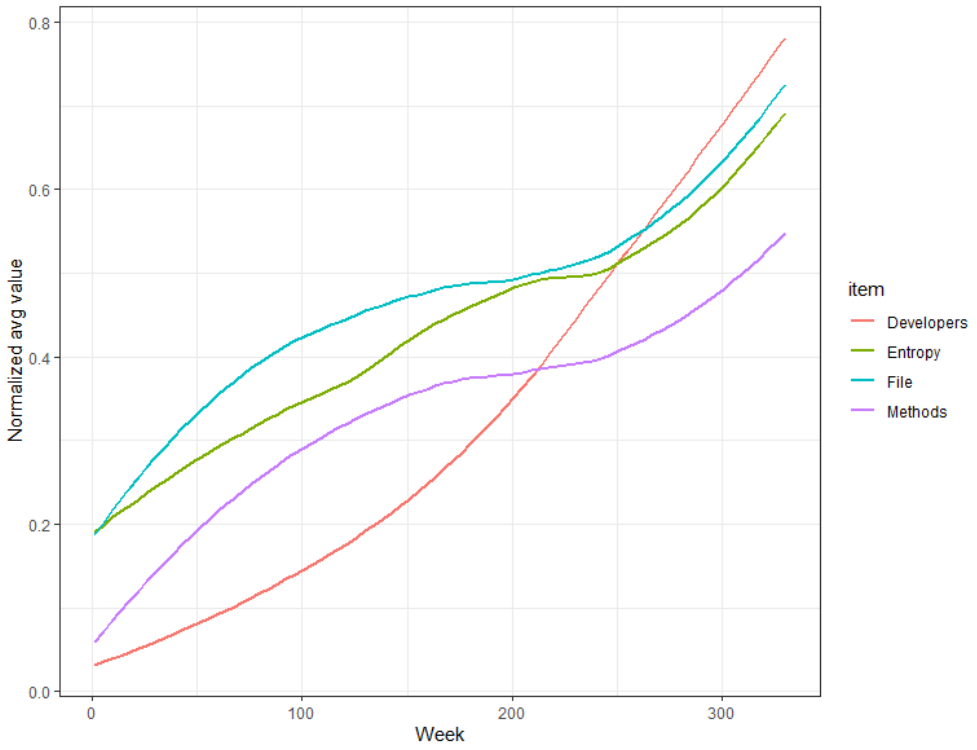

Figure 3.3: Week-wise average for each metrics across all projects.

see that the highest correlation value between entropy and method for "argparse4j" project is 0.58 (shown by the circled vertical line in (a) at a lag of 7. Similarly, (b) in figure 4.6 shows the correlation between entropy and developer with the highest correlation value of 0.56 for the project "clj-ds" and (c) shows and the correlation between entropy and total files for project "cloudhopper-smpp" with the highest correlation value of 0.47 at lag 6. Table 3.4 shows the statistics of the cross-correlation values with the average lags for each pair of time series across 158 projects.

Next, we calculate the significance of the correlation for each project for all three pairs. A correlation is significant when the absolute value is greater than, $\frac{2}{\sqrt{n-|k|}}$, where n is the number of observations and k is the lag [9]. We found that 150 out of 158

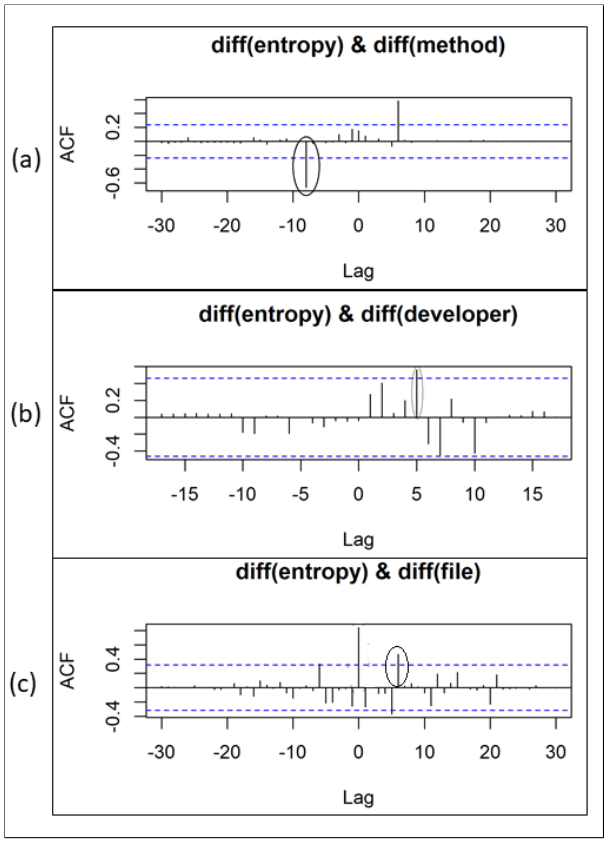

Figure 3.4: Cross correlation values of two time series

projects have ave statistically significant cross-correlation between entropy and number of methods. For the pair entropy and developers, we found that 135 projects have a statistically significant correlation, and between entropy and total files, 143 projects have a statistically significant correlation. We also looked into the lags between the pair of time series. The fourth column of the table 3.4 shows the average lags for each paper of time series. Due to space limitations, we report the plots for each paper, for each of the 158 projects in the companion website [162].

Observation 2: Entropy is moderately correlated with other project metrics.

Table 3.4: Distribution of the correlation value of each pair of time series.

	Maximum	Minimum	Median	Avg lags
Entropy and Methods	0.94	0.067	0.45	7.48
Entropy and Developers	0.94	0.082	0.36	10.51
Entropy and Files	0.96	0.07	0.35	7.5

3.4 Discussion

Our analysis found that the overall entropy scores increase over the lives of OSS projects (Figure 3.1). One of the reasons behind this increasing trend could be the practice of software developers implementing quick fixes with immature design solutions instead of the optimal solution due to the time constraints. Over time, these bad design solutions build-up, and software design degrades, and entropy (which is a measurement of software degradation [35]) increases.

However, when we investigate the entropy trend for each of the projects over 330 weeks, we found four distinct categories of trends, as mentioned in section 3.3.1. From our analysis, we found that 76% of the projects' trends are statistically significant. Among these four categories, category 2 and category 4 showed a low and decreased entropy over time. Though these categories have a small number of projects (See table 3.3), we decided to manually check some of them to find out if there is any specific practice that helping these projects to maintain or decrease the entropy of their project. We selected one project from category 2 and 3 projects from category 4 and went through the GitHub pages for these projects. For all five projects, we noticed that the instructions for contributors are well defined. These projects provided specific information to contributors

regarding the structure of the codebase, which package contains important classes, preferred development tools, testing instructions on how to format the code, specific coding style, etc. These projects also have specific instructions on where to report a bug, if they are using any issue tracking system or not. One of the projects named "mapdb" has a section called current news, where the project highlights the ongoing works. Most of these things are missing from the projects with an increasing entropy trend. More research is needed before drawing any guidelines that could help the OSS projects to keep low entropy as we only looked into the GitHub pages for five projects.

Next, we looked at the correlation between entropy and other project metrics in a time series data. We found that, on average, entropy moderately correlates with total methods, files, and contributors. As contributors are associated with entropy, it will be interesting to investigate how they contribute in increasing entropy, is it their lack of knowledge of software entropy or lack of tool support or lack of knowledge on the codebase, etc. Further research is needed to answer these questions.

Previous research showed that entropy is correlated with bugs [211, 107] and fail to maintain low software entropy will degrade the overall software quality. One of the possible ways to decrease software entropy could be a regular discussion on software design and keeping the design document updated. However, for OSS projects, this step will be very hard as developers of OSS projects are geographically distributed, and for it's voluntary characteristics, it is scarce for an OSS project to have updated design documents or in some cases, design documents.

To the best of our knowledge, no popular IDEs report on entropy. As monitoring entropy is essential for projects, developers need a tool that monitors entropy of their working source code in real-time and provides developers information about current entropy scores and the links between the working class/method to other classes/methods.

3.5 Threats to Validity

Our 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 could not be mitigated, and it is possible that our mitigation strategies may not have been sufficient.

All the projects in our corpus have been collected from a single source, GitHub. Since we used only GitHub, our findings may be limited to open source projects from GitHub. However, we believe that a large number of projects sampled more than adequately addresses this concern.

All our subject projects are written in Java, but entropy is not specific to Java. Any programs written in a programming language will be prone to a high entropic situation.

Our set of project metrics are only related to source code and developers who write the code. We selected these metrics to see the correlation between entropy and project evolution in terms of it's size and number of contributors. However, we can not guarantee that our set of metrics is exhaustive.

We selected the week as our unit of measure for this study, where previous research used different ways of partitioning time (releases, commits, months, year, etc.). However, our goal was to identify general trends of entropy across projects, week as the unit of measure gives us enough detailed insight into the evolution of projects.

3.6 Related Work

Entropy is a metric used to measure the naturalness of code. Bianchi et al. [35] have initially introduced software entropy. It represents an overall measure of the degree of quality degradation induced by maintenance interventions, which tend to make code

increasingly "chaotic". Tan and Mookerjee [243] analyze the effect of software entropy on maintenance costs on a sample of closed source software projects.

Olague et al. [191] used entropy as one of the metrics to explain the changes that a class undergoes between versions of an object-oriented program. According to the study, high entropic classes tend to change more than classes with lower entropy. Yuet al. [263] combined entropy with component-dependency graphs to measure component cohesion. Entropy was also used by Snider [235] in an empirical study to measure the structural quality of C code.

Chatzigeorgiou et al. [56] proposed entropy is as a design quality metric for object-oriented programs. The difference between the two systems' entropy provides insight into the quality of the design in terms of how flexible it has been during the enhancement of its functionality.

Entropy has also been used for bug prediction. In a study, Ray et al. [211] found a correlation between buggy code and entropy. According to their study, buggy code has higher entropy than non-buggy code. Hassan et al. [107] compared the entropy with the number of changes and the number of previous bugs in a bug prediction model and found that entropy is a better predictor. D'Ambros et al. [66] extended Hassan's [107] work and found that source code metrics better describe the entropy of changes. Canfora et al. [48] found that the change entropy decreases after any refactoring of the code. Chakraborty et al. [54] used entropy score from statistical language models and used it for spectrum-based bug localization (*SBBL*). They also found a significant improvement compared to standard *SBBL*. In a recent study by Zhang [271], they used cross-entropy with traditional metric suites in a defect prediction model and found that the performance of prediction models is improved by an average of 2.8% in F1-score.

Previous studies have investigated entropy from different perspectives. However, to

the best of our knowledge, no studies have investigated the evolution of entropy in OSS projects and their relationship with project evolution.

3.7 Conclusion

In this study, we aimed to understand how entropy evolves in OSS projects over time. We study the history of 158 open source projects and found strong evidence that though different types of entropy trends exist, however on average, entropy increases over time. In our corpus, we identified four different entropy trends.

In our study, we investigate the correlation between entropy evolution and project evolution (in terms of size and number of contributors). To find the correlation between entropy and other project metrics, we conducted cross-correlation analysis as all the data in our corpus are time series. We found that entropy has a moderate correlation with all the project metrics, methods, contributors, and files (see table 3.4). In our manual inspection of the projects that show a decreasing trend in entropy evolution, we found that the GitHub page of these projects is well documented for the user and the contributors.

Our work showcases that further research is needed to understand which activities or practices by the project community could maintain low entropy over time. The researcher community should also try to come up with tools to help the projects manage the chaos that high entropic code makes over time.

On the Relationship Between Design Discussions and Design Quality : A Case Study of Apache Projects

Umme Ayda Mannan, Iftekhar Ahmed, Carlos Jensen, Anita Sarma

To appear in Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE),2020.

Chapter 4: On the Relationship Between Design Discussions and Design Quality: A case Study of Apache Projects

4.1 Introduction

Good design is key to ensuring performance, robustness and maintainability of software [250]. To be able to create "high quality" products developers need to not only be aware of the design of the system, but also about the underlying design decisions. If developers are unaware of the design decisions they can make changes contrary to what was intended, setting others up for clashes, which in turn can degrade the overall design quality of the system [249].

Often, design decisions are decided and shared informally among developers of a project [137], making it a key a challenge to document and share design decisions [137]. This becomes a bigger challenge for open source software (OSS) for several reasons. First, these projects are distributed, involving 100's of developers who are geographically or temporally separated. This means that purposeful discussions are the only way for developers to coordinate and build together. Second, because of their often volunteer nature, OSS projects rarely produce updated design documents and in some cases lack formal design documentation altogether [46]. In such cases, design discussions are the only form of design documentation.

Recent work has started to investigate how OSS developers discuss design, and how they find these design discussions later. For example, Brunet et al. [46] and Viviani et al.

[251, 250] examined how design discussions are embedded in pull request comments and how it can be difficult for developers to piece together these discussions. Researchers have also developed techniques for detecting design discussions from a (single) communication channel using Machine Learning (ML) techniques [46, 251, 250, 161].

However, we still have gaps in our understanding of how design discussions are conducted in OSS projects—What channels are used? Who participates in these discussions? Who implements design-related changes? Answers to these questions are especially pertinent for OSS projects since they are known to use different channels (project mailing list, issue tacking systems, pull requests, code, etc. [46]) to discuss their work. Design discussions fragmented across channels can make it difficult for developers to retrieve and reconcile the scattered pieces of design discussions. This was indeed the case as one of the respondents in our survey of 130 software developers mentioned "*discussions are fragmented between several[sic] systems and not structured for easy searching*".

Even if developers are able to find the design discussions, it is yet unknown to what extent design discussions in OSS actually relate to the design quality of the project. Without such knowledge, OSS projects might be discussing design under incorrect assumptions (of how design decisions can be retrieved and used) and researchers might miss opportunities for improving the state of the art. To the best of our knowledge, no prior work has investigated how design discussions are fragmented across multiple channels in OSS and the relation between design discussions and design quality.

In this work, we seek to understand two broad aspects of design discussions in OSS projects:

RQ1: How are design related discussion conducted in OSS projects?

RQ2: What is the relationship between design discussions and design quality in OSS

projects?

To answer our research questions, we performed a mixed method empirical study of 37 Apache projects. These projects use multiple communication channels (according to their official documentation) making them ideal candidates for our study. We opted to investigate Apache projects because these projects have well documented discussion procedures. Apache projects initiate discussions on mailing list and once decided the high level design, the discussion moves to issue tracking system and finally the code is submitted using pull request, where another round of discussion may occur [89].

We started by extracting design discussions from multiple channels (developer mailing list, issue tracking systems–Jira and Bugzilla, and pull requests). This gave us a corpus of $1,661,922$ distinct discussions. We then developed an ML classifier that can identify design discussions from either of the four aforementioned communication channels. Applying the classifier on our corpus, we found that overall 9.34% of all discussions touch on some design aspect. We then validated our findings via a survey of 130 developers from the Apache projects.

Finally, we evaluated the association of design discussions with design quality. Design quality could mean different things to different people. For example, it could mean poor feature selection, terrible UX design or bad architecture design. While trying to identify a suitable metric for measuring design quality, researchers have investigated different metrics such as Coupling Between Object(CBO), COupling Factor(COF), Weighted Method per Class(WMC), and code smells etc. [50, 69, 156, 193, 167, 21]. We opted to use the occurrence of code smell as a proxy for (poor) software design quality since a large number of studies have used and validated code smells for the aforementioned purpose [181, 69, 156, 193, 167, 21, 181, 20, 163, 78, 237, 238].

Our paper makes the following contributions:

- We present a first study investigating the relationship between design discussions and design quality.
- We present the results of a survey of 130 software developers examining the difficulties that developers face in locating and acting on design discussions in OSS projects.
- Based on our results, we outline implications for developers and researchers, make suggestions for improving existing tools and guidelines to better support design discussion.

The paper is structured as follows Section 4.2 provide a review of prior research efforts. In Section 4.3, we present our methodology, the demographics of our corpus, approach of mining discussions, machine learning classifier to classify discussion, data collection process and surveying developers for answering our intended research questions. In Section 4.4, we present our findings. Section 4.5 discusses the results and outlines implications for developers and researchers. Section 4.7 concludes with a summary of the key findings and future work.

4.2 Related Work

In the context of software engineering, design is defined both as a process [92] and an artifact [210]. Researchers have investigated both aspects. While investigating the process aspect, researchers looked into how and where developers discuss design. Previous studies [46, 233] show that in OSS projects design discussions takes place in platforms

like email communications, issue trackers, pull request etc. Brunet et al. [46] found that 25% of the discussions are about design. Though many channels are available for discussion, all prior studies focused on analyzing discussions in only one channel [251, 250, 46]. Since many channels are available, discussions can be fragmented and an in-depth analysis involving all channels is required to gather a better understanding about how and when design discussions happen. To the best of our knowledge, ours is the first study analyzing multiple communication channels in this context.

Researchers have also looked into the artifact aspect of design. Though OSS projects share most of the information in written form, it is very often to find an updated design document in a project's archive. Brunet et al. [46] examined 90 popular projects from GitHub and found that 68% of these did not have any kind of design documents. Cherubini et al. found that developers mostly convey design decisions through temporary drawings [58]. Soria et al. [236] found that design knowledge generated during verbal meetings was not captured in the project archive. Researchers also found that design decisions are lost over time [58].

Since design in OSS is not well documented, recovering it becomes difficult with time. To help recover design decisions, researchers have proposed many techniques. One technique is recovering design by reverse engineering [60] or inferring from structural designs from code and logs [183]. Antoniol et al. [22] showed that design can also be recovered by tracing links between code and documentation. Another study recovered architectural design from source code with commits and issues [226].

Researchers have also applied ML based techniques to retrieve design discussions. Brunet et al. applied a machine learning classifier to automatically label design related discussions [46]. Viviani et al. [250] applied a classifier to automatically locate paragraphs in pull request discussions that related to design. Mahadi et al. [161] trained a classifier

on the dataset created by Brunet et al. [46] and tested it on the dataset of Viviani et al. [250]. However, both of the dataset include discussions only from pull requests. In our study, we build a machine learning classifier that automatically label design related discussion from different communication channels (project's mailing lists, issue tracking systems and pull requests).

Different ways of measuring design quality has been investigated. Many researchers [69, 156, 193, 167, 21, 181, 20, 163, 78, 237, 238] have used code smells [91] to quantify design quality. Code smells are symptoms of poor design and implementation choices [91]. Initially, it was introduced to identify potential maintainability issues in software system, however later on it has been associated with bugs [156, 192, 139], and fault-proneness [104, 265]. Researchers have also investigate how design quality evolves over time and found that small changes accumulating over time cause the design drift [249]. Ahmed et al. [21] found that software design quality, measured using code smell, gets worse over time. Researchers have also studied the relationship between project activities and design quality [181, 201] and found that code reviews have a significant influence on the reduction of code smells. However, there are no studies investigating the relationship between design discussions and design quality. Our study takes the first step towards filling this gap in research.

4.3 Methodology

The goal of this study is to understand how design discussions take place in OSS projects and the relationship between design discussions and design quality. In the following subsections we describe the pipeline we followed to collect, process and analyze the data.

4.3.1 Data Collection

4.3.1.1 Project selection

Our overarching goal is to identify design discussions happening in OSS projects and investigate their relationship with design quality. We start by selecting 37 Apache projects written in Java. We selected projects from Apache because they use multiple communication channels (mailing list, issue tracking system, pull request etc.) and have specific instructions for using these channels. Selecting random projects from GitHub without well documented discussion procedure would make it difficult to ensure that we have identified all communication channels. Since our goal was to understand how discussions are scattered across all channels, identifying all channels is of the utmost importance. We selected projects written in Java since it is one of the most popular programming languages [17], and there are more design quality (code smell) detection tools available for Java than other languages [87]. We downloaded the git repositories for these projects to collect various information such as code smell,project duration, developer information etc. Table 4.1 provides a summary of our selected projects.

Table 4.1: Project Statistics

Dimension	Max	Min	Average	Median
Line count	18,474,542	82,303	1,770,088.82	1,074,659.50
Duration (weeks)	1,063	214	606	560.14
# Developers	1,852	21	226.44	105.50
Total Commits	80,277	3,561	22,688.05	18,083.50
Total Discussions	117,995	594	86,599.33	38,195

4.3.1.2 Discussion collection

For these 37 Apache projects, we found the list of communication channels used by them. From developer mailing list we collected $1,437,753$ email bodies as discussion. We also collected $67,027$, $134,312$ and $22,362$ discussions from Bugzilla, JIRA and pull requests respectively. The entire thread of interactions (emails-grouped by the subject header, comments grouped by issueID) was the unit of analysis and considered a discussion. In total, our initial data set contained $1,661,922$ discussions.

4.3.1.3 Unit of measure selection

To investigate how design discussions correlate with design quality over time, we could use different ways of partitioning time. Some researchers [121, 122] have used releases as the unit of time, other have used individual commits, or discrete-time units (years, months, weeks, days) [21, 170]. Individual commits are the only "level" measure but would be too fine-grained for our purpose. Thus, similar to Ahmed et al. [21], we selected the week as our unit of measure because it gives us enough detailed insight into the evolution of projects. We then checked the distribution of commits across the history of the projects and found that the majority (90%) of the projects had an active history of 560 weeks or less. We cut off our analysis at 560 weeks in order to prevent extremely long lived projects from skewing the results.

4.3.2 Building the Discussion Classifier

To answer our research questions, we need to separate design discussions from other discussions. As manual classification is not a practical option to classify $1,661,922$ discussions, we use machine learning techniques. We followed the protocol of Brunet et al. [46] with some improvisations suggested by Mahadi et al. [161].

4.3.2.1 Step 1: Manual classification of Sample data

We wanted to make sure that our manually labeled dataset of discussions contains discussions from all four channels. For this, we selected discussions for manual labeling by following the steps: (1) initial random selection of 500 discussions from each channel to ensure each channel is represented, and (2) an additional random selection of $3,000$ discussions agnostic of the channel type. In table 4.2, column three represents the total number of discussions taken from each channel after these two steps.This resulted in total $5,000$ discussion which were used to build and evaluate our ML classifier.

Next, two of the authors manually labeled $2,000$ discussions independently (40% of the corpus). We did not discuss any specific rules except for focusing on the structural design aspects of the code to classify the discussions. This was to avoid bias in our independent classification of the discussions. Also, our initial manual investigation of the discussions found that keywords used in design-related discussions are diverse. As a result, only focusing on keywords would not be able to capture all the design discussions. Instead of focusing on keywords, we focus on the semantics. While reading the discussion, the researchers paid attention to discussion regarding "structural design", "code architecture", anything regarding the "restructuring of code", and those kinds of high-level topics. To

remove subjectivity, multiple researchers labeled each discussion either as a design or non-design discussion. Then we calculated the inter-rater reliability using Cohen's Kappa and found a Kappa value of 0.88. Cohen's kappa is a statistic that assesses the degree of agreement between the codes assigned by two researchers working independently on the same sample [190]. Values of Cohen's kappa fall between 0 and 1, where 0 indicates poor agreement and 1 indicates perfect agreement. According to the thresholds proposed by Landis and Koch [148], kappa value of 0.88 indicates an almost perfect agreement between the researchers. Once the agreement was reached, one author manually classified the rest of the sample data. In our final sampled data, 998 (20%) discussions are design related and $4,002$ (80%) are non-design related discussions. Table 4.2 shows the distribution of manually classified sample discussions across channels.

Table 4.2: Distribution of manually classified sample across different Channels.

Channel	Total Discussions	Total Sample Discussions	# of Design Discussion	# of Non Design Discussion
Mailing List	1,437,753	3,000	600	2,400
Bugzilla	67,027	647	129	518
JIRA	134,312	797	159	638
Pull Request	22,830	556	110	446
Total	1,661,922	5,000	998	4,002

4.3.2.2 Step 2: Natural Language pre-processing

We followed some pre-processing steps to clean the data before applying a machine learning classifier and used the NLTK library [12] for this. We remove stop words and applied Lemmatization to normalized the data. We added some domain-specific words in the

stop word list that are not part of the predefined English stop words list [13]. For example, we added name of the days in week, name of the months, special character sequence such as "»" or "#" etc. The full list of stop words can be found in our companion website [254]. Then we use lower-case letter conversion. After removing stop words and doing the letter-case conversion, we used TF–IDF for vectorizing the natural language words into numerical vectors. More specifically, we used TfidfVectorizer [16] provided by Scikit-Learn.

4.3.2.3 Step 3: Machine learning classifier

Using the manualy classified corpus, we train four different machine learning classifiers, a Multinomial Naive Bayes (MNB) [11], a Decision Tree(DT) [8], a Support Vector Machine (SVM) [15] and a Logistic Regression (LR) [10]. We use Python Scikit-learn library [223] to implement the classifiers. We used 10-fold cross validation to train and evaluate the classifiers [125]. This validation approach randomly divide the manually classified data set into 10 groups of equal size. The first group is treated as a validation set, and the classifier is fit on the remaining 9 groups. The mean of the 10 executions is used as an estimation of classifier's accuracy. 10-fold cross validation has been recommended in the field of applied machine learning [147].

After 10-fold cross validation, we compared the classifiers using the Area Under the Receiver Operating Characteristic Curve (AUC) [83]. AUC returns a scores from 0 to 1 to represent prediction performance of a classifiers. A classifier with an AUC score higher than 0.5 indicates that it is performing better than random chance. An AUC score above 0.7 is often considered to have adequate classification performance [214]. We choose to use AUC instead of F1-score for comparing classifier performances for several

reasons. First, it is independent of prior probabilities [33]. Second, AUC is not biased by the size of test data. Finally, AUC provides a "broader" view of the performance of the classifier since both sensitivity and specificity for all threshold levels are incorporated in calculating AUC [209].

Table 4.3: AUC for classifiers

Classifier name	AUC
Decision Tree	**0.86**
Logistic Regression	0.77
SVM	0.74
Multinomial Naive Bayes	0.50

Table 4.3 shows that the Decision Tree achieved the highest AUC score of 0.86 compare to other classifiers. We, therefore, used the Decision Tree classifier to label the remaining discussions automatically. We performed hyper-parameter optimization using randomized search [32] for all four classifiers before selecting the decision tree. Randomized search sets up a grid of hyper-parameter values and selects random combinations to train the model and score on the validation data. The number of search iterations is set based on time/resources; in our case, it was 10. We search for the optimal parameter settings using the Scikit-Learn RandomizedSearchCV function. Our hyper-parameter grid for the best classifier (Decision Tree) includes "max_depth","max_features", "min_samples leaf" and "criterion". The final tuned decision tree parameters for our classifier were "min_samples_leaf: 5, max_depth:3, max_features: 5, criterion: gini".

4.3.3 Developer Categorization

To answer our research questions, we needed to identify the status of developers in the project and their participation in discussion. We used the number of commits contributed by individual contributors in the code base as the criterion to classify developer as core or non-core of the project [179, 21].

In order to calculate the status of developers, we start by collecting developer names and corresponding email addresses from project repository. After doing so, we noticed some miss matches. For example, multiple slightly different names with the same email addresses and same name with different email addresses. In our data set, 45% of the developer names had this issue. To identify the unique list of developers, we did a two way matching of name and email address. First, we identified all names attached to each email address and also all email addresses attached to a name. Then we identify all unique pairs of name and email addresses. We were able to match 98.5% of email addresses with names.

Next, we categorized developers into core and non-core groups according to their code contributions in the projects. Open source contribution follows a power law, where 20% of contributors are responsible for 80% of the contributions [179]. We follow the same rule to identify the core and non-core developers. We consider a developer as *core* if the developer is among the top 20% of developers in that project (calculated by the number of commits authored). Otherwise the developer is *non-core*.

4.3.4 Code Smell Collection

To examine the relationship between design discussions and design quality we collected code smell (indicator of design quality) for the 37 selected projects.

4.3.4.1 Code Smell Detection Tool Selection

We used inFusion [7, 195], a commercial tool to identify code smells. We selected inFusion for several reasons: First, inFusion detects a wide range (20) of code smells [195]. As a commercial tool, inFusion is no longer available for download. However, there is an open source version of the tool called iPlasma [6], which is available. It uses static code metrics to identify design related issues. Details of the metrics along with the definition of the code smells are provided in the companion website [254]. Second, a previous study by Ahmed et al. [21] showed that inFusion has a high precision of 0.84, recall of 1.00 and an F-measure of 0.91. Also, inFusion scales well to large code bases. Finally, many other studies [84, 88, 119] used inFusion as their code smell detection tool.

4.3.4.2 Measuring Code Smells

For each of the 37 projects, we collected code smells by running inFusion on their code base. First, we collected the number of code smells in each code smell category per week. Then we add all smells across all categories to collect total code smells for each project.

4.3.5 Data Analysis

We calculated the number of design discussions and code smells for each project over a period of 560 weeks. To compare these two time series, first we normalized the data since the number of discussions and code smells will vary according to the size of development team and project. There are many ways of normalization and most commonly used one is dividing the data by the lines of code. However, in our case, dividing the number of discussions by total lines of code does not necessarily give us a meaningful measure. Instead, we normalized both the number of design discussions per week and number of code smells per week using the feature scaling [40], which gives a score between 0 and 1 (Equation 4.1).

$$\text{Rescaled value} = \frac{x - min(x)}{max(x) - min(x)} \tag{4.1}$$

Where,

x = each data point.

min(x) = The minimum among all the data points.

max(x) = The maximum among all the data points.

Our goal is to identify any correlation between two time series data which are the number of design discussions per week and the number of code smells per week. We calculate the cross-correlation between design discussions per week and count of code smells per week individually for each project for this purpose [177]. As the first step of time series analysis, we start by checking if there are any visible trends. If a time series exhibits a visible trend, we need to remove the trend before further analysis. This is called detrending. We applied first differencing method to detrend both time series [177] and after that we calculate the cross-correlation.

4.3.6 Survey

To validate our findings from mining archival data, we conducted an online survey of software developers working in Apache projects. In this section we describe the survey design, participant selection criteria, pilot testing, data collection, and data analysis.

4.3.6.1 Survey design

We designed an online survey to identify the issues associated with accessing and implementing design related discussions using Qualtrics survey system [14]. We gathered demographic information to understand the respondents' backgrounds (e.g., number of years as professional developer, number of years contributing in open source project). Next, we asked them how they conduct design related discussion in the OSS project and what are the problems they faced while finding design related discussions. We asked the respondents to rate their difficulty level of finding design discussions in each communication channels using a likert scale where the options are *Extremely easy, Somewhat easy, Neither easy nor difficult, Somewhat difficult, Extremely difficult.* We used multiple choice for some of the questions. A text box option was provided for respondents in the multiple choice questions in case they want to share the reason behind their choice. The survey instrument is available in the companion website[254].

4.3.6.2 Participant Selection

For our survey we recruited software developers from the 37 Apache projects we analyzed. To build our participants list, we created the list unique e-mail addresses of individuals

from the version control system. In total, we had 7,682 unique email addresses in our list. We selected a random sample of 2,000 potential participants from the list and sent a targeted e-mail inviting them to our survey.

4.3.6.3 Pilot Survey

To help ensure the validity of the survey, we asked Computer Science professors and graduate students (Two professors and 5 Ph.D. students) with experience in OSS and in survey design to review the survey. To make sure that the questions were clear and complete, we conducted several iterations of the survey and rephrased some questions according to their feedback. We also focused on the time limit to ensure that the participants could finish it under 10 minutes. The survey was anonymous but at the end of the survey, we gave the participants a choice to receive a summary of the study through email.

4.3.6.4 Data Collection

After sending invitation email to 2,000 potential participants, 1,980 invites were delivered via Qualtrics [14]. 20 failed to deliver likely due to invalid e-mail addresses. 946 of those emails bounced and we received 22 automatic reply notifying the respondent's absence. Our final count of potential participants were 1,012. According to the Software Engineering Institute's guidelines for designing an effective survey [136], *"When the population is a manageable size and can be enumerated, simple random sampling is the most straightforward approach"*. This is the case for our study with a population of 7,682 software developers from the selected Apache projects.

We received 110 responses from 1,012 email requests during first 10 days. Then we sent a reminder email. After the reminder, we received 30 more responses in the next 10 days. We sent out a second reminder email 10 days after the first reminder and got 12 additional responses. In total, we received 152 responses from 1,012 email requests (15.01% response rate). Previous studies in Software Engineering field have have reported response rates between 5.7% [202] and 7.9% [171]. We discarded 22 partial responses, which left us with 130 responses (12.9% response rate after excluding the partial responses). Software development experience of our respondents varies from 3 years to more than 20 years, with an average of 15.21 years.

4.3.6.5 Data Analysis

We collected the ratings our respondents provided for each question, converted these ratings to Likert scores from 1 (Extremely difficult) to 5 (Extremely easy) and computed the average Likert score. We also extracted comments and texts from the "other" fields by the survey respondents explaining the reasons behind their choices. To further analyze the results, we applied Scott-Knott Effect Size Difference (ESD) test [244] to group the difficulty level of finding design discussion in each channel into statistically distinct ranks according to their Likert scores. Tantithamthavorn et al. [244] proposed ESD as it does not require the data to be normally distributed. ESD leverages hierarchical clustering to partition the set of treatment means (in our case: means of Likert scores) into statistically distinct groups with non-negligible effect sizes.

4.4 Results

We structure our study findings based on our research questions.

4.4.1 Design Discussions in OSS projects (RQ1)

As a first step, we investigate how often and where OSS contributors discuss design related matters:

RQ 1.1: What is the prevalence of design related discussion and which channels are used for such discussions?

To answer this research question, we collected all discussions from developer mailing list, issue tracking system (JIRA and Bugzilla) and pull requests. We built a machine learning classifier (discussed in section 4.3.2) that labeled $155,175$ (9.34%) discussions as design related out of $1,661,922$ total discussions.

Some example of design discussions labeled by the classifier are:

- *"I do not think this is by design. I think it is so today because... UI should be able to send this every time a call is made - please open a JIRA for this".*
- *"I'm really confused. I have tried looking at the code, but I got lost in the tangle of Contexts, Containers, Wrapper, Valves, Dispatchers...and I gave up. BTW, Craig, is there a design document anywhere for Catalina?".*
- *"did you guys ever come up with any sort of design document? Looking back at the last chatter, we were still in a localfs WAL capability".*

Next, we analyze where these design discussions take place by grouping discussion by communication channel. Table 4.4 shows the percentage of design discussions distributed

across the three types of channels.

Table 4.4: Channel wise design discussions.

Communication Channel	Design Discussions	% of total Design Discussions
Dev-mailing list	138,891	89.51%
Issue tracking system	14,986	9.66%
Pull requests	1298	0.84%

Dev-mailing lists contains the vast majority of design related discussions, accounting for 89.51% of the total design discussions from all channels, with 9.34% of the total discussions in that channel. Issue tracking (comments) were the next used communication channel with 9.66% of all design discussions across all channels being conducted there and 7.44% of Issue Tracker discussions being about design. Pull requests were the least used medium—a scant 0.84% of design related discussions being through pull request comments; and 5.69% of all pull request comments being about design.

Survey Triangulation: Our survey responses (from 130 developers) confirm these findings, which indicates that project mailing list and issue tracking system were the top two communication channels for discussing design. Project mailing list was considered the preferred channel by 57.69% of the respondents, whereas issue tracking system was the preferred choice for 54.62% respondents (see Figure 4.1).

Observation 3: Design discussions in our selected projects appears across all channels but project mailing list is the top choice among developers to discuss design.

We also ask our participants to rate the difficulty of retrieving design discussion from the three communication channels. To that end, we presented the communication

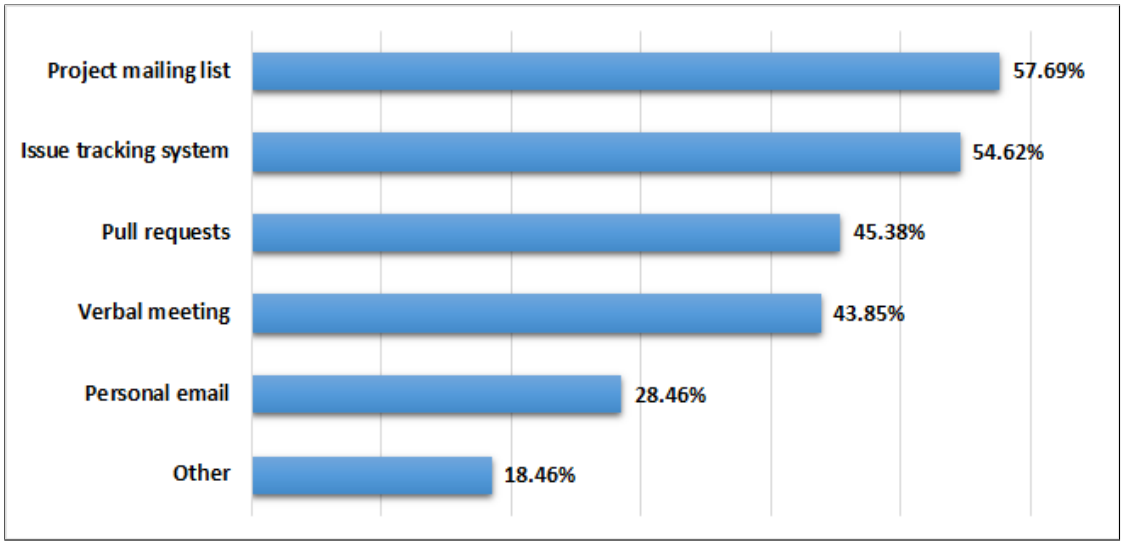

Figure 4.1: Communication channels used by survey respondents

channels and asked the developers to rate them (See Section 4.3.6.5 for details). Dev-mailing list ranked the most difficult and issue tracking system ranked the least difficult according to the Scott-Knott ESD test in terms of means of Likert scores for all the respondents.

Observation 4: Mailing list is the most difficult channel in terms of retrieving design discussions.

Respondents from the survey mentioned that they often also use *unofficial channels* like personal emails, verbal meetings etc. for design related discussion. In fact, 76.21% respondents mentioned using "verbal meeting". This is troubling since these conversations are not archived and inaccessible to others or for later review. Another 23.79% mentioned "personal email" as a medium for design discussions in the project. While this leaves some trace, it is still not recorded in the project archives or available to the community—reducing transparency of decision making in the project.

Observation 5: Design discussions occur via both documented and undocumented channels.

RQ 1.2: Who are involved in design discussions in the projects?

We answer this question by categorizing the developers into core and non-core developers according to their amounts of code contribution to the project (See section 4.3.3 for details). It is important to investigate this question because, if design discussions in a project are controlled by a small, core group of contributors, it can leave out a large group of non-core contributors. In such a situation, non-core members may be unaware of design decisions and implications of these decisions on their proposed changes, making these changes sub-optimal or incompatible with the rest of the project.

In our dataset, a majority of the design discussions were from the core group (67.48%); The non-core group was responsible for 28.54% design discussions. At first glance, it seems that non-core groups are in the minority and unable to participate, but recall that by definition non-core groups have fewer overall (code) contributions, which also translates to fewer instances of participation in discussions. Therefore, we normalize the design discussion of each group with the total participation in discussions using the equation below.

$$\text{Rescaled value} = \frac{\text{Total design discussions by the group}}{\text{Total discussions done by the group}} \tag{4.2}$$

This normalization shows that the mean of design discussions by core developers is 0.12, whereas the mean design discussions of the non-core developers is 0.06. These numbers indicate that after accounting for the amounts of contributions, core members were more involved in the design discussions of the project. While the difference between the groups is not statistically significant (Welch Two Sample t-test, $p > 0.05$), the effect

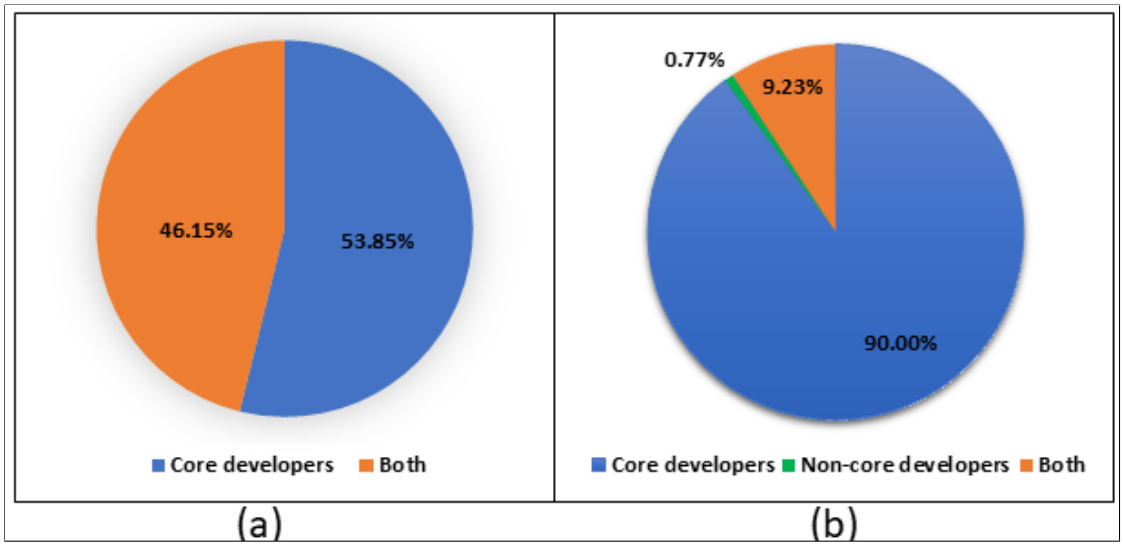

Figure 4.2: Percentage of survey responders describing participation of core and non core developers in a) design discussion, b)implementing design discussion

size is small (0.30, Cohen's d [61]).

Survey Triangulation: In our survey, we asked respondents, "Who are more involved in design related decisions?" after defining core and non-core groups. 46% of the respondents answered that both groups were involved in design related discussions in their projects (See Figure 4.2.a); confirming our results showing that both groups are active in design discussions.

Observation 6: Both core and non-core developers participate in design discussions, with some difference in the amount of contributions between the two groups.

RQ 1.3: Who implements changes resulting from design discussions?

Given that both core and non-core groups participate in design discussions, the next question is whether there are any differences in who makes design-related code changes. It is possible that, non-core developers participate in discussions, but core developers–who typically have longer tenure and have a broader view of the project—serve as gatekeepers

to design related changes.

To investigate this, we first identified the assignee of each issue in the issue tracking systems (JIRA and Bugzilla). In our data set, 14,986 issues were related to design discussions. Of these 7,310(48.77%) were assigned to core developers and 4,696(31.33%) to non-core developers. We did not find any assignee for 2,980(19.88%) issues.

In the next step, we linked the assigned issues to the commits in GitHub, so as to discount those issues that did not result in changes to the project (and remain uncommitted to the version history). Out of 12,006 issues (combined issues of core and non-core) we could link 11,228 (93.52%) issues. Of these 11,228 linked issues, 7,530 (67.06%) were authored by core developers and 3,698 (32.93%) by non-core developers. We were unable to link 778 (6.48%) issues due to the missing link problem [27, 215].

To understand whether core developers were more involved in implementing design related changes, we compared the mean number of design-related changes by core developers with that of non-core developers. The results show that the two groups are significantly different (Welch Two Sample t-test, $p < 0.05$), with a medium effect size (Cohen's d [61] of 0.77).

Survey Triangulation: Among the 130 survey respondents, 90% reported that design discussions were implemented mostly by core developers (See Figure 4.2.b) which triangulates our findings.

Observation 7: Core developers are responsible for implementing a majority of design discussions despite core and non-core developers participate equally in design discussions.

RQ 1.4: How do design discussion evolve ?

When a project matures, it is possible that it needs fewer design discussions as the design is already stable by then. However, on the other hand, as the project grows, adding new

features may require more detailed discussions so as to ensure that different features are compatible. Additionally, in the case of OSS projects, as the project matures and gains visibility, more contributors join the project bringing with them "fresh" design ideas or needing to understand the core design decisions, which can affect how design discussions occur in these projects.

We tracked the design discussion evolution of each project over a period of 560 weeks. First we analyze the overall design discussion trend across all 37 projects, we found that in most projects, design discussions follow a *decreasing* trend, and discussions start to drop after approximately 250 weeks (Figure 4.3).

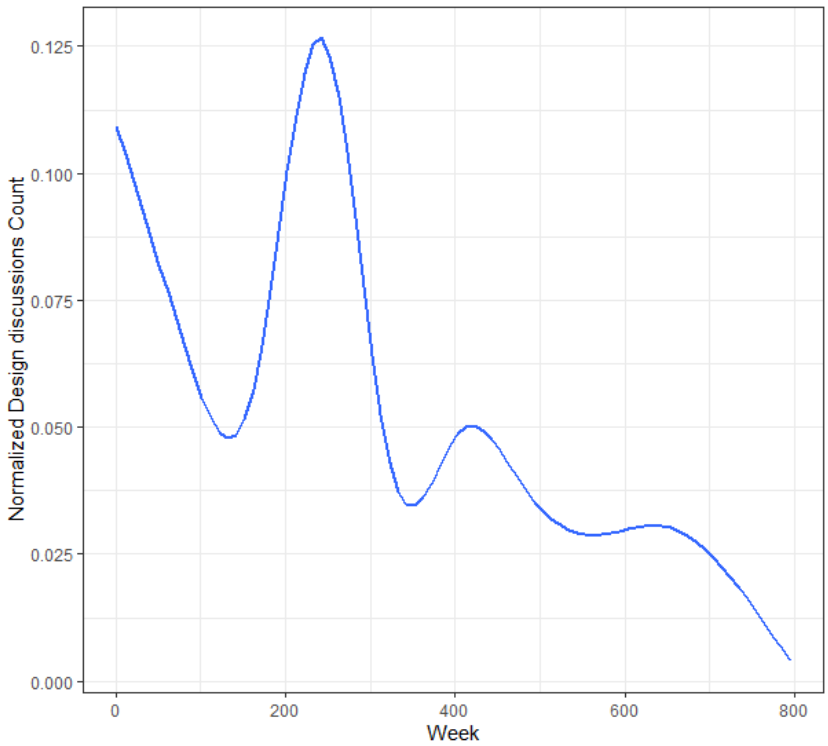

Figure 4.3: Week-wise average design discussions across all projects.

When comparing across the projects, three different trends emerged. Figure 4.4

presents examples of these trends using specific projects from our dataset. Table 4.5 shows the percentage of projects in each design discussion trend category. The first trend, *Decreasing discussions*, includes projects where design discussions decrease as the project matures, manifests in "Maven" project. The second trend, *Constant discussions* shows the amount of design discussions stay stable over time, despite several peaks; "Log4j" is an exemplar of this trend. *Increasing discussions* trend includes projects where design discussions increase as seen in the project "ant".

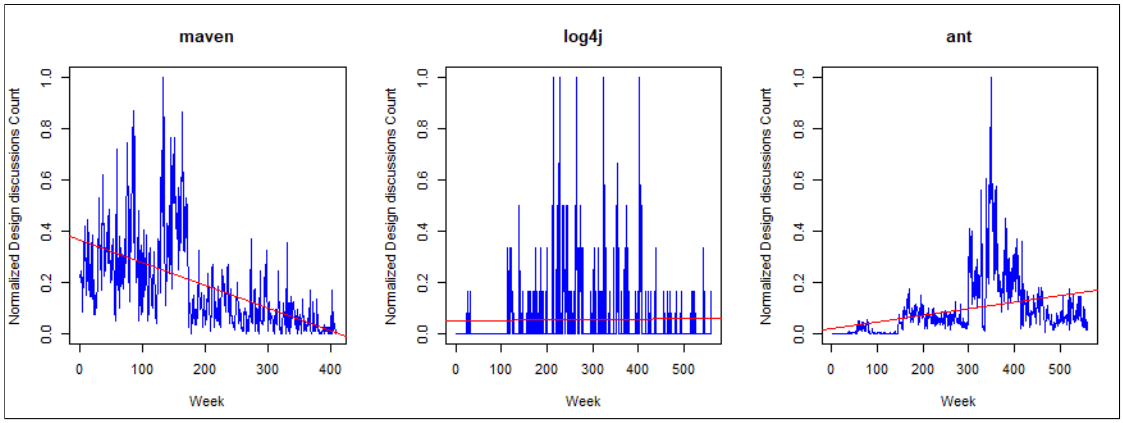

Figure 4.4: Week wise project's design discussion trend.

Table 4.5: Number of projects in each design discussion trend category

General Trend	Number of projects(%)
Decreasing	21 (56.76%)
Constant	11 (29.73)%
Increasing	5 (13.51)%

Observation 8: Distinct patterns of design discussion trends exists but on an average design discussions decrease over time.

4.4.2 Design Discussions vs. Design Quality

RQ 2: What is the association between design discussions and design quality in OSS projects?

Projects that discuss their design are likely to have fewer design quality issues. However, only discussing about it will not have any positive impact on the design quality of the project. Hence investigating the relationship between design discussions and design quality can help to inform the role of design discussions in improving (or maintaining) design quality.

We start by looking into the general trend of average code smell count across all projects. Figure 4.5 shows the *increasing* trend of the average number of code smells across all projects. This is similar to the findings of Ahmed et. al [21].

Since design discussions per week and code smell count per week both are time series data, a time series analysis is required to identify the correlation between these two, which is called cross-correlation. We do time series analysis individually for each project. Time series analysis requires a pre-processing step before doing the actual correlation analysis [177]. Due to space limitations, we report the results after each pre-processing step in the companion website[254].

Next, we calculate the cross-correlation between these two time series (design discussions per week and code smell count per week). Figure 4.6 shows the result of this step for qpid project as an example. From the figure we see that the highest correlation value is 0.16 (shown by the circled vertical line) at lag of 20. Next, we calculate the significance of correlation for each project. A correlation is significant when the absolute value is greater than, $\frac{2}{\sqrt{n-|k|}}$, where n is the number of observations and k is the lag [9].

We found that 20 out of the 37 projects (54.05%) have statistically significant cross-

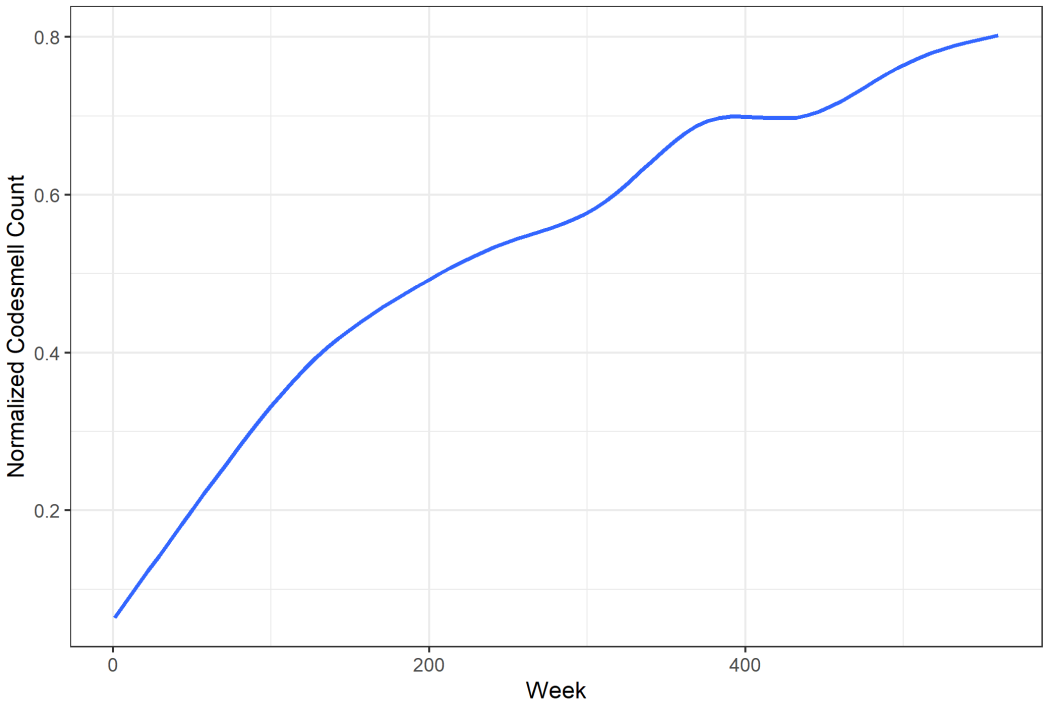

Figure 4.5: Average code smell count per week across all projects.

correlation between design discussions and code smells count. However, the cross-correlation values are small (Maximum cross-correlation is 0.34 and the minimum one is 0.003). We also looked into the lags between two time series and found that the length of the lag varies from 1 to 22 weeks between two time series. But in most projects the lag between two time series is 13 weeks. Due to space limitations, we report the plots for each of the 37 projects in the companion website [254].

Observation 9: Design discussions and design degradation are weakly correlated.

As design issues are always increasing and design related discussions are not helping project's design quality, we wanted to check if any specific group of developers are re-

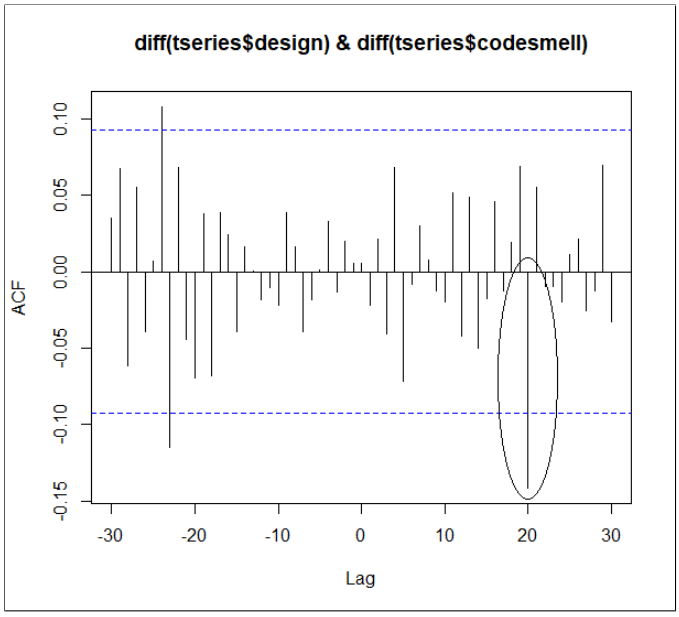

Figure 4.6: Cross correlation values of two time series

sponsible for adding design issues in the projects. First we collect the total number of commits for each developer that add or remove at least one code smell. We normalized the number of commits (that add or remove code smell) for each developers using the following equation.

$$\text{Rescaled value} = \frac{\text{Total number of smell related commits}}{\text{Total number of commits}} \tag{4.3}$$

We found that, 66.35% of the commits by core developers and 33.65% commits by non-core developers added at least one code smell in the code base. However we can see from figure 4.7 that core developers remove more code smells than they add, whereas the opposite is true for non-core developers.

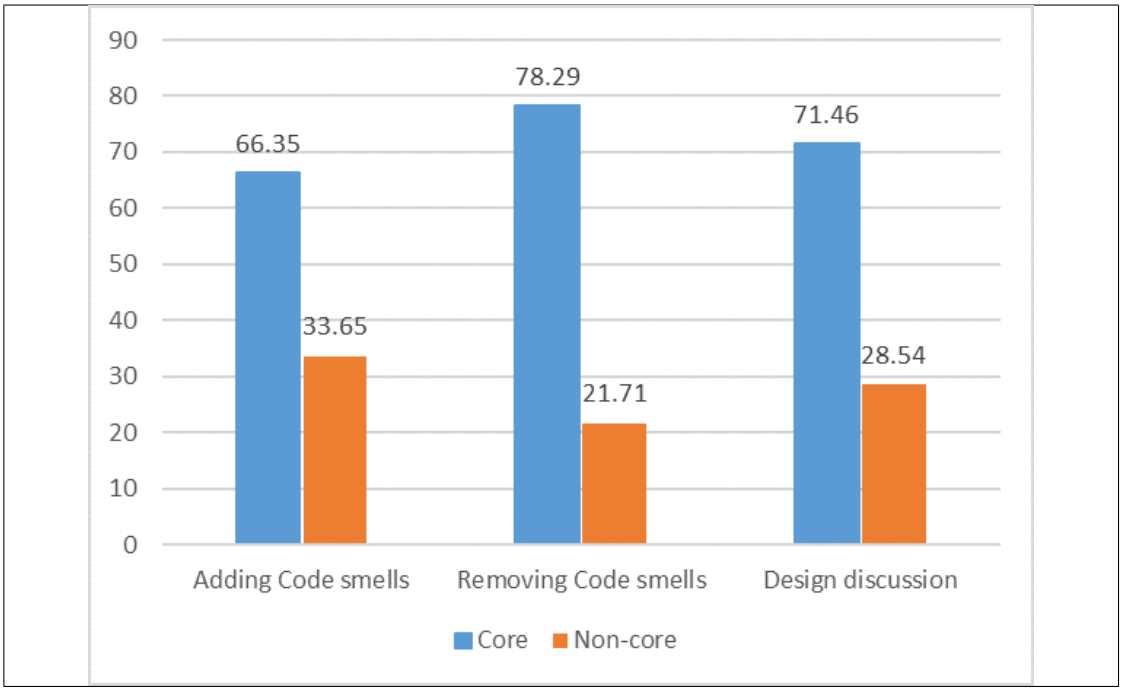

Figure 4.7: Adding, removing code smells versus design discussion participation.

Observation 10: Core developers are adding and removing more design issues in code base than non-core developers.

We also investigate if there is any difference between adding and removing code smells for each group. We found that there is no statistically significant difference in terms of number of code smell adding and removing for the core developers (Welch Two Sample t-test, $p > 0.05$). We found similar result for non-core group (Welch Two Sample t-test, $p > 0.05$).

4.5 Discussion

In this section, we discuss the results presented in the previous section and present practical implications of our study for researchers, and tool builders.

The development culture in Apache projects is not very different than that of other OSS projects. If anything, it's more systematic. For example, while merging a change, Apache developers have to provide a reference to the issue related to the submitted change as a part of the commit message. This helps to preserve an activity trail between the issue and the commit [90]. This is not a common practice in most of the OSS projects. Since such well managed projects struggle to find design related discussions in their communication channels, it is high likely that other OSS projects struggle even more.

Through mining we found that despite the presence of multiple channels the vast majority of design discussions occur over email (Table 4.4). However, our survey results show that both project mailing lists and issue tracking systems are the preferred channel for developers to discuss design related issues (Figure 4.1). One reason behind this could be that mailing lists allow developers to have lengthier discussions. Developers may therefore initiate discussions in the mailing list. Only after deciding on a design issues, they move to issue tracking system to work on those changes.

Using mailing lists for design discussions has drawbacks. Finding a specific design discussion from email archive can be extremely difficult as one of the survey respondents explained "*existing documentation is about usage and rarely about design, design decisions are lost in mail. It's very hard and time consuming to find a email thread*".

The other problem related to using mailing lists for design discussion comes from the fact that emails eventually need to translate into an issue in the *Issue Tracking System*. There may not be a one-to-one mapping between an email discussion and a corresponding

issue. This can lead to fragmented discussions, lost information etc. Making it difficult to retrieve design discussions and decisions from the archive. In our survey, 56.92% of respondents confirmed that retrieving design information from email archives is a difficult and time consuming task. Our study takes the first step towards showing that it is possible to automatically identify (and retrieve) design related discussions from multiple discussion archives (all three communication channels). However, further research to develop more efficient automated techniques to locate design discussions would be beneficial before sharing with end users.

A manual inspection of discussion contents across the three communication channels showed that a majority of design related discussions started on the mailing list and then shifted to the issue tracking system. Therefore, in order to understand the rationale of *why* a design change is made, developers need to be able to link discussion threads across multiple channels. Currently there are no tools to support such discussion tracking. For example, tools could allow context sensitive search on archives and help identify design discussions related to a specific piece of code. Similarly, another useful feature could be the ability to tag design related discussions and link them to the pull requests or to the part of code where the relevant implementation exists.

Identifying design discussions could also help in automated review assignment. Current recommender systems for task assignment or code reviews use developers' expertise [245, 264] to identify the best match. Identifying individuals who have been involved in design discussions and therefore have a bigger picture of the intention of changes can prove beneficial for such recommender systems. Further research can help in the design of approaches for automatically selecting contributors involved in a specific design discussion and assigning them higher priority.

Apart from the issue of design discussions being fragmented across multiple channels,

some of the discussions also occur through personal emails, verbal meetings etc. These unofficial discussions are not recorded in the project archives and are not accessible to developers who were not part of these discussions. We posit that this may be one of the reasons why we don't see a strong association between design quality and design discussions in our analysis. Discussion in unofficial channels can also make it difficult for developers to gauge the impact of their contribution on design. Further research is required to answer many of the questions raised by these observations: what types of discussions occur over the unofficial channels, who participate in these discussions, and what is the impact of such discussions on tracking design decisions and the overall quality of the project.

We also investigated who are mostly engaged in design discussions and design implementation. From both mining and survey we found that though both core and non-core developers are participating in design discussions and core developers are more involved implementing design discussions than non-core developers. Some reasons (as alluded to by the survey respondents) about why this might be occurring are: it is hard to find discussions across multiple channels, lack of knowledge of where to look for information, project culture where non-core developers do not work on design related changes, and design related changes when raised by non-core members are overlooked by the core group. Some of these can be fixed by better tooling, others need changes to the project culture. Researchers need to investigate these reasons and devise monitoring mechanisms to recommend necessary measures to project leaders when needed.

Finally, 79.23% of our survey respondents said that they do not continuously monitor the design quality of a project and even worse they do not check the impact of their contribution on design quality. One reason for this might be the lack of tool support. Current code smell detection tools do not support just-in-time analysis for the most

recent changes. These tools analyze the whole code base, which make it difficult to incorporate them into the regular development workflow. Moreover, the tools mentioned by the respondents–SpotBugs, JaCoCo, Simian, Checkstyle, SonarQube etc.– provide information regarding design quality (number of code smells). However, to the best of our knowledge none of these tools provide insight or links to relevant design discussions and how the actual design deviates from those. They also do not provide refactoring options to developers. This might be one of the reasons why we see such accumulations of code smells. Developers do not act upon these warnings unless they have either (i) a high return on investment (e.g., eliminating the smell has an immediate value), or (ii) tools make it easy to eliminate the smells. While developers might not see the immediate benefit of eliminating code smells (as smells usually have long-term effects on code maintenance), we encourage tool builders to close the gap between detection and correction of code smells.

4.6 Threats to Validity

We have taken care to ensure that our results are unbiased, and have tried to eliminate the effects of random noise, however some biases are unavoidable and in some cases it's possible that our mitigation strategies may not have been effective.

The data set used for the study contained $1,661,922$ discussions from 37 Apache projects. We analyzed discussions from the official communication channel archives. However, developers also use personal emails, chats, offline discussion, etc. for design related discussions which are not available on these archives and are not part of our analysis. Since we picked open source projects only from the Apache Software Foundation, our findings may not be generalizable for all open source projects. However, our

goal here was to perform a case study of a single ecosystem where projects have similar development processes. Similarly, we surveyed developers only from he Apache Software Foundation. The characteristics of these developers may not be representative of all developers in other open source projects.

We categorized the developers into core and non-core groups using a threshold of number of commits in the code base for each developer. Some of the developers could have been categorized as non-core according to our criteria though they were actually core developers who focus on large contributions rather than frequent contributions, or simply focus on architecture and high-level design (high value contributions).

For code smell detection , we used inFusion [7], which is a static source code analysis tool. Code smells that are intrinsically historical, such as Parallel Inheritance, are difficult to detect by just using static source code analysis [197]. So, the number of occurrences of such "intrinsically historical" smells will be different when historical information based smell detection technique is used. Also, InFusion detects 22 code smells, which is not a exhaustive list of code smell. There could be more smells which InFusion cannot detect.

It is always possible that the participants misunderstand the survey questions. To mitigate this threat, we conducted a pilot study with experts in OSS and survey design. We updated the survey based on the findings of these pilot studies.

4.7 Conclusions

In this paper, we present the results of our investigation of design discussions, their evolution, and their association with the project's design quality. Our mixed method empirical study of 37 Apache projects and survey of 130 developers revealed that design discussions are fragmented across multiple communication channels. This fragmentation

of design discussions likely makes it difficult for developers to keep track of the agreed upon design decisions. According to the respondents, though mailing list is the most difficult channel (56.92% of respondents) when it comes to retrieving design discussions, it is the most frequently used channel for design discussions in the Apache projects. Interestingly, survey respondents mentioned that other side channels such as personal emails (28.46% respondents) and verbal meeting (43.85% respondents) were also used to conduct design discussions. Further, the average number of design discussions decrease, but code smells increase, as the project evolves. These factors could be playing a role in the low association of design discussions with design quality.

Our work, showcases that further research is needed to: a) understand the state, evolution, and impact of design discussions that are fragmented across multiple communication channels–ours is just a start, b) analyze the disconnect between the design discussions and design quality in OSS projects, and c) build tool support to help developers find and link different design discussions.

An Empirical Examination of the Relationship Between Code Smells and Merge Conflicts

Iftekhar Ahmed, Caius Brindescu, **Umme Ayda Mannan**, Anita Sarma, Carlos Jensen

2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (pp. 1-10)

Chapter 5: An Empirical Examination of the Relationship Between Code Smells and Merge Conflicts

5.1 Introduction

Modern software systems are becoming more and more complex and requires a large development team to develop and maintain. Modern Version Control Systems (VCS) have made parallel development easier by streamlining and coordinating code management, branching, and merging. This enables large teams to work together efficiently. But it has been shown that this process is sometimes halted when isolated private development lines are synchronized and the developer runs into merge conflicts. Conflicts distract the developers as they have to interrupt their workflow to resolve them. Developers have to reason about the conflicting changes and find an acceptable merging solution. This process of conflict resolution can itself introduce bugs. Prior work has found that in complex merges, developers may not have the expertise or knowledge to make the right decisions [67, 189] which might degrade the quality of the merged code. Researchers have looked at many ways of preventing merge conflicts, and make developer's lives easier when they do occur.

Researchers have proposed workspace awareness tools [36, 64, 109, 219, 246] that help prevent merge conflicts by making the developers aware of each other's changes. Also, new merge techniques [23, 24, 134] have been proposed that would reduce the number of merge conflicts. However, little research has been devoted to the causes of

merge conflicts. Are there any endemic issues that arise from the design itself? We are interested in knowing whether the design of the codebase has an effect on the merge conflicts and what is its impact on the overall quality.

Just like merge conflicts, bad design can inflict pain on developers. Bad design makes maintenance and future changes difficult and error prone. Code smells, an indication of bad design, imply that the structure of the code is badly organized. This can lead to developers stepping on each other's toes as they make their changes. This, in turn, can lead to merge conflicts.

If there are "fundamental flaws" in the design itself, as the project grows, and the codebase grows in size and complexity, understanding and working around these "rough spots" becomes more challenging. Thus, the chances of creating a conflict increases because of the need to generate workarounds. This means that as projects grow, merge conflicts should be more likely to occur, especially around the smelly parts of the code. We aim to examine whether there is a correlation between the two, to examine whether such a link is credible.

In order to evaluate the design we look at the code smells [149]. We investigate if there is a connection between entities that contain code smells, the code smells they contain, and the merge conflicts that surround the smelly entities.

It is important to note that not all smells are created equal. Some might be more associated with a merge conflict than others. For example, a class is considered a God Class if it contains an oversized part of the entire functionality of the final product. Therefore, any changes have a high likelihood of involving changes in the God Class. When multiple developers are working, they all have a high likelihood of touching the God class. This can easily lead to merge conflicts down the road. If the changes involved are not trivial then the task of merging them will be not trivial as well.

In this paper, we investigate the following questions:

RQ1: Do program elements that are involved in merge conflicts contain more code smells?

RQ2: Which code smells are more associated with merge conflicts?

RQ3: Do code smells associated with merge conflicts affect the quality of the resulting code?

To answer these questions, we investigated 143 projects. Across them, we had 36,122 merge commits, out of which 6,979 were conflicting. We identified 7,467 code smells instances across our whole corpus. We found that merge conflicts involved more "smelly" program elements than merges that did not conflict. Our results also show that not all code smells are created equal. Some are more likely to cause problems than others. When we looked at the difficulty of merge conflicts, we found that some of the smells are more likely to be involved in semantic merge conflicts than others. Finally, we found that code smells have a negative impact on code quality.

5.2 Related Work

5.2.1 Code smells and their impact

Various measures of software quality have been proposed. Boehm et al. [39], and Gorton et al. [99], to mention a few, have explored measures including completeness, usability, testability, maintainability, reliability, efficiency etc. Some of these metrics are difficult to measure, especially in the absence of requirement documents or other supporting information. Researchers have also used code smells as a measurement of software quality [166, 167], though smells are often focused on future maintainability issues. The

concept of code smells was first introduced by Fowler [91]. Code smells are symptoms of poor design and implementation choices [91] in code base which eventually affect the maintainability of a software system [146]. Studies also showed that there is an association between code smells and bugs [156, 192] and code maintainability problems [91]. Code smells also leads to design debt. Zazworka et al. [265] found that the God Class smell is related to technical debt. Ahmed et al. [21] found how software gets worse over time in terms of design degradation. They analyzed 220 open source projects in their study and confirmed that ignoring the smells leads to "software decay".

Researchers have proposed many different approaches for detecting code smell, such as metric based [69, 70, 149, 156, 166] and meta-model based [180]. Researchers used different techniques for identifying code smells. Fontana et al. [86] used machine learning techniques for detecting code smells. Researchers also used both static analysis [69, 70, 156] and techniques that rely on the evaluation of successive versions of a software system [130, 149, 197].

5.2.2 Work related to code smells and bugs

Researchers have also considered the relationship between the presence of code smells and bug appearance in the code base. Khomh et al. [139] showed that classes affected by design problems ("code smells") are more likely to contain bugs in the future. Hall et al. [104] also found relationships between code smells and fault-proneness. According to their study some code smells indicate fault-proneness in the code base but the effect size is small (under 10%). Zazworka et al. [265] found that God Classes are fault-prone in some cases. Li et al. [166] also studied the relationship between code smells and the probability of faults in industrial systems, and found that the Shotgun Surgery smell was

correlated with a higher probability of faults. To the best of our knowledge no work has tried to research on the relationship between code smells and how it impacts collaborative work flow, specifically merging individual works.

5.2.3 Merge conflicts

Several studies have been done on identification of conflicts and developers' awareness about potential conflicts. Awareness is frequently defined as an understanding of the activities of others to give a context for one's activities [74], which is a very important issue in Global Software Engineering (GSE) [220]. Researchers have looked at different techniques of avoiding merge conflicts by increasing the developer's awareness of the changes others made to the source code. Biehl et al. [36] proposed FastDash, which sends notifications about potential conflicts when two or more developers are modifying the same file. Another awareness tool called Syde by Hatori et al. [109] consider the source code changes at Abstract Syntax Tree (AST) level operations to detect conflicts by comparing tree operations. Da Silva et al. [64] introduced Lighthouse, which is another tool for increasing awareness among developers about the conflict. Palantír by Sarma et al. [219] detects the changes made by other developers and show them in a graphical, non-intrusive manner. Servant et al. [225] also presented a tool and visualization that can be used to understand the impact of developers' changes to prevent indirect conflicts.

Guimaraes et al. [101] introduce WeCode which continuously merges uncommitted and committed changes in the IDE to detect merge conflicts as soon as possible. Brun et al. [45] used the similar approach in Crystal, to detect both direct and indirect conflicts. A software development model presented by Dewan et al. [71] aims to reduce conflicts by notifying developers who are working on the same file.

5.2.4 Work related to merge conflict resolution

Researchers have also studied different ways of managing the merge of developers' changes to efficiently resolve conflicts. This resolution could be either in an automated way or by preserving and presenting a useful context for the developer trying to resolve the conflict. A comprehensive survey of merge approaches was done by Mens [174]. Apel et al. [24, 23] presented a merging technique called semi structured merge. This considers the structure of the code which is being merged. Operation based merging by Lippe et al. [159] considers all the changes performed during development, in addition to the result, when merging.

Kasi and Sarma [134] present a technique of avoiding merge conflicts by scheduling tasks in a way that the probability of a conflict is minimized. SafeCommit by Wloka et al. [256] uses a static analysis approach to identify changes in a commit with no test failure. They proposed to use this approach when detecting indirect conflicts.

5.2.5 Conflict categorization

Researchers have come up with different ways of categorizing conflicts. Sarma et al. [219] grouped conflicts into two categories. One is direct conflicts, where the changes conflict directly. The other is indirect conflicts, where the files don't conflict directly, but integrating the changes cause build or test failures. Similarly, Brun et al. [45], categorized conflicts as first level (textual) conflicts and second level (build and test failure) conflicts. Buckley et al. [47] proposed a taxonomy of changes based on properties like time of change, change history, artifact granularity etc. Their taxonomy deals with software changes in general or conflicts at a coarser level.

5.2.6 Tracking code changes and conflicts

Researchers have proposed various algorithms for tracking individual lines of code across versions of software. Canfora et al. [49] proposed an algorithm that uses Levenstein edit distance to compute similarity of lines, matching "chunks" of changed code. Zimmerman et al. [273] proposed annotation graphs which works at the region level for tracking lines. Godfrey et al. [98] described "origin analysis", a technique for tracking entities across multiple revisions of a code base by storing inexpensively computed and easily comparable "fingerprints" of interesting software entities in each revision of a file. These fingerprints can then be used to identify areas of the code that are likely to match before applying more expensive techniques to track code entities. Finally, Kim et al. [143] propose an algorithm, SZZ, for tracking the origin of lines across changes.

5.3 Methodology

Our goal was to identify the effect of design issues on merge conflicts and the quality of the resulting code (whether these changes are associated with bug fixes or other improvements.)

Here we discuss the various steps of collecting data: (1) selecting the sample of projects for the study, (2) identifying which merge commits lead to merge conflicts, (3) tracking the lines of code through different versions and merges to investigate how the code evolved and which lines were associated with conflicts, (4) identifying code smells at the time of the conflicting merge commit. Next, we determine the nature of the code updates (e.g. was the commit a result of a bug fix or a new feature etc.) taking place on those lines. In order to do this, we manually classify a subset of the commits as bug-fix

related or other. We train a machine learning classifier to classify the rest. Finally, we build a model to predict the total number of bug fixes that would occur on a conflicting line that also contained a code smell. The following subsections describe each of these steps in detail.

5.3.1 Project Selection Criteria

We wanted to make sure that our findings would be representative of the code developed in real world, thus we selected active, open source projects hosted in GitHub. 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 [17]. The second was the availability of code smell detection tools for Java, as compared to other programming languages. Further, for ease of building and analyzing the code, we select projects using the Maven [5] build system.

We started by randomly selecting 900 projects, the first to show up when using the GitHub search mechanism. From these, we eliminated aggregate projects (which could skew our results), leaving 500 projects. After eliminating projects that did not compile (for reasons such as unavailable dependencies, or compilation errors due to syntax or bad configurations), 312 projects remained. Finally, we eliminated projects our AST walker, implemented using the GumTree algorithm [81], could not handle. This left us with a total of 200 projects.

Next, we removed projects that were too small, that is, having fewer than 10 files, or fewer than 500 lines of code. We also removed projects that had no merge conflicts. These selection criteria were used, since we are interested in the effect of design issues and merge conflicts in moderately large, collaborative projects. Our final data set contained

143 projects. Table 5.1 provides a summary of features and other descriptive information of the projects in our study.

Table 5.1: Project Statistics

Dimension	Max	Min	Average	Std. dev.
Line count	542,571	751	75,795	105,280.1
Duration (Days)	6,386	42	1,674.54	1,129.11
# Developers	105	4	74.76	83.19
Total Commits	30,519	16	3,894.48	5,070.73
Total Merges	4,916	1	252.60	522.73
Total Conflicts	227	1	25.86	39.49

We also manually categorized the domain of the projects by looking at the project description and using the categories used by Souza et al. [68]. Table 5.2 has the summary of the domains of the projects.

Table 5.2: Distribution of Projects by Domain

Domain	Percentage
Development	61.98%
System Administration	12.66%
Communications	6.42%
Business & Enterprise	8.10%
Home & Education	3.11%
Security & Utilities	2.61%
Games	3.08%
Audio & Video	2.04%

5.3.2 Code smell detection tool selection

We chose to use InFusion [7] to identify code smells because it has been found to identify the broadest set of smells [87]. Researchers have found that the metric-based approach

identified by Marinescu [166] has the highest recall and precision (precision: 0.71, recall: 1.00) for finding most code smells [222]. InFusion uses this same principle and set of thresholds for identifying code smell, which was another reason for using InFusion. Researchers [21] have evaluated the smell detection performance of InFusion where they found it to have precision of 0.84, recall of 1.00 and an F-measure of 0.91.

5.3.3 Conflict Identification

Since Git does not record information about merge conflicts, we had to recreate each merge in the corpus in order to determine if a conflict had occurred. We used Git's default algorithm, the recursive merge strategy, as this is the most likely to used by the average Git project. From our sample of 143 projects we extracted 556,911 commits. This included 36,122 merge commits. The average number of merge commits was 253. Out of all the merges, 6,979 (19.32%) were identified as leading to a conflict. The distribution of merge conflicts is shown in Figure 5.1. We see that projects experience an average of 25 merge conflicts, or 19.32% of all merges. Merge conflicts, therefore, are a common part of the developer experience.

We then collected statistics regarding each file involved in a conflict. We tracked the size of the changes being merged, the difference between the two branches (in terms of LOC, AST difference, and the number of methods and classes involved). To determine the AST difference, we used the Gumtree algorithm [81]. We also tracked the number of authors involved in the merge.

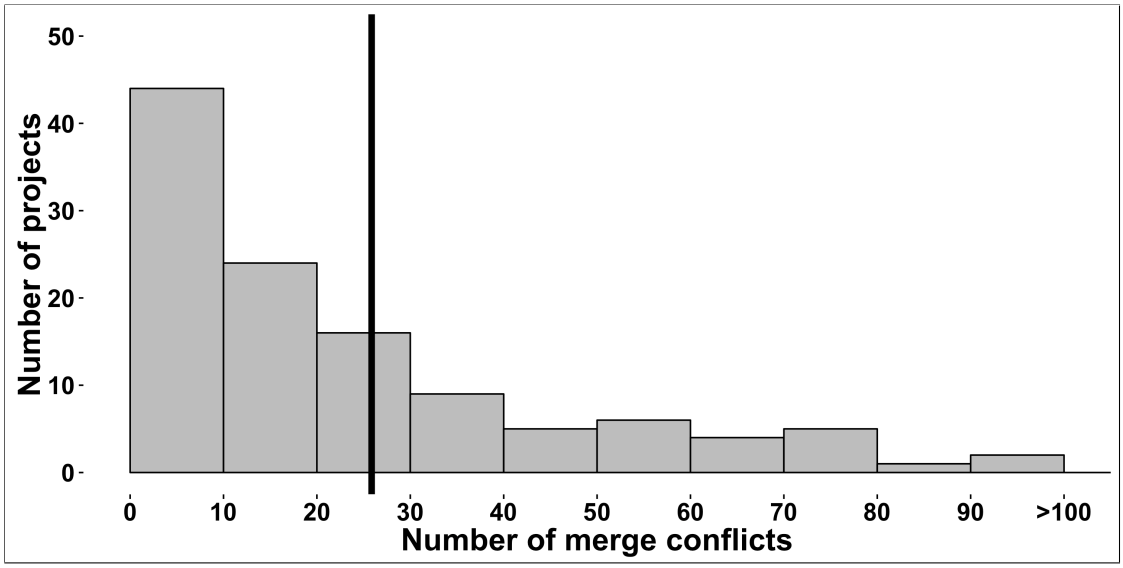

Figure 5.1: Distribution of merge conflicts. The vertical line represents the mean (25.86)

5.3.4 Conflict Type Classification

To answer our second research question, we needed to categorize the conflicts based on the type of changes (e.g., whitespace or comment added vs. variable name changed).

We identified two categories of conflicts. The first one being semantic conflicts which requires understanding the pro-gram logic of the changes in order to successfully resolve the conflict. The other type of conflict is non-semantic which easier and less risky to resolve since they do not affect the pro-grams' functionality. We manually classify 606 randomly sampled commits. We classify each conflict based on the type of changes causing the merge conflict (e.g., whitespace or comment added vs. variable name changed). Two of the authors coded 300 of these commits using qualitative thematic coding [81]. They achieved an interrater agreement of over 80| on 20% of the data: we obtained a Cohen's Kappa of 0.84. Having reached an agreement, one of the authors classified the remaining 306 commits. The codes and their definitions are given in Table 5.3.

Table 5.3: Conflict Categories

Category	Definition	Example
Semantic	Conflicts involving semantic changes.	A refactoring and a bug fix involving the same lines.
Non-Semantic	Conflicting changes in formatting/comments	One of the branches contains only formatting changes (white space) .

To train the classifier (to differentiate between semantic and non-semantic commits) we use a set of 24 features, in-cluding: the total size of the versions (LOC) involved in a conflict, the number of statements, methods and classes involved in the conflict. Details of the features are in the accompanying website [16]. We use the set of 606 (10%) commits as training data for a machine learning classifier. We used Adaptive Boost (AdaBoost) ensemble classifier that can only be used for binary classes. We categorized the 6,979 conflicting commits. We use 10 fold cross-validation to test the performance of our classier. The precision of predicting the semantic conflicts is high at 0.75.

5.3.5 Measuring Code Smells and Tracking Lines

For each of the 6,979 conflicting commits we collected the code smells that were associated with a conflict. We needed to track them to measure the effect of having the smell and being involved in a conflict on the quality of the resulting code.

We use GumTree [81] for our analysis, as it allows us to track elements at an AST level. This way we can track only the elements that we are interested in (statements), and ignore other changes that do not actually change the code. The GumTree algorithm works by determining if any AST node was changed, or had any children added, deleted or modified. The algorithm maps the correspondence between nodes in two different trees,

which allows it to accurately track the history of the program elements. This algorithm has unique advantages over other line tracking algorithms, such as SZZ [143]. These advantages include: ignoring whitespace changes, tracking a node even if its position in the file changes (e.g. because lines have been added or deleted before the node of interest), and tracking nodes across refactoring, as long as the node stays within the same file. Using this technique, we can track a node even when it has been moved, for example, be-cause of an extract method refactoring.

For each node (in the AST) involved in a conflict and having a smell, we identify all future commits that touched the file containing said node and tracked the AST node forward in time. For Java, it is possible for multiple statements to be ex-pressed in the same line (e.g., a local variable declaration in-side an if statement). In this case, we considered the innermost statement, as this gives the most precise results.

5.3.6 Commit Classification

In order to answer our third research question, we needed to categorize the type of change for a code commit. For our purpose, code commits can be broadly grouped into one of two categories: (1) bug-fixes and improvements (modifying existing code), and (2) Other — commits that introduced new features or functionality (adding new code) or commits that were related to documentation, test code, or other concerns. Two key problems with this classification are: (1) it is not always trivial to determine which category a commit falls under, and (2) larger projects see a huge amount of activity. Manual classification of all commits was not an option, and we decided to use machine learning techniques for this purpose, rather than limiting the statistical power of our study (especially as arbitrarily dropping the most active subjects would clearly potentially introduce a large

bias into our results.)

In order to build a classifier, we randomly selected and manually labeled a set of 1,500 commits. The first two authors worked independently to classify the commits. Their data sets had a 33% overlap, which we used to calculate the inter-rater reliability. This gave us a Cohen's Kappa of 0.90. In our training dataset, the portion of bug-fixes was 46.30%, with 53.70% of the commits assigned to the Other category. Some keywords indicating bug-fixes or improvements were Fix, Bug, Resolves, Cleanup, Optimize, and Simplify, and their derivatives. Anything that did not fit into this pattern was marked as Other.

Not all bug-fixing commits include these keywords or direct reference to the issue-id; commit messages are written by the initial contributor, and there are few guidelines. A similar observation was made by Bird et al. [37], who performed an empirical study showing that bias could be introduced due to missing linkages between commits and bugs. This means that we are conservative in identifying commits as bug-fixes.

We trained a Naive-Bayes (NB) classifier and a Support Vector Machine (SVM) by using the SciKit toolset [203]. We used 10% of the data to train the classifier. We applied the classifiers to the training data using a 10-fold cross-validation. As before, we used the F1-score to measure and compare the performance of the models. The NB classifier outperformed the SVM. Therefore, we used the NB classifier to classify our full corpus.

Table 5.4 has the quality indicator characteristics of the NB classifier. Tian et al. [246], suggest that for keyword-based classification the F1 score is usually around 0.55, which also occurs in our case. While our classifier is far from perfect, it is comparable to "good" classifiers in the literature, and we believe it is unlikely for the biases to have a confounding effect on our analysis. Since our analysis only relies on relative counts of bug-fixes for statements, so long as we do not systematically under count bug-fixes for only some statements, our results should be valid.

Table 5.4: Naive Bayes Classifier Details

	Precision	Recall	F1-measure
Bug-fix	0.63	0.43	0.51
Other	0.74	0.86	0.80

For each line of code resulting from a merge conflict, we count the number of (future) commits in which it appears, as long as those commits are identified as bug-fixes. We stop the tracking when we encounter a commit that is classified as Other. Our reasoning is that once an element has seen a change that is not a bug-fix, it is no longer fair to assume that subsequent bug fixes are associated with the original merge conflict.

5.3.7 Regression analysis

In order to answer our third research question that is related to the effect of code smells on quality of the resulting code, we needed to build a regression model to identify the impact of code smell on the number of bug-fixes that occur on lines of code that are associated with a code smell and a merge conflict. We use Generalized Linear Regression [61]. The dependent variable (count of bug fixes occurring on smelly and conflicting lines) follows a Poisson distribution. Therefore, we use a Poisson regression model with a log linking function.

In order to build our model, we collect information about the smells and the conflicts. We use Understand [4] to count the number of references to, and from other files to the files that are involved in a conflict. We collect this information as a proxy for the importance of the file. We assume that the more a file is referenced by other files, the more central that file is, and hence more important. Any change in these central files

can increase the chance of a change being required in other files, and therefore lead to multiple developers making changes to these files, which can in turn lead to conflicting changes.

We also collect the following factors for each commit such as the difference between the two merged branches in terms of LOC, AST difference, and the number of methods and classes being affected. Our intuition is that larger "chunks" of changes should have a higher chance of causing a conflict. We also calculate the number of authors who made commits to the branches that were merged, since there is a higher likelihood of conflicts if multiple developers are involved.

We also determine the experience level of each developer by splitting them into two categories: core and non-core. To calculate the category for each developer, we split the development history into quarters. For each commit a developer is classified as core if he is in the top 20% of the developers in that quarter (calculated by the number of commits). Otherwise he is non-core. We use this process because, in open source projects, authors come and go. Also, an author can be classified as core and non-core in different quarters, depending on his contribution to the project.

After collecting these metrics, we checked for multicollinearity using the Variance Inflation Factor (VIF) of each predictor in our model [61]. VIF describes the level of multicollinearity (correlation between predictors). A VIF score between 1 and 5 indicates moderate correlation with other factors, so we selected the predictors with VIF score threshold of 5. This step was necessary since the presence of highly correlated factors forces the estimated regression coefficient of one variable to depend on other predictor variables that are included in the model.

5.4 Result

5.4.1 RQ1: Do program elements that are involved in merge conflicts contain more code smells?

As a first step, we collect the total number of code smells for each of the 6,979 conflicting commits in our dataset. Table 5.5 contains the percentage of each smell and the percentage of projects that have a particular smell. We find that external and internal duplication have a much higher instance than others when considering the percentage of smells in the dataset. However, about 50% of projects have Data Class and SAP Breakers smells.

Table 5.5: Percentage of Code smells

Smell	% of smells in the full dataset	% of projects w/ smell
External Duplication	42.79	22.53
Internal Duplication	**34.05**	**23.80**
Feature Envy	4.04	28.42
Data Clumps	3.71	20.36
Intensive Coupling	3.50	14.30
Data Class	3.18	48.05
Blob Operation	2.58	30.05
Sibling Duplication	2.35	10.86
SAP Breakers	1.52	52.76
God Class	**0.89**	**19.10**
Schizophrenic Class	0.58	20.00
Message Chains	0.33	5.34
Tradition Breaker	0.17	6.33
Refused Parent Bequest	**0.19**	**5.25**
Shotgun Surgery	0.01	1.72
Distorted Hierarchy	**0.003**	**0.36**

We next compare the mean number of code smells associated with each merge commit, for cases when they conflict and for cases when they do not conflict. Note that a commit can involve multiple files, and a file can contain multiple smells. We calculate the total number of smells for each file. For example, a conflicting merge commit in the `commandhelper` project (with the SHA1 of `a91faa`) contains one conflicting file, and that conflicting files contains a total of 8 smells.

The mean number of smells in conflicting program elements is **6.54**, whereas the mean for non-conflicting program elements is **1.92**. The results are statistically significant (Mann-Whitney test, $U = 6.24e^6$, $p < 4.77e^{-10}$.); we use the non-parametric Mann-Whitney test since our population is not normally distributed. Therefore, we find that *program elements that are involved in merge conflicts are, on average, more smelly than entities that are not involved in a merge conflict.*

5.4.2 RQ2: Which code smells are more associated with merge conflicts?

Next, we compare the occurrence of each individual smell across conflicting and non-conflicting commits. Since we are performing multiple tests, we have to adjust the significance value accordingly to account for multiple hypothesis correction. We use the Bonferroni correction, which gives us an adjusted p-value of 0.0031.

For 12 out of 16 total smells, we find significant differences (Mann-Whitney test, $\alpha < 0.0031$) between the means of conflicting and non-conflicting commits. The conflicting commits have a higher incidence of smells. Table 5.6 presents the results for code smells where the difference was significant along with the p-values of individual comparisons.

Table 5.6: Mean Number of Smells in Conflicts VS. Non-Conflict Commits Calculated per Commit

Smell	Smells in conflicts	Smells in non conflict	p-value
God Class	1.23	0.25	0.0001
Data Clump	0.65	0.27	0.0001
Sibling Duplication	0.58	0.10	0.000001
Data Class	0.47	0.12	0.000001
Distorted Hierarchy	0.45	0.05	0.000001
Unnecessary Cou-pling	0.33	0.10	0.0001
Internal Duplication	0.24	0.08	0.000001
SAP Breaker	0.12	0.07	0.000001
Tradition Breaker	0.10	0.05	0.00007
Blob Operation	0.07	0.06	0.0001
Message Chain	0.04	0.03	0.00062
Shotgun Surgery	0.01	0.00769	0.00021

The following are the top 5 smells in terms of their (mean) numbers per conflict: *God Class, Data Clump, Sibling Duplication, Data Class and Distorted Hierarchy.* It is worth noting that the distribution of smells per conflict (Table 5.6) is different from Table 5.5. This is because in Table 5.6 we are looking only at the smells that affect the entities involved in merge conflicts, whereas Table 5.5 shows all the smells in the project. This discrepancy is an effect of the fact that merge conflicts exhibit a different smell pattern compared to the overall project.

Next, we perform two steps. First, we investigate the correlation between each smell and the merge conflicts to identify which of the above smells are more strongly associated with conflicts. Then, we categorize merge conflicts into semantic and non-semantic conflicts to further explore the associations of smells to these types of conflicts.

Code smells and conflicts. We perform a correlation analysis between the count of smells and merge conflicts to distill which of the smells from Table VI are more closely

associated with conflicts, and should be attended to. We use the Kendall correlation test because it is a non-parametric test and it is more accurate with a smaller sample size. As we perform the tests for each smell, we are splitting out data into smaller chunks. Therefore, the Kendall correlation test is more appropriate.

We find that, except for *External Duplication, Schizophrenic Class, SAP Breaker and Data Class* all smells are correlated with merge conflicts (Kendall correlation test, $\alpha < 0.0031$). We report the statistically significant results in Table 5.7.

Table 5.7: Correlation Between Conflict and Smell Count.

Smell	Correlation	p-value
God class	0.18	<0.0001
Internal Duplication	0.17	<0.0001
Distorted Hierarchy	0.13	<0.0001
Refused Parent Bequest	0.10	<0.0001
Message Chain	0.10	<0.0001
Data clump	0.09	<0.0001
Feature Envy	0.09	<0.0001
Tradition Breaker	0.09	<0.0001
Blob Operation	0.08	<0.0001
Shotgun Surgery	0.07	<0.0001
Unnecessary Coupling	0.05	0.00007
Sibling Duplication	0.04	0.00021

The three strongest correlation to conflicts are with the following smells: God Class, Internal Duplication and Distorted Hierarchy. These smells all relate to cases where object-oriented design principles of encapsulation and structuring is not well used, leading to problems with developers making conflicting parallel changes. We discuss these reasons further in Section 5.5.

Types of conflicts and their classification. Not all conflicts are the same, some involve changes to the actual code structure and require the developer to understand

the logic behind the changes before they can be integrated (*semantic conflicts*), whereas others can be formatting or cosmetic changes (*non-semantic*). *Semantic* conflicts are inherently harder to resolve. Therefore, we investigate whether specific types of code smells are more likely to occur with *semantic conflicts*. We use the conflict classification methodology in Section 5.3.4.

Recall, we manually labeled 606 conflicts to classify them into *semantic or non-semantic*, which we then use for the automated classification of 6,979 commits. We present the distribution between the manual and automatic classification in Table 5.8. The distributions of *semantic and non-semantic conflicts* in the automatically classified data match the distribution of our manual labeling (training data), which shows the efficacy of the automated classifier.

Table 5.8: Conflict Types Based on Their Frequency of Occurrence

Category	# of Conflicts	% of total (classifier)	% of total (training)
Semantic	5,250	75.23%	76.12%
Non-Semantic	1,729	24.77%	23.88%

Semantic conflicts are more common (76.12% in the manually labeled data and 75.23% in the automated classified data), as compared to the *non-semantic* conflicts (23.88% in manually labeled and 24.77% in automated classified data).

Semantic conflicts and code smells. To understand if there is any correlation between *semantic* conflicts and the types of code smells we perform the Kendall correlation test for each smell in the presence of *semantic* merge conflicts (in our total dataset). We use the Kendall correlation test and found significant correlation ($\alpha < 0.0031$) only for *Internal Duplication and Blob Operation*. Table 5.9 IX contains all correlations, where the cells marked with ** are significant.

Table 5.9: Smell Categories for Semantic Conflicts (Significance Level $\alpha < 0.0031$).

Smell	Correlation	p-value
Blob Operation **	0.05	0.0030
Internal Duplication **	0.07	0.0002
Message Chain	0.01	0.4970
Refused Parent Bequest	0.03	0.0492
SAP Breaker	-0.02	0.2652
Schizophrenic Class	-0.03	0.0832
Shotgun Surgery	-0.008	0.6597
Sibling Duplication	0.031	0.1029
Tradition Breaker	0.018	0.3291
Unnecessary Coupling	-0.01	0.5681
Data Class	-0.02	0.1524
Data Clumps	0.030	0.1103
Distorted Hierarchy	0.034	0.0670
Feature Envy	0.009	0.6206
God Class	0.050	0.0072

Since the correlation for both *Blob Operation and Internal Duplication* are small, we perform an odds-ratio test to understand which of these smells are more likely to be involved in a Semantic merge conflict, as compared to entities that do not have these smells, but were involved in a conflict. Since we are performing two comparisons, we have to adjust the significance value to adjust for multiple hypothesis testing. Like in the previous sections, we performed a Bonferroni correction, which gives us significance value of $\alpha = 0.0025$ to test at.

We performed an odds ratio test (Fisher's exact test) for the *Blob Operations* and find that they are 1.7 times more likely to be involved in a *Semantic merge conflict* (odds ratio: 1.77, $p = 0.0024$). *Blob Operations* are methods that are very complex and have many responsibilities. Therefore, any change to the method will likely impact multiple lines, which may intersect with logical changes made by another developer to the same

method. This explains the high likelihood of their involvement in *Non-Semantic conflicts*.

For *Internal Duplication*, we found that they are 1.55 times more likely to be involved in merge conflict (odds ratio: 1.55, $p = 0.0001$.) We attribute this to the fact that, because of duplication, a change has to be repeated in multiple locations. This increases the chances of developers making overlapping changes.

5.4.3 RQ3: Do code smells associated with merge conflicts affect the quality of the resulting code?

We aim to model the effects of code smells on the bugginess of a line of code involved in a merge conflict. As defect prediction literature has already identified several factors (e.g., the size of the module under investigation [77], number of committers [255], centrality of files [53]) that affect bugginess, we include them in our model also. Table 5.10 lists the final set of factors that we use, which include metrics that are code-based (F1, F2), change-related (F5-F8), author-related (F3, F4), and code smells. We compute whether a developer is core or non-core based on our methodology in Section 5.3.7.

To answer our third research question, we build two Generalized Linear Models (GLM). The first contains the number of code smells as a factor, and the second does not. The first (Poisson regression) model is built with a log linking function as explained in Section 5.3.7. After filtering the factors with VIF ≤ 5, we had a set of 8 factors out of 43 factors. All eight factors were statistically significant (see Table 5.10). The predicted value is the total number of bug fixes occurring on a line of code that was involved in a merge conflict. Note that smell count was a significant factor in the model ($p < 0.05$), with an estimate of 0.427.

Table 5.10: Poisson Regression Model Predicting Bug-fix Occurrence on Lines of Code Involved in a Merge conflict

Factor#	Factor	Estimate	p-value
F1	In Deps	3.195	<0.0001
F2	Out Deps	-0.053	<0.0001
F3	Noncore author	-3.799	<0.0001
F4	No. Authors	0.129	<0.0001
F5	No. Classes	-0.373	<0.0001
F6	No. Methods	0.244	<0.0001
F7	AST diff	0.001	<0.0001
F8	LOC diff	0.00002571	<0.0001
F9	Number of Smells	0.427	<0.0001

The McFadden Adjusted R^2 [115] of this model is 0.47. We calculated McFadden's Adjusted R^2 as a quality indicator of the model because there is no direct equivalent of R^2 metric for Poisson regression. The ordinary least square (OLS) regression approach to goodness-of-fit does not apply for Poisson regression. Moreover, adjusted R^2 values like McFadden's cannot be interpreted as one would interpret OLS R^2 values. McFadden's Adjusted R^2 values tend to be considerably lower than those of the R^2. Values of 0.2 to 0.4 represent an excellent fit [115].

To understand the impact that code smells have, we built the same model by removing the total number of smells as a factor. This decreased the adjusted R^2 from 0.47 to 0.44. We can therefore conclude that code smells have a significant impact on the final quality of the code. Since McFadden's adjusted R^2 penalizes a model for including too many predictors, had the code smells not mattered, removing it could have increased the adjusted R^2 instead of reducing it.

5.5 Discussion

To the best of our knowledge, we are the first to investigate the association of code smells with that of merge conflicts, and their impact on the bugginess of the merged results (line of code). We find that program elements that are involved in merge conflicts contain, on average, 3 times more code smells than program elements that are not involved in a merge conflict.

Not all code smells are equally correlated to merge conflicts. 12 out of the 16 code smells that co-occur with conflicts are significant associated with merge conflicts. The top five code smells from this list are: *God class, Message Chain, Internal Duplication, Distorted Hierarchy and Refused Parent Bequest. Interestingly, the only (significant) code smells associated with Semantic conflicts are Blob Operation and Internal Duplication.*

All the above code smells arise when developers do not fully exploit the advantages of object-oriented design, leading to high coupling, duplication, or large containers. These factors lay the groundwork for parallel conflicting efforts, where developers step on each other's toes. For example, the *Blob Operation* is a large and complex method that grows over time becoming hard to maintain. In such a situation, multiple developers may need to make changes to the same method and, therefore, collide when merging. Similarly, *Internal Duplication* arises when code is duplicated, which bloats methods and makes it hard to ensure all clones evolve in the same way. In such a situation, developers might have to "touch" multiple parts of the method to ensure all clones are being updated, causing situations of parallel, conflicting edits.

It is interesting to observe that Semantic merge conflicts are associated with smells at the method level. For example, *the Blob Operation and Internal Duplication smells* are 1.77 times and 1.55 times, respectively, more likely to be present in a *semantic*

conflict as compared to a non-semantic conflict. This indicates that bloated methods or duplicated code in methods increase the spread of the change a developer is likely to make, which in turn increases the likelihood of two or more changes conflicting during a merge. Prior work has associated code duplication with negative consequences such as increased maintenance cost [217, 145] and faults [29, 128] . Our findings indicate that duplication also negatively impacts the collaborative workflow by making it difficult to merge changes.

It is worth noting that while smells, such as *God Class* have a significant correlation with overall merge conflicts, they do not have a significant correlation with *semantic merge conflicts*. We posit that a large container (class) with cohesive logical units (methods) can lead to multiple developers making parallel changes that are localized to specific areas (methods) and do not intersect. In these cases, when changes are merged conflicts can arise because of the movement of code or formatting changes (non-semantic conflicts). The same reasoning is also applicable for *Distorted Hierarchy, Refused Parent Bequest and Message Chain*. In contrast, as discussed earlier method-level smells seem are correlated with *semantic conflicts*.

To the best of our knowledge, ours is the first empirical study to investigate the effects of merge conflicts and code smells on the bugginess of code. We found that the presence of code smells on the lines of code involved in a merge conflict has a significant impact on its bugginess (see Table 5.10). Including code smells as a factor increases the McFadden's adjusted R^2 value from 0.44 to 0.47. Since McFadden's adjusted R^2 penalizes a model for including too many predictors, an increase in the value signifies that adding code smells as a factor was valuable.

We find that factors such as incoming-dependencies and the number of code smells have the highest correlation estimate, indicating their importance to the model. We find

that some factors, such as non-core author, number of classes, and outward dependencies have a negative effect on bugginess. This is counter intuitive. We had assumed that changes from multiple non-core authors are more likely to be buggy. We believe that the following reasons lead to this surprising outcome. It might be the case that non-core contributors are more thorough and put more effort towards submitting code that is less bug prone. Or it might be the process via which newcomers' contributions are accepted. For example, core developers might pay more attention to changes coming from non-core contributors. Further empirical studies on the differences in review processes for core vs. non-core developers will be interesting. We also found that the number of classes involved in a conflict has a negative correlation to its bugginess. This might be because changes that involve multiple classes are more likely to be refactoring or licensing changes, and therefore, less likely to introduce bugs.

Implications: Our findings have a number of implications for software practitioners, tool builders and researchers.

Code smells have been historically associated with maintenance issues, which are known to be a problem in the long term. However, developers are often unaware of code smells. Yamashita et al. found that a considerable portion (32%) of developers did not know about code smells [257]. Our findings shed a different light on the impact of code smells and on the importance of addressing them. Our results show that code smells are an immediate concern for day-to-day activity such as merging changes.

Merge conflicts delay the project by requiring an examination of the conflict, and disrupting the developers' workflow. Anecdotal evidence shows that developers hate resolving conflicts. Developers are known to follow informal processes (e.g., check in partial code, email the team about impending changes etc.) or rush to commit their work in an effort to avoid having to resolve conflicts [67]. A developer may also choose to

delay the incorporation of others' work, fearing that a conflict may be hard to resolve [67]. Such processes can have a detrimental effect on team productivity and morale. This situation can only become worse as the project evolves on two fronts. First, the number of code smells is likely to increase as the project ages [21]. Second, there is a likelihood of increase in merge conflicts as more developers start to contribute. Our results indicate that practitioners should pay more attention to code smells, as it will not only make the code quality better, but will also help them minimize the number of merge conflicts they need to resolve.

Practitioners, when investigating the root cause of a merge conflict can start by looking for smelly program elements in the code. Moreover, since changes that involve entities containing code smells are more likely to lead to semantic merge conflicts, integrators (or code reviewers) should pay particular attention to and attempt to remove code smells when reviewing commits. Practitioners should also pay attention to "good" software engineering processes when they deal with smelly program elements. For example, when changes are being made to smelly parts of the code base developers should merge more frequently and perform more thorough code reviews.

Our results show that code smells are a good predictor of merge conflict and the level of difficulty of that conflict. Therefore, tool builders can use the information of incidence of code smells to support distributed work – either in predicting likelihood of conflicts or their difficulty. Code smells can also be used as a factor to schedule tasks (e.g., program elements that have code smells should not be edited in parallel) or assign tasks (e.g., developers with higher experience should work on smelly program elements).

Our results have implications for researchers. Since code smells together with merge conflicts can predict bugginess, researchers can use this information in bug prediction models to increase their effectiveness. To the best of our knowledge, no merge conflict

prediction tool exists. Our results show that code smells have a strong association with merge conflicts, therefore, researchers can use this information to predict impending merge conflicts. Our results also have implications in testing. For example, increasing the test coverage of smelly lines that were involved in a merge conflict can be used as an objective/fitness function in the field of search based software engineering.

5.6 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.

Bias due to sampling: 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 and not generalizable to commercial programs. However, the threat is minimal since we analyze a large number of projects spanning eight different domains.

Bias due to tools used: The smell detection 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 [197]. So the number occurrence of such "intrinsically historical" smells should be different when historical information based smell detection technique is used.

Secondly, we used the Gumtree algorithm [81] for tracking program elements across commits. However, the algorithm used is unable to track program elements across renames or movement to another folder. Further, refactoring that involves modification of scope, such as moving the code out of the current compilation unit also causes the

algorithm to lose track of the program element after refactoring.

Bias Due to using classifiers: We use machine learning to group conflicts into the two categories, and to determine whether a commit was a bug-fix. As with any classifier, we have some mislabeling. While our results do not require those results to be anywhere near perfect, this threat is low as our classifiers have good F1-measure and high precision.

Regarding the bug-fix classifier, our recall and precision measures are on par with past work [37]. Since our analysis relies on relative count of bug fixes, as long as we do not systematically undercount bug fixes, our results are valid.

Finally, we have assumed that all bugs were found and fixed by developers when we use it as a metric of bugginess of merged lines of code. This may not always be true, and hence our results are conservative.

5.7 Conclusion

In this paper, we study the history of 143 open source projects, from which we extract 6,979 merge conflicts to see if there is any correlation between code smells and merge conflicts. We found that entities involved in merge conflicts contain almost 3 times more code smells than non-conflicting entities.

To have a better understanding of the effect of code smells on merge conflicts, we categorized conflicts into semantic conflicts – changes to the AST and hard to resolve – and non-semantic – changes that are cosmetic. We found two method-level code smells (Blob Operation and Internal Duplication) to be significantly correlated with semantic conflicts. More specifically, methods that contained the Blob Operation and Internal Duplication smells were more likely to be involved in a semantic merge conflict, by 1.77 times and 1.55 times respectively. We also found that code smells have a significant

impact on the final quality of the code. Count of code smells was a significant factor when we modeled the bugginess of lines of code involved in a merge conflict.

Our results show that code smells, thought to be a maintenance issue and often neglected by practitioners, have an immediate impact in how distributed development is managed. Their presence is not only associated with difficult merge conflicts (semantic), but also with the likely-hood of bugs getting introduced in the code base.

Floss Now or Go to the Dentist: An Empirical Assessment of Bug-proneness Prediction

Umme Ayda Mannan, Iftekhar Ahmed, Caius Brindescu, Carlos Jensen, Anita Sarma

Under review in Information and Software Technology (IST).

Chapter 6: Floss Now or Go to the Dentist: An Empirical Assessment of Bug-proneness Prediction

6.1 Introduction

Assuring the reliability of software is very important to the Software Engineering community. However, it is also inherently a resource constrained activity. Real-world software systems have more bugs than developers can identify and fix [150]. Moreover, bug fixing is effort intensive; Kim et al. [141] reported that the time to fix a bug ranges from 100 to 200 days. Therefore, any technique that allows developers to reliably identify buggy parts of the code to guide their bug-fixing efforts is helpful. One such technique is bug prediction. In the last decade, researchers have investigated a wide range of bug prediction models based on different types of metrics, such as metrics about the code [30, 175, 274, 275, 268, 268], historical data [107, 95, 100, 144, 185, 186], and developers' interaction information [205, 207, 151].

These bug prediction models have largely focused on predicting bugs in the current version or the next version of the code base (*bugginess at a given time*). That is, these techniques aim to isolate the parts of the code that are likely to be buggy so as to facilitate bug-fixing efforts. While narrowing down the likely location of bugs in the current code base is useful, proactively identifying parts of the code base that is bug-prone (*i.e., more likely to be buggy now and in the future*), can have a bigger impact. In this paper, we focus on the latter, an activity we refer to as bug-proneness prediction. This can help

developers minimize the chances of the occurrences of future bugs, which can in turn reduce maintenance time and costs.

To explain the difference, between bug prediction and bug-proneness prediction, let us draw an analogy between these and a visit to the dentist. If we consider a bug in our code–the digital equivalent of a cavity–seeking the aid of a dentist is the equivalent of filing a bug report and seeking remedy to a bug. Bug prediction models can help developers identify the location of problems, much like an X-ray can help a dentist spot a hidden cavity. These are both reactive actions; a problem has occurred, and it needs remedy. Bug-proneness prediction is the equivalent of our dentist discovering weaknesses in the enamel or plaque buildup. While these may not need immediate attention, our dentist may direct us to floss and perform preventative care to prevent or slow down the development of a cavity in the future. As is the case with dental health, an ounce of prevention can be worth a pound of cure with bugs, as some of these can impact the safety and security of systems and their users. In summary, bug prediction focuses on predicting bugs in the current version or the next version. Bug-proneness prediction focuses on identifying the portion of code which will be buggy in all future versions because of how it is written "now". Bug-proneness is not only about identifying buggy parts in version N+1 (based on N), but also for predicting more temporally removed versions (N+n future versions).

While there are many approaches to bug (or defect) prediction, these primarily focus on predicting bugs in the current version [42, 102, 54, 102, 94, 175, 113, 240, 133] or in the next version [199, 116, 31, 185, 205, 38, 26] of a code base. Through a literature survey of papers in top Software Engineering conferences from 2008 to 2019 (see Section 6.2), we found only 2 studies [20] and [107] that identify bug-prone parts of a code base across multiple versions.

Ahmed et al. [20] were able to predict bug-proneness at an accuracy level of 0.70. Hassan et al. [107] used a change complexity metric to predict faults in software systems (with R^2 statistic ranging from 0.26 to 0.71) by training on two years of data and predicting faults on the next two years (multiple versions). Note that though these studies do not use the term bug-proneness, their goal is the same as ours.

There is potential to improve the accuracy of bug-proneness prediction. We believe this because extensive research on bug prediction [30, 175, 274, 275, 268, 268, 95, 100, 144, 185, 186, 205, 207, 151] has been able to obtain higher prediction accuracy than for bug-proneness prediction (AUC value more than 0.90 [94, 94]). Therefore, in this paper, we attempt to improve the accuracy of bug proneness prediction by leveraging the metrics that have been used in past bug prediction models.

As a first step to achieve our goal, we build a baseline bug-proneness model following the approach used by Ahmed et al. [20], so as to compare our final model with the baseline model. In their model, Ahmed et al. used code smells and a combination of product and process metrics (referred as traditional metrics in the rest of the paper):

1. Code smells [91] serve as an indication of the overall design quality of the code base [167]. They also reveal social aspects, such as a lack of developers' expertise, a lack of understandability of the code [257], or a lack of conformation to the development processes [20].

2. Code metrics relate to information about the code itself, such as its complexity, number and type of dependencies, etc.

3. Process metrics relate to information about development activities, such as number of developers, number of commits, etc.

Next, we identify metrics that have been useful in improving bug prediction models

and can be leveraged to improve bug-proneness models. Ahmed et al. [20] reviewed 43 different process and code related metrics from the bug prediction literature and selected eight metrics to use with code smell in their bug-proneness prediction model. In searching for metrics for improving bug-proneness prediction we found *one additional metric* not considered by them–entropy–which has been found to be useful in bug prediction [211]. Entropy is a measurement of the naturalness of code [117] and is defined by the syntactic difference of an entity (e.g. method) and the rest of the code base. Intuitively, the more "different" a piece of code is from its surrounding code, the more "unnatural" (higher entropy) it is. Highly entropic code is harder to maintain and has been associated with bugs (bugginess in current version) [211]. We, hypothesize that high entropic code would make the code base more difficult to understand and maintain, and therefore make it more bug-prone. This leads to our first research question:

RQ1: How useful is entropy in predicting bug-proneness?

Finally, we investigate the interplay of these different metrics to identify the most accurate bug-proneness prediction model. This is reflected in our second research question:

RQ2: What is the accuracy of the enhanced bug-proneness prediction model?

We investigate different mixed models to determine whether these models are mature enough for wide use by developers. As existing research has shown that combining different types of metrics [208, 38]) can significantly improve the accuracy of bug-prediction models.

To answer our research questions, we conducted a large scale empirical study. We sampled 120 projects from GitHub [97], the repositories of which contained over 500,000 commits, with an average of over 3,800 commits per project. For each version in the

repository, and for each method and file, we identified the code smells and calculated the entropy. We also tracked each statement throughout the history of the project, and all the bugfixes that affected it, to calculate the bug-proneness of each line of code. Using this information, along with other process and code related metrics collected from the project repositories, we built four bug-proneness prediction models. We do this in order to determine whether bug proneness prediction is accurate enough to serve as a preventative practice among developers.

The contributions of this paper are as follows:

- An analysis of relationship between *bug-proneness*, and entropy (naturalness in the code) for 120 sampled projects from Github.
- Evidence that there is a correlation between entropy and bug-proneness of a program element, indicating that lack of naturalness has a negative impact on the quality of software in future.
- Evidence that combining code metrics (i.e. Entropy) along with socio-technical metrics (i.e. code smells) and process metrics (i.e. number of authors) can significantly improve the bug-proneness prediction accuracy.

6.2 Literature Survey of Bug/Defect prediction models

We conducted a literature survey on existing bug/defect prediction models to get an overview of the prediction horizon of models (i.e., how far into the future these models focus). We reviewed all full conference and journal papers published in English and related to bug/defect prediction models from 2008 to 2019 (11 years) that appeared in top conferences and journals in Software Engineering: ICSE, FSE, ASE, ICSME, MSR,

ESEM,ISSRE,ICPC, ISSTA, IST, CSMR, FASE, TSE, ESE and IST.

We identified 1082 relevant papers by querying the four key computer science libraries (ACM, IEEE, Elsevier and Springer) using keywords such as, "defect prediction", "fault prediction", "bug prediction", "bug-proneness prediction". Then the first two authors read the abstract and the introduction of these papers to determine whether a paper was to be included in the study by applying the inclusion and exclusion criteria in Table 6.1. Because our goal was to identify and classify clusters or families of approaches to bug prediction, we excluded replication studies, studies comparing the effectiveness of approaches defined elsewhere, or papers improving on the effectiveness of a previously defined approach. While obviously important and impactful work, these papers did not change the theoretical landscape and could skew our results. This resulted in 46 papers.

Table 6.1: Inclusion and exclusion criteria for our literature survey.

Inclusion criteria (a paper **must be**...)	Exclusion criteria (a paper **must not be**...)
- An empirical study. - Introducing a new bug/defect prediction model or improving an existing one by adding or removing metrics from the model. - Focused on predicting bugs/defects in units of a software system. - Bug/defect is the main output (dependent variable).	- A replication study. - A comparison between two or more existing bug/defect prediction models without improving the model. - Related to improving feature selection for bug/defect prediction model. - Focused on: testing, inspection, reliability modeling, debugging and fault tolerance.

From these 46 papers, we categorized the bug/defect prediction models based on their prediction focus. We found three categories of models that predict bugs/defect prone code: those focused on (1) the current version—58.70%, (2) the next version—36.95%, and (3) any future version—4.35%. The last category (2 papers) included approaches to identify bug-prone parts of the code either in near future ($N+1$ version) or in the distant

future ($N+n$ version). From the small number of papers in category 3, we conclude that the concept of bug-proneness is still novel and under-researched. Table 6.2 summarizes our findings.

Table 6.2: Summary of our literature survey

Year	Total number of Bug/defect prediction papers	Papers on models for current version	Papers on models for next version	Papers on models for future versions
2008	5	1 [142]	4 [205, 173, 135, 255]	0
2009	5	1 [248]	3 [276, 38, 268]	1 [107]
2010	4	2 [26, 172]	2 [34, 176]	0
2011	1	1 [95]	0	0
2012	4	4 [94, 113, 133, 108]	0	0
2013	1	1 [240]	0	0
2014	3	1 [112]	2 [116, 261]	0
2015	3	3 [258, 188, 111]	0	0
2016	8	5 [42, 260, 259, 132, 267]	3 [31, 253, 199]	0
2017	5	2 [187, 160]	2 [57, 72]	1 [20]
2018	3	3 [54, 271, 262]	0	0
2019	4	3 [118, 65, 200]	1 [270]	0
Total	46	27	17	2

6.3 Methodology

To conduct our empirical study on bug-proneness prediction, we start by creating a dataset of 120 open source projects, described in (Section 6.3.1). For these projects we collected traditionally used process and product metrics (Section 6.3.4), code smell (Section 6.3.3), entropy (Section 6.3.2) and bug data (Section 6.3.5). Next, we built regression models with different combinations of traditional metrics, entropy and code

smells to predict the total number of bug fixes occurring on a statement (Section 6.3.6).

6.3.1 Project Selection Criteria

We wanted to make sure that our findings would be representative of the code developed in real world, thus we selected active, open source projects hosted in GitHub. We chose Java because it is the most popular programming language [17], and there are more code smell detection tools for Java than for other languages [87]. We adopted the three project selection criteria from the study by Ahmed et al. [20]– searching and selecting initial project, project size and number of files in a project. In this study we added additional selection criteria as discussed next.

To eliminate potential sampling bias we randomly selected 900 projects; the first to show up when using the GitHub search mechanism. Next, to ensure that the projects are representative of real-world projects, and not throw-away or class projects, we followed the guidelines proposed by Kalliamvakou et al. [131]. According to these guidelines, we eliminated those repositories that are not "actual" projects, have very few commits, are inactive, are not related to software development, or are personal projects. This left us with 500 projects. From these projects, we selected projects that met the following minimum needs: (a) project size (must have more than 10 files and 500 lines of code), (b) number of developers (from 4 to 105) and (c) project age (more than six weeks). This resulted in our final data set containing 120 projects. Table 6.3 provides a summary of our selected projects.

Table 6.3: Project Statistics

Dimension	Max	Min	Average	Std. dev.
Line count	542,571	751	91,792.43	114,950.6
File count	4,644	14	1,073.45	1,169.35
Duration (Days)	6,386	42	1,765.60	1,129.95
# Developers	288	4	74.10	61.30
Total Commits	30,519	113	4,362.20	5,200.38

6.3.2 Measuring Entropy

For our study we first calculate entropy for each java source file. To calculate the entropy of each java source file we used Python's `Entropy` library [2]. The library uses Shannon's entropy [3]. Other researchers also used Shannon's entropy [48] $H(X)$, which is calculated using the following formula:

$$H(X) = \sum_{i=1}^{n} P(x_i) \log_2 P(x_i) \tag{6.1}$$

In Equation 6.1, X is a java method. And x_i is a term (in our case a token in a java method) in X. $P(x_i)$ is the probability of a change occurring in a method. n is the total number of tokens in each java method. We will get a maximum entropy when all tokens have the same probability of having a change: $P(x_i) = 1/, \forall i = 1, 2, \ldots, n$. We will get minimum entropy if for a token i, $P(x_i) = 1$, and for all other tokens, other than i, $P(x_j) = 0, \forall j \neq i$.

6.3.3 Code Smell Collection

6.3.3.1 Code Smell Detection Tool Selection

To perform code smell analysis for the selected projects, we selected inFusion [7], a commercial tool. As a commercial tool, inFusion is no longer available for download. However, there is an open source version of the tool called iPlasma [6], which is available. We favor inFusion for several reasons. First, inFusion detects a wide range of 20 different code smells. Table 6.4 lists the smells and the metrics used by inFusion for calculating the smells. Details of the metrics along with the definition of the code smells are provided in the companion website [254]. Second, a previous study by Ahmed et al. [21] showed that inFusion has a very high precision of 0.84, recall of 1.00 and an F-measure of 0.91. Also, inFusion scales well to large code bases. Finally, many other studies [84, 88, 119] used inFusion as their code smell detection tool.

6.3.3.2 Measuring Code Smells

For each of the 120 projects, we collected the code smells by running inFusion [7] for each commit in the project code base. First, we collect the number of code smells in each code smell category for every file. Then we added all smells across all categories to collect the total number of code smells for each file. Overall, we found instances of 13 code smell categories in our study.

Table 6.4: List of Smells and Metrics used to measure each code smell by InFusion

Smell	Metrics
Sibling Duplication	LB, LOC, SDC, SEC
Blob Class	CYCLO, LOC, LOCOMM,MAXNESTING, NOAV, NOPAR, TCC, WOC
Blob Operation	CYCLO, LOC, LOCOMM, MAXNESTING, NOAV
Internal Duplication	LB, LOC, SDC, SEC
Schizophrenic Class	NOPUBM, TCC
Data Class	CW, NOACCM, NOPUBA, WOC
God Class	CPFD, TCC, WOC
Data Clump	NOPAR
External Duplication	LB, LOC, SDC, SEC
Tradition Breaker	NAS, NOM, AMW, WOC
Shotgun Surgery	CYCLO, ICDO, ICIO, LOC, MAXNESTING, OCDO, OCIO
Refused Parent Bequest	AMW, BOVR, BUR, NOM, NOPRTM, WOC
Message Chain	OCIO, ODD
Unstable Dependencies	IF
SAP Breakers	IF, NOAC, NOC
Intensive Coupling	DR, MAXNESTING, OCIO
Distorted Hierarchy	DIT, NOCHLD, PNAS
Feature Envy	ATFD, FDP, LDA

6.3.4 Collecting Traditional Metrics

For our study, we selected Ahmed et al. [20] proposed model as our base bug-proneness prediction model. So, we collect the same process and product related metrics (referred as traditional metrics) for our study. These metrics were selected from defect prediction literature (e.g., the size of the module under investigation [77], number of committers [255] and centrality of files [53]). Table 6.5 lists the final set of traditional metrics used as dependent variables in the bug-proneness prediction model by Ahmed et al. [20] including metrics that are code-based (F1, F2), change-related (F5-F8), and author-related (F3, F4). After collecting code smell and entropy related information we collected the above

Table 6.5: Traditional metrics used for bug proneness prediction model [20].

Factor	Metric
F1	Inward Dependency
F2	Outward Dependency
F3	Number of Non core author
F4	Number of Authors
F5	Number of Classes
F6	Number of Methods
F7	Number of AST nodes
F8	Line of code

mentioned traditional metrics (Table 6.5) from our data set. We use Understand [4] to count the number of references to, and from other files for each file. We also collect the number of line for each file of a project, number of classes, number of methods in a file, number of AST nodes, number of developers who made commits in the code base for each project.

To determine the total number of non core developers, we examine the experience level. We split the development history into quarters and for each quarter we consider a developer as *core* if he is in the top 20% of developers in that quarter (calculated by the number of commits authored). Otherwise he is *noncore*. Open source contributors follow a power law, where 20% of contributors are responsible for 80% of the contributions [179]. We used this as our categorization criteria.

6.3.5 Bug Data Collection

We identified the total number of bugs for each statement. For this, we used a machine learning classifier (Section 6.3.5.1) to classify all the commits in each project's repository into two categories: *Bugfix* and *Other*. Finally, we collect the number of bug fixes for

each (program) statement (Section 6.3.5.2).

6.3.5.1 Commit Classification

In order to distinguish between commits, we classify them into two groups: (1) *Bugfix*-commits that fix the bugs (modifying existing code), and (2) *Other*— commits that introduced new features or functionality (adding new code) or commits were related to documentation, test code, or other concerns. This could be done by either linking issue tracker data (on bug fixes) to commits or by applying machine learning technique.

Researchers have used SZZ [232] algorithm to identify bug fixing commits [94, 211]. SZZ links bug reports from issue tracking system to the bug fixing commit by using regular expressions to identify explicit references to bug reports in commit messages. Unfortunately, many of the OSS projects do not link the bug tracking and the version control systems [94] properly. Unsurprisingly, this was the case for our data set of 120 OSS projects. A manual inspection of our projects showed 78% did not have a culture of putting references from the bug tracking system into the GitHub commit. Therefore, we opted to not use SZZ(e.g., [232]) to identify the bug fixing commits. Instead, we used machine learning techniques to classify the commits as (1) it is not trivial to manually determine the category of a particular commit from the project repository and (2) large projects have a high number of commit activity in the repository [20].

We built our ML classifier by following the protocol of Ahmed et al. [20]. We first manually classified 1,500 commits by reading the commit messages to build and test our classifier. As a part of this process, two authors labeled 500 commits into two categories independently by searching for key words like “Fix,” “Bug,” “Resolve,” “Cleanup,” “Optimize,” “Simplify,” and their derivatives to place a commit in the *Bugfix* category. Then we

calculated the inter-rater reliability and found a Cohen's Kappa of 0.90. Cohen's kappa is a statistic that assesses the degree of agreement between the codes assigned by two researchers working independently on the same sample [190]. Values of Cohen's kappa fall between 0 and 1, where 0 indicates poor agreement and 1 indicates perfect agreement. According to the thresholds proposed by Landis and Koch [148], our kappa value of 0.90 indicates strong agreement between researchers. Once agreement was reached, one author manually classified the rest of the sample data. In our final training data, 46.30% of commits were assigned to the *Bugfix* category and 53.70% were assigned to the *Other* category.

Using these manually labeled data, we built two different classifiers, a Naive-Bayes (NB) and a Support Vector Machine (SVM) classifier using the Python Scikit-learn library [223]. We used 80% of the manually labeled data to train the classifiers and tested them on the remaining 20% of data. We used 10-fold cross validation to train and evaluate the classifiers [125]. This validation approach randomly divided the manually classified dataset into 10 groups of equal size. The first group is treated as a validation set, and the classifier is fit on the remaining 9 groups. The mean of the 10 executions is used as an estimation of classifier's accuracy. 10-fold cross validation has been recommended in the field of applied machine learning [147].

Table 6.6: Details of both classifiers

Classifier	Precision	Recall	F1-score
Naive-Bayes	0.63	0.43	0.51
Support Vector Machine	0.54	0.40	0.46

Among the two classifiers NB classifier outperformed the SVM, in terms of precision and recall which is similar to what was was reported by Ahmed et al.[20]. Table 6.6 has

the quality indicator characteristics of both classifier. We performed hyper parameter optimization using randomized search [213]. Tian et al. [247], suggest that for keyword-based classification the F1-score is usually around 0.55, which happened in our case. While our classifier is far from perfect, it is comparable to “good" classifiers in the literature. Further, our classifier does not systematically under-count or over-count bug fixes, indicated by precision of the classifier in identifying different types of commits in training data. Therefore, we believe it is unlikely for the biases to have a confounding effect on our analysis.

6.3.5.2 Line Tracking

We tracked each statement to measure the effect of code smell and entropy on the quality of the code. In order to determine when a program element (statement, block, method, or class) was changed, and track its history, we used the GumTree Algorithm [82]. For each element of interest, we considered it changed if the corresponding AST node was changed, or had any children that were added, deleted or modified. The algorithm maps the correspondence between nodes in two different trees, which allowed us to accurately track the history of the program elements.

GumTree has unique advantages over other line tracking algorithms, such as SZZ [143]. For example, GumTree ignores white space changes and can track a node even when it’s position changes within a file. Therefore, GumTree can correctly track statements that have been moved by, for example, an Extract Method refactoring.

To track statements, we used the version of the source code at epoch (In this case time of the first commit the statement appeared) to determine which AST node resided at that particular line. We only considered the commits that touch the file of interest. We

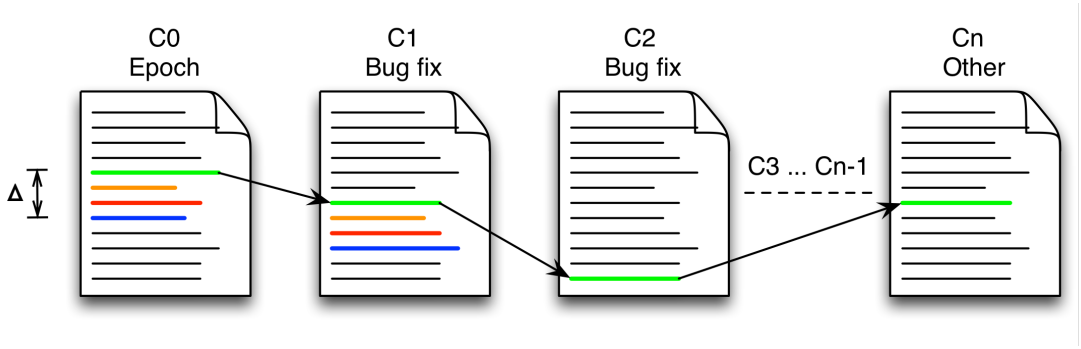

Figure 6.1: An overview of the tracking algorithm. The lines marked with Δ represent the statements. For each line we track all the modifications forward in time. We stop when we hit a commit that was classified as *Other*.

then tracked that AST node forward in time, taking note of the commits that changed that particular node. We stopped the tracking when the statement hit a commit that classified as "other", as shown in Figure 6.1. Then we collected the number of bug fix commits for each statement. For Java, it is possible for multiple statements to be in the same line (for example, a local variable declaration statement inside an if statement). In this case, we considered the innermost statement, as this gives the most precise results.

6.3.6 Data Analysis

Before performing the data analysis, we performed the Anderson-Darling normality test [212] on our data related to entropy score, code smell count, traditional metrics values and bug count. We found a p-value of less than 0.05, which indicates our data does not have a normal distribution. We did the Anderson-Darling normality test to choose the suitable statistical tests for the data, as many of the statistical test assume the data is normally distributed.

To identify the impact of the collected metrics on the number of bug-fixes that occur

on lines of code, we build regression models. For this purpose, we use Generalized Linear Regression [61] where we use collected metrics (traditional metrics (Table 6.5), entropy and code smells) as independent variables to predict the total number of bug fixes on a line of code (dependent variable). In our dataset, the number of bug fixes occurring on a line of code follows a Poisson distribution. We decided to use logistic regression since it performs best among the available regression techniques [103]. We used the `varImp` function in the `caret` [51] package for this purpose. Since we are using a regression from the Generalized Linear Model (GLM) category, no tuning is needed because the model has no hyper-parameters and assumes a logit relationship between response and predictors [221].

In order to answer our second research question, we compare multiple regression models built with different combinations of traditional metrics (Table 6.5), entropy and code smells. To compare the models, we follow the methodology suggested by Stockl et al. [239] where we build regression models to check if there is any improvement in prediction accuracy using Leave-One-Out model of validation, starting with the baseline model built using number of code smells and traditional metrics following the approach used by Ahmed et al. [20] and then adding, in turn, entropy and the interaction of code smells and entropy. However, before using code smell and entropy in a model, we check the correlation between them by performing a correlation analysis between count of smells and entropy. We used Kendall's rank correlation test. We chose Kendall's rank correlation test over Spearman's because it is more robust [96]. Also, Kendall's tau allows for an easier and clearer interpretation of the results. We also checked for multicollinearity using the Variance Inflation Factor (VIF) of each predictor in our models [61] for our data set. VIF describes the correlation between predictors. A VIF score between 1 and 5 indicates moderate correlation with other factors, and these selected the predictors

are with $VIF < 5$. This step was necessary since the presence of highly correlated factors forces the estimated regression coefficient of one variable to depend on other predictor variables that are included in the model. See Table 6.7 for the models with the independent variables used in this study.

Table 6.7: Value of independent variables used to predict number of bug fixes for each regression model

Regression Model	Value of independent variables
$Model_{Base}$	Traditional metrics and Code Smell
$Model_{E}$	Entropy
$Model_{CSE}$	Code Smell and Entropy
$Model_{TE}$	Traditional metrics and Entropy
$Model_{Final}$	Traditional metrics, Entropy, Code Smell and interaction of Code Smell and Entropy

6.4 Results

In the following sections, the collected and observed results for the research questions stated above are presented. We start by building the baseline model following the approach used by Ahmed et al. [20] to compare the performance of the final model with the baseline model.

6.4.1 Baseline Bug-proneness model

Our goal is to improve bug-proneness prediction accuracy by adding new features to the prediction model proposed by Ahmed et al. [20], so we use the same set of metrics as Ahmed et al. to build the baseline model. After collecting the traditional metrics and

code smells used by Ahmed et al.(Discussed in section 6.3.4) we build a Poisson regression model with a log linking function as explained in section 6.3.6. Ahmed et al. reported a McFadden Adjusted R^2 [115] of 0.47 for their model. We got a similar McFadden Adjusted R^2 value of 0.45 using our model. We were, therefore, able to replicate Ahmed et al.'s findings in our data set.

We use AUC (Area Under the receiver operating characteristic Curve) to assess the performance of the prediction models. Since the predicted value of our multinomial logistic regression model is the total number of bug fixes occurring on a line of code, we treat this as a multi-class classification problem. We use multi-class AUC defined by Hand et al. [105], where the reported AUC is the mean of several AUC. We use AUC instead of McFadden Adjusted R^2 for several reasons. First, it is independent of prior probabilities [33]. Second, AUC is a better measure of classifier performance, because it is not biased by the size of test data. Finally, AUC provides a "broader" view of the performance of the classifier since both sensitivity and specificity for all threshold levels are incorporated in calculating AUC. Prior studies comparing bug prediction models have also used AUC [72, 94, 95, 266].

From Table 6.8, we can see that the AUC value of the$Model_{Base}$ is 0.88, which indicates a high accuracy in predicting bug-proneness; AUC values between $0.5 \leq AUC \leq 1.0$ are considered as highly accurate [72, 1].

Table 6.8: AUC value of the Poisson regression models

Model	Independent variables	AUC
$Model_{Base}$	Traditional metric+Code Smell	0.88
$Model_E$	Entropy	0.89

6.4.2 Predicting Bug-Proneness Using entropy (RQ1)

Recall, high entropy in code has been found to be a strong predictor of bugs in the current code base [117]. Therefore, we hypothesized that entropy would also be a good predictor for bug-proneness. To test this hypothesis we built a bug-proneness prediction model ($Model_E$) using only entropy as independent variable.

From Table 6.8, we can see that $Model_E$ has a high accuracy–0.89–in predicting bug-proneness; Accuracy between $0.5 \leq AUC \leq 1.0$ is considered high [72, 1]. Using entropy alone gives a higher AUC than the model proposed by Ahmed et al. [20].

However, since previous research has indicated that combining different metrics (process and code metrics [208]) can significantly improve the prediction accuracy, we next investigated whether entropy can be added as a factor to the existing model, $Model_{Base}$ and to what extent it can improve the prediction of bug-proneness.

Observation 11: Using entropy alone gives a prediction accuracy that is as good as the $Model_{Base}$ for predicting bug-proneness.

6.4.3 A Combination Model for Predicting Bug-Proneness (RQ2)

Given that entropy can predict bug-proneness, we next investigated whether it can be used in conjunction with the other metrics in our baseline model. Before creating a combination model that includes entropy along with the other metrics we first needed to investigate the association between code smells and entropy. Given our intuition that code smells and entropy both reflect structural issues in the code base, it is possible that they are closely associated. If that is the case then both these metrics should not be included in the model as we need to control for multicollinearity among variables so as to not negatively impact the performance of the model.

6.4.3.1 Association between code smells and entropy

To investigate whether and to what extent code smells are associated with entropy, we first calculated the smells for each of the 120 projects in our data set. Table 6.9 contains the percentage of each smell and the percentage of projects that have a particular smell. Table 6.9 shows that code smells are prevalent across a majority of these projects. Also, there is a wide range of smells impacting projects.

Internal and external duplication are the top code smells in our data set, making *duplication* one of most prevalent code smells. This is along the lines of findings in prior work [21].

Table 6.9: Percentage of code smells

Smell	% of smells in the full data set	% of projects with smell
External Duplication	46.80	22.53
Internal Duplication	37.42	23.80
Feature Envy	4.04	28.42
Data Clumps	3.71	20.36
Data Class	3.27	48.05
Blob Class	2.58	30.05
God Class	0.89	19.10
Schizophrenic Class	0.58	20.00
Message Chains	0.33	5.34
Tradition Breaker	0.17	6.33
Refused Parent Bequest	0.19	5.25
Shotgun Surgery	0.01	1.72
Distorted Hierarchy	0.01	0.36

Next, we calculated the level of entropy in the code base that has these smells. To do this, we calculated the entropy for each method in a file using the expression in Section 6.3.2. The distribution of entropy score for our data set of 120 projects is shown

in Figure 6.2. The median entropy score of our data set is 7.68 with a standard deviation of 0.39. This is similar to the numbers reported by Hindle et al. [117] in their data set of 10 java projects; Another instance where we *replicate past research findings through our data set.*

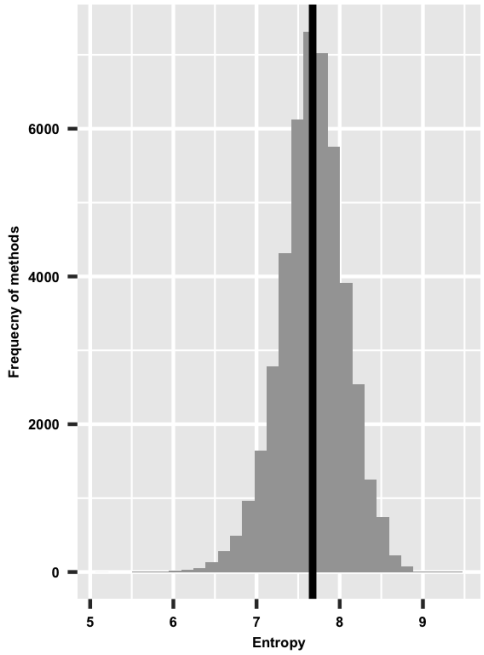

Figure 6.2: Histogram of the entropy distribution. The median is 7.68, marked by the solid line

The mean entropy of program elements with code smells is 7.73*, whereas the mean entropy for non-smelly program elements is* 7.66, as shown in Figure 6.3. We did not remove the outliers from the analysis because they were evenly distributed around the mean, and would not negatively impact our statistical tests.

To test whether the populations (entropy scores of code with or without code smells) were different, we performed a non-parametric Mann-Whitney test since our population

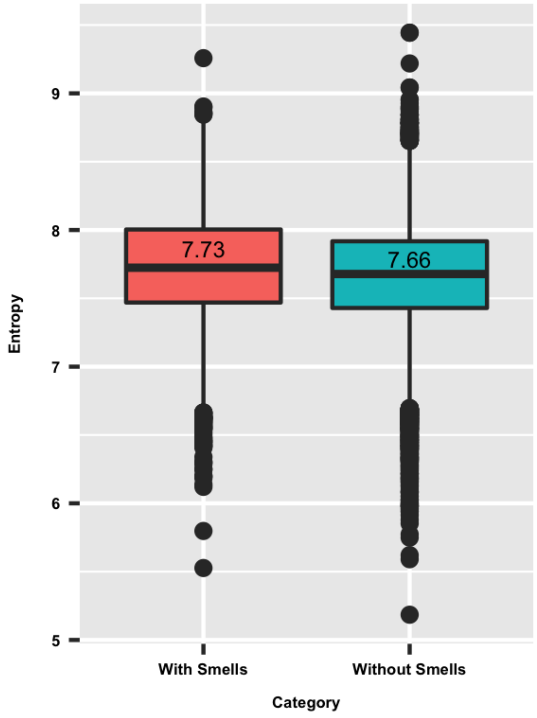

Figure 6.3: Entropy of Smelly code vs. Non-smelly code

was not normally distributed. We found that the population means are statistically significant ($p < 2.2 \times 10^{-16}$), although their effect size is negligible (0.17, Cohen's d).

We next investigated the association between total code smell count of a project and entropy to see if there is any correlation between them. We found a positive, but small correlation of 0.04 (Kendall's τ) between code smells and entropy using Kendall's rank correlation test ($p < 2.2 \times 10^{-16}$).

Correlation with Individual Code Smells: Since we found a correlation between total counts of code smell and entropy, we unpack this association further to investigate whether particular code smells are more closely associated with entropy. The intuition being particular code smells, such as God Class or Data Clumps might make the code

become more "unnatural" than the rest of the code. We performed Kendall's rank correlation test between the counts of each smell and entropy to distill which smells from Table 6.10 are more associated with entropy. Since we performed multiple tests, we had to adjust the significance value to account for multiple hypothesis correction. We used the Bonferroni correction [110], which gave us an adjusted α value of 0.003 to be used as the significance level.

All code smells in our data set are *significantly* correlated with entropy, but the correlation values are all less then or equal to 0.05 as shown in Table 6.10. We see that the bottom three (line break in table) are negatively correlated, whereas the other 10 are positively correlated (Kendall's τ, $\alpha < 0.003$).

The three strongest correlations of entropy to code smell are with the following smells: *Feature Envy, Blob Class,* and *God Class* with Kendal τ of 0.05. These smells all relate to the cases where the object-oriented design principles of encapsulation and structuring are not well used, leading to maintainability problem and bug-proneness. In Section 6.5 We discuss these findings and whether classes of code smell have implications for entropy.

6.4.3.2 Association between Traditional metrics and entropy:

As our final goal was to build a bug-proneness prediction model using traditional metrics, code smells and entropy, we needed to make sure that these factors were not closely associated with each other. We already investigated the association between code smells and entropy in the previous section and found them to be negligible. In this section, we will discuss our analysis on the correlation between our selected traditional metrics and entropy.

To check this association, we performed correlation tests between entropy and and

Table 6.10: Correlation between entropy and smell count

Smell	correlation	p-value
Blob Class	0.05	1.07×10^{-37}
Feature Envy	0.05	9.32×10^{-1}
God Class	0.05	2.47×10^{-4}
Schizophrenic Class	0.02	4.97×10^{-1}
Refused Parent Bequest	0.02	1.10×10^{-6}
External Duplication	0.02	5.19×10^{-11}
Data Clump	0.01	8.57×10^{-1}
Data Class	0.01	8.24×10^{-1}
Shotgun Surgery	0.01	9.61×10^{-3}
Internal Duplication	0.01	2.41×10^{-3}
Tradition Breaker	-0.01	3.84×10^{-5}
Message Chain	-0.01	4.46×10^{-5}
Distorted Hierarchy	-0.03	3.09×10^{-1}

the traditional metrics from the projects in our dataset. Table 6.11 shows the correlations between the traditional metrics and entropy. The correlation between each traditional metric and entropy are statistically significant, except the correlation between entropy and number of authors and Line of code. However, regardless of the significance values, the correlation values (Kendal τ) are very small.

Given that the correlation between traditional metrics, code smells and *entropy* is very small, all of these metrics can be used in a model simultaneously without sacrificing the statistical restrictions.

Table 6.11: Correlation between traditional metrics and entropy

Metric Type	Traditional metric	Correlation with entropy
Code metrics	Inward Dependency	0.13
	Outward Dependency	0.28
	Number of Classes	0.07
	Number of Methods	-0.04
	Number of AST nodes	-0.06
	Line of code	**0.01**
Process Metrics	Noncore author	-0.24
	Number of Authors	**-0.03**

6.4.3.3 Bringing it all together: Code smells, entropy and traditional metrics for predicting bug-proneness

Here we investigate how traditional metrics, code smells and *entropy* can be used together in a model for predicting bug-proneness.

After building the $Model_{Base}$ (Using the model proposed by Ahmed et al. [20]) with our data, we build regression models to check if there is any improvement in prediction accuracy using a Leave-One-Out model of validation, starting with the model built using the number of code smells and traditional metrics and then adding, in turn, entropy and the interaction of code smells and entropy. The first model includes eight traditional metrics, the second one includes the number of code smells and the final model includes traditional metrics (shown in Table 6.5) and number of code smells. As we are trying to evaluate the fit of a model, leave-one-out is the best approach, because it allows us to identify which factors contribute most to the model's accuracy.

Next, we compare our final model ($Model_{Final}$) with the $Model_{Base}$ and the other models that we build following leave-one-out approach. Table 6.12 shows the AUC

value for each of the models. From the table we see that code smells, entropy and the interaction of code smells and entropy have a significant impact on the bug-proneness of code and we achieve 4% improvement in AUC over using code smells along with traditional metrics ($Model_{Base}$). 4% improvement is a substantial improvement, given that strong predictors are needed to achieve satisfactory increment in AUC value and even a large gain in performance may not yield a substantial increase in AUC value [204]. We performed a Mann-Whitney U test to check whether the achieved improvement in AUC between the final model and the baseline model is statistically significant or not. We did not find any statistically significant difference (p-value > 0.05).

Observation 12: Code smells along with entropy and traditional metrics can increase the bug-proneness prediction AUC up to 4% compare to the base model.

Table 6.12: AUC of the Poisson regression models using both entropy and code Smells

Model	Independent variables	AUC
$Model_{Base}$	Traditional metric+Code Smell	0.88
$Model_{CSE}$	Code Smell+Entropy	0.87
$Model_{TE}$	Traditional metric+Entropy	0.90
$Model_{Final}$	Traditional metric+Code Smell+Entropy +Code Smell*Entropy	0.92

Table 6.13 lists the 11 factors from our $Model_{Final}$ that were identified as significant ($p < 0.05$). $Model_{Final}$ includes metrics that are change-related (F1, F3), code-based (F6, F7, F10, F11), author-related (F4, F9), number of code-smells (F8), entropy (F2) and the interaction term of number of code smell and entropy (F5). Factors F2 and F5 are our addition to the$Model_{Final}$. Looking at the coefficients and relative importance of individual predictors from our $Model_{Final}$ model, we see that entropy is clearly high in importance (362.41). Among the other predictors, socio-technical metrics such as

dependencies, no. of authors etc. are significant. This also provides a strong signal that bug-proneness is not purely technical in nature but also a manifestation of socio-technical metrics.

Observation 13: Bugs are not purely technical in nature, they are a manifestation of socio-technical factors.

Table 6.13: Poisson regression model predicting bug-fix occurrence on lines of code (Factors sorted by relative importance)

Factor	Metrics	Estimate	Relative Importance
F1	Outward Dependency	0.12	496.94
F2	**Entropy**	0.99	362.41
F3	Inward Dependency	-0.17	320.20
F4	No. Authors	-0.35	257.87
F5	**# of Smells × Entropy**	-0.16	256.98
F6	No. Methods	0.69	115.75
F7	No. Classes	-0.10	94.00
F8	# of Smells	0.13	92.74
F9	Noncore author	-0.19	51.56
F10	Number of AST nodes	-0.01	38.83
F11	LOC	0.01	32.63

6.5 Discussion

Although bug-proneness prediction can help developers save future effort and cost, there is very little research on improving bug-proneness prediction accuracy. In fact, a literature survey on existing bug/defect prediction models over last 11 years resulted in identifying only 2 papers that investigated bug-proneness of code either in near future ($N+1$ version) or in the distant future ($N+n$ version). As this research is relatively new, we start our study by using one of the most recent works on bug-proneness prediction by Ahmed et al. [20] as baseline and find new metrics to achieve higher bug-proneness prediction

accuracy. We also looked into different defect prediction metrics that were not considered by Ahmed et al. and decided to investigate entropy in detail due to its high bug prediction power [211]. Our results show that we can use entropy for bug-proneness prediction and by only using entropy as a factor we get a prediction accuracy that is as good as the existing one (See table 6.8).

We also investigate the connection between code smells and entropy. To the best of our knowledge, we are the first to do that. Our key intuition is that since highly entropic code tends to be buggy, and code smells are also associated with bugs, smelly code should have higher entropy (i.e., be more "unnatural") compared to non-smelly code. And there should be association between entropy and code smells. We find that program elements that are smelly indeed have higher entropy when compared to elements that are not smelly. The mean entropy for code with smells is higher when compared to code without smells, 7.73 and 7.66 respectively.

To get a more nuanced view of the relationship between entropy and code smells, we investigated the individual code smells and their association with entropy. The top three code smells associated with entropy are: Blob Class, Feature Envy and God Class (correlation value is 0.05). All these code smells arise when developers do not fully exploit the advantages of object-oriented design, which leads to bloating of code, duplication, or large containers. For example, Blob Classes are very large and complex. Because of their size, they are also more likely to be strongly coupled to other classes and have low-cohesion. This situation becomes worse over time as such classes tend to bloat and non-cohesive functionality tend to pile up. Due to the lack of cohesiveness, different kinds non-related functions are placed in proximity which eventually leads to decreased repetition a.k.a. high entropy (Section 6.3.2).

We also find that Message Chain, Tradition Breaker and Distorted Hierarchy have

a negative correlation with entropy. Distorted Hierarchy is related to an inheritance hierarchy that is unusually narrow and deep. Having an inheritance hierarchy that is too deep may cause maintainers "to get lost" in the hierarchy, making the system in general harder to maintain. This also increases the repetition of program constructs as the depth increases and leads to reduction in entropy. In this case, while these code smells increase likelihood of bug-proneness (as identified in [20]), it does reduce entropy. This suggests that the relation between these code smells and *entropy* should be further investigated, especially from the perspective of developers and maintainers to better understand how these different metrics affect them and to what extent.

Although there is correlation between code smells and entropy, the effect size is pretty small (0.04, Kendall's τ). However, when used together in a model, code smells and entropy combination improves the accuracy of bug-proneness prediction by 4%. Intuitively, smelly code with high entropic value should be more difficult to understand and more error-prone compared to a code with only code smells or only high entropy. That may be one of the reasons why the combined presence of code smell and entropy has more predictive power then either of them individually.

Also, the mean entropy of code with smells is higher compared to code without smells, however their practical significance is negligible (0.17, Cohen's d). We hypothesize that the negligible difference may be related to the fact that the majority of the code base is affected by smell. Since entropy measures how different the code is relative to code surrounding it and if code smells are prevalent all over the code base, the difference in the repetition patterns (as measured by entropy) may not be large, hence resulting in negligible difference in mean entropy.

Despite 10 out of the 13 code smells having positive correlation with entropy, the interaction term is negative (-0.16), which may be occur due to the distribution of

these smells in our data set. The negative interaction term most likely occurs due to the overwhelming presence of duplication code smells (shown in Table 6.9). Since, duplication is the most frequent code smell in our data set; Increased repetition of programming constructs due to duplication may be causing reduction in entropy. Further research is required to understand how and why the combination of code smells and entropy improve the accuracy of bug-proneness prediction.

We also investigate the relative importance of different factors in the $Model_{Final}$ (Table 6.13), we see that outward dependency is the most important factor. Which is inline with the findings of prior research, showing that if dependencies are not updated accordingly it could introduce bugs in the code [140, 272]. Inward dependency is another important factor which has negative regression coefficient. This could be due to the fact that when a file become more central, it gets more feedback for improvement, because any bug in this central file is likely to be noticed by others. As a result of such continuous bug-fixing, central files are less bug-prone. Another important factor which has negative coefficient is number of authors (See Table 6.13) which is inline with the *Linus's Law* that states, "Given enough eyeballs, all bugs are shallow". This is also similar to the findings of Sen et al. [224], who found that number of developers reflect the "healthiness" of an open source project.

6.6 Implications

In this section, we present practical implications of our study for researchers, tool builders, and application developers.

Researchers: We have shown that a combination of code smells, traditional (process and code related) metrics and entropy improves the accuracy of bug-proneness prediction.

This is inline with prior research which indicates, models using a combination of different categories of metrics perform better as compared to models built using a single category [103]. This means that bug-proneness prediction that includes entropy may be further improved if additional metrics (i.e. static code metrics, process metrics, technical network metrics etc.) are added.

In this paper, we have identified how code smells and entropy together can affect bug-proneness. However, it is open research to understand how the combination of code smell and entropy can help in identifying other issues in software development. For example, code smells are associated with maintainability issues and intuitively a smelly code with high entropy should be more difficult to maintain compared to a code with only code smells. Further investigation is required to answer such questions.

Also, researchers working on identifying and predicting code smells should note that, though entropy and code smells are correlated, the effect size is negligible. Therefore, using entropy for predicting code smells might not be very fruitful and vice versa.

We find that code smells that are highly associated with entropy are not focused in the literature. Similar to the findings of [21], we find that except for "Feature Envy," none of the other code smells with high association with entropy (shown in Table 6.10) are among the most researched code smells in literature. Therefore, we encourage researchers to prioritize code smells based on not only their prevalence in real-world applications, but also on their impact on code quality (i.e bug proneness).

Previous findings on developers' perception [196] has shown that developers don't think all code smells to be problematic and our empirical findings indicate that code smells with high association with entropy are in fact some of those code smells. For example, developers have divided opinions about Feature Envy, which is a one of the top code smells both in terms of prevalence and association with entropy in our corpus.

Therefore, Software Engineering educators need to prioritize these code smells and explain their impact on the code quality in terms of bug-proneness, entropy etc.

Tool builders: To the best of our knowledge, no popular IDEs report on entropy and the existing tools/plugins for code smell [85] only detect specific code smells. Our findings show that code smells impact code quality and developers don't perceive them as important [257, 242], and are likely to ignore them. We posit two things.

First, entropy should be monitored. Moreover, when particular code smells arise along with an increase in entropy, impacted parts of the code base should be highlighted and prioritized for analysis. The prioritization could be based on the identified severity of the code smells along with their association with entropy.

Second, tools need to provide actionable guidance for developers to detect and remove code smells. Currently, there is a gap in existing tools. Such lack of guidance from tools and the perceived "only long-term effect" of code smells make developers not focus on code smells. We therefore, encourage tool builders to close the gap between the detection and the correction of code smells.

Developers: Developers can use bug-proneness prediction to minimize their future maintenance cost. To identify the bug-prone code they can use different metrics. For example, developers can identify bug prone parts of the code base by only using entropy (which gives a higher AUC value than our $Model_{Base}$, Table 6.8). However, we find that, if they consider all factors in conjunction, they will have a significantly higher return on investment. Our results also have implications for testing. The correlation between code smells, entropy, and bug proneness can help in developer's testing effort. For example, increasing the test coverage of smelly lines with high entropy can be used as an objective function in the field of search based software engineering [106].

6.7 Threats to Validity

We have taken care to ensure that our results are unbiased, and have tried to eliminate the effects of random noise, but it's possible that our mitigation strategies may not have been effective.

Bias Due to Sampling : To ensure representativeness of our samples, we used search results from the GitHub repository of Java projects. We picked all projects that we could retrieve given by the GitHub API. However, our sample of programs could be biased by skew in the projects returned by GitHub. GitHub's selection mechanisms favoring projects based on some unknown criteria may be another source of error. Since we used only a single source (GitHub) our findings may be limited to open source projects from GitHub. However, we believe that the large number of projects sampled more than adequately addresses this concern.

Bias due to tools used: Code smells that are intrinsically historical, such as Parallel Inheritance, are difficult to detect by just using static source code analysis [197]. So, the number of occurrences of such "intrinsically historical" smells will be different when historical information based smell detection technique is used.

Secondly, we used the Gumtree algorithm [82] to track program elements across commits. However, this algorithm is unable to track program elements across renames or movement to another folder. Further, re-factoring that involves modification of scope, such as moving the code out of the current compilation unit also causes the algorithm to lose track of the program element after refactoring.

Bias Due to Commit Classification: Our determination of commits as bug-fixes depend on a learned classifier. As with any classifier, we have some mislabeling. While our results do not require those results to be anywhere near perfect, this threat is low as our

classifiers have good F1-measure and high precision. Our recall and precision measures are on par with past work [37, 247]. Moreover, our classifier do not systematically under-count bug fixes, indicated by precision of the classifier in identifying different types of commits in training data (See section 6.3.5.1). We have assumed that all bugs were found and fixed by developers when we use it as a metric of bug-proneness. This may not always be true, and hence our results are conservative.

Bias Due to lack of completeness of used Metrics: Our set of code and process metrics are diverse and widely used in literature. However, we can not guarantee that our set of metric are exhaustive but we based our metric selection on prior work [103, 208, 252] which ranges from code ownership and developer experience to file activities. We therefore argue that our set of metrics are comprehensive.

6.8 Related Work

In this section, we describe related works on using different kind of metrics for bug prediction and bug-proneness prediction.

6.8.1 Prediction using process metrics

A large volume of research has been done on bug prediction. Researchers have proposed and validated various metrics that helped to improve the prediction accuracy. These metrics are also grouped into broader categories. One such category is process related metrics. This category consists of metrics such as change metrics [66]. Graves et al. [100] used the number of changes to build Generalized Linear Models. Ostrand et al. [194] used fault and change history and achieved good performance in predicting not only the

location of the bug, but also the number of bugs. Nagappan et al. [186] found that the number of subsequent, consecutive changes is a strong predictor of bugs. Shihab et al. [229] predicted surprise bugs in files using the count of changes affecting the file. Yang et al. [258] built their defect prediction model using change metrics and achieved an average F1-score of 0.45. A number of other studies [33, 31, 205, 173] used process metrics as factors in their studies to predict bugs/ defects in software system and achieve an AUC up to 0.92.

6.8.2 Prediction using code related metrics

Code related metrics represent another category that is used in bug/defect prediction. Basili et al. [30] investigated the impact of the CK metrics suite on software quality. Menzies et al. [175] showed the usefulness of static code metrics in building prediction models using the NASA data set. Using the same data set Lessmann et al. [154] evaluated different machine learning algorithms and found that the difference between those algorithms is mostly not statistically significant. However, this ceiling effect is reported to disappear when focusing only on maximizing detection and minimizing false alarm rates [176]. The applicability of lines of code (LOC) to predict bug was demonstrated in [268]. An extensive empirical study with 38 different metrics and multivariate models to predict the fault-prone modules of the Apache web-server is presented in [274]. Nagappan et al. [185] built statistical regression models using relative measures of code churn as an independent variable and detect defects in software system with an accuracy of 89%. Liang et al. [157] used a combination of object-oriented metrics, and statistics collected from the warnings of various static analysis tools as their feature set. Gyimothy et al. [102] showed that object-oriented metrics are useful predictors of faults. Brun et

al. [44] used the Daikon dynamic invariant detector to generate runtime properties and used them as the feature set. Kim et al. [142] used change metadata such as length of change log, changed LOC (added Δ LOC + deleted Δ LOC) along with a combination of object-oriented metrics as their feature set. They also reported the highest level of accuracy compared to Gyimothy et al. [102] and Brun et al. [44]. Bowes et al [42] used mutation metrics for fault prediction. Zimmermann et al. [274] used social network analysis on dependency information to build prediction models. A study by Hassan [107] used change complexity metrics to build fault prediction model using data from the second and third years of a project to predict faults in the fourth and fifth years.

6.8.3 Prediction using code smells

Another type of metric used for bug prediction are code smells, which is a socio-technical metric. Initially the concept of code smell was developed to indicate parts of code base that reflected sub-optimal design and implementation choices [91] and adversely affect the maintainability and quality of a software system [146, 167]. Studies found an association between code smells and bugs [192]. Palomba et al. [198] used smell intensity to improve bug prediction accuracy. Studies also show relationship between bug-proness and code smells. Falessi et al. [80] found that classes exhibiting code smells (violating several quality rules) are five times more bug-prone than classes without smells. Hall et al. [104] found relationships between code smells and fault-proneness. According to their study some code smells indicate fault-proneness in the code base but the effect size is small (under 10%). Another study by Taba et al. [240] used anti-patterns as one of the metrics with other process and code related metrics to predict bugs in the current version. Khomh et al. [19] showed that classes affected by code smells are more likely to be buggy in the

future. Ahmed et al. [20] also used code smell for future bug-proneness prediction.

6.8.4 Prediction using entropy

Entropy is a metric used to measure the naturalness of code. It has also been used for bug prediction. In a study, Ray et al. [211] found a correlation between buggy code and entropy. According to their study buggy code has higher entropy than non-buggy code. Hassan et al. [107] compared the entropy with the amount of changes and number of previous bugs in a bug prediction model and found that entropy is a better predictor. D'Ambros et al. [66] extended Hassan's [107] work and found that source code metrics better describe the entropy of changes. Gerardo et al. [48] found that the change entropy decreases after any refactoring of the code. Chakraborty et al. [54] used entropy score from statistical language models and used it for spectrum based bug localization (*SBBL*). They also found a significant improvement compared to standard *SBBL*. In a recent study by Zhang [271], they used cross-entropy with traditional metric suites in a defect prediction model and found that the performance of prediction models is improved by an average of 2.8% in F1-score. To best of our knowledge, no one has used entropy to predict bug-proneness. In this study, we used entropy to predict parts of the code base that is bug-prone (*i.e., more likely to be buggy in the future*).

6.9 Conclusions

In this paper, we investigated whether and to what extent we can improve the bug-proneness prediction accuracy. Finding the parts of the code likely to be buggy in the future using bug-proneness prediction can help developers save future maintenance cost.

Though a plenty of work have been done on predicting bugs in current version, there is a gap in the research of bug-proneness prediction. To fill that gap, we tried to come up with a better prediction model by adding other metrics to the existing one [20]. We added one metric, "entropy" to the model proposed by Ahmed et al. [20] to get a better prediction model. Our choice of investigating entropy as a predictor of bug-proneness was motivated by the findings that show that high-entropic code is correlated with bugginess in the (current version) of the code base.

Our large scale empirical analysis shows that entropy is a good predictor for *bug-proneness prediction* and as the correlation between entropy and code smells and traditional metrics are very low, we can use them together in a model to get a better prediction model. So we enhanced the model proposed by Ahmed et al. [20] by adding entropy, and found that the $Model_{Final}$ built with the combination of code smells, entropy, and traditional metrics achieves a 4% gain in prediction accuracy compared to the prediction model proposed by Ahmed et al. [20].

The importance of this work is two fold. First, our results show that we can achieve an improvement of 4% in predicting the future occurrence of bugs over current state of the art method [20]. While this is an important step forward, it also serves as a call to action for future research to investigate other metrics that can further improve the prediction accuracy.

Second, it provides empirical evidence that there is a statistically significant association between entropy and code smells, with marginal effect size, which indicate that code smells are "unnatural", something that is intuitive, but hasn't been studied thus far. However, this "unnaturalness" is marginal and raises the question whether code smells are something developers create naturally, instead of the commonly held belief that code smells are a product of bad programming habits and a lack of experience. This

result also identifies the need for further research to understand the underpinning of how and why developers create code smells and how the unnaturalness of code smells along with the unnaturalness of entropy can impact various aspects of software quality such as understandability.

Chapter 7: Conclusion and Future Work

Our goal in this dissertation was to investigate the impact of design quality on the long-term health of OSS projects and how do projects manage this. To achieve this high-level goal, we conducted a number of empirical studies focusing on four research questions: (1) How does design quality in OSS projects evolve over time? (2) How do OSS projects manage design quality over time? (3) How does design quality impact 'high-coordination need' work like merging changes? and (4) How can design quality metrics be used to improve the long-term health of projects?

In the first part of this thesis, we examined the evolution of design quality in OSS projects. We looked into the evolution of two design quality metrics: code smells and entropy. We found that both code smells and entropy scores increase over the life of open source projects. This indicates that as the projects grow older and bigger, design issues tend to accumulate faster than they are resolved, and as a consequence, overall design quality degrades over time.

In the second part, we examined the process OSS projects follow to manage design quality. We found that though OSS projects use multiple communication channels for design-related discussions, developers prefer the mailing list. We also found that the correlation between design discussions and design quality is low. In our survey, developers mentioned that retrieving design decisions from the official channels and the offline discussions that are not in the project archives are very difficult and sometimes impossible. Consequently, it was hard for developers who are not part of these discussions to gauge the impact of their contribution to design quality. These factors could be playing a role

in the low correlation of design discussions with design quality.

In the third part, we examined how design quality impacts day-to-day activities like code merging and as well as the long-term health of projects. Our findings showed that (I) code involved in merge conflicts contain three times more design issues (code smells) than non-conflicting ones, (II) code smells, which are thought to be an indicator of maintenance issues and often neglected by developers, have an immediate impact on how distributed development is managed, (III) design quality impacts the project's long-term health, future bug-proneness. We also built a bug-proneness prediction model using design quality and traditional metrics and achieved an AUC value of 0.92. Finding the parts of the code likely to be buggy in the future using bug-proneness prediction can help developers save future maintenance costs.

In summary, the design quality of OSS projects degrades over time, impacting both day-to-day activities and the long-term health of the projects. One of the main reasons behind this design degradation is the lack of tools for developers to monitor design quality. Current code smell detection tools do not support just-in-time analysis and refactoring suggestions for the most recent changes. To the best of our knowledge, no tool shows the current entropy score for a piece of code. One of our future research direction is to build such tools to help developers monitor design quality. Another reason behind this degradation could be developers' limited access to design decisions. One way to help developers is to build tools to find and link different design discussions across multiple channels. Our study (Chapter 4) takes the first step towards automatically identify (and retrieve) design-related discussions from multiple discussion archives. However, further research is needed to develop more efficient automated techniques to locate design discussions and link them to a specific code.

Bibliography

[1] Auc value. https://www.researchgate.net/post/What_is_the_value_of_the_area_under_the_roc_curve_AUC_to_conclude_that_a_classifier_is_excellent. Accessed: 2018-08-22.

[2] Entropy library. https://pypi.python.org/pypi/entropy/0.9//. Accessed: 2017-03-1.

[3] Shannon entropy. https://en.wiktionary.org/wiki/Shannon_entropy. Accessed: 2017-03-1.

[4] Understand: static code analysis tool. https://scitools.com/feature/. Accessed: 2010-09-30.

[5] Apache software foundation. apache maven project. http://maven.apache.org, 2002. Accessed: 2015-08-17.

[6] iplasma. http://loose.upt.ro/iplasma/, 2009. Accessed: 2019-08-17.

[7] Infusion. http://www.intooitus.com/inFusion.html, 2017. Accessed: 2014-01-01.

[8] Decision trees. https://scikit-learn.org/stable/modules/tree.html, 2019. Accessed: 2019-08-17.

[9] Interpret all statistics and graphs for cross correlation. https://support.minitab.com/en-us/minitab/18/help-and-how-to/modeling-statistics/time-series/how-to/cross-correlation/interpret-the-results/all-statistics-and-graphs/, 2019. Accessed: 2019-08-13.

[10] Logistic regression. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html, 2019. Accessed: 2019-08-17.

[11] Multinomial naive bayes classifier. https://scikit-learn.org/stable/modules/naive_bayes.html, 2019. Accessed: 2019-08-17.

[12] Natural language toolkit, 2019.

[13] Nltk stop words. https://pythonspot.com/nltk-stop-words/, 2019. Accessed: 2019-08-17.

[14] qualtrics. https://www.qualtrics.com/, 2019. Accessed: 2019-08-17.

[15] Support vector machine. https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html, 2019. Accessed: 2019-08-17.

[16] Tfidfvectorizer. https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html, 2019. Accessed: 2020-08-17.

[17] Tiobe index. http://tiobe.com/index.php/content/paperinfo/tpci/index.html, 2019. Accessed: 2019-08-17.

[18] Software quality in 2012: A survey of the state of the art. http://sqgne:org/presentations/2012-13/Jones-Sep-2012:pdf, 2020. Accessed: 2020-06-12.

[19] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *Software maintenance and reengineering (CSMR), 2011 15th European conference on*, pages 181–190. IEEE, 2011.

[20] Iftekhar Ahmed, Caius Brindescu, Umme Ayda Mannan, Carlos Jensen, and Anita Sarma. An empirical examination of the relationship between code smells and merge conflicts. In *Empirical Software Engineering and Measurement (ESEM), 2017 ACM/IEEE International Symposium on*, pages 58–67. IEEE, 2017.

[21] Iftekhar Ahmed, Umme Ayda Mannan, Rahul Gopinath, and Carlos Jensen. An empirical study of design degradation: How software projects get worse over time. In *Empirical Software Engineering and Measurement (ESEM), 2015 ACM/IEEE International Symposium on*, pages 1–10. IEEE, 2015.

[22] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE transactions on software engineering*, 28(10):970–983, 2002.

[23] Sven Apel, Olaf Leßenich, and Christian Lengauer. Structured merge with auto-tuning: balancing precision and performance. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 120–129, 2012.

[24] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. Semistructured merge: rethinking merge in revision control systems. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 190–200, 2011.

[25] Erik Arisholm and Dag IK Sjoberg. Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software. *IEEE Transactions on software engineering*, 30(8):521–534, 2004.

[26] Alberto Bacchelli, Marco D'Ambros, and Michele Lanza. Are popular classes more defect prone? In *International Conference on Fundamental Approaches to Software Engineering*, pages 59–73. Springer, 2010.

[27] Adrian Bachmann and Abraham Bernstein. Software process data quality and characteristics: a historical view on open and closed source projects. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, pages 119–128. ACM, 2009.

[28] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999.

[29] Tibor Bakota, Rudolf Ferenc, and Tibor Gyimothy. Clone smells in software evolution. In *2007 IEEE International Conference on Software Maintenance*, pages 24–33. IEEE, 2007.

[30] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on software engineering*, 22(10):751–761, 1996.

[31] Kwabena Ebo Bennin, Koji Toda, Yasutaka Kamei, Jacky Keung, Akito Monden, and Naoyasu Ubayashi. Empirical evaluation of cross-release effort-aware defect prediction models. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 214–221. IEEE, 2016.

[32] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[33] Abraham Bernstein, Jayalath Ekanayake, and Martin Pinzger. Improving defect prediction using temporal features and non linear models. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, pages 11–18. ACM, 2007.

[34] Nicolas Bettenburg and Ahmed E Hassan. Studying the impact of social structures on software quality. In *2010 IEEE 18th International Conference on Program Comprehension*, pages 124–133. IEEE, 2010.

[35] Alessandro Bianchi, Danilo Caivano, Filippo Lanubile, and Giuseppe Visaggio. Evaluating software degradation through entropy. In *Proceedings Seventh International Software Metrics Symposium*, pages 210–219. IEEE, 2001.

[36] Jacob T Biehl, Mary Czerwinski, Greg Smith, and George G Robertson. Fastdash: a visual dashboard for fostering awareness in software teams. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 1313–1322, 2007.

[37] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 121–130. ACM, 2009.

[38] Christian Bird, Nachiappan Nagappan, Harald Gall, Brendan Murphy, and Premkumar Devanbu. Putting it all together: Using socio-technical networks to predict failures. In *Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on*, pages 109–119. IEEE, 2009.

[39] Barry W Boehm, John R Brown, and Mlity Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, pages 592–605. IEEE Computer Society Press, 1976.

[40] Grady Booch. *Object oriented analysis & design with application*. Pearson Education India, 2006.

[41] Grady Booch, Robert A Maksimchuk, Michael W Engle, Bobbi J Young, Jim Connallen, and Kelli A Houston. Object-oriented analysis and design with applications. *ACM SIGSOFT software engineering notes*, 33(5):29–29, 2008.

[42] David Bowes, Tracy Hall, Mark Harman, Yue Jia, Federica Sarro, and Fan Wu. Mutation-aware fault prediction. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 330–341. ACM, 2016.

[43] Lionel C Briand, William M Thomas, and Christopher J Hetmanski. Modeling and managing risk early in software development. In *Proceedings of 1993 15th International Conference on Software Engineering*, pages 55–65. IEEE, 1993.

[44] Yuriy Brun and Michael D Ernst. Finding latent code errors via machine learning over program executions. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 480–490. IEEE, 2004.

[45] Yuriy Brun, Reid Holmes, Michael D Ernst, and David Notkin. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 168–178, 2011.

[46] João Brunet, Gail C Murphy, Ricardo Terra, Jorge Figueiredo, and Dalton Serey. Do developers discuss design? In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 340–343. ACM, 2014.

[47] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5):309–332, 2005.

[48] Gerardo Canfora, Luigi Cerulo, Marta Cimitile, and Massimiliano Di Penta. How changes affect software entropy: an empirical study. *Empirical Software Engineering*, 19(1):1–38, 2014.

[49] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. Identifying changed source code lines from version repositories. In *MSR*, volume 7, page 14, 2007.

[50] Eugenio Capra, Chiara Francalanci, and Francesco Merlo. An empirical study on the relationship between software design quality, development effort and governance in open source projects. *IEEE Transactions on Software Engineering*, 34(6):765–782, 2008.

[51] Caret. Caret. `https://cran.r-project.org/web/packages/caret/`. Accessed: 2018-03-1.

[52] Michelle Cartwright and Martin Shepperd. An empirical investigation of an object-oriented software system. *IEEE Transactions on software engineering*, 26(8):786–796, 2000.

[53] Marcelo Cataldo and James D Herbsleb. Coordination breakdowns and their impact on development productivity and software failures. *IEEE Transactions on Software Engineering*, 39(3):343–360, 2013.

[54] Saikat Chakraborty, Yujian Li, Matt Irvine, Ripon Saha, and Baishakhi Ray. Entropy guided spectrum based bug localization using statistical language model. *arXiv preprint arXiv:1802.06947*, 2018.

[55] Alexander Chatzigeorgiou and Anastasios Manakos. Investigating the evolution of bad smells in object-oriented code. In *2010 Seventh International Conference on the Quality of Information and Communications Technology*, pages 106–115. IEEE, 2010.

[56] Alexander Chatzigeorgiou and George Stephanides. Entropy as a measure of object-oriented design quality. In *Proceedings of the Balkan Conference in Informatics (BCI)*, pages 565–573, 2003.

[57] Tse-Hsun Chen, Weiyi Shang, Meiyappan Nagappan, Ahmed E Hassan, and Stephen W Thomas. Topic-based software defect explanation. *Journal of Systems and Software*, 129:79–106, 2017.

[58] Mauro Cherubini, Gina Venolia, Rob DeLine, and Andrew J. Ko. Let's go to the whiteboard: How and why software developers use drawings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 557–566. ACM, 2007.

[59] Shyam R Chidamber, David P Darcy, and Chris F Kemerer. Managerial use of metrics for object-oriented software: An exploratory analysis. *IEEE Transactions on software Engineering*, 24(8):629–639, 1998.

[60] Elliot J. Chikofsky and James H Cross. Reverse engineering and design recovery: A taxonomy. *IEEE software*, 7(1):13–17, 1990.

[61] Jacob Cohen, Patricia Cohen, Stephen G West, and Leona S Aiken. *Applied multiple regression/correlation analysis for the behavioral sciences*. Routledge, 2013.

[62] Henry Coles. Pit mutation testing: Mutators. `http://pitest.org/quickstart/mutators`.

[63] Ward Cunningham. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1992.

[64] Isabella A Da Silva, Ping H Chen, Christopher Van der Westhuizen, Roger M Ripley, and André Van Der Hoek. Lighthouse: coordination through emerging design. In *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, pages 11–15, 2006.

[65] Hoa Khanh Dam, Trang Pham, Shien Wee Ng, Truyen Tran, John Grundy, Aditya Ghose, Taeksu Kim, and Chul-Joo Kim. Lessons learned from using a deep tree-based model for software defect prediction in practice. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 46–57. IEEE, 2019.

[66] Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, 2012.

[67] Cleidson RB De Souza, David Redmiles, and Paul Dourish. " breaking the code", moving between private and public work in collaborative software development. In *Proceedings of the 2003 International ACM SIGGROUP conference on Supporting group work*, pages 105–114, 2003.

[68] Lucas Batista Leite De Souza and Marcelo de Almeida Maia. Do software categories impact coupling metrics? In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 217–220. IEEE, 2013.

[69] Ignatios Deligiannis, Martin Shepperd, Manos Roumeliotis, and Ioannis Stamelos. An empirical investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software*, 65(2):127–139, 2003.

[70] Ignatios Deligiannis, Ioannis Stamelos, Lefteris Angelis, Manos Roumeliotis, and Martin Shepperd. A controlled experiment investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software*, 72(2):129–143, 2004.

[71] Prasun Dewan and Rajesh Hegde. Semi-synchronous conflict detection and resolution in asynchronous software development. In *ECSCW 2007*, pages 159–178. Springer, 2007.

[72] Dario Di Nucci, Fabio Palomba, Giuseppe De Rosa, Gabriele Bavota, Rocco Oliveto, and Andrea De Lucia. A developer centered bug prediction model. *IEEE Transactions on Software Engineering*, 44(1):5–24, 2018.

[73] M Doliner et al. Cobertura-a code coverage utility for java.

[74] Paul Dourish and Victoria Bellotti. Awareness and coordination in shared workspaces. In *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, pages 107–114, 1992.

[75] Bart Du Bois, Serge Demeyer, Jan Verelst, Tom Mens, and Marijn Temmerman. Does god class decomposition affect comprehensibility? In *IASTED Conf. on Software Engineering*, pages 346–355, 2006.

[76] F Brito e Abreu and Walcelio Melo. Evaluating the impact of object-oriented design on software quality. In *Proceedings of the 3rd international software metrics symposium*, pages 90–99. IEEE, 1996.

[77] Khaled El Emam, Saïda Benlarbi, Nishith Goel, and Shesh N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering*, 27(7):630–650, 2001.

[78] Andre Eposhi, Willian Oizumi, Alessandro Garcia, Leonardo Sousa, Roberto Oliveira, and Anderson Oliveira. Removal of design problems through refactorings: are we looking at the right symptoms? In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 148–153. IEEE, 2019.

[79] Michael Fagan. Design and code inspections to reduce errors in program development. In *Software pioneers*, pages 575–607. Springer, 2002.

[80] Davide Falessi and Alexander Voegele. Validating and prioritizing quality rules for managing technical debt: An industrial case study. In *Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on*, pages 41–48. IEEE, 2015.

[81] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 313–324, 2014.

[82] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324, New York, NY, USA, 2014. ACM.

[83] Tom Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[84] Vincenzo Ferme, Alessandro Marino, and F Arcelli Fontana. Is it a real code smell to be removed or not? In *International Workshop on Refactoring & Testing (RefTest), co-located event with XP 2013 Conference*, 2013.

[85] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, page 18. ACM, 2016.

[86] Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, and Alessandro Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21(3):1143–1191, 2016.

[87] Francesca Arcelli Fontana, Elia Mariani, Andrea Mornioli, Raul Sormani, and Alberto Tonello. An experience report on using code smells detection tools. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 450–457. IEEE, 2011.

[88] Francesca Arcelli Fontana and Marco Zanoni. On investigating code smells correlations. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 474–475. IEEE, 2011.

[89] Apache Software Foundation. Apache developers' contributors' overview. `http://apache.org/dev/`, 2019. Accessed: 2019-08-17.

[90] Apache Software Foundation. How should i apply patches from a contributor. `http://www.apache.org/dev/committers.html#applying-patches`, 2019. Accessed: 2019-08-17.

[91] Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[92] Peter Freeman and David Hart. A science of design for software-intensive systems. *Communications of the ACM*, 47(8):19–21, 2004.

[93] Mark Gabel and Zhendong Su. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 147–156. ACM, 2010.

[94] Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C Gall. Method-level bug prediction. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 171–180. ACM, 2012.

[95] Emanuel Giger, Martin Pinzger, and Harald C Gall. Comparing fine-grained source code changes and code churn for bug prediction. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 83–92. ACM, 2011.

[96] Andrew R Gilpin. Table for conversion of kendall's tau to spearman's rho within the context of measures of magnitude of effect for meta-analysis. *Educational and psychological measurement*, 53(1):87–92, 1993.

[97] GitHub Inc. Software repository. `http://www.github.com`.

[98] Michael W Godfrey and Lijie Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.

[99] Ian Gorton and Anna Liu. Software component quality assessment in practice: successes and practical impediments. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 555–558. IEEE, 2002.

[100] Todd L Graves, Alan F Karr, James S Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on software engineering*, 26(7):653–661, 2000.

[101] Mário Luís Guimarães and António Rito Silva. Improving early detection of software merge conflicts. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 342–352. IEEE, 2012.

[102] Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software engineering*, 31(10):897–910, 2005.

[103] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2012.

[104] Tracy Hall, Min Zhang, David Bowes, and Yi Sun. Some code smells have a significant but small effect on faults. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):33, 2014.

[105] David J Hand and Robert J Till. A simple generalisation of the area under the roc curve for multiple class classification problems. *Machine learning*, 45(2):171–186, 2001.

[106] Mark Harman, Phil McMinn, Jerffeson Teixeira De Souza, and Shin Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In *Empirical software engineering and verification*, pages 1–59. Springer, 2012.

[107] Ahmed E Hassan. Predicting faults using the complexity of code changes. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 78–88. IEEE, 2009.

[108] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. Bug prediction based on fine-grained module histories. In *2012 34th international conference on software engineering (ICSE)*, pages 200–210. IEEE, 2012.

[109] Lile Hattori and Michele Lanza. Syde: a tool for collaborative software development. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 235–238, 2010.

[110] Winston Haynes. Bonferroni correction. In *Encyclopedia of Systems Biology*, pages 154–154. Springer, 2013.

[111] Peng He, Bing Li, Xiao Liu, Jun Chen, and Yutao Ma. An empirical study on software defect prediction with a simplified metric set. *Information and Software Technology*, 59:170–190, 2015.

[112] Peng He, Bing Li, and Yutao Ma. Towards cross-project defect prediction with imbalanced feature sets. *arXiv preprint arXiv:1411.4228*, 2014.

[113] Zhimin He, Fengdi Shu, Ye Yang, Mingshu Li, and Qing Wang. An investigation on the feasibility of cross-project defect prediction. *Automated Software Engineering*, 19(2):167–199, 2012.

[114] Vincent J Hellendoorn, Premkumar T Devanbu, and Alberto Bacchelli. Will they like this? evaluating code contributions with language models. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 157–167. IEEE, 2015.

[115] David A Hensher and Peter R Stopher. *Behavioural travel modelling*. Taylor and Francis, 1979.

[116] Kim Herzig. Using pre-release test failures to build early post-release defect prediction models. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 300–311. IEEE, 2014.

[117] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 837–847. IEEE, 2012.

[118] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 34–45. IEEE, 2019.

[119] Mario Hozano, Henrique Ferreira, Italo Silva, Baldoino Fonseca, and Evandro Costa. Using developers' feedback to improve code smell detection. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1661–1663. ACM, 2015.

[120] Watts S. Humphrey, Terry R. Snyder, and Ronald R. Willis. Software process improvement at hughes aircraft. *IEEE software*, 8(4):11–23, 1991.

[121] C. Izurieta and J. M. Bieman. How software designs decay: A pilot study of pattern evolution. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pages 449–451, Sep. 2007.

[122] C. Izurieta and J. M. Bieman. Testing consequences of grime buildup in object oriented design patterns. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 171–179, April 2008.

[123] Clemente Izurieta and James M Bieman. How software designs decay: A pilot study of pattern evolution. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pages 449–451. IEEE, 2007.

[124] Clemente Izurieta and James M Bieman. Testing consequences of grime buildup in object oriented design patterns. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 171–179. IEEE, 2008.

[125] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[126] Anton Jansen, Jan Bosch, and Paris Avgeriou. Documenting after the fact: Recovering architectural design decisions. *Journal of Systems and Software*, 81(4):536–557, 2008.

[127] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013.

[128] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 485–495. IEEE, 2009.

[129] Huzefa Kagdi, Michael L Collard, and Jonathan I Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of software maintenance and evolution: Research and practice*, 19(2):77–131, 2007.

[130] Huzefa Kagdi, Malcom Gethers, Denys Poshyvanyk, and Michael L Collard. Blending conceptual and evolutionary couplings to support change impact analysis in source code. In *2010 17th Working Conference on Reverse Engineering*, pages 119–128. IEEE, 2010.

[131] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101. ACM, 2014.

[132] Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E Hassan. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering*, 21(5):2072–2106, 2016.

[133] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2012.

[134] Bakhtiar Khan Kasi and Anita Sarma. Cassandra: Proactive conflict minimization through optimized task scheduling. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 732–741. IEEE, 2013.

[135] Yomi Kastro and Ay

e Basar Bener. A defect prediction method for software versioning. *Software Quality Journal*, 16(4):543–562, 2008.

[136] Mark Kasunic. Designing an effective survey. Technical report, Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 2005.

[137] Michael Keeling and Runde Joe. Architecture decision records in action, 2017.

[138] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. In *2009 16th Working Conference on Reverse Engineering*, pages 75–84. IEEE, 2009.

[139] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering*, 17(3):243–275, 2012.

[140] Miryung Kim, Dongxiang Cai, and Sunghun Kim. An empirical investigation into the role of api-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 151–160. ACM, 2011.

[141] Sunghun Kim and E James Whitehead Jr. How long did it take to fix bugs? In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 173–174. ACM, 2006.

[142] Sunghun Kim, E James Whitehead Jr, and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.

[143] Sunghun Kim, Thomas Zimmermann, Kai Pan, E James Jr, et al. Automatic identification of bug-introducing changes. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 81–90. IEEE, 2006.

[144] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498. IEEE Computer Society, 2007.

[145] Rainer Koschke. Survey of research on software clones. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[146] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *Ieee software*, 29(6):18–21, 2012.

[147] Max Kuhn and Kjell Johnson. *Applied predictive modeling*, volume 26. Springer, 2013.

[148] J Richard Landis and Gary G Koch. An application of hierarchical kappa-type statistics in the assessment of majority agreement among multiple observers. *Biometrics*, pages 363–374, 1977.

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

[150] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 3–13. IEEE, 2012.

[151] Taek Lee, Jaechang Nam, DongGyun Han, Sunghun Kim, and Hoh Peter In. Micro interaction metrics for defect prediction. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 311–321. ACM, 2011.

[152] Meir M Lehman and Juan F Ramil. Software evolution and software evolution processes. *Annals of Software Engineering*, 14(1-4):275–309, 2002.

[153] Meir M Lehman, Juan F Ramil, Paul D Wernick, Dewayne E Perry, and Wladyslaw M Turski. Metrics and laws of software evolution-the nineties view. In *Proceedings Fourth International Software Metrics Symposium*, pages 20–32. IEEE, 1997.

[154] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4):485–496, 2008.

[155] Wei Li and Sallie M Henry. Object-oriented metrics which predict maintainability. 1993.

[156] Wei Li and Raed Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of systems and software*, 80(7):1120–1128, 2007.

[157] Guangtai Liang, Ling Wu, Qian Wu, Qianxiang Wang, Tao Xie, and Hong Mei. Automatic construction of an effective training set for prioritizing static analysis warnings. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 93–102. ACM, 2010.

[158] R Liesenfeld. Jmockit - a developer testing toolkit for java. `http://code.google.com/p/jmockit/`.

[159] Ernst Lippe and Norbert Van Oosterom. Operation-based merging. In *Proceedings of the fifth ACM SIGSOFT symposium on Software development environments*, pages 78–87, 1992.

[160] Jinping Liu, Yuming Zhou, Yibiao Yang, Hongmin Lu, and Baowen Xu. Code churn: A neglected metric in effort-aware just-in-time defect prediction. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 11–19. IEEE, 2017.

[161] Alvi Mahadi, Karan Tongay, and Neil A Ernst. Cross-dataset design discussion mining. 2019.

[162] Umme Ayda Mannan. Companion website. `http://web.engr.oregonstate.edu/~mannanu/entropyeval/index.html`. Accessed: 2020-07-28.

[163] Umme Ayda Mannan, Iftekhar Ahmed, Rana Abdullah M Almurshed, Danny Dig, and Carlos Jensen. Understanding code smells in android applications. In *Proceedings of the International Workshop on Mobile Software Engineering and Systems*, pages 225–234. ACM, 2016.

[164] Mika Mantyla, Jari Vanhanen, and Casper Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pages 381–384. IEEE, 2003.

[165] Cristina Marinescu, Radu Marinescu, Petru Florin Mihancea, Daniel Ratiu, and Richard Wettel. iplasma: An integrated platform for quality assessment of object-oriented design. In *Proceedings of the 21st IEEE International Conference on Software Maintenance - Industrial and Tool volume, ICSM 2005, 25-30 September 2005, Budapest, Hungary*, pages 77–80, 2005.

[166] Radu Marinescu. Detecting design flaws via metrics in object-oriented systems. In *Technology of Object-Oriented Languages and Systems, 2001. TOOLS 39. 39th International Conference and Exhibition on*, pages 173–182. IEEE, 2001.

[167] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 350–359. IEEE, 2004.

[168] Raúl Marticorena, Carlos López, and Yania Crespo. Extending a taxonomy of bad code smells with metrics. In *7th ECCOP International Workshop on Object-Oriented Reengineering (WOOR)*, page 6. Citeseer, 2006.

[169] Robert C Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[170] S. McIntosh. Build system maintenance. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 1167–1169, May 2011.

[171] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. Discipline matters: Refactoring of preprocessor directives in the# ifdef hell. *IEEE Transactions on Software Engineering*, 44(5):453–469, 2017.

[172] Thilo Mende and Rainer Koschke. Effort-aware defect prediction models. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 107–116. IEEE, 2010.

[173] Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 13–23, 2008.

[174] Tom Mens. A state-of-the-art survey on software merging. *IEEE transactions on software engineering*, 28(5):449–462, 2002.

[175] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering*, 33(1):2–13, 2007.

[176] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and AyBener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, 2010.

[177] Andrew V Metcalfe and Paul SP Cowpertwait. *Introductory time series with R*. Springer, 2009.

[178] Subhas C Misra and Virendra C Bhavsar. Relationships between selected software measures and latent bug-density: Guidelines for improving quality. In *International Conference on Computational Science and Its Applications*, pages 724–732. Springer, 2003.

[179] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, July 2002.

[180] Naouel Moha, Jihene Rezgui, Yann-Gaël Guéhéneuc, Petko Valtchev, and Ghizlane El Boussaidi. Using fca to suggest refactorings to correct design defects. In *Concept Lattices and Their Applications*, pages 269–275. Springer, 2008.

[181] Rodrigo Morales, Shane McIntosh, and Foutse Khomh. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 171–180. IEEE, 2015.

[182] Rodrigo Morales, Aminata Sabane, Pooya Musavi, Foutse Khomh, Francisco Chicano, and Giuliano Antoniol. Finding the best compromise between design quality and testing effort during refactoring. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 24–35. IEEE, 2016.

[183] Hausi A Müller and Karl Klashinsky. Rigi-a system for programming-in-the-large. In *Proceedings of the 10th international conference on Software engineering*, pages 80–86. IEEE Computer Society Press, 1988.

[184] Emerson Murphy-Hill and Andrew P Black. An interactive ambient visualization for code smells. In *Proceedings of the 5th international symposium on Software visualization*, pages 5–14. ACM, 2010.

[185] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 284–292. ACM, 2005.

[186] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy. Change bursts as defect predictors. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 309–318. IEEE, 2010.

[187] Jaechang Nam, Wei Fu, Sunghun Kim, Tim Menzies, and Lin Tan. Heterogeneous defect prediction. *IEEE Transactions on Software Engineering*, 44(9):874–896, 2017.

[188] Jaechang Nam and Sunghun Kim. Clami: Defect prediction on unlabeled datasets (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 452–463. IEEE, 2015.

[189] Antti Nieminen. Real-time collaborative resolving of merge conflicts. In *8th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, pages 540–543. IEEE, 2012.

[190] John Noll, Sarah Beecham, and Dominik Seichter. A qualitative study of open source software development: The open emr project. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 30–39. IEEE, 2011.

[191] Hector M Olague, Letha H Etzkorn, and Glenn W Cox. An entropy-based approach to assessing object-oriented software maintainability and degradation-a method and case study. In *Software Engineering Research and Practice*, pages 442–452, 2006.

[192] Steffen Olbrich, Daniela S Cruzes, Victor Basili, and Nico Zazworka. 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*, pages 390–400. IEEE Computer Society, 2009.

[193] Gustavo Ansaldi Oliva, Igor Steinmacher, Igor Wiese, and Marco Aurélio Gerosa. What can commit metadata tell us about design degradation? In *Proceedings of the 2013 International Workshop on Principles of Software Evolution*, pages 18–27. ACM, 2013.

[194] Thomas J Ostrand, Elaine J Weyuker, and Robert M Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[195] Thanis Paiva, Amanda Damasceno, Eduardo Figueiredo, and Cláudio Sant'Anna. On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development*, 5(1):7, 2017.

[196] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. Do they really smell bad? a study on developers' perception of bad code smells. In *Software maintenance and evolution (ICSME), 2014 IEEE international conference on*, pages 101–110. IEEE, 2014.

[197] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. Detecting bad smells in source code using change history information. In *Automated software engineering (ASE), 2013 IEEE/ACM 28th international conference on*, pages 268–278. IEEE, 2013.

[198] Fabio Palomba, Marco Zanoni, Francesca Arcelli Fontana, Andrea De Lucia, and Rocco Oliveto. Smells like teen spirit improving bug prediction performance using the intensity of code smells. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 244–255. IEEE, 2016.

[199] Annibale Panichella, Carol V Alexandru, Sebastiano Panichella, Alberto Bacchelli, and Harald C Gall. A search-based training algorithm for cost-aware defect prediction. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 1077–1084, 2016.

[200] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. Fine-grained just-in-time defect prediction. *Journal of Systems and Software*, 150:22–36, 2019.

[201] Luca Pascarella, Davide Spadini, Fabio Palomba, and Alberto Bacchelli. On the effect of code review on code smells. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 12 2020.

[202] Leonardo Passos, Rodrigo Queiroz, Mukelabai Mukelabai, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Jesus Padilla. A study of feature scattering in the linux kernel. *IEEE Transactions on Software Engineering*, 2018.

[203] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.

[204] Michael J Pencina, Ralph B D'Agostino, and Joseph M Massaro. Understanding increments in model performance metrics. *Lifetime data analysis*, 19(2):202–218, 2013.

[205] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. Can developer-module networks predict failures? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 2–12. ACM, 2008.

[206] C Lakshmi Prabha and N Shivakumar. x. In *2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS)*, pages 583–588. IEEE, 2020.

[207] Foyzur Rahman and Premkumar Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 491–500. ACM, 2011.

[208] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 432–441. IEEE, 2013.

[209] Gopi Krishnan Rajbahadur, Shaowei Wang, Yasutaka Kamei, and Ahmed E Hassan. The impact of using regression models to build defect classifiers. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 135–145. IEEE, 2017.

[210] Paul Ralph and Yair Wand. A proposal for a formal definition of the design concept. In *Design requirements engineering: A ten-year perspective*, pages 103–136. Springer, 2009.

[211] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the naturalness of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 428–439. ACM, 2016.

[212] Nornadiah Mohd Razali, Yap Bee Wah, et al. Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests. *Journal of statistical modeling and analytics*, 2(1):21–33, 2011.

[213] Alfonso Rojas-Domínguez, Luis Carlos Padierna, Juan Martín Carpio Valadez, Hector J Puga-Soberanes, and Héctor J Fraire. Optimal hyper-parameter tuning of svm classifiers with application to medical diagnosis. *IEEE Access*, 6:7164–7176, 2018.

[214] Daniele Romano and Martin Pinzger. Using source code metrics to predict change-prone java interfaces. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 303–312. IEEE, 2011.

[215] Bilyaminu Auwal Romo, Andrea Capiluppi, and Tracy Hall. Filling the gaps of development logs and bug issue data. 2014.

[216] Vlad Roubtsov et al. Emma: a free java code coverage tool. `http://emma.sourceforge.net/`.

[217] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68, 2007.

[218] Geoffrey G Roy and Valerie E Veraart. Software engineering education: from an engineering perspective. In *Proceedings 1996 International Conference Software Engineering: Education and Practice*, pages 256–262. IEEE, 1996.

[219] Anita Sarma, Zahra Noroozi, and André Van Der Hoek. Palantír: raising awareness among configuration management workspaces. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 444–454. IEEE, 2003.

[220] Anita Sarma and Andre Van Der Hoek. Towards awareness in the large. In *2006 IEEE International Conference on Global Software Engineering (ICGSE'06)*, pages 127–131. IEEE, 2006.

[221] Patrick Schratz, Jannes Muenchow, Jakob Richter, and Alexander Brenning. Performance evaluation and hyperparameter tuning of statistical and machine-learning models using spatial data. *arXiv preprint arXiv:1803.11266*, 2018.

[222] Jan Schumacher, Nico Zazworka, Forrest Shull, Carolyn Seaman, and Michele Shaw. Building empirical support for automated code smell detection. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 8. ACM, 2010.

[223] scikit learn. `http://scikit-learn.org/stable/`, 2019. Accessed: 2019-08-1.

[224] Ravi Sen, Chandrasekar Subramaniam, and Matthew L Nelson. Open source software licenses: Strong-copyleft, non-copyleft, or somewhere in between? *Decision support systems*, 52(1):199–206, 2011.

[225] Francisco Servant, James A Jones, and André Van Der Hoek. Casi: preventing indirect conflicts through a live visualization. In *Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering*, pages 39–46, 2010.

[226] Arman Shahbazian, Youn Kyu Lee, Duc Le, Yuriy Brun, and Nenad Medvidovic. Recovering architectural design decisions. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 95–9509. IEEE, 2018.

[227] Claude Elwood Shannon. A mathematical theory of communication. *ACM SIGMOBILE mobile computing and communications review*, 5(1):3–55, 2001.

[228] Tushar Sharma, Pratibha Mishra, and Rohit Tiwari. Designite: A software design quality assessment tool. In *Proceedings of the 1st International Workshop*

on Bringing Architectural Design Thinking into Developers' Daily Activities, pages 1–4, 2016.

[229] Emad Shihab, Audris Mockus, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. High-impact defects: a study of breakage and surprise defects. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 300–310. ACM, 2011.

[230] Forrest Shull, Vic Basili, Barry Boehm, A Winsor Brown, Patricia Costa, Mikael Lindvall, Daniel Port, Ioana Rus, Roseanne Tesoriero, and Marvin Zelkowitz. What we have learned about fighting defects. In *Proceedings eighth IEEE symposium on software metrics*, pages 249–258. IEEE, 2002.

[231] Dag IK Sjøberg, Aiko Yamashita, Bente CD Anda, Audris Mockus, and Tore Dybå. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156, 2012.

[232] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.

[233] T. J. Smiley. The uses of argument. by s. e. toulmin, professor of philosophy, university of leeds. [cambridge: at the university press. 1958. vii, 261 and (index) 2 pp. 22s. 6d. net.]. *The Cambridge Law Journal*, 16(2):251–252, 1958.

[234] Connie U Smith and Lloyd G Williams. Software performance antipatterns. In *Proceedings of the 2nd international workshop on Software and performance*, pages 127–136, 2000.

[235] Greg Snider. Measuring the entropy of large software systems. *HP Laboratories Palo Alto, Tech. Rep*, 2001.

[236] Adriana Meza Soria and André van der Hoek. Collecting design knowledge through voice notes. In *Proceedings of the 12th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 33–36. IEEE Press, 2019.

[237] Leonardo Sousa, Anderson Oliveira, Willian Oizumi, Simone Barbosa, Alessandro Garcia, Jaejoon Lee, Marcos Kalinowski, Rafael de Mello, Baldoino Fonseca, Roberto Oliveira, et al. Identifying design problems in the source code: A grounded theory. In *Proceedings of the 40th International Conference on Software Engineering*, pages 921–931, 2018.

[238] Leonardo Sousa, Roberto Oliveira, Alessandro Garcia, Jaejoon Lee, Tayana Conte, Willian Oizumi, Rafael de Mello, Adriana Lopes, Natasha Valentim, Edson Oliveira,

et al. How do software developers identify design problems? a qualitative analysis. In *Proceedings of the 31st Brazilian Symposium on Software Engineering*, pages 54–63, 2017.

[239] Dietmar Stöckl, Katy Dewitte, and Linda M Thienpont. Validity of linear regression in method comparison studies: is it limited by the statistical model or the quality of the analytical input data? *Clinical Chemistry*, 44(11):2340–2346, 1998.

[240] Seyyed Ehsan Salamati Taba, Foutse Khomh, Ying Zou, Ahmed E Hassan, and Meiyappan Nagappan. Predicting bugs using antipatterns. In *2013 IEEE International Conference on Software Maintenance*, pages 270–279. IEEE, 2013.

[241] Ladan Tahvildar and Kostas Kontogiannis. Improving design quality using meta-pattern transformations: a metric-based approach. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(4-5):331–361, 2004.

[242] Davide Taibi, Andrea Janes, and Valentina Lenarduzzi. How developers perceive smells in source code: A replicated study. *Information and Software Technology*, 92:223–235, 2017.

[243] Yong Tan and Vijay S Mookerjee. Comparing uniform and flexible policies for software maintenance and replacement. *IEEE Transactions on Software Engineering*, 31(3):238–255, 2005.

[244] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 43(1):1–18, 2016.

[245] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 141–150. IEEE, 2015.

[246] Yuan Tian, Julia Lawall, and David Lo. Identifying linux bug fixing patches. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 386–396. IEEE, 2012.

[247] Yuan Tian, Julia Lawall, and David Lo. Identifying linux bug fixing patches. In *Software Engineering (ICSE), 2012 34th International Conference On*, pages 386–396, 2012.

[248] Burak Turhan, Tim Menzies, Ayse B Bener, and Justin Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 14(5):540–578, 2009.

[249] Jilles Van Gurp and Jan Bosch. Design erosion: problems and causes. *Journal of systems and software*, 61(2):105–119, 2002.

[250] Giovanni Viviani, Michalis Famelis, Xin Xia, Calahan Janik-Jones, and Gail C Murphy. Locating latent design information in developer discussions: A study on pull requests. *IEEE Transactions on Software Engineering*, 2019.

[251] Giovanni Viviani, Calahan Janik-Jones, Michalis Famelis, Xin Xia, and Gail C Murphy. What design topics do developers discuss? In *Proceedings of the 26th Conference on Program Comprehension*, pages 328–331. ACM, 2018.

[252] Romi Satria Wahono. A systematic literature review of software defect prediction: research trends, datasets, methods and frameworks. *Journal of Software Engineering*, 1(1):1–16, 2015.

[253] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 297–308. IEEE, 2016.

[254] Companion website. Companion website. `https://fse2020.wixsite.com/designdiscussion`. Accessed: 2018-05-20.

[255] Elaine J Weyuker, Thomas J Ostrand, and Robert M Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Software Engineering*, 13(5):539–559, 2008.

[256] Jan Wloka, Barbara Ryder, Frank Tip, and Xiaoxia Ren. Safe-commit analysis to facilitate team software development. In *2009 IEEE 31st International Conference on Software Engineering*, pages 507–517. IEEE, 2009.

[257] Aiko Fallas Yamashita and Leon Moonen. Do developers care about code smells? an exploratory survey. In *WCRE*, volume 13, pages 242–251, 2013.

[258] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 17–26. IEEE, 2015.

[259] Yibiao Yang, Mark Harman, Jens Krinke, Syed Islam, David Binkley, Yuming Zhou, and Baowen Xu. An empirical study on dependence clusters for effort-aware fault-proneness prediction. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 296–307. IEEE, 2016.

[260] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 157–168, 2016.

[261] Yibiao Yang, Yuming Zhou, Hongmin Lu, Lin Chen, Zhenyu Chen, Baowen Xu, Hareton Leung, and Zhenyu Zhang. Are slice-based cohesion metrics actually useful in effort-aware post-release fault-proneness prediction? an empirical study. *IEEE Transactions on Software Engineering*, 41(4):331–357, 2014.

[262] Tingting Yu, Wei Wen, Xue Han, and Jane Huffman Hayes. Conpredictor: concurrency defect prediction in real-world applications. *IEEE Transactions on Software Engineering*, 45(6):558–575, 2018.

[263] Yong Yu, Tong Li, Na Zhao, and Fei Dai. An approach to measuring the component cohesion based on structure entropy. In *2008 Second International Symposium on Intelligent Information Technology Application*, volume 3, pages 697–700. IEEE, 2008.

[264] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. Automatically recommending peer reviewers in modern code review. *IEEE Transactions on Software Engineering*, 42(6):530–543, 2015.

[265] Nico Zazworka, Michele A Shaw, Forrest Shull, and Carolyn Seaman. Investigating the impact of design debt on software quality. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, pages 17–23. ACM, 2011.

[266] Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. Towards building a universal defect prediction model. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 182–191. ACM, 2014.

[267] Feng Zhang, Quan Zheng, Ying Zou, and Ahmed E Hassan. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 309–320. IEEE, 2016.

[268] Hongyu Zhang. An investigation of the relationships between lines of code and defects. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 274–283. IEEE, 2009.

[269] Min Zhang, Tracy Hall, and Nathan Baddoo. Code bad smells: a review of current knowledge. *Journal of Software Maintenance and Evolution: research and practice*, 23(3):179–202, 2011.

[270] Weiqiang Zhang, Shing-Chi Cheung, Zhenyu Chen, Yuming Zhou, and Bin Luo. File-level socio-technical congruence and its relationship with bug proneness in oss projects. *Journal of Systems and Software*, 156:21–40, 2019.

[271] Xian Zhang, Kerong Ben, and Jie Zeng. Cross-entropy: A new metric for software defect prediction. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 111–122. IEEE, 2018.

[272] Minhaz F Zibran, Farjana Z Eishita, and Chanchal K Roy. Useful, but usable? factors affecting the usability of apis. In *2011 18th Working Conference on Reverse Engineering*, pages 151–155. IEEE, 2011.

[273] Thomas Zimmermann, Sunghun Kim, Andreas Zeller, and E James Whitehead Jr. Mining version archives for co-changed lines. In *MSR*, volume 6, pages 72–75, 2006.

[274] Thomas Zimmermann and Nachiappan Nagappan. Predicting subsystem failures using dependency graph complexities. In *Software Reliability, 2007. ISSRE'07. The 18th IEEE International Symposium on*, pages 227–236. IEEE, 2007.

[275] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 531–540. IEEE, 2008.

[276] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 91–100, 2009.