An Exploratory Study of Design Discussions that Occur During Peer Code Review

Farida El Zanaty

Master of Engineering

Department of Electrical and Computer Engineering

McGill University

Montreal,Quebec

October, 2019

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Engineering

TO MY ANGEL MOTHER,
whose wish was to become a software engineer, and then gave birth to two.

ACKNOWLEDGEMENTS

First and foremost, I want to thank Dr. Shane McIntosh. Thank you, wholeheartedly and genuinely, for your mentorship, and for always leading by example. Thank you for teaching me about work ethics, integrity and discipline, as much as you have taught me about software engineering research. I am honestly grateful.
Thank you, Dr. Jin L.C. Guo, for taking the time to examine this thesis. I am inspired by you for being a successful female professor in a male-dominated software community.
I extend my gratitude to Al Ghurair Foundation, for believing in me. Your scholarship has provided me with a degree, and life experience. It is my stepping stone to make a difference in the world, and fulfill your vision.
Thank you, Toshiki Hirao, for your hard work in the foundational work of this thesis, and your friendship. Thank you to Kehilya Gallaba, Shivashree Vysali, Noam Rabbani, Christophe Rezk and Ray Wen, my Software Rebels family.
Mille merci, Farid Namek, mon ami et mon cousin, pour ton effort de traduire mon abstract.
This thesis would not have been possible without all of you.

To my Family.
Thank you, Dawdy, whom without, my homesickness would have gotten the best of me. You are my home, and this degree is yours as much as it is mine.
Thank you, papa, for being my best friend, and for living with me some of our toughest days.
Thank you, Moustafa, my little brother, for teaching me strength, and love. Thank you for being the kindest person I know.
And most importantly, thank you, mama, for investing in me your lifetime, and doing everything in your capacity to give me the best possible education, since day one. I hope I pay back a fraction of the constant love and support you surround me with, by making you proud. I hope you feel like all the time, money and effort you invested in me, did not go in vain.

Zero thanks to Canadian winters, which have certainly made this degree unnecessarily harder to achieve. As a result, I have died a little on the inside.

ABSTRACT

Code review is a well-established software quality practice where developers critique each others' changes. A shift towards automated detection of low-level issues (e.g., integration with linters) has, hypothetically, freed reviewers up to focus on higher level issues, such as software design. Yet in practice, little is known about the extent to which design is discussed during code review. To bridge this gap, in this thesis, we set out to study the frequency and type of design discussions in code reviews. We first perform an empirical study on the code reviews of the OpenStack Nova (provisioning management) and Neutron (networking abstraction) projects. We manually classify 2,817 review comments from a randomly selected sample of 220 code reviews. We then train and evaluate classifiers to automatically label review comments as design-related or not. Finally, we apply the classifiers to a larger sample of 2,506,308 review comments to study the characteristics of reviews that include design discussions. Our manual analysis indicates that (1) design discussions are still quite rare, with only 9% and 14% of Nova and Neutron review comments being related to software design, respectively; and (2) design feedback is often constructive, with 73% of the design-related comments also providing suggestions to address the concerns. Furthermore, code changes that have design-related feedback have a statistically significantly increased rate of abandonment (Pearson χ^2 test, DF=1, $p < 0.001$). To further explore the phenomenon, we survey 94 software developers and triangulate the responses with respect to our quantitative and qualitative results. The results indicate that 1) discussing design is indeed perceived to

be an important topic in code review, which benefits the author/reviewer (peer mentorship), the development team (collaborative problem solving, change awareness), and the end-product (improved code quality); and that 2) software developers in an industry setting prioritize the performance of the system over other design concerns. Given the relative sparseness of design discussions during code review and its role as a primary motivation for conducting code review, more may need to be done to foster such discussions among contributors.

RÉSUMÉ

L'audit de code est une pratique de qualité de logiciel bien établie où les développeurs critiquent leurs propre modifications. Un changement vers une detection automatisée des problémes de niveau faible (ex: integration avec lint) a, en théorie, permis aux développeurs de se concentrer sur les problémes de plus haut niveau, tel que le désign du logiciel. Cependant, en pratique, peu est connu quant à létendue de la discussion au sujet du désign lors de l'audit du code. Pour combler ce fossé, dans la dissertation, nous nous sommes lancés dans létude de la fréquence et le type de l'audit du désign dans le cadre de l'audit du code. Nous effectuons une étude empirique sur les audits de code des projets OpenStack Nova (gestion de l'approvisionnement) et Neutron (abstraction de réseau). Nous classifions manuellement 2,817 commentaires d'audit d'un échantillon aléatoire de 220 audits de code. Nous procédons alors à entrainer et évaluer les classificateurs à automatiquement étiqueter les commentaires d'audit comme étant relatif au désign ou non. Finalement, nous appliquons les classificateurs à un échantillon plus large de 2,508,308 commentaires d'audit pour étudier les caractéristiques des audits qui incluent une discussion du désign. Notre analyse manuelle indique que (1) les discussions du désign sont encore assez rares, avec seulement 9% et 14% des commentaire d'audit de Nova et Neutron respectivement, qui sont en rapport avec le désign du logiciel; et (2) les commentaires du désign sont souvent constructifs, avec 73% des commentaires en rapport avec le désign qui fournissent des propositions pour addresser les préoccupations. De plus, les changement de code qui contiennent des commentaires relatifs au désign ont, statistiquement,

un taux d'abandon plus élevé. Afin de trianguler nos résultats, nous avons effectués un sondage sur 94 développeurs de logiciels pour nous aider à renforcer nos observations que 1) discuter du désign est en effet important, puisque c'est avantageux à l'individu, l'équipe de développement, et le produit final, et que 2) les développeurs de logiciels, dans le cadre professionnel, priorisent la performance du système, par dessus toutes autres inquiétudes quant au désign. Puisque la discussion du désign est la motivation principale pour la conduite de l'audit de code, il faudrait en faire plus pour encourager telles discutions entre les contributeurs.

Related Publications

Earlier version of the work in this thesis was published as listed below:

- *An Empirical Study of Design Discussions in Code Review.* Farida El Zanaty, Toshiki Hirao, Akinori Ihara, Kenichi Matsumoto, and Shane McIntosh. To appear in Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, p. 11:1–11:10 ACM (ESEM 2018)

The following work is not directly related to the content of this thesis, but was produced in parallel to the research performed for this thesis.

- *Automatic Recovery of Missing Issue Type Labels.* Farida El Zanaty, Christophe Rezk, Sander Lijbrink, Willem van Bergen, Mark Cote, and Shane McIntosh. Submitted to IEEE Software (status: under review).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
Introduction

Code reviewing is a broadly adopted practice where one developer (the author) makes a change to the code, and other developers (the reviewers) critique the premise, content, and structure of the change [17]. Code review practices are realized in a variety of ways, such as formal code inspections [43], informal walkthroughs [57], lightweight tool-based reviews [3], or checklist-based reviews [45].

Since code review is expensive in terms of human effort [5], recent code review systems are shifting towards automated detection of low-level issues. For example, the Review Board project[1] incorporates linting and static code analysis results directly in the code review interface [5]. This trend towards automation of menial tasks, has, hypothetically, freed reviewers up to focus on higher-level issues, such as design.

Software design plays a central role in the success of the produced software [53]. As a result, design decisions are essential during development, since design decisions are linked to the maintainability and extensibility of a codebase [19, 27, 11]. From an internal perspective, better design has been linked to lower rates of restructuring, better understandability, reduced coding effort and faster reworks of the system

[1] `https://www.reviewboard.org`

[19, 18, 53]. Moreover, from an external perspective, design quality has been linked to the quality of the end-product, both quantitatively and qualitatively [53, 15].

1.1 Problem Statement

Although better design is associated with better software quality, little is known about the type and frequency of design discussions during code review.

> ***Thesis statement:*** Hypothetically, the broader role that automation has taken in the code review processes has freed up human review participants to focus on higher-level concepts, such as design. Archives of code review data can be leveraged to better understand the prevalence and characteristics of design discussions that occur during the code review process.

More specifically, it is unclear how design discussions take place and what type of design issues are developers concerned about. Hence, in this thesis, we set out to study the frequency, type and characteristics of the design discussions that take place during code review.

1.2 Thesis Overview

Figure 1.1 provides an overview of the scope of this thesis. We first provide the necessary background material (yellow boxes):

Chapter 2: Before delving into the study of design discussions that occur during code review, we first provide the reader with background information, and define key terms and processes that we will use throughout the thesis.

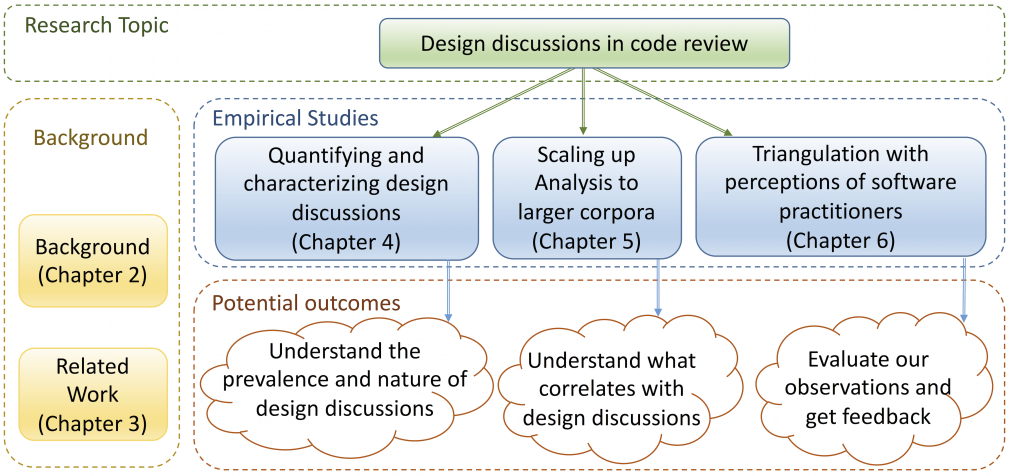

Figure 1–1: An overview of the scope of this thesis.

Chapter 3: In order to situate this thesis with respect to prior research, we present a survey of research on code review and software design.

Next, we shift our focus to the main body of the thesis. In this thesis, we analyze design discussions using three research methodologies (blue boxes), motivated by their potential outcomes (orange clouds).

Chapter 4: In order to better understand the *prevalence* and *type* of design discussions, we manually analyze 2,817 code review comments from two open source software repositories. We answer the following two research questions:

(RQ1)How often is design discussed during code review?

(RQ2)Which design issues are discussed most often?

Chapter 5: While we could derive rich insights from our manual analysis, it does not scale well to larger samples of data. Thus, we explore the applicability of automated approaches to the labeling of code review comments as design-related or not. We first train and evaluate models using supervised learning

approaches, and then apply them to 2,506,308 review comments. This larger sample allows us to explore the relationship (or lack thereof) between design discussions and other characteristics of code review. We answer the following two research questions:

(RQ3)How accurately can our classifiers identify design-related comments?

(RQ4)Is design-related discussion correlated with the characteristics of code changes?

Chapter 6: We evaluate the observations of prior chapters by surveying 94 software developers who regularly participate in code reviews in industrial settings. The survey responses yield further insight into the perceptions of developers about design discussions that occur during code reviews.

1.3 Thesis Contributions

This thesis demonstrates that:

- Design discussions rarely occur during code review, with only 9%–14% sampled reviews containing some form of design feedback (Chapter 4).
- When design discussions do occur, they are typically *uni-directional*, with feedback flowing from reviewers to authors; however, the feedback is often *constructive*, including suggestions to address the identified design flaws (Chapter 4).
- Design issues most often arise due to a lack of global context of the codebase (Chapter 4).
- The rate at which reviewers provide design feedback correlates with the reviewer's (project-specific) experience (Chapter 5).

- Most survey participants perceive that design is an important topic to be discussed during code reviews, highlighting benefits to the author (e.g., peer mentorship), the team (e.g., change awareness), and the product (e.g., improvements to the code quality) (Chapter 6).
- Survey participants tended to prioritize properties that impact end-user experience with the system (e.g., performance) over the other design issues (Chapter 6).

1.4 Thesis Organization

The remainder of this paper is organized as follows: Chapter 2 provides background information, while Chapter 3 surveys previous research related to software design and code reviews. Chapter 4 describes our qualitative study that quantifies and characterizes design feedback, while Chapter 5 discusses the large-scale analysis that followed. Chapter 6 triangulates our observations from prior chapters with respect to the perceptions of software practitioners. Chapter 7 discloses the threats to the validity of our study. Lastly, Chapter 8 draws conclusions and discusses promising directions for future research.

CHAPTER 2
Background

2.1 What is software design?

Software design is a term that as been re-defined numerous times. Since a crisp definition is needed to our work, we need to select an appropriate definition. We chose to adopt Brunet et al.'s [11] definition, which states that:

> *"As an artifact, design is a representation of how a portion of the code should be organized. As an activity, design is the process of discussing the structure of the code to organize abstractions and their relationships."*

Design is an important aspect of the software development process, especially in large systems. However, software systems are constantly evolving, and are rewritten several times during their lifetime [42]. This makes evaluating a software system's design a challenging issue. Previous work shows that, in practice, design is mostly based on subjective opinions, rather than objective metrics and/or systematic assessment [28]. Fowler and Beck [7] even claim that when detecting bad smells in code, *"no set of metrics rivals informed human intuition."*

Prior work has shown that software design is linked to the quality of the software [15, 35]. Moreover, *design debt*, i.e., compromises to best design practices taken for practical reasons, negatively affects the quality of the software and wastes development time and effort [58].

Indeed, software design is subjective to some degree. We conjecture that the subjectivity of software design needs to be resolved in an equitable manner to achieve a cohesive software team dynamic. Moreover, compromises to design are generally perceived to negatively impact software quality (e.g., technical debt). Hence, we expect that code review would be an ideal place to discuss design implications of evolving software systems.

2.2 What is code review?

'Code review' is a pillar of modern software quality assurance practices. It typically occurs before a code change is integrated to the codebase. Participants in the review process take an author, reviewer, and verifier roles. The author is the party responsible for creating the code changes under review and typically have a stake in seeing the code changes through to integration with the central code repository. Verifiers assess whether code changes meet concrete quality criteria. While in theory, verifiers may be human participants, in practice, the role is typically filled by automatic bots. Reviewers are responsible for critiquing the premise, form, and content of the code changes under review.

Since appearing in the 1970's [16], code reviews have taken on several formats. Early variants [16] empathized process rigour, including checklists for reviewers, and a structured meeting process with additional participants in administrative roles, such as moderators, scribes, and readers. Recently, a more lightweight variant as well as the more lightweight, tool-supported, asynchronous variant [44] has become almost ubiquitous, in part due to the rise of globally distributed software development teams.

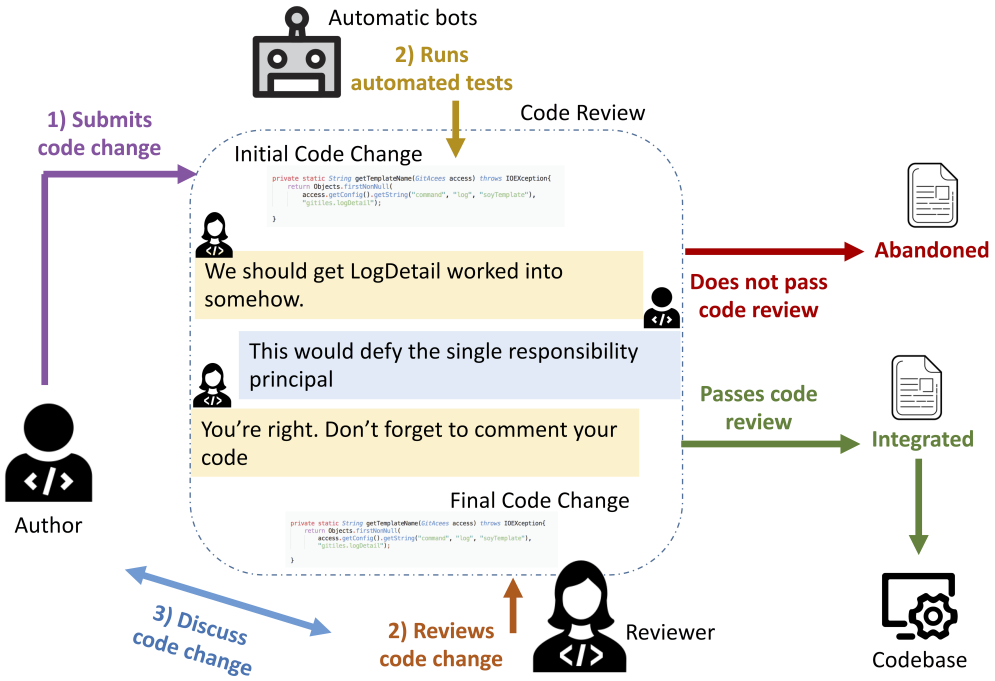

Figure 2–1: The workflow of a typical code review

2.3 Motivational Example

In this section, we present a motivational example to highlight how the automatic bots freed time for human reviewers to discuss code organization and design.

Figure 2–1 provides a concrete example of a code review, an author first submits their code change (Step 1), which then enters the 'code review' phase. It is then simultaneously reviewed by a human author, and analyzed by automated bots (Step 2). For example, Figure 2-4 shows a software developer, Mo, who submits a new code change to the system. Mary, a more experienced software developer, gets a notification to review Mo's code change. When Mary checks the code change to review it, she realizes that the automatic bots of the system have already started running all

test suite against it. They have also checked code conventions and compliance with their project's coding standards using linters. She figures that the automatic bots are more accurate and conclusive at checking for functional errors by their exhaustive and systematic test cases, so she checks the design of the code change with respect to the whole system. Mary, as a reviewer, leaves inline comments to Mo, asking him to move method `convertData` to the class `DataHandling`, to promote cohesion, and reduce unnecessary coupling. She also points out that the `retrieveData` method is redundant, as another method by the name `getData` already exists. This initiates a discussion between the author and reviewer (Step 3), which allows the author to address the received feedback and resubmit code changes with incorporated suggested changes (i.e., Steps 1–3 can be repeated until code change passes review). Mo, as the author of the code change, first and foremost wants his change to be accepted and successfully integrated to the codebase. He replies back to the inline comments, and resubmits another version of the code change with all of Mary's feedback addressed. Mary then approves the code change, which has also passed all the tests and evaluations run by the automatic bots. The code change is then successfully integrated into the codebase.

CHAPTER 3
Related Work

In this chapter, we present the related work with respect to code review practice, design quality assurance, and code review characteristics dimensions.

3.1 Code Review Practice

Code review is a common practice that is introduced to improve the code quality [4]. Code reviews add value to a software organization by detecting bugs early, increase productivity, and can even improve project documentation [20]. Several studies set out to explore the factors of code reviews that affect software quality. For instance, reviewer involvement has been linked with incidence rates of post-release defects [30, 31, 49], defect proneness [25], design anti-patterns [35], and security vulnerabilities [32]. Moreover, the documentation level of code reviews has been linked with the level of maintainability of the produced code [37, 8].

Code reviews have a much broader scope [2, 6] than only improving software quality. Czerwonka et al. [14] found that code reviews often do not find functionality issues that should block a code submission. Bacchelli and Bird [3] found that code reviews at Microsoft aim to not only fix bugs, but also transfer knowledge among team members. Mäntylä and Lassenius [29] found that there are three evolvability issues raised for every functionality issue during code reviews. Beller et al. [8] found a similar ratio of evolvability and functional issues are fixed during code reviews. These findings motivated the search for design discussions in code reviews, which

are also considered non-functional issues. This thesis strives to delve into software design—a non-functional aspect of review discussion.

Better design has been linked to lower rates of restructuring, better understandability, reduced coding effort, and faster reworks of the system [18, 19, 53]. While prior work has analyzed aspects of code review related to code quality [25, 30, 31, 35, 49], little is known about the prevalence and type of design discussion in code reviews. To bridge this gap, this thesis focuses on feedback content by analyzing how often reviews discuss design issues and what kinds of design issues are raised.

3.2 Design Quality Assurance

Software design is essential to ensure the quality of the software product [11, 19, 27]. For instance, Kemerer and Paulk [24] showed that adding design reviews to projects at the SEI led to software products that were of higher quality. Brunet et al. [11] found that only 25% of discussions are design-related in commits, issues, and pull requests. Our findings in Chapter 4 complements the prior work, indicating a minority (i.e., only 9% and 14% of comments) are design-related in OpenStack Nova and Neutron projects.

Design-related discussions may broach various subjects. Tao et al. [56] found that code changes that comprise design issues like suboptimal solutions and incomplete fixes (e.g., "Shallow Fix") are often linked with the rejection of the code changes in the Eclipse and Mozilla projects. Our findings in Chapter 5 also suggest that, in general, design issues present in code reviews, including "Shallow Fix", are associated with the abandonment of code changes in the OpenStack Nova and Neutron.

Sedano et al. [47] found that duplicated work like "Redundant Code" and "Unnecessary Complexity" is a frequently occurring type of software development waste. Our results in Chapter 5 show that design-related comments are raised due to not only "Redundant Code" and "Unnecessary Complexity", but also "Module Coupling", "Performance", "Side Effect", "Shallow Fix", and "Code Misplacement". Yamashita and Moonen [55] found that 32% of developers are not aware of the code smells in their own code. Paixao et al. [39] found that 62% of the time, developers do not discuss the architectural impact of their changes on the system, suggesting a lack of awareness of them. Our findings in Chapter 5 also suggest that a lack of awareness of the global context of the codebase is linked to incidence rates of design comments.

3.3 Code Review Characteristics

Code review characteristics have been analyzed to understand their relationship with integration decisions. Tsay et al. [52] showed that code changes that attract many comments are less likely to be accepted. Indeed, McIntosh et al. [30, 31] and Thongtanunam et al. [49] have argued that the amount of discussion that was generated during review should be considered when making integration decisions. Gousios et al. [21] found that only 13% of pull requests are rejected due to technical reasons. Our findings in Chapter 5 suggest that design-related reviews are likely to struggle with integration.

Prior work showed the importance of author expertise in code reviews. Bosu et al. [10] found that module-based expertise shares a link with the perceived usefulness of a code review (as expressed by authors of the code changes). Meneely et al. [33]

examined the association between the number of commits of developers and security problems in the Red Hat Enterprise Linux 4 kernel. Bosu and Carver [9] found that code changes written by inexperienced authors tend to receive little review participation. Our findings in Chapter 5 suggest that authors of code changes that receive design-related code review feedback tend to have less project-specific expertise than the authors of code changes that do not receive design-related feedback. However, in those design-related reviews, reviewers often provide constructive feedback, which includes suggestions to help inexperienced authors to resolve the design problems.

CHAPTER 4
Quantifying and Characterizing Design Discussions

In this chapter, we present our qualitative study design and results. The *goal* of the study in this chapter is to understand the type and prevalence of design discussions that occur during code review.

4.1 Introduction

Hypothetically, the modern trend of partially-automated code reviews has freed up human review participants to discuss the high-level concepts like design of the codebase, since the correctness and functionality of code changes are automatically assessed. Despite the importance of design and its positive effect on the software quality, little is known about the extent to which design is discussed during code reviews. We, therefore, set out to study the frequency and type of design discussions in code reviews. To achieve our goal, we perform an empirical study on the code reviews of the Nova and Neutron projects from the OpenStack community. In this chapter, we take an exploratory qualitative approach, where we manually classify 2,817 review comments from a randomly selected sample of 220 code review discussions to address the following two research questions:

(RQ1) How often is design discussed during code review?

Motivation: A common motivation for code review is to improve code quality [3]. Since design is an important aspect of code quality [19, 27, 11], we set out to study how often design issues are discussed during code review.

Table 4–1: An overview of the subject systems.

Product	Scope	Studied Period	#Code Changes	#Devs
Nova	Provisioning management	09/2011 to 01/2018	30,972	1,852
Neutron	Networking abstraction	07/2013 to 01/2018	16,894	1,177

(RQ2) Which design issues are discussed most often?

Motivation: Since design is a multi-faceted concept, to deepen our results from RQ1, we would like to know which aspects of design are being discussed during code review when design issues have been raised.

4.2 Study Design

In this section, we present our rationale for selecting the subject projects, as well as our manual analysis of the code review discussions.

4.2.1 Subject Projects

To address our research questions, we aim to perform an empirical study on large and actively maintained software projects that regularly perform code reviews. Our qualitative analysis requires a considerable investment of manual effort, which makes large-scale analysis of data from several software projects impractical. Therefore, we select for analysis the OpenStack community—a software community that has heavily invested in (a) its code review process (e.g., all changes must be approved for integration by two core reviewers)[1] and (b) the integration of automation into its

[1] `https://docs.openstack.org/infra/manual/developers.html\#project-gating`

code review process (e.g., before integration, all changes are scanned by and must pass code-style and static analysis verification bots). The OpenStack community develops a set of software tools that provision and manage virtual and physical resources that comprise a cloud computing environment.[2] The code reviewing process of OpenStack projects is managed by the Gerrit Code Review tool,[3] and its data can be accessed via a REST API.[4] We select the two most active OpenStack projects for analysis. Nova is responsible for provisioning management, while Neutron manages networking abstraction. Table 4.2.1 provides an overview of the studied projects.

4.2.2 Code Review Process

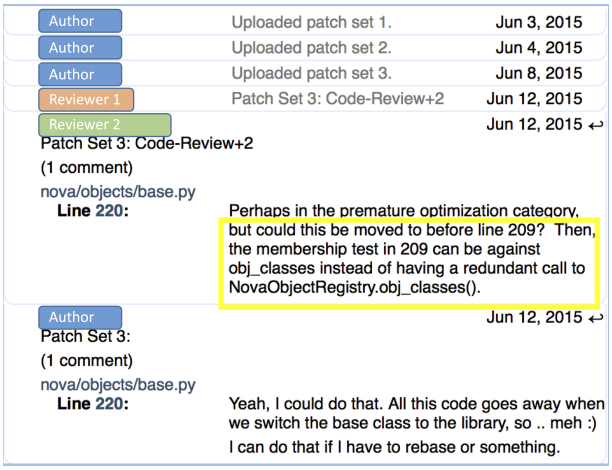

Figure 4–1: An example of a review discussion on Gerrit

[2] https://www.openstack.org

[3] https://www.gerritcodereview.com/

[4] https://gerrit-review.googlesource.com/Documentation/rest-api.html

Gerrit is a web-based tool that facilitates code review and integration management. The Gerrit workflow of OpenStack begins with an author uploading an initial revision of their code changes for review. These changes are then scanned by verification bots that check for code style issues using linters, common mistakes using static analysis, and functional issues using automated tests. After these checks are complete, other developers can review the changes, writing comments for the author to consider, providing general comments about the whole change or inline comments about specific parts of the change. Authors of other reviewers may reply to review comments, creating genera; or inline discussion threads.

Authors may address the feedback of verification bots and reviewers by uploading new revisions of their changes. This process is repeated until either (1) verification bots and reviewers are satisfied, and the code changes are approved for integration into project repositories; or (2) the author abandons the changes. Finally, for changes that have been approved for integration, a final set of integration tests are executed to ensure that other concurrently developed changes do not cause problems when they are integrated with the proposed changes.

4.2.3 Data Preparation and Processing

Figure 4–2 provides an overview of our data preparation and processing approach, which is comprised of a sample selection, design review identification and taxonomy generation steps. We describe each step below.

Representative Sample Selection (DP1). Using the REST API provided by Gerrit, we download 527,287 reviews within the studied time frame, which begins

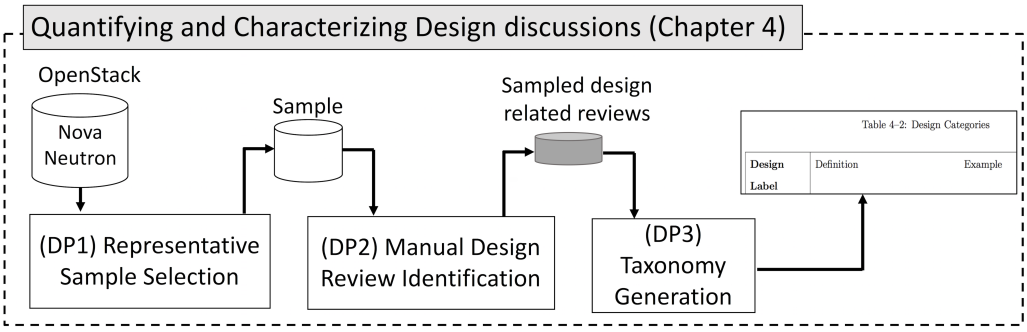

Figure 4–2: An overview of our data preparation and processing approach for Chapter 4

with community adoption of Gerrit (July 2011) and ends at the time of data collection (January 2018).

Since manual classification of a dataset of this size is impractical, we randomly sample reviews in each subject project. We randomly select NOVA and NEUTRON reviews for analysis. To cover as broad of a set of design issues as possible, similar to prior work [40], we aim to achieve *saturation* [34] with our sample. To do so, we continue to classify randomly selected review comments until we find no new types of design issues for 50 reviews. We achieved saturation after classifying 100 NOVA reviews with 1,357 comments and 120 NEUTRON reviews with 1,460 comments.

Manual Design Discussion Clarification (DP2). We identify design-related review comments by manually classifying review comments as design-related or not. Design-related review comments are identified according to the definition of design introduced by Brunet et al. [11]: *"As an artifact, design is a representation of how*

a portion of the code should be organized. As an activity, design is the process of discussing the structure of the code to organize abstractions and their relationships."

Together with a collaborator, the author of this thesis participated in the manual classification process. The process involved classifying each comment in the discussion of the sampled review as design-related or not. When necessary, the code changes themselves were analyzed for context. When the author of this thesis and collaborator disagree, the case was discussed until a consensus was reached.

Each comment may be tagged with multiple design issues; however, we find that multiple design concerns are rarely raised within one comment (only 18 instances in our sample of 2,817 comments). The whole discussion is then tagged with all of the unique labels of its comments. For example, Figure 4–1 shows that the comment surrounded by the yellow square is tagged as design-related due to a "Code Misplacement" issue.

In addition to labeling every code review discussion with one or more raised design issues, we also record if alternative solutions have been provided. For instance, a comment in a discussion can be labelled as "code misplacement" without a solution (e.g., "This method should not be in this class"), or with a solution (e.g., "This method should be in Class X, not Class Y").

To ensure that the labels that appeared later in the sample were not overlooked in the earlier comments, we perform a second pass over all of the labels after the initial classification pass. This classification process took seven days of full-time work of the two researchers who participated in the process. We use this classified sample to address RQ1 and RQ2.

Taxonomy Generation (DP3). After our manual classification (DP2), we apply open card sorting to construct a taxonomy from our tag data [46]. This taxonomy helps us to extrapolate general themes from our detailed tag data. During the card sorting process, the tagged comments are merged into cohesive groups.

4.3 Results

In this section, we present the results of our qualitative analysis with respect to RQ1 and RQ2. For each question, we describe our approach for addressing it followed by the results that we observe.

(RQ1) How often is design discussed during code review?

Approach. To address RQ1, we examine how often reviewers discuss design issues in the Nova and Neutron projects. To do so, we manually tag comments that are related to design issues (See DP2). The tags focus on the design concerns of authors and reviewers. We also differentiate between authors and reviewers to capture the difference in participation behaviour for both roles.

Similar to prior studies [3, 38, 48], we apply open card sorting to construct a taxonomy from our tag data. This taxonomy helps us to extrapolate general themes from our detailed tag data. The card sorting process is comprised of three steps. First, the author of this thesis and a collaborator perform the classification of review comments as per DP2. Second, we recursively group related labels by topic to produce a hierarchy of design issues. Finally, we count the occurrences of design-related comments in each studied project.

To estimate the degree to which the researchers agree about the design issues, we performed a pilot study of 50 review comments. The thesis author and collaborator independently labeled the 50 review comments. We then compute the Cohen's Kappa inter-rater agreement measurement, which was observed to be 0.72, indicating 'Substantial Agreement' [26].

Results. Observation 1—Design issues are not commonly discussed. Figure 4–3 shows a distribution of design issues that emerged during manual classification and the percentage of alternative solutions that were provided for each of the identified issues. We find that 33% and 35% of reviews have at least one design-related comment in Nova and Neutron, respectively. However, we find that only 9% and 14% of Nova and Neutron review comments are related to design, which implies that even in design-related code reviews, design discussions are short and scarce.

Observation 2—Authors rarely engage in design-related discussions. Most of the design-related comments are initiated by reviewers, whereas 11% and 21% of those comments elicit authors' responses in Nova and Neutron, respectively. This is alarming as low review participation has been associated with poor software quality [30, 51, 36, 12, 25].

We suspect that this tendency may have to do with the power that (core) reviewers hold over authors. Indeed, authors need reviewers to approve their changes in order for them to be integrated into the project. Thus, authors may feel compelled to simply do what the reviewer asks. A more rigorous analysis is needed to confirm (or reject) this suspicion.

Design concerns are not commonly discussed in code reviews. Moreover, authors rarely engage in design-related discussions with reviewers.

(RQ2) Which design issues are discussed most often?

Approach. To address RQ2, we examine the types of design issues in the studied projects. To do so, we examine the properties of the taxonomy that our open card sorting produced (See DP2 and DP3).

Results. Table 4–2 shows our resulting hierarchy-less taxonomy of mutually exclusive design issues that have been encountered and identified during manual classification [41]. Furthermore, we include some non-design labels, namely "Readability" and "Documentation", both of which can easily be confused with design. Despite being unrelated to the functionality of the code, these categories are concerned with the code cosmetics and understandability, respectively, rather than design.

Table 4–2: Design Categories

Design Label	Definition	Example

Module Coupling	Comments that point out unnecessary dependencies between software artifacts.	*"so my proposal was to have the resource tracker create the ResourceProvider object (separately from the ComputeNode object). That way the ComputeNode and ResourceProvider objects don't have any interaction needed between them."*
Redundant Code	Comments that identify duplicated or unnecessary code that needs to be eliminated. Usually due to not reusing existing functionality or, simply, the addition of unwanted code.	*"nit: This is the default, and already set by the superclass. It would be cleaner to omit it."*
Performance	Comments that point out execution issues in terms of time and memory usage, which can be optimized using alternative approaches, while still maintaining correct functionality.	*"[This alternative] uses a much more efficient exponential doubling algorithm. It runs in 0.03 seconds versus a very long time for your implementation (I stopped waiting after a couple of minutes)."*

Shallow Fix	Comments that point out limitations in the breadth of a fix in terms of the design of the code. This could be a fix that has a ripple effect on the code, or a fix that could potentially be generalized to fix other areas in the code suffering from the same problem.	*"The VIP port is not too different from any other port, so why checking only VIP's ip then? I think such check needs to be added to create_port method of db_base_plugin_v2, and not to LBaaS plugin only."*
Side Effect	Comments pointing at code changes that might potentially break other parts of the code due to its design	*"Hardcoding this breaks non-qemu libvirt hypervisors. [..]"*

Unnecessary Complexity	Avoidable complications in the code that can and should be further simplified without affecting the correct functionality of it.	*"The code here is going to query subnets table and fetch subnets objects, but those subnet objects are only used to check whether their enable_dhcp attribute is True or not. So, why not directly query Subnets table [..], and check whether at least one record exists."*
Code Misplacement	Unsuitable or wrong code placement of software artifacts. Better placement might render the artifacts reachable and reusable by other modules or artifacts.	*"[..] We should mode that map from neutron.agent.firewall to some security group common module and re-use it. "*
Non-Design Labels	These labels can be confused for design, but are not. They are shown as way to better understand the difference between what is considered as Design-related, according to our taxonomy.	

Readability	Not considered design. Related to the cosmetics of the code. It includes but is not limited to: indentation, naming conventions and spacing.	*"I think this should be renamed to SimpleTenantUsage"* *"recommend just overriding the numaTopology variable instead of the awkward "fittedTopology" variable name :)"*
Documentation	Intuitiveness, understandability and writing style of commit messages, code comments and code documentation.	*"So I think this might be change that doesn't require an API version bump, but we need a lot more context in the commit message to make that clear."* *"you removed this function [..] if it's your desired action, at least should be mentioned in commit message"*

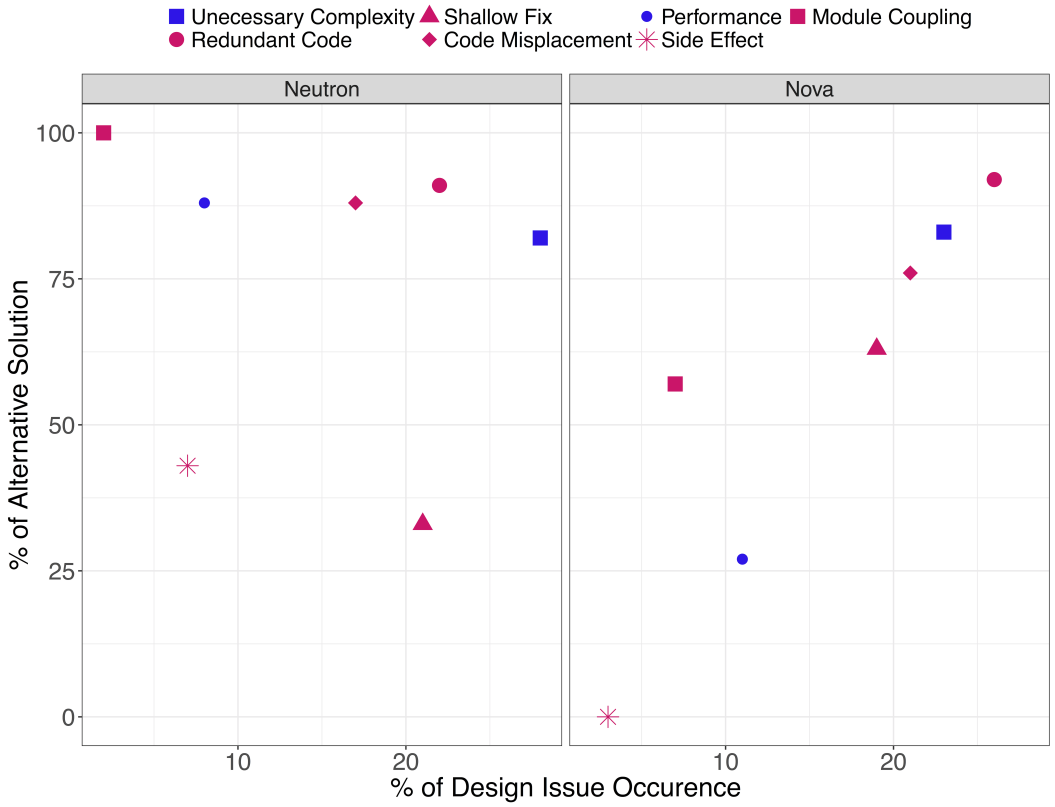

Figure 4–3: The rate of each design category across design-related comments

Table 4–3: The frequencies of different design categories found in code reviews, in Nova and Neutron projects, during manual classification.

Design Category	Frequencies %	
	Nova	Neutron
Unnecessary Complexity	23	28
Redundant Code	26	22
Shallow Fix	19	21
Code Misplacement	11	17
Performance	11	8
Side Effect	3	7
Module Coupling	7	2

Observation 3–Design issues are often raised due to lack of awareness of the global context of the codebase. Figure 4–3 shows the type of design issues that emerge during the sampled code reviews. We observe that all of the design issues except for "Performance" and "Unnecessary Complexity" (blue shapes in Figure 4–3) are linked to an incomplete understanding of the codebase. Altogether, a lack of global context awareness accounts for 67% and 65% of the sampled design issues in Nova and Neutron, respectively (red shapes in Figure 4–3, and Table 4–3). For instance, duplication may occur because the author is not aware of the existing functionality, while shallow fixes, code misplacement, side effects, and coupling concerns may occur because the author is not aware of the hierarchy and relationships between modules in the codebase. However, their emergence in code reviews is a positive indication that these types of issues are being noticed and corrected during the reviewing process.

Observation 4—Most of the design-related discussions are constructive, offering alternative solutions that mitigate the raised design issues. Collectively, 73% of the analyzed design comments provide an alternative solution to help authors to address the raised design issues. We believe that this is an indication of a healthy reviewing community, since such feedback is likely more constructive than simply pointing out flaws.

As Figure 4–3 shows, reviewers are less likely to provide alternative solutions for design issues like "Code Misplacement", "Performance", "Module Coupling", and "Side Effect" than the other design issues. Indeed, 93% of the time, reviewers provide alternative solutions to address "Redundant Code".

For example, in review #319327,[5] the reviewer pointed out that the functionality already exists: "We actually have a method to do that in [..]". The author fixed the issue and was eventually able to merge the code change.

Most design issues are raised due to a lack of awareness of the global context of the codebase. Moreover, most of those comments are constructive, providing alternative solutions.

4.4 Chapter Summary

In this chapter, we qualitatively explore the frequency and type of design discussions that occur during code reviews. While we make some general observations, it is difficult to scale the analysis up, due to the considerable investment of manual effort that is required. Thus, in the next chapter, we explore the applicability of automated approaches that would remove this barrier.

[5] `https://review.openstack.org/\#/c/319327/`

CHAPTER 5
Large-Scale Analysis of Design Discussions

In this chapter, we present the study design and the results of our quantitative analysis with respect to RQ3 and RQ4. For each question, we describe our approach for addressing it followed by the results that we observe.

5.1 Introduction

The manual effort required to perform the analysis of Chapter 4 presents barriers to large-scale analysis of design discussions. For example, manually classifying 2,817 code review comments took ten days of full-time work, by two classifiers. The entire dataset of code review comments available is 2,506,308, which is practically impossible to manually-classify. In this chapter, we aim to remove that barrier by proposing automated techniques capable of performing the high-level labeling task (design-related or not). To do so, we use the manually classified sample to train and evaluate classifiers to automatically label code review comments as design-related or not. Those classifiers are applied to a large sample of 2,506,308 review comments, and the results are used to address the following two research questions:

(RQ3) How accurately can our classifiers identify design-related comments?

Motivation: Classifiers that can automatically identify design-related comments would be helpful to study the characteristics of code changes that elicit design-related feedback. Since the usefulness of classifiers depends on their accuracy, we first set out to train and evaluate classifier performance.

(RQ4) Is design-related discussion correlated with the characteristics of code changes?

Motivation: Design-related discussion may be more likely to appear under certain conditions. For example, since novices may not understand the global context of the codebase (see RQ2), design discussions during their reviews may occur more frequently. Moreover, design-related discussion may be associated with the outcome of code changes. For example, design-related discussion may be associated with higher rates of abandonment, since design issues may require a rewrite of the code change. We set out to test such hypotheses using the automatically classified review data.

5.2 Study Design

In this section, we present our data preprocessing steps required to build a prediction model that detects design discussions in code reviews.

5.2.1 Quantitative Data Preparation

Figure 5–1 provides an overview of our preparation processes (DP4 and DP5) for addressing RQ3 and RQ4 of our quantitative analysis.

Automatic Design Review Classification (DP4). To address RQ3, we train classifiers that can automatically identify design-related comments by applying machine learning techniques to our original review data. We select five popular techniques for our study: Multinomial Naïve Bayes (MNB), Support Vector Machines (SVM), Decision Tree, k-Nearest Neighbours, and Random Forest.

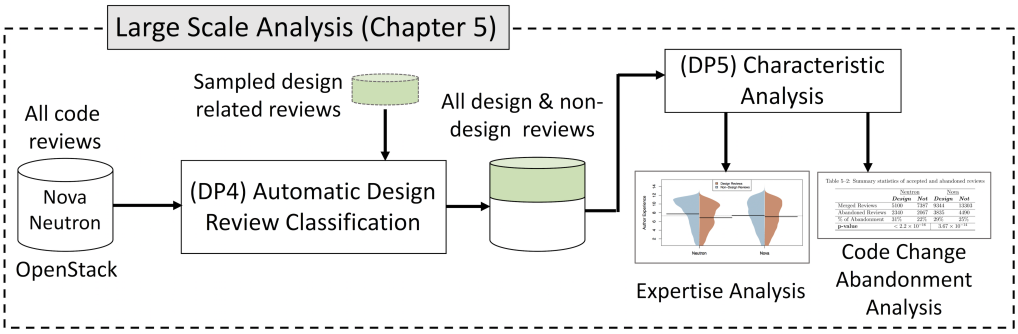

Figure 5–1: An overview of our data preparation approach for Chapter 5

We begin the automatic classification by pre-processing the raw comment text. First, we remove stop words, i.e., words that add little semantic meaning to a document, using the stopword list of the Python NLTK module.[1] Next, to mitigate the impact of conjugation, we apply the Porter stemmer to each surviving word.[2] Finally, we convert each comment to a vector that is composed of the Term Frequency-Inverse Document Frequency (TF-IDF) weights of its words. In a nutshell, terms that rarely appear in comments, and often appear within a comment are of higher weight for a comment.

To estimate the performance of our models on unseen data, we perform ten-fold cross-validation, where the sample data is split into ten folds of equal size, nine folds are used to train the model, and the remaining fold is used to test the model. The

[1] `https://www.nltk.org/api/nltk.corpus.html/module-nltk.corpus`

[2] `http://www.nltk.org/howto/stem.html`

process is repeated ten times (where each fold is treated as the testing fold once), and the mean performance results are reported.

Characteristic Analysis (DP5). To address RQ4, we study the relationship between incidences of design-related discussion and pre-/post-integration code change characteristics. In terms of pre-integration properties, we analyze author experience because we conjecture that authors with low experience may be more susceptible to making design mistakes. We estimate the author's expertise using heuristics that we derive from prior work [50]. Since the primary focus of our study is on reviewing, similar to Thongtanunam et al. [51], our author's expertise heuristic counts the number of prior reviews that the author has written or reviewed.

In terms of post-integration properties, we analyze the rate of review abandonment, since it is the primary post-integration property. We identify reviews that are abandoned (i.e., not integrated). To do so, we classify the design-related/unrelated comments that DP4 generates into accepted or abandoned categories. We exclude ongoing reviews (i.e., those with the "OPEN" status) from this analysis because it is not clear if they will be accepted or abandoned yet.

5.3 Results

(RQ3) How accurately can our classifiers identify design-related comments?

Approach. To address RQ3, we train classifiers models that can automatically identify design-related comments. Our models learn a word vector of a comment where words are weighted by TF-IDF scores. For this analysis, we use the comments that were not included in the sample that was analyzed in Chapter 4.

Handling Data Imbalance: Our dataset is imbalanced—only 9% and 14% of NOVA and NEUTRON comments are design related. To reduce bias towards the majority class (non-design), the training data must be re-sampled. Our re-sampling process is implemented using a combination of oversampling of the minority class and undersampling of the majority class. We do so because prior work has shown that such a combination performs better than either oversampling or undersampling in isolation [13, 1].

Choosing Performance Metrics: We use the common classifier performance measures of recall, precision, and F1-score. Recall is the ratio of actual design-related comments that a classifier can detect ($Recall = \frac{TP}{TP+FN}$), and precision is the ratio of correctly predicted design-related comments to the total number of reviews that were predicted to be design-related ($Precision = \frac{TP}{TP+FP}$). There is a tradeoff between recall and precision. To account for this, we use the F1-score, which is the harmonic mean of recall and precision: $F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$

The above-mentioned performance metrics (i.e., recall, precision, and F1-score) depend on the threshold that is selected to classify instances as design-related or not (typically set to 0.5). Choosing different thresholds may yield different results. To get an overall idea of the performance across thresholds, we additionally use the Area Under the receiver operating characteristics Curve (AUC). AUC ranges from 0 to 1, and larger AUC values indicate better prediction performance. The advantage of AUC is its robustness toward imbalanced data, since the curve is obtained by varying the classification threshold over all possible values.

Table 5–1: Binary Classification 10-fold Cross-Validation Results with balanced datasets

Algorithm	Neutron				Nova			
	Prec.	***Recall***	***F1***	***AUC***	***Prec.***	***Recall***	***F1***	***AUC***
MNB	0.66	0.76	0.68	0.85	0.59	0.69	0.60	0.79
SVM	0.70	0.71	0.70	0.84	0.63	0.62	0.62	0.76
Decision Tree	0.65	0.66	0.65	0.66	0.59	0.61	0.59	0.61
KNN	0.59	0.60	0.36	0.70	0.63	0.55	0.56	0.55
Random Forests	0.69	0.64	0.65	0.80	0.66	0.55	0.56	0.75

Evaluating the Prediction Model: We compare the performance of our classifiers to that of traditional baselines like random guessing and zeroR. By construction, a random guessing approach would achieve an AUC of 0.5. Hence, AUC values above 0.5 indicate performance that suggests the classifier contains some information. In addition, a zeroR classifier is a classifier that always reports the class of interest. Hence, in our scenario, a zeroR classifier achieves a perfect 1.0 recall and a precision equal to the rate of design-related comments. Since these values are unreasonably high (recall) and low (precision), we compare our classifiers to the F1-score of the zeroR classifier to counteract the extreme values.

Results. Observation 5—Multinomial Naïve Bayes achieves the best recall values. Table 5–1 shows that MNB achieves recall values of 0.69 and 0.76 in Nova and Neutron, respectively. As for the general performance of all classifiers, the precision ranges from 0.59 to 0.70, whereas the recall ranges from 0.55 to 0.76. Overall, we observe that Neutron has better performance values than Nova. We suspect that this is due to Neutron having more design comments in its training data (14%) compared to Nova (9%).

Observation 6—Our classifiers outperform zeroR by 43 percentage points in terms of F1-score. Since zeroR achieves 9% (Nova) and 14% (Neutron) of precision, the best classifier (MNB) outperforms zeroR by 50 (Nova) and 52 (Neutron) percentage points in terms of precision. Moreover, the AUC ranges from 0.55 to 0.85, indicating that our models outperform random guessing by five to 35 percentage points.

Observation 7—Our classifiers strengthen the finding that design discussions are scarce. Following Observation 1, our classifiers suggest that 43% and 44% of all code reviews contain at least one design comment in Nova and Neutron, respectively. However, when calculating the percentage of design-related comments with respect to all review comments, they only amount to 4% and 6% in Nova and Neutron, respectively.

> Multinomial Naïve Bayes achieves the best recall values among our classifiers, outperforming a zeroR classifier by 43 percentage points in F1-score. Moreover, our automatic classification complements our manual classification finding, suggesting that design discussions are indeed rare.

(RQ4) Is design-related discussion correlated with the characteristics of code changes?

Approach. To address RQ4, we first apply our top-performing classifier from RQ3 to the entire set of review comments from Nova and Neutron to classify them as design-related or not. Then, we compare the author's experience scores and rates of the abandonment of code changes in the reviews that contain design-related comments and those that do not. Note that the top-performing classifier has plenty

of room for improvement. A more accurate model may improve the robustness of these results.

The author's experience score is the number of times that an author has participated in the reviews of a given project prior to the review at hand. We then compare the author's expertise values using bean plots. Bean plots are boxplots in which the vertical curves summarize and compare the distributions of different data sets [23], which in our case correspond to the author's expertise of design-related reviews and reviews without design comments. The horizontal black lines indicate the mean of the author's expertise for design-related reviews (orange) and non-design reviews (blue), shown in Figure 5–2.

We estimate the statistical significance of the difference using two-sided, unpaired Mann-Whitney U tests ($\alpha = 0.05$) and the practical significance of the difference using Cliff's delta (δ). The effect is considered negligible when $0 \leq |\delta| < 0.147$, small when $0.147 \leq |\delta| <$, medium when $0.33 \leq |\delta| < 0.474$, and large otherwise. We use Mann-Whitney U tests and Cliff's delta since these are non-parametric tests and our data is not normally distributed. We check the statistical significance of the change in the rate of abandonment using a Pearson χ^2 test (one degree of freedom, $\alpha = 0.05$).

Results. Observation 8—Authors of design-related reviews tend to have lower expertise. Figure 5–2 shows that design-related reviews tend to have lower author expertise than reviews without design comments in both studied projects. Mann-Whitney U tests show that there is a statistically significant difference in

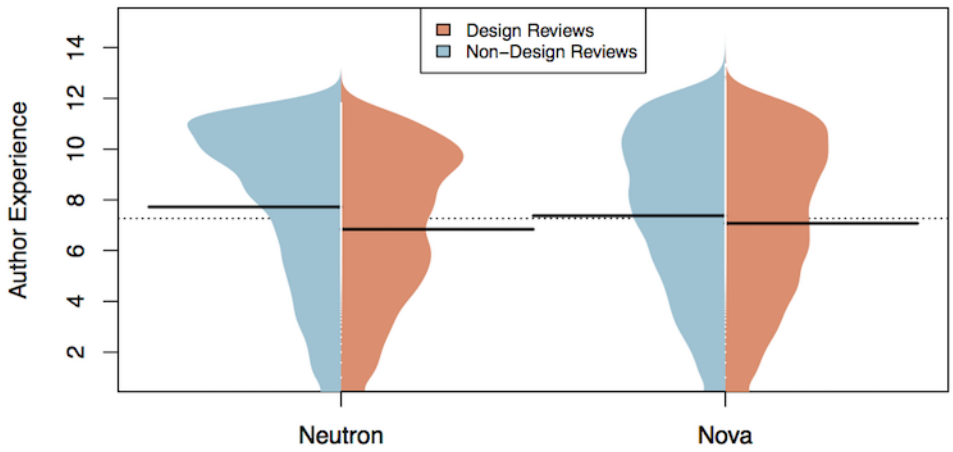

Figure 5–2: Difference in author experience in design-related and design-unrelated code reviews

expertise between design-related and non-design-related reviews in Nova and Neutron ($p < 2.2 \times 10^{-16}$ in both cases). However, Cliff's delta values suggest that the practical difference is negligible (-0.09 in Neutron and -0.06 in Nova). Although the difference is practically negligible, the direction generally agrees with Observation 3, i.e., the authors of design-related reviews are less likely to be aware of the global context of the codebase.

Table 5–2: Summary statistics of accepted and abandoned reviews

	Neutron		Nova	
	Design	***Not***	***Design***	***Not***
Merged Reviews	5100	7387	9344	13303
Abandoned Reviews	2340	2067	3835	4490
% of Abandonment	31%	22%	29%	25%
p-value	$< 2.2 \times 10^{-16}$		3.67×10^{-14}	

Observation 9—Design-related reviews are more likely to be abandoned. Table 5–2 shows that 29% and 31% of design-related reviews are abandoned

in Nova and Neutron, respectively. When compared with reviews without design comments, we find that design-related reviews account for an increase of four and nine percentage points in Nova and Neutron, respectively. A Pearson χ^2 test shows that there is a statistically significant difference between design-related and reviews without design comments in terms of the rate of abandonment in both studied projects. These results complement those of Tao et al. [56], who observed that design-related reviews were less likely to be accepted in Mozilla.

Authors of design-related reviews tend to have less expertise than reviews without design comments. Moreover, design-related reviews are more likely to be abandoned in our subject projects.

5.4 Chapter Summary

In this chapter, we demonstrate that coarse-grained classifiers show promise for identifying design-related review comments. Using those classifiers, we scale our analysis up to our entire dataset of code reviews to navigate the relationship between properties of a code review and the incidence of design-related feedback. While the observation of Chapters 4 and 5 shed light on the phenomenon of design discussions that occur during code review, we are missing the pragmatic perspectives of developers who regularly participate in the code review process. Thus, the next chapter sets out to gather practitioner perspectives and triangulate them with respect to our qualitative and quantitative observations.

CHAPTER 6
Triangulation through Surveying Software Practitioners

In this chapter, we present the design and results of the survey that we conducted to triangulate our observations. Triangulation is a mixed-methods research tool that connects different quantitative and qualitative observations by inquiring within the evaluation community [22]. In our case, our evaluation community is comprised of software practitioners who regularly participate in code reviews. The goal of this chapter is to better understand how software developers define design, and the role that it plays in their code reviewing behaviour.

6.1 Survey Design

Broadly speaking, our survey consists of five main sections, each of which focuses on one aspect of design discussion in the peer code review process. The first section explores the definition and importance of design in the context of code review, while the second section explores design concerns. The third and fourth sections investigate the prevalence and importance of the different types of design discussions that occur during code reviews, while the fifth section examines the strength of the relationship between design discussions and software experience.

Multiple choice and Likert scale questions are quantified easily, while free-form questions are intentionally formulated to solicit open-ended responses to allow developers to express their opinions and perspectives in complete thoughts. Using the

same open coding approach as Chapter 4, responses are coded by the author of this thesis, and a collaborator to extrapolate trends for further analysis [41].

6.2 Demographic Information

The survey was posted on the developer mailing lists of the studied OpenStack community and the Gerrit Community. It was also circulated via social media platforms such as Linkedin, Twitter, and Facebook, to maximize exposure and engagement. To encourage participation in the survey, respondents were entered into a raffle for a $50 gift card. The survey yielded 94 respondents with varying expertise levels, backgrounds, geographical locations, and work domains. All respondents are software professionals who self-report that they participate regularly in code reviews.

Our respondents have a median of five years of experience, with a mean of eight years, and values range from as much as 27 years to as little as one year. In addition, 15% of our respondents have contributed to OpenStack which is the community that we focused on in Chapters 4 and 5. Respondents work in a broad variety of sectors of the software market, including (but not limited to) cloud operations, telecommunications, e-commerce, finance, aerospace and web, mobile and video game development. Lastly, the respondents are geographically dispersed, with respondents having affiliations in Africa, Asia, Europe, and North America.

6.3 Importance and Frequency of Design

Participants were shown our adopted definition of design, originally introduced by Brunet *et al.* *"As an artifact, design is a representation of how a portion of the code should be organized. As an activity, design is the process of discussing the structure of the code to organize abstractions and their relationships"* [11].

Figure 6–1: Importance of Design

Question #1: *With the definition in mind, in your opinion, how important is design during code review? Why?*

Observation 10—The majority of respondents agree that design discussion is important and shares a link with software quality. Figure 6–1 show that 89% of respondents believe that discussing design during code review is 'important' (39%) or 'very important' (50%).

Respondents were given the opportunity to elaborate upon the rationale behind their choice with a free-form answer box. 82% of respondents provided responses. Table 8.4 (Appendix A) shows the coded results, which indicate that the majority of the respondents perceive that there is a strong link between software design with software quality aspects. For example, one respondent speculates that "One of the crucial aspects of any new code written is how it fits into the current [design] or what kind of [design] it forms itself. A lack of design for any code eventually lends itself to become unmaintainable and difficult to extend." Broadly speaking, the software quality responses fit under 'Code Cleanliness' and 'Performance' categories, which are further decomposed into sub-categories. We find that 71% of the responses are associated with 'Code Cleanliness', while 12% of the responses are associated with "Software Performance". 'Code Cleanliness' includes reusability, maintainability,

extensibility, sustainability, and code organization, while 'Software Performance' encompasses scalability, efficiency, as well as time and memory resources.

Observation 11— The majority of respondents agree that design discussion should take place during code review. We find that 88% of the respondents agree that design is an integral part of a proper code review. Several responses especially touch upon the difficulty to automate design checks. For example, one respondent speculated that: "Design is something that [is] hard to check automatically, so humans must check that during code-review." Along the same line of thought, another respondent remarked: "[Design Concerns] are the hardest to detect with an automated tool, and their side effects can appear too late in the development process which makes [them] more expensive to fix."

On the other hand, a considerable minority expressed opposing views. Indeed, 15% of the participants believe that design should be discussed prior to code reviews, not during them. For instance, a respondent shared that "Design discussions should take place in design discussion/grooming sessions. While writing maintainable and reusable code is crucial in software design, code reviews tend to get bulky and boring, with multiple parallel threads of debates when design decisions are discussed there."

The majority of survey participants agree that design is important, and should be discussed during code reviews, as it is associated with better software quality. If left unchecked, however, design discussions may bloat the code review process.

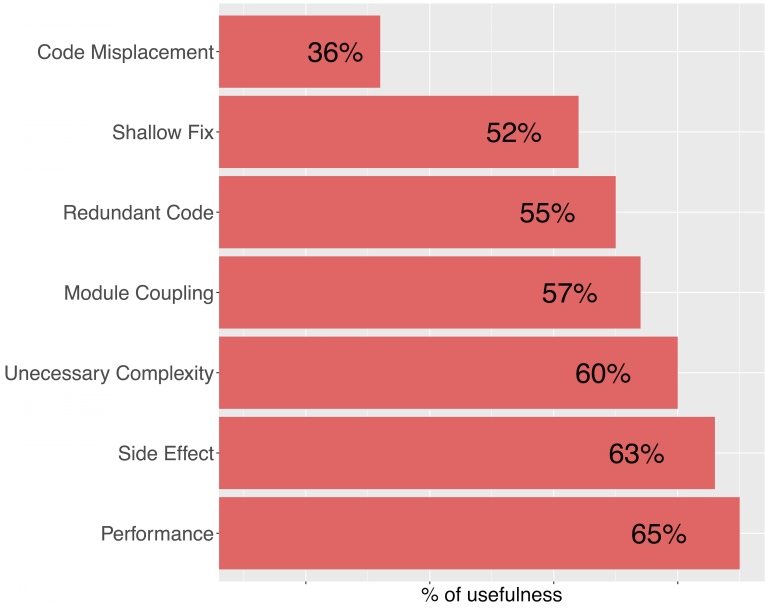

Figure 6–2: Usefulness of design concerns according to the survey participants

6.4 Design Concerns

Participants were shown the definition of the seven design concerns that we discovered in Chapter 4.

Question #2: *In your opinion, which design concerns are the most useful?*

Observation 12—'Performance' is the design concern that is most frequently perceived to be important. Figure 6–2 shows the number of respondents who selected each design concern. 65% of respondents selected 'Performance'. The value of performance feedback is highlighted by the free-form feedback from one respondent: "Performance is always important, without it, nothing is useful commercially." On the other hand, 'Code Misplacement' ranks as the least useful, since only 36% of developers selected it. Indeed, a respondent clarifies why they did

not select 'Code Misplacement': "Code misplacement is also less important than the other topics, and can be easily addressed [in follow-up code changes]."

Observation 13—The potential for end-user impact is a key criterion for evaluating the importance of design concerns. Software design is rarely a characteristic that is associated with an impact on end-user experience. Yet, interestingly, 7% of the free-form answers to Question #2 argue that their selection of important design concerns are based on whether said concern affects the end-user of the software. While 7% is not an especially large percentage, we do believe it is worth noting because it emerged naturally without respondents being prompted to select it as an option. For example, respondents pointed out:

- Product: "Redundant code and module coupling while nice [minimize, they do] not necessarily [break] or []affect] the end product too much. [...]"
- End-Users: "These concerns/issues will directly impact the users of the software. It will likely flow to them as errors, slow performance etc."
- User Experience: "I selected these ones since they can affect the end user's experience like (Performance, Side Effects, Shallow Fix)[...]"
- User Experience: "Performance because slow performance affects UX and [...]"

Observation 14—Respondents also notice the importance of awareness of the global context of the codebase (Observation 3). Respondents also point out the importance of recognizing the global context of the codebase during code review. For example, one respondent speculates that: "Redundant code is a symptom of [a developer] who doesn't understand code design as a whole", while

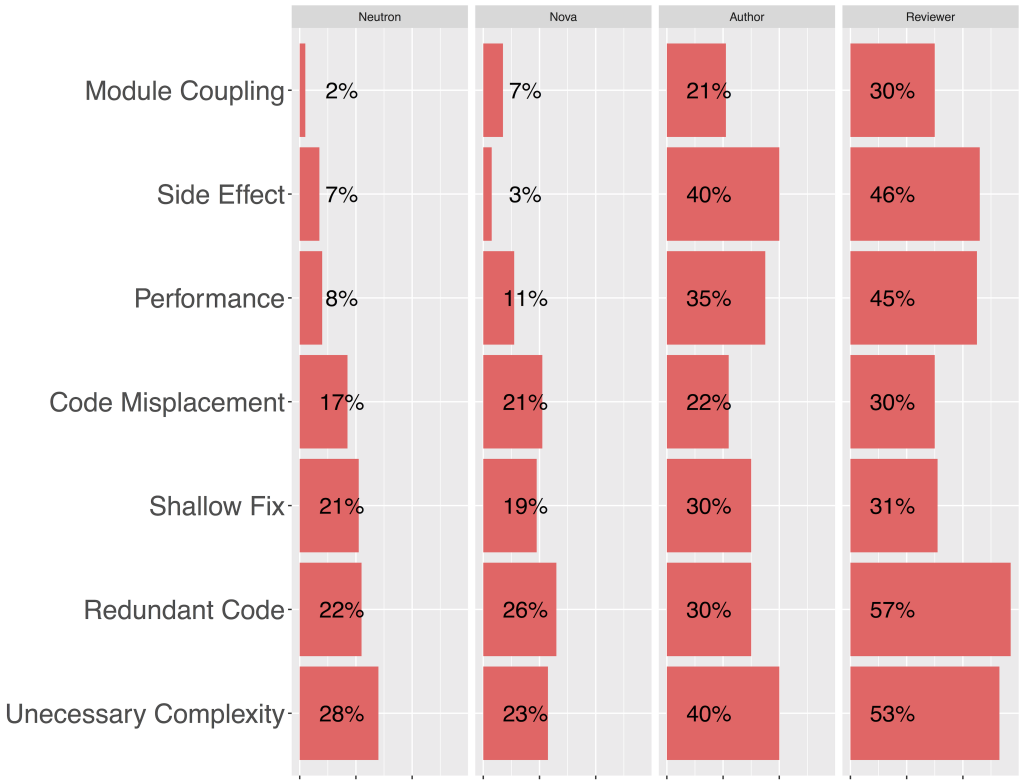

Figure 6–3: The frequencies of different design concerns found in code reviews, compared to the perceived frequencies of them according to survey participants

another speculates that: "Because sometimes, especially when fixing bugs, we miss the whole picture [...]". These responses complement Observation 3, where we noted that most design concerns arise due to a lack of awareness of the global context of the codebase.

Question #3: *In your experience, which design concerns are raised most frequently on code changes that you have authored/reviewed?*

Figure 6–3 shows the rates at which respondents perceive that different design concerns are raised. The differences between the perceptions of respondents and our

observations from Chapter 4 are highlighted. In addition, the difference in perceptions when respondents are in the author and reviewer roles are shown.

Observation 15—Respondents perceive 'Unnecessary Complexity' and 'Redundant Code' as the most frequent design concerns across all the different settings, which agrees with our findings. Interestingly, the two most frequent design concerns that we have found in the OpenStack community, are the same two design concerns that respondents perceive as the most frequent, as shown in Figure 6–3.

On the other hand, respondents perceive that 'Performance' and 'Side Effect' occur relatively frequently, while we find that they were rarely raised in the OpenStack community. More specifically, design concerns that are related to 'Performance' seem to be of great significance to practicing software developers, which might suggest that in propriety development settings, performance may be a bigger pain point that in the studied OpenStack context.

These reinforce the importance of Observation 12, which states that 'Performance' is the design concern that resonates with the largest proportion of respondents.

Question #4: *Can you think of other design concerns that you believe are important?*

28 respondents shared their input to this question. After coding the responses, we found five respondents (29%) named *readability*-related concerns such as 'naming conventions' and 'redundant comments' as *other* design concerns. However, we have chosen to exclude 'Readability' (recall Table 4–2). We did so to conform to a definition of design which focuses on higher-level issues, such as codebase organization

and class structure, rather than more cosmetics issues, such as the readability of the code. Moreover, nowadays, readability concerns can be improved by automatic approaches (e.g., Linters, IDE configuration). This confusion emphasizes the difficulty of clearly carving the boundary of design in practice.

Another three of the respondents (10%) mentioned (violations of) the 'Single Responsibility Principal' as an important design issue that is missing in our taxonomy of design concerns. We revisited our data from Chapter 4, and found that such violations are labelled under the 'Module Coupling' and 'Code Misplacement' categories of our taxonomy. This is because violations of the single responsibility principle manifest as functionality appearing within a module where it does not belong, i.e., *misplacing the code*, which results in undesirable *coupling* between modules that should not (directly) interact.

Additionally, 7% of the answers mention 'Testability', i.e., the degree to which the code as written lends itself to being (automatically) tested. After revisiting our data from Chapter 4, we confirmed that Testability did not explicitly arise in our sample; however, we did observe examples of 'Code Misplacement' and 'Module Coupling' that imply an underlying concern of testability.

Finally, one respondent mentioned 'Backwards Compatibility' as a missing design concern. After revisiting data from Chapter 4, we note that these issues map to our 'Side Effect' category.

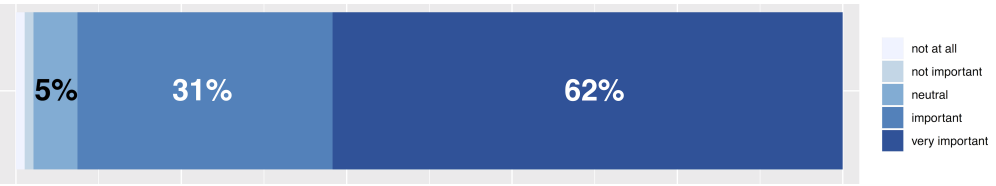

Figure 6–4: Importance of Design Discussion

Respondents tend to prioritize design concerns by their potential impact on end-users. This helps to explain why 'Performance' is the design concern that resonated with the largest proportion of our respondents.

6.5 Design Discussions

Question #5: *When a design concern is raised, in your opinion, how important is it that the authors and reviewers engage in a discussion about it? and Why?*

Observation 16—Respondents believe that design discussion benefits the individual, the development team, and the end-product. As Figure 6–4 shows that most respondents perceive that discussing design is beneficial, as 94% of the answers fall under 'important' and 'very important'. Table 8–2 (Appendix B) shows the coded results of the free-form answers, which provides further insight into the rationale for this. The responses fit into four high-level categories: 'Team Gains' (74%), 'Personal Gains' (52%), 'Product Gains' (48%), and 'No Discussion Needed' (16%).

As the names suggest, the first three categories encompass positive benefits for the individual, the development team, and the end-product, respectively, while the

last category is comprised of responses that are opposed to design discussions during code reviews.

'Team Gains' were mentioned in 74% of the responses—the highest-ranking justification for the importance of design discussion according to our respondents. The majority of the answers were further decomposed into: 'Reaching Consensus', 'Opportunities for Growth', 'Group Problem-Solving', 'Team Sync-Up', 'Understanding Trade-Offs', 'Information-Sharing' and the 'Understandability of Design' sub-categories. For example, one respondent remarked that: "[it is important that] software engineers know what are the constraints, limitations, and performance of the code they develop.", while another respondent claimed that: "Both parties should understand the concern that is raised and the various tradeoffs each solution will [present]."

'Personal Gains' were mentioned in 52% of the responses, which were further decomposed into: 'Understanding Other Perspectives', 'Receiving Feedback for Author', 'Avoiding Repetition of Mistakes', 'Explaining the Rationale' behind the code change, and 'Suffering/Overcoming Power Dynamic between Authors and Reviewers' sub-categories. This agrees with our earlier suspicion that a power dynamic is present between authors and reviewers. Indeed, in Observation 2, we conjectured that authors rarely engage in design discussions because of the power dynamic between authors and reviewers.

For example, one respondent remarked that: "[authors should] make sure that [they] answer reviewers for reasons of politeness. It would be rude to ignore the time and effort someone put in to give that feedback", while another encouraged engagement

in the discussions because "Two sided discussions are very important, if people have egos about their software and reject improvements, the product will suffer."

'Product Gains' was mentioned in 48% of the responses, which were further decomposed into: 'Enhancing Product Quality', 'Preventing Design Erosion', 'Maintainability' and 'Reusability' sub-categories. For example, one respondent remarked that: "because everyone needs to understand the issue and why it is important to [address, for example] for code reading [or] reusability"

'No Discussion Needed' was mentioned in 16% of the responses, which were further decomposed into two sub-categories: 'Design Concerns Should be Resolved Later', and 'Trivial Solutions to Design Concerns' should not be discussed as it wastes time. For example, one respondent remarked that: "In a review, you do not discuss. You highlight the issue and move on. Resolution is done after the review", while another stated that: "the importance [of design discussion depends] on the complexity of the review, if the review is trivial, a discussion can be a bad allocation of time."

The majority of the respondents believe discussing design is beneficial for the individual, the development team, and the end-product. Other respondents raised concerns that design discussion should not slow development progress, and if raised, could be addressed in follow-up work.

6.6 Experience

Question #6: *In your experience, how quickly are new contributors able to provide valid design feedback?*

Figure 6–5: How quickly our respondents think new contributors are able to provide valid design feedback.

Figure 6–6: The type of contributor that tends to provide the mosts design feedback, according to our respondents.

Observation 17—Respondents perceive that software experience is associated with providing design feedback, and engaging in design discussions. Figure 6–5 shows that the responses are skewed to the 'Expert' side of the scale, where 52% chose 'Intermediate'-experience as the starting level that is able to provide valid design feedback, while 34% of the responses opted for even higher experience (i.e. 'Advanced' and 'Expert'). Additionally, Figure 6–6 compliments this finding, as it shows that 84% of respondents chose 'Advanced'- and 'Expert'-level developers as the type of contributers that are more *likely* to provide useful design feedback during code review. This compliments our Observation 8, where we observed that experience levels of reviewers who provide design feedback are statistically-significantly higher than others.

Developers' perceptions agree with our quantitative observations that project-specific experience is related to the likelihood of providing design feedback.

6.7 Chapter Summary

In this chapter, we triangulate our prior observations from Chapters 4 and 5 with respect to the perceptions of 94 software practitioners who regularly participate in code reviews. In the next chapter, we discuss the threats to the validity of the studies in this thesis.

CHAPTER 7
Threats To Validity

Being comprised of empirical studies, this thesis is subject to threats to its validity. In this chapter, we describe those threats with respect to external, internal and construct validity types.

7.1 External Validity

External validity concerns have to do with the generalizability of our studies. Due to the manually intensive nature of our qualitative study (Chapter 4), we chose to focus our analysis on two open-source projects from one community (OpenStack). Due to our small sample size, it is difficult to draw general conclusions about software projects. However, the goal of this thesis is not to provide a theory of design discussions in code reviews that applies to all systems. Instead, we believe that our study could serve as a baseline for design classification and design involvement in code reviews. Future replication studies will help to broaden our insights into more general trends.

Moreover, the insights that we glean from our survey of 94 software practitioners (Chapter 6) are likely incomplete as well. On the other hand, the survey respondents work in different sectors of the software industry and are geographically dispersed. Furthermore, our respondents cover a broad range of experience levels ranging from one to 27 years of professional experience in the software industry.

7.2 Internal Validity

Internal validity concerns may be raised if other plausible hypotheses may explain our observations. Since we select a sample of review comments in our qualitative analysis (Chapter 4), sampling bias might have an impact on our conclusions. For example, the studied projects may have dedicated explicit effort to design improvement at particular time periods. If those periods were over or undersampled, our results may be skewed. To mitigate this threat, we randomly selected reviews for manual classification, so that our findings are not bound to a certain period in the project history.

7.3 Construct Validity

Construct validity concerns may creep into our study if our measurements are misaligned with the phenomena we set out to study. Our detection of design-related comments (Chapter 4) is a manual process that may be subjective. However, we mitigated this threat by having two coders agree about each classified result. The coders are both graduate students with industrial experience in code reviewing practices. Moreover, an individually classified sample yielded a Cohen's Kappa of 0.72 (substantial agreement). To facilitate further refinement of our classification, we provide our definition and shared understanding of each design category, as well as examples in Table 4–2. To further strengthen our observations, as suggested by Ralph [41], we conduct a developer survey (Chapter 6) to see if our observations resonate with the community under analysis.

CHAPTER 8
Conclusion

In this chapter, we conclude this thesis by summarizing its contributions, discussing the broader implications of our observations for stakeholders in the code review process, and proposing promising directions for future research.

8.1 Contributions and Findings

Code review is a common software quality assurance practice. A shift towards automated detection of low-level issues has, hypothetically, freed reviewers up to focus on higher-level issues, such as software design. Yet in practice, little is known about the extent to which design is discussed during code review.

In this thesis, we set out to better understand the frequency and type of design-related discussions in code reviews. To achieve our goal, we qualitatively and quantitatively analyze design-related comments in the OpenStack Nova and Neutron projects, and then we triangulate our findings by surveying 94 software practitioners. We make the following observations:

- Design concerns are not commonly discussed in code reviews. Moreover, authors rarely engage in design-related discussions with reviewers.
- Most design issues are raised due to a lack of awareness of the global context of the codebase. However, most design-related comments are constructive, providing suggestions to mitigate the issue.

- Authors of design-related reviews tend to have lower expertise than those reviews without design-related comments, and providing valid design feedback is believed to be linked with higher software expertise.
- Design-related reviews are more likely to be abandoned than reviews without design-related comments.
- The majority of survey respondents perceive design discussions that occur during code review as an important aspect, which positively affects the author/reviewer (e.g., peer mentorship), the development team (e.g., collaborative problem-solving) and the end-product (e.g., improvements to the code quality). Others caution against it by justifying that design discussions bloat code review, and slow the development process.
- Design concerns that are associated with end-user experience (e.g., performance) resonated with the largest proportion of survey respondents.

8.2 Practical Implications

Our results suggest that: (a) the modern shift code review automation is not enough to ensure that design is discussed during code reviews; (b) design classifiers could be integrated into the reviewing interface to increase the transparency of the reviewing process; and (c) authors should not be intimidated to engage in design-related discussions, since reviewers usually offer constructive suggestions for improvement, and this will widen their codebase knowledge and result in a better end-product. Below, we describe the practical implications of our results for software organizations, tool developers, and authors of code changes in more detail.

8.2.1 Software Organizations

Software organizations should encourage reviewers to take design aspects into account. Only 9% and 14% of code review comments are design-related in NOVA and NEUTRON, respectively (Observation 1). Moreover, 11% and 21% of those comments elicit authors' responses in NOVA and NEUTRON (Observation 2). Despite increasing rates of automation of reviewing tasks in the OPENSTACK community, these observations suggest that design-level discussions are still rare. Indeed, automation of menial tasks is not enough to achieve high rates of high-level review discussion.

Design-related reviews are likely to struggle with their integration. We find that 67% and 65% of the design issues observed are due to a lack of awareness of the global context of the codebase in NOVA and NEUTRON, respectively (Observation 3). This may explain why we observe that those design-related reviews are significantly more likely to be abandoned (Observation 9).

8.2.2 Tool Developers

Integration of design classifiers into the reviewing interface would increase the transparency of the reviewing process. Our design classifiers can detect design discussions in past code reviews. If these classifiers are integrated into the code review user interface, integration decisions could be based on a clearer understanding of the review process. This could be combined with risk estimates (e.g. the impact of system deliverables [54]) to help determine whether the rigour of the code reviewing process is too low to integrate the corresponding code changes into the codebase.

8.2.3 Authors of Code Changes

Authors should not be intimidated to engage in design-related discussions. Authors rarely engage in design-related discussion (Observation 2). We conjecture that author-reviewer power dynamics may be at play, since reviewers have authority over the acceptance of code changes. However, in design-related reviews, reviewers mostly offer constructive feedback to help authors handle their design issues (Observation 4), which in return will broaden their global knowledge of the codebase (Observations 3, 15).

8.3 Opportunities for Future Research

Below, we outline several promising directions for future research.

8.3.1 Can we improve our automatic classification?

In chapter 5, we automatically label design-related comments to perform a large-scale analysis of design discussions during code reviews. Other code review characteristics and metrics can be used to improve the classifiers' performance, such as the author, the reviewer, their project-specific expertise level, the component of the code being discussed, the number of files, and file names, etc. More accurate and precise classifiers will result in more solid findings.

8.3.2 How do design discussions differ between open-source and proprietary settings?

In Chapters 4 and 5, we study the OpenStack Nova and Neutron projects qualitatively and qualitatively. This can be replicated in a proprietary setting, where developers are often under direct pressure by customer demands, as well as financial and time constraints. Our initial conjecture is that proprietary software organizations

may prioritize other concerns such as user-experience, performance, and functional aspects, and thus, discuss design during code reviews even more than open-source projects.

8.3.3 Is design discussion associated with design or software quality?

In Chapter 5, design classifiers are used to scale up our qualitative study, since manually classifying the entire dataset of code reviews is practically infeasible. However, the development of a tool that comprises a project- and team-specific design classifier can also be used to determine the level of rigour in the code reviewing process. Using these values, we may be able to study how design discussions associate with concepts like design quality (e.g., incidence rates of design anti-patterns) or software quality (e.g., incidences of post-release defects). This is inspired by prior work, which demonstrated how different aspect of the code reviews are associated with the quality of the software [25, 30, 31, 35, 49], as well as our Observations 10 and 16 that discussing design is beneficial to the development team, and is linked to the quality of the software produced.

8.4 Replication Package

To enable future work, we have made the manually classified reviews and the scripts that we used to analyze them available online.[1]

[1] `https://github.com/software-rebels/DesignInCodeReviews-ESEM2018`

Appendix A

Table 8–1: Open Coding for the Importance of Design.

Category	%	Survey Response Example
Code Cleanliness	71%	
Reusability	5%	"When you organize the code and design it well , the team will be able to reuse some of it . That will help us have an easy maintainable product ."
Maintainability	36%	"I am a strong believer in clean code. Structure and relationships are important when it comes to maintenance and code that's clean."
Extensibility	14%	"Design is also heavily impacted by the purpose of the software. If the purpose were to be extended, the design would need to allow for this."
Code Organization	5%	"Because design must be well understood when reviewing code. It helps in better understanding and may result in review comments that would change the code structure to a better one."
Understandability	11%	"Understanding how the system is meant to work is important in reviewing someone's attempt to change how it actually works."

Performance	12%	
Scalability	2%	"Proper design allows the code to be easily modified and scaled later on"
Efficiency	10%	"Design of code is important for maintenance, memory and feature extension in big projects."
Execution Tracing	11%	
Traceability	6%	"If the implementation is not following the design, this will cause integration problems, as well as untraceable issues as the code is not following the design, hence not following the sw requirements."
Debugability	1%	"The design of the code has effects beyond just the functionality delivered. It affects [...] how easy it would be to debug the code."
Testability	4%	"I often find a more-structured design results in clearer unit and functional tests for the code, also the code coverage of these designs tend to be higher."
Source of Defects	5%	"Code Chaos is the main source of bugs"
Timing of Design	27%	
In code review	4%	"Design is very important in multi-tiered applications so it comes naturally in a code review."

Before code review	15%	"Design should be agreed on before code review"
Revisited in code review	8%	"Sometimes, design and architecture decisions should be made prior to starting the coding process. The review should then validate the adherence to those."
Definition of Design	26%	
Important	21%	"Design determines everything from variable names right up to the entire architecture of the solution itself."
Not top priority	4%	"I chose not to mark it as a 5 because that would place as at the top of the importance list. Although I do think it is an important factor to look for, I would say there are a few other things to look for in a review which are more beneficial."
Can't be automatically checked	1%	"Design is something that is hard to check automatically, so humans must check that during code-review"
Cost of Design	21%	
Cost of good design	11%	"Design is very important for the long term health of the project and the proper functioning of code. But - to quote an old aphorism - we must not "let the perfect become the enemy of the good"."
Cost of bad design	5%	"Without design, people can fall to suboptimal solutions more easily."

Technical Debt	3%	"Bad design can affect app performance and also cause future code debt"
Discussion != good design	1%	"Discussion is likely necessary but is not a requirement of "design process". Surprisingly good design can come even from individuals without discussion. Also bad designs could be results of many discussions"
Design is impractical	1%	"most of pre-designed projects [..] have less issues after deployment but to be honest designing takes time, that sometimes it could be impractical in some proof-of-concept projects"
Design Knowledge	11%	
Clear Communication	5%	"Code review is the best opportunity to build a shared understanding and consensus on the design."
Power Dynamic	1%	"people tend towards leniency in the reviews, people, especially developers are not amazing communicators, so they are afraid of being 100% open and honest."
Easier Onboarding	2%	"A good, consistent design helps newer project participants to become contributors more quickly, in my experience."
Shared Code Ownership	3%	"Discussing it during code reivew help the verification of the understandability of the design (which is important in corporate since it is rarely the case that only one person write and maintain the code)."

Appendix B

Table 8–2: Open Coding for the Importance of discussing design during code review.

Category	%	Survey Response Example
Team Gains	74%	
Consensus	22	"Given both reviewer and author have strong opinions about their ideas or design decisions, [code review] could be beneficial."
Opportunity for growth	19	"Design decisions must be discussed. It is a learning opportunity for both the reviewer and the reviewee"
Sync up	9	"Discussion will put all parties on the same page."
Group problem solving	9	"To have a fruitful discussion about how to fix this."
Information sharing	7	"[so that] software engineers know what are the constraints, limitations, and performance of the code they develop."
Understanding trade-offs	4	"Both parties should understand the concern that is raised and the various tradeoffs each solution will give."
Understandability of design	4	"Code review is very important for the development team to understand design."
Personal Gains	52%	

Rationale	15	"We have to know the reason behind this implementation."
Other Perspectives	13	"Because it's very important for the reviewer to understand the author's point of view and vice versa. As that would reach an ultimate solution."
Feedback for author	9	"As an author, I gain experience by discussing with my code reviewer. It's like supervision, and it highlights the mistakes that I was unaware of."
Avoid repetition of mistakes	8	"All parties need to get an understanding of why the design was selected and what the resolution is in order to prevent the same issues occuring again."
Power dynamic	7	"Two sided discussions are very important, if people have egos about their software and reject improvements, the product will suffer."
Product Gains	48%	
Product Quality	22	"Two reasons: to enhance the overall quality of the end product, and [...]"
Maintainability	14	"Its important to get a high quality usable maintainable code."
Prevent Design Erosion	6	"Design should be maintained through all code merges."
Reusability	6	"because everyone needs to understand the issue and why it is important to avoid such thing for code reading and reusability"
Against Discussion	16%	

Trivial	9%	"Sometimes the concern and resolution are both obvious, and no discussion is needed.", "the importance depend on the complexity of the review, if the review is trivial, a discussion can be a bad allocation of time."
Resolve Later	3%	"In a review, you do not discuss. You highlight the issue and move on. Resolution is done after the review."
Deadlines	2%	"Sometime, there is a tight deadline, so we pass the design in the review.", "It is important but sometimes developers don't have time for that and only care for a working code for immediate submission."
Intended design != correct functionality	2%	"Because following intended design is not necessary for working software in most cases."

References

[1] Rana Alkadhi, Manuel Nonnenmacher, Emitza Guzman, and Bernd Bruegge. How do developers discuss rationale? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 357–369. IEEE, 2018.

[2] Guzzi Anja, Bacchelli Alberto, Lanza Michele, Pinzger Martin, and van Deursen Arie. Communication in open source software development mailing lists. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 277–286, May 2013.

[3] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 international conference on software engineering*, pages 712–721. IEEE Press, 2013.

[4] Richard A Baker Jr. Code reviews enhance software quality. In *Proceedings of the 19th international conference on Software engineering*, pages 570–571. ACM, 1997.

[5] Vipin Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 931–940. IEEE, 2013.

[6] Tobias Baum, Olga Liskin, Kai Niklas, and Kurt Schneider. Factors influencing code review processes in industry. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 85–96, New York, NY, USA, 2016. ACM.

[7] Kent Beck, Martin Fowler, and Grandma Beck. Bad smells in code. *Refactoring: Improving the design of existing code*, pages 75–88, 1999.

[8] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. Modern code reviews in open-source projects: Which problems do they fix? In *Proceedings of the 11th working conference on mining software repositories*, pages 202–211. ACM, 2014.

[9] A. Bosu and J. C. Carver. Impact of peer code review on peer impression formation: A survey. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 133–142, Oct 2013.

[10] Amiangshu Bosu, Michaela Greiler, and Christian Bird. Characteristics of useful code reviews: An empirical study at microsoft. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 146–156, Piscataway, NJ, USA, 2015. IEEE Press.

[11] 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.

[12] Felivel Camilo, Andrew Meneely, and Meiyappan Nagappan. Do bugs foreshadow vulnerabilities?: A study of the chromium project. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 269–279. IEEE Press, 2015.

[13] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[14] Jacek Czerwonka, Michaela Greiler, and Jack Tilford. Code reviews do not find bugs: how the current code review best practice slows us down. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 27–28. IEEE Press, 2015.

[15] Fernando Brito e Abreu and Walcelio Melo. Evaluating the impact of object-oriented design on software quality. In *Software Metrics Symposium, 1996., Proceedings of the 3rd International*, pages 90–99. IEEE, 1996.

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

[17] Michael E Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 38(2/3):258, 1999.

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

[19] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[20] Tom Gilb, Dorothy Graham, and Susannah Finzi. *Software inspection*, volume 253. Addison-Wesley Reading, 1993.

[21] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 345–355, New York, NY, USA, 2014. ACM.

[22] Jennifer Greene and Charles McClintock. Triangulation in evaluation: Design and analysis issues. *Evaluation review*, 9(5):523–545, 1985.

[23] Peter Kampstra. Beanplot: A boxplot alternative for visual comparison of distributions. *Journal of Statistical Software, Code Snippets*, 28(1):1–9, October 2008.

[24] Chris F Kemerer and Mark C Paulk. The impact of design and code reviews on software quality: An empirical study based on psp data. *IEEE transactions on software engineering*, 35(4):534–550, 2009.

[25] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W Godfrey. Investigating code review quality: Do people and participation matter? In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 111–120. IEEE, 2015.

[26] J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174, 1977.

[27] Craig Larman. *Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Interative Development*. Pearson Education India, 2012.

[28] Mika V Mantyla. Developing new approaches for software design quality improvement based on subjective evaluations. In *Proceedings. 26th International Conference on Software Engineering*, pages 48–50. IEEE, 2004.

[29] Mika V. Mäntylä and Casper Lassenius. What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering*, 35(3):430–448, 2009.

[30] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 192–201. ACM, 2014.

[31] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):2146–2189, 2016.

[32] Andrew Meneely, Alberto C. Rodriguez Tejeda, Brian Spates, Shannon Trudeau, Danielle Neuberger, Katherine Whitlock, Christopher Ketant, and Kayla Davis. An empirical investigation of socio-technical code review metrics and security vulnerabilities. In *Proceedings of the 6th International Workshop on Social Software Engineering*, SSE 2014, pages 37–44, New York, NY, USA, 2014. ACM.

[33] Andrew Meneely and Laurie Williams. Secure open source collaboration: An empirical study of linus' law. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 453–462, New York, NY, USA, 2009. ACM.

[34] Matthew B Miles, A Michael Huberman, Michael A Huberman, and Michael Huberman. *Qualitative data analysis: An expanded sourcebook*. sage, 1994.

[35] 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 *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 171–180. IEEE, 2015.

[36] 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.

[37] Stacy Nelson and Johann Schumann. What makes a code review trustworthy? In *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on*, pages 10–pp. IEEE, 2004.

[38] Kerzazi Noureddine, Khomh Foutse, and Adams Bram. Why do automated builds break? an empirical study. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 41–50, Sept 2014.

[39] Matheus Paixao, Jens Krinke, DongGyun Han, Chaiyong Ragkhitwetsagul, and Mark Harman. Are developers aware of the architectural impact of their changes? In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 95–105. IEEE Press, 2017.

[40] Rigby Peter C. and Storey Margaret-Anne. Understanding broadcast based peer review on open source software projects. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 541–550, May 2011.

[41] Paul Ralph. Toward methodological guidelines for process theories and taxonomies in software engineering. *IEEE Transactions on Software Engineering*, 2018.

[42] Jack W Reeves. What is software design. *C++ Journal*, 2(2):14–12, 1992.

[43] Jason Remillard. Source code review systems. *IEEE software*, 22(1):74–77, 2005.

[44] Peter C Rigby, Daniel M German, and Margaret-Anne Storey. Open source software peer review practices: a case study of the apache server. In *Proceedings of the 30th international conference on Software engineering*, pages 541–550. ACM, 2008.

[45] Guoping Rong, Jingyi Li, Mingjuan Xie, and Tao Zheng. The effect of checklist in code review for inexperienced students: An empirical study. In *Software Engineering Education and Training (CSEE&T), 2012 IEEE 25th Conference on*, pages 120–124. IEEE, 2012.

[46] Louis Rosenfeld, Peter Morville, and Jakob Nielsen. *Information architecture for the world wide web.* " O'Reilly Media, Inc.", 2002.

[47] Todd Sedano, Paul Ralph, and Cécile Péraire. Software development waste. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 130–140, Piscataway, NJ, USA, 2017. IEEE Press.

[48] Mini Shridhar, Bram Adams, and Foutse Khomh. A qualitative analysis of software build system changes and build ownership styles. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering*

and Measurement, ESEM '14, pages 29:1–29:10, New York, NY, USA, 2014. ACM.

[49] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. Investigating code review practices in defective files: An empirical study of the qt system. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 168–179, Piscataway, NJ, USA, 2015. IEEE Press.

[50] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. Review participation in modern code review. *Empirical Software Engineering*, 22(2):768–817, Apr 2017.

[51] 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.

[52] Jason Tsay, Laura Dabbish, and James Herbsleb. Influence of social and technical factors for evaluating contribution in github. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 356–366, New York, NY, USA, 2014. ACM.

[53] John Vlissides, Richard Helm, Ralph Johnson, and Erich Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49(120):11, 1995.

[54] Ruiyin Wen, David Gilbert, Michael G. Roche, and Shane McIntosh. BLIMP Tracer: Integrating Build Impact Analysis with Code Review. In *Proc. of the International Conference on Software Maintenance and Evolution (ICSME)*, page To appear, 2018.

[55] Aiko Yamashita and Leon Moonen. Do developers care about code smells? an exploratory survey. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 242–251. IEEE, 2013.

[56] Tao Yida, Han Donggyun, and Kim Sunghun. Writing acceptable patches: An empirical study of open source project patches. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 271–280, Sept 2014.

[57] Edward Yourdon. *Structured walkthroughs*. Yourdon Press, 1989.

[58] 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.