UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

VICTOR DA CUNHA LUNA FREIRE

CHARACTERIZATION OF DESIGN DISCUSSIONS IN MODERN CODE REVIEW

CAMPINA GRANDE - PB
2021

Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Characterization of Design Discussions in Modern Code Review

Victor da Cunha Luna Freire

Tese submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Doutor em Ciência da Computação.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Engenharia de Software

João Arthur Brunet Monteiro e Jorge Cesar Abrantes de Figueiredo

(Orientadores)

Campina Grande, Paraíba, Brasil

F866c Freire, Victor da Cunha Luna.

Characterization of design discussions in modern code review / Victor da Cunha Luna Freire. – Campina Grande, 2021.

87 f. : il. color.

Tese (Doutorado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2021.

"Orientação: Prof. Dr. João Arthur Brunet Monteiro, Prof. Dr. Jorge Cesar Abrantes de Figueiredo".

Referências.

1. Engenharia de Software. 2. Revisão de Código. 3. Revisão de Código Moderna (MCR). 4. Design de Software. 5. Discussão de Design. I. Monteiro, João Arthur Brunet. II. Figueiredo, Jorge Cesar Abrantes de. III. Título.

CDU 004.41(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA SEVERINA SUELI DA SILVA OLIVEIRA CRB-15/225

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
POS-GRADUACAO CIENCIAS DA COMPUTACAO
Rua Aprigio Veloso, 882, - Bairro Universitario, Campina Grande/PB, CEP 58429-900

FOLHA DE ASSINATURA PARA TESES E DISSERTAÇÕES

VICTOR DA CUNHA LUNA FREIRE

CHARACTERIZATION OF DESIGN DISCUSSIONS IN MODERN CODE REVIEW

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação como pré-requisito para obtenção do título de Doutor em Ciência da Computação.

Aprovada em: 17/08/2021

Prof. Dr. JOÃO ARTHUR BRUNET MONTEIRO, Orientador, UFCG

Prof. Dr. JORGE CESAR ABRANTES DE FIGUEIREDO, Orientador, UFCG

Prof. Dr. TIAGO LIMA MASSONI, Examinador Interno, UFCG

Prof. Dr. LEANDRO BALBY MARINHO, Examinador Interno, UFCG

Prof. Dr. MARCO TÚLIO DE OLIVEIRA VALENTE, Examinador Externo, UFMG

Prof. Dr. UIRA KULESZA, Examinador Externo, UFRN

Resumo

Revisão de Código Moderna (MCR) é uma atividade leve cada vez mais popular para melhorar a qualidade do software. Na MCR, os desenvolvedores participam de várias discussões que são armazenadas em ferramentas de suporte a MCR. Ao analisar essas discussões, pesquisadores descobriram que há uma quantidade considerável de informações de design dentro delas. Eles também obtiveram resultados positivos nos seus estudos de técnicas para identificar automaticamente as discussões de design nas revisões. No entanto, a maior parte desta pesquisa é quantitativa e não analisou a fundo questões como, por exemplo, como os desenvolvedores conduzem discussões de design e que tipo de informação de design eles discutem. Para recuperar informações de design de forma mais eficaz nas discussões de revisão de código, é necessário saber como os desenvolvedores discutem design durante as revisões de código para poder distinguir as informações de design do resto. Além disso, é necessário saber que tipo de informação de design existe nessas discussões e qual é a sua forma. Com o objetivo de compreender melhor a MCR e o processo de como design é discutido na MCR a fim de preencher as lacunas de conhecimento atuais, realizamos um estudo qualitativo para caracterizar as informações de design na MCR por meio da aplicação de Straussian Grounded Theory (GT) a um conjunto de dados de projetos de software de código aberto (OSS) da Apache Software Foundation. Como resultado, produzimos um modelo de como os desenvolvedores discutem design durante a revisão de código, uma classificação dos tipos de informações de design discutidas na MCR e uma base de dados de discussões de design. Acreditamos que nosso trabalho será de grande ajuda em pesquisas futuras que objetivem extrair informações de design de discussões da MCR de uma maneira que seja útil para os profissionais.

Palavras-chave: revisão de código; revisão de código moderna; design de software; discussão de design.

Abstract

Modern Code Review (MCR) is an increasingly popular lightweight activity for improving software quality. As part of MCR, developers participate in a number of discussions which are stored in tools for supporting the process. By analyzing these discussions, researchers found that there is a considerable amount of design information within them. They also had positive results in their studies of techniques for automatically identifying design discussions in the reviews. However, most of this research is quantitative and has not thoroughly analyzed questions such as how developers conduct design discussions and what topics of design they discuss. To retrieve design information more effectively from code review discussions, it is necessary to know how developers discuss design during code reviews in order to be able to distinguish design information from the rest. Furthermore, it is necessary to know what kind of design information exist in these discussions. With the goal of better understanding MCR and the process of how design is discussed in MCR in order to fill the current knowledge gaps, we performed a qualitative study to characterize design information in MCR by applying Straussian Grounded Theory (GT) to a dataset of design discussions from open source software (OSS) projects of the Apache Software Foundation. As a result, we produced a model of how developers discuss design during code review, a classification of types of design information discussed in MCR and a dataset of design discussions. We believe our work will be of significant help in future research aiming to extract design information from MCR discussions in a manner that is useful to practitioners.

Keywords: code review; modern code review; software design; design discussion.

Agradecimentos

Este trabalho não seria possível sem a ajuda inestimável de inúmeras pessoas. Gostaria de agradecer principalmente a:

- Aos meus pais, Pedro Aurélio e Rosângela, e ao meu irmão, Leonardo,
- À minha amada, Maraíza,
- Aos demais parentes,
- Aos meus orientadores, João Arthur e Jorge Abrantes,
- Aos membros da banca avaliadora, Tiago Massoni, Uirá Kulesza, Marco Túlio Valente e Leandro Marinho,
- Aos demais professores do DSC, especialmente: Dalton Serey, Franklin Ramalho, Joseana Fechine, Andrey Brito, Jacques Sauvé,
- Aos amigos e colegas do projeto ePol e do SPLAB,
- Aos funcionários da COPIN e do SPLAB, especialmente, Lilian,
- Aos amigos, Felipe, Alana, Erick, Poliana, Corina, Arthur, Davi, Diego Pedro, Guilherme e Laércio,
- Ao professor Kerly Monroe,
- À CAPES pelo auxílio financeiro,
- A todos que me ajudaram nessa trajetória e que ainda não foram citados.

Muito obrigado!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Code review is a very popular activity for improving software quality. It originated in 1976 with the work of Fagan on the first code review process [17]. Fagan's code review process and similar formal ones are known as software inspections and several studies have provided evidence of their efficacy. However, despite all this evidence, practitioners tend to not use inspections because of how costly and time-consuming they are [12][27][47].

In contrast to the early code review processes which were formal and costly, lightweight code review processes known as Modern Code Review (MCR) were created and they are now commonplace, especially in open source software (OSS) projects [2]. MCR usually consists of reviewing changesets in a distributed and asynchronous manner with the assistance of specialized tools [22][23][41]. Even though MCR processes requires less time than software inspections, they still positively affect software quality [2][5][7].

We conducted a systematic mapping study of MCR to obtain an overview of the research in the field of MCR (Appendix A). We identified and classified 177 studies about MCR that were published between the years of 2002 and 2018 to create a map of the published peer-reviewed research on MCR. Principally, we observed that the field of MCR has been growing fast and steadily since 2011 and that a wide variety of topics within MCR are being studied. In particular, we noticed that the topic of design discussions within MCR needs more research.

A large number of discussions about the software is often generated as a consequence of using MCR. Because of the tendency of MCR to be distributed and asynchronous, the developers usually discuss the changesets under review in writing and these discussions are commonly stored for future reference.

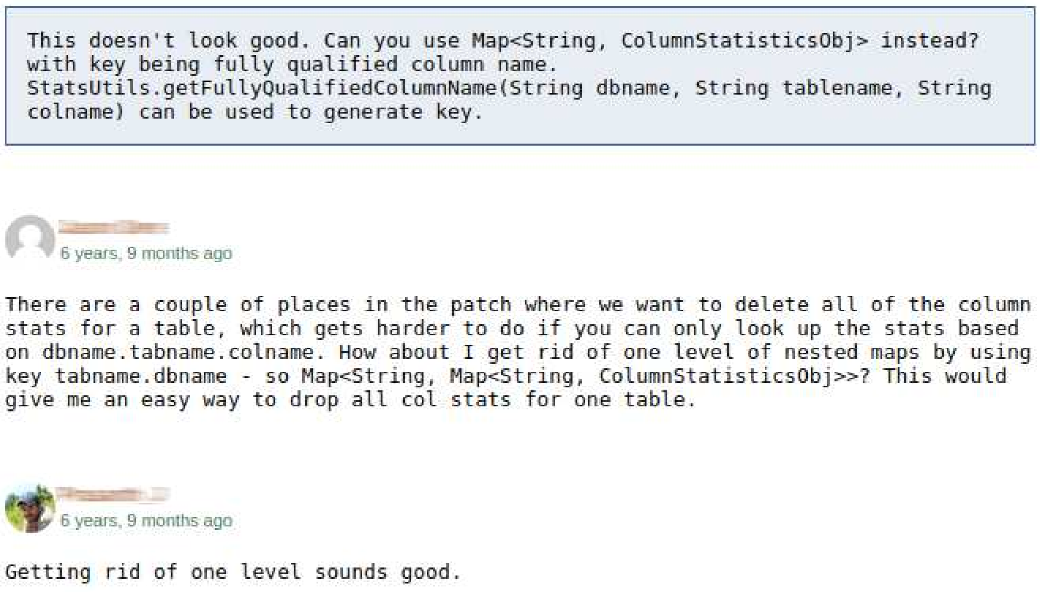
By analyzing these discussions, researchers have found that there is a considerable amount of design information present in MCR discussions, although the quantity of such information varies in each project. Sutherland and Venolia found that design information was abundant in the code review emails at Microsoft and that those emails were used by project members to retrieve rationales for past design decisions [50]. Brunet et al. analyzed 102,122 discussions from 77 projects hosted in GitHub and found that on average 25% of discussions have design information [9]. Zanaty et al. found that an average of 12% of comments have design information in a sample of 1,357 comments from OpenStack Nova and 1,460 comments from OpenStack Neutron projects [58].

1.2 Problem

While a number of studies have investigated design information in MCR, these have mostly used quantitative methodologies and usually focus on automatically identifying design discussions. Indeed, few studies have investigated how developers conduct design discussions in MCR and what kind of design information is present.

Merely identifying which discussions and comments are about design in a review is not enough for many purposes. For instance, a discussion might be about a proposed design change which the developers rejected in the end. These rejected design changes would likely not be useful if a developer wanted to extract design rules from the discussions. Figure 1.1 shows a real example of such a discussion. The reviewer initially suggests using a simpler data structure but this is rejected by the developer. However, the developer subsequently proposes another design change which the reviewer agrees with.

Before designing tools to leverage the design information in MCR, it is necessary to know what type of design information is present in the discussions. For example, if developers often talk about architectural concerns in MCR, then it might be possible to extract architectural rules from these discussions. Conversely, if the discussions are mostly about low-level design such as method signatures and class responsibilities, then other solutions



```
This doesn't look good. Can you use Map<String, ColumnStatisticsObj> instead?
with key being fully qualified column name.
StatsUtils.getFullyQualifiedColumnName(String dbname, String tablename, String
colname) can be used to generate key.
```

[redacted]
6 years, 9 months ago

```
There are a couple of places in the patch where we want to delete all of the column
stats for a table, which gets harder to do if you can only look up the stats based
on dbname.tabname.colname. How about I get rid of one level of nested maps by using
key tabname.dbname - so Map<String, Map<String, ColumnStatisticsObj>>? This would
give me an easy way to drop all col stats for one table.
```

[redacted]
6 years, 9 months ago

```
Getting rid of one level sounds good.
```

Figure 1.1: An example of a discussion with a proposed design change which was eventually rejected.

focusing on this kind of information must be evaluated.

Thus, there is a gap in the research regarding how developers discuss design during MCR and what types of design information are discussed. We hypothesize that filling this knowledge gap would be valuable for future research on several open problems in the area such as (1) automatically extracting design information to produce documentation, (2) automatically extracting design information to verify design conformance and (3) tools with better support for reviewing design in MCR.

1.3 Solution

Our goal is to achieve a deep understanding of the process of design discussion during MCR and of the content present in design discussions. Therefore, we performed a characterization study of design discussions in MCR. In particular, we wanted to investigate the following research questions:

- **RQ1:** How do developers discuss design during code review?
- **RQ2:** What topics of design are discussed during code review?

Before conducting this characterization study, we had to obtain a dataset of design discussions in MCR. To this end, we replicated studies which quantitatively analyze the presence of design discussions in MCR. Specifically, we conducted a study where we created an automatic binary classifier capable of recognizing which discussions are about design and applied it to 267,843 review discussions from the Apache Software Foundation (ASF). As a result, we found further evidence that design is frequently discussed in MCR and we built a public dataset of 108,458 design discussions.

Straussian Grounded Theory (GT) is a qualitative research methodology which is well suited to examine research questions like RQ1 and RQ2. The GT research methodology enables researchers to investigate not only how a process happens but also what happens in it [15].

For this reason, we used GT to analyze the dataset of design discussions from the the ASF. We examined 180 review requests which had design information and analyzed 30 of them in depth to generate as a result: a model of how developers discuss design during MCR and a classification of types of design discussed in MCR.

We observed that design review is a significant part of MCR which happens implicitly in code review discussions. Furthermore, we found that design review can be seen as a process where developers engage in design discussions over design concerns throughout MCR while performing a number of review actions which affect these design concerns. Moreover, our classification of types of design discussed in MCR is composed of 93 concepts representing types of design concerns, which were grouped into a hierarchy that mainly splits them into two major categories: design changes and design issues. We also identified 45 concepts representing comments that were not about software design primarily because these will help other researchers clearly understand what we considered as being software design.

1.4 Contributions

Briefly, the main contributions of this work are:

- A model of how developers discuss design during MCR;
- A classification of types of design information discussed in MCR;

- A public dataset of 108,458 design discussions from the Apache Software Foundation that can be used in other studies.

1.5 Outline

This thesis is structured as follows: Chapter 2 provides the necessary background for understanding this work, Chapter 3 presents our study of automatically identifying design discussions which produced the dataset used in the characterization study, Chapter 4 describe our characterization of design in MCR and Chapter 5 concludes this work by outlining what we found and what the implications are.

Chapter 2

Background

2.1 Modern Code Review (MCR)

In the past decade, lightweight code review processes known as Modern Code Review (MCR) [2] became increasingly more popular not only on the OSS projects where they originated but also in proprietary software. In contrast to traditional software inspections, MCR is usually performed in a distributed and asynchronous manner with the assistance of specialized tools [22][23][6]. These characteristics have led MCR to be less costly than inspections and, as a result, it enjoys a higher popularity.

During MCR, it is common for at least part of the discussions to happen in text and within a MCR tool. Therefore, ample data is generated during these discussions which could be leveraged to improve the software process.

Although there are many tools for performing MCR and the review processes of each software development team have their own particularities, the underlying review process does not change much (Figure 2.1) [21]. After a developer has performed a set of code changes in the database which he finds satisfactory, he uses an MCR tool to group these as a changeset and he shares this changeset with others developers who will act as reviewers. Then, the reviewers read the changeset and write comments where they ask for clarification, identify bugs, suggest better solutions and etc. Afterwards, the author of changeset respond to these comments and improve the code according to the discussions. This exchange between the author of the changeset and the reviewers continues until the reviewers are either satisfied with the changeset and accept it or they do not approve of it and reject it.

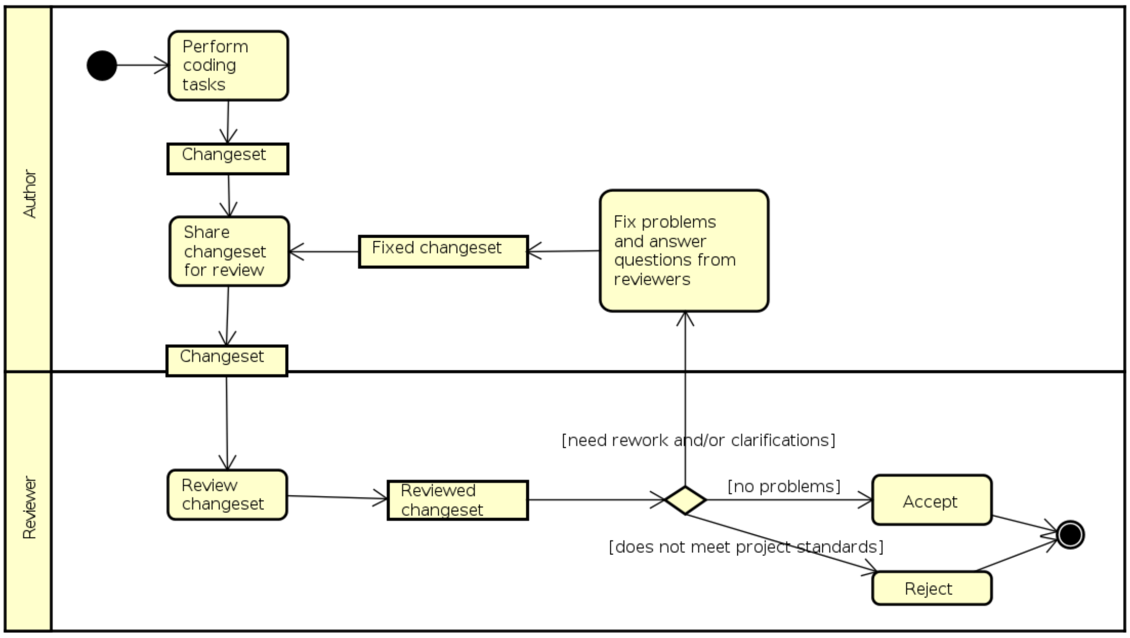

Author
Reviewer
Perform coding tasks
Changeset
Share changeset for review
Fixed changeset
Fix problems and answer questions from reviewers
Changeset
Review changeset
Reviewed changeset
[need rework and/or clarifications]
[no problems]
Accept
[does not meet project standards]
Reject

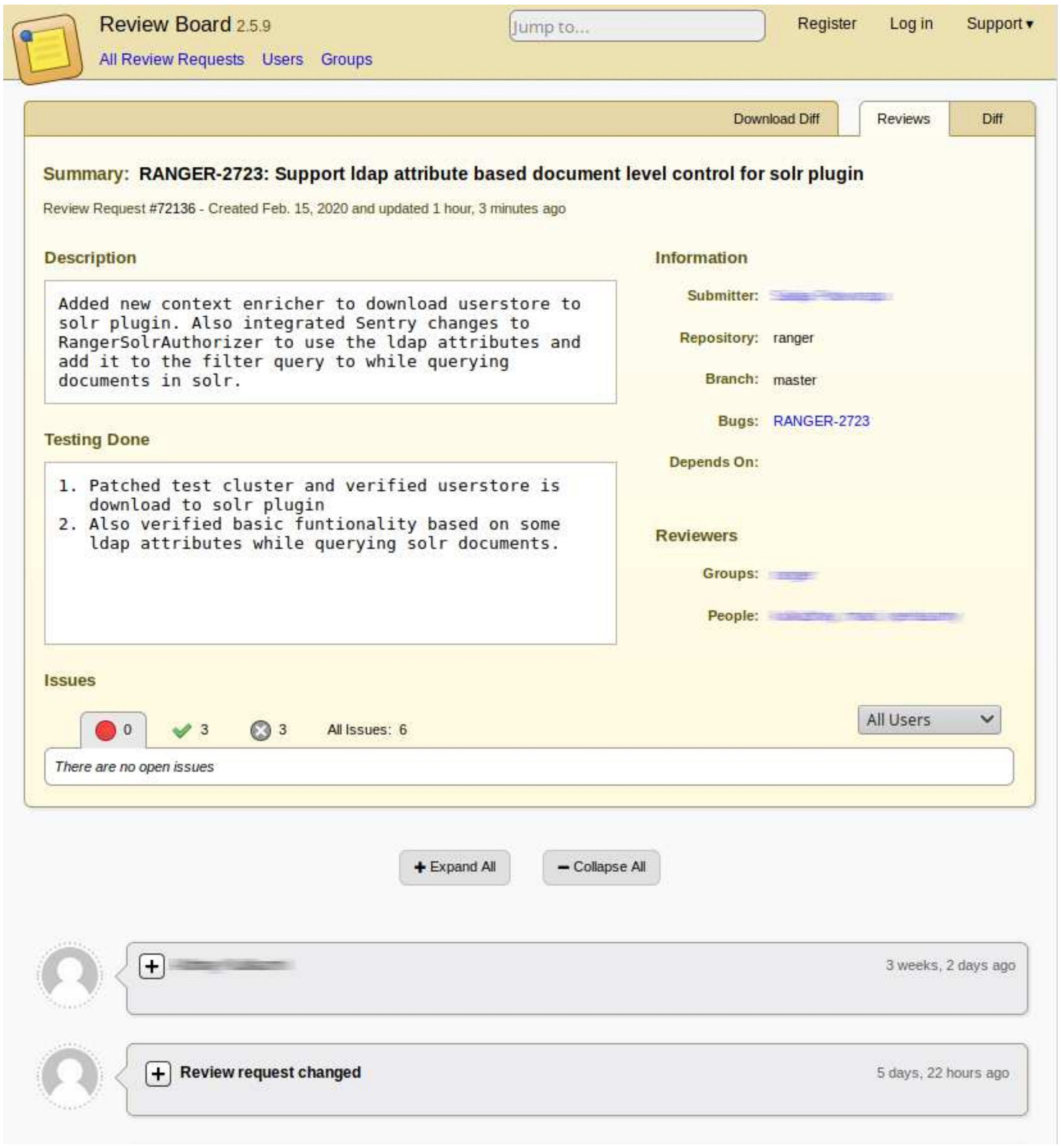
Figure 2.1: Workflow of modern code review. [21]

2.1.1 Review Board

Review Board is a OSS tool for supporting the process of MCR [6]. It allows developers to share changesets, discuss them and decide whether to approve and reject these changesets.

The core concept of Review Board is a review request, which is how they represent a changeset. Review requests contain metadata (e.g. name of the author of the review request, summary of the changes), screenshots, attachments and, most importantly, the files and diff to be reviewed.

Developers add reviews to a review request throughout the MCR process. Each review contains general comments about the whole changeset and/or diff comments which are about specific parts of the diff. It is also possible to reply to these general comments or to the diff comments.



Summary: RANGER-2723: Support ldap attribute based document level control for solr plugin

Review Request #72136 - Created Feb. 15, 2020 and updated 1 hour, 3 minutes ago

Description

```
Added new context enricher to download userstore to
solr plugin. Also integrated Sentry changes to
RangerSolrAuthorizer to use the ldap attributes and
add it to the filter query to while querying
documents in solr.
```

Testing Done

```
1. Patched test cluster and verified userstore is
   download to solr plugin
2. Also verified basic funtionality based on some
   ldap attributes while querying solr documents.
```

Information

Submitter:	
Repository:	ranger
Branch:	master
Bugs:	RANGER-2723
Depends On:	

Reviewers

Groups:	
People:	

Issues

● 0 ✓ 3 ⊗ 3 All Issues: 6 — All Users

There are no open issues

Figure 2.2: A review request of the Apache Software Foundation in Review Board. [18]

2.2 Software Design

2.2.1 Definitions

Even though design is a fundamental part of most software processes, there is not a clear and unambiguous definition of software design that is widely accepted by researchers and practitioners. Classic and seminal works in the field of software engineering (SE) present different definitions for software design.

For example, two classical SE books describe software design as a process immediately following requirement analysis which translates the requirements into a design model which will be followed during subsequent construction activities. They divide the design process into architectural design, component design, interface design and data design [42][48].

Similarly, Clemens et al., Budgen and Bourque and Fairley also see software design as being a process happening between requirements and construction, but they divide software design into architectural design and non-architectural design (detailed design) [13][10][8].

Perry and Wolf in their seminal work on software architecture used the term "design" to refer to what other authors commonly call "detailed design", i.e. the process executed after the software architecture is specified and which defines the details of each component of the architecture [38].

Likewise, Rozanski and Woods also see design as the process which is executed after the architecture is defined. In addition, they believe that design is exclusively concerned with translating the system requirements into a specification for building the system and therefore does not consider the needs of all stakeholders. In their view, architecture definition (architectural design) acts as a bridge between requirements analysis and design [44].

The lack of a clear, precise and widely accepted definition of design negatively impacts both research and practice. In research, a lack of such definition prevents the creation of an integrated body of knowledge on design because studies using different definitions might be incompatible with one another. In practice, confusion over what design is or not might lead to conflicts among the software engineers and stakeholders.

Considering the importance of having a clear and detailed definition of software design, we adopted in this work the definition proposed by Ralph and Wand which aims to fulfill this need [43]. Ralph and Wand identified 33 definitions for design in the literature (including design in other fields) and analyzed them using four quality criteria: coverage, meaningfulness, unambiguousness and ease of use. All of the 33 definitions had problems according to one or more the chosen quality criteria. Given these results, Ralph and Wand proposed a new definition for design that builds upon these existing definitions but which aims to be clear and precise and to satisfy the aforementioned four quality criteria.

Ralph and Wand define design as both a noun and a verb. As a noun, design is "a specification of an object, manifested by some agent, intended to accomplish goals, in a particular environment, using a set of primitive components, satisfying a set of requirements, subject to some constraints. As a verb, to design is "to create a design, in an environment (where the designer operates)".

They explained each of the highlighted concepts in their definition. The specification is a detailed structural description of how the object is to be built using the set of available primitives. Primitives are the basic components available to the design agent which can be used to build the object. The environment means either the object environment which is where the object will operate or the agent environment which is where the agent works. Requirements are structural or behavioral properties required of the object. Goals differ from requirements in that they are more abstract statements indicating what the stakeholders wish the design object accomplishes, i.e. goals specify the impact of the object in the object environment. Constraints are similar to requirements and they are structural or behavioral restrictions on the object.

To put these concepts in perspective, let us consider an example in the context of software engineering where the design object is a software system for the payroll of a business. In this example, the primitives would include modules, classes, relationships, data structures and algorithms. The design agents are a group of senior software engineers. The goal is that the stakeholders expect the system to automate the payroll while complying with governmental regulations. Part of the behavioral requirements are that the system tracks previously made payments and that users can examine this data. A structural requirement is that the system must interface with another human resources system within the company. The environment for the payroll system is the business and it includes other in-house systems and a set of outside systems that it must interact with such as the government tax system. A constraint is that the system must be fast enough to process the payments in the first five days of each month.

Design is usually separated into high-level design (software architecture) and low-level design (detailed design or non-architectural design). We explain this separation in the next two subsections.

2.2.2 Software Architecture (High-level Design)

The architecture of a software is a set of architectural elements and their relationships. Its goal is to satisfy the functional and non-functional requirements (quality properties) of the software. Architectural elements are often called modules and components and each element is responsible for a set of features which are accessible by well defined interfaces [44].

Although every software has its own architecture, not all of them has this architecture properly documented. The documentation of an architecture is called an architectural description.

There are multiple models for creating an architectural description but most of them use the concept of views to manage the complexity of architecture [44][13]. Each of these views focuses on a specific part of the architecture so as not to overwhelm the readers of the documentation with information.

Rozanski and Woods presents a process for creating an architectural definition which revolves around the concepts of stakeholders, views, viewpoints and perspectives in their. More specifically, stakeholders are the individuals or groups interested in the software, a view is a representation showing certain aspects of the architecture and how these address the software requirements, a viewpoint is a template for creating a view and a perspective is a set of guidelines similar to a viewpoint but it affect multiple views and helps the architecture meet a desired quality property such as security, performance and availability [44].

They introduce 7 viewpoints for creating views for an architectural definition, namely: context, functional, information, concurrency, development, deployment, and operational. For example, the context viewpoint helps architects create a view that specifies the scope of the software and how it will interact with its environment (other software, organizations and people) [44].

As for perspectives, they describe 9 of them: security, performance and scalability, availabilty and resilience, evolution, accessibility, development resource, internationalization, location, regulation and usability. To illustrate, the security perspective presents a set of activities and guidelines on how to ensure the architecture meets the security needs of the stakeholders such as controlling access to information, limiting the effect of potential security breaches and auditing data modifications [44].

2.2.3 Detailed Design (Low-level Design)

Low-level design, which is also known as non-architectural design and detailed design, consists of specifying each architectural element in detail. In the case of object-oriented software design, this step would define a number of elements such as classes, interfaces and methods and how they compose each of the architectural elements [48].

There is not a clear boundary between architecture and detailed design. Not only do authors have different definitions on what separates them but also these definitions are often not clear enough to determine with certainty if an element is part of either the architecture or the detailed design.

For example, Clements et al. believes the architect establishes the boundary between the two in each software project. Design decisions which enable the software to meet its requirements are considered architectural decisions and the other decisions are non-architectural decisions. Thus, software architecture is the set of architectural decisions and detailed design is the set of non-architectural decisions [13].

At the same time, Budgen states that the architecture is an abstract model of the solution which does not include details of the architectural elements and that detailed design is concerned with the details of each of these architectural elements [10].

2.3 Machine Learning and Natural Language Processing

In this section, we focus on the core concepts of machine learning and NLP necessary to understand the problem of binary classification of text.

2.3.1 Supervised Learning

Supervised learning consists of generating an approximate function (hypothesis) $h(x)$ that is as close as possible to a real function $f(x)$ when given a set of input-output pairs (x_i, y_i) where $y_i = f(x_i)$. When the output of $f(x)$ is restricted to a finite set of values, the problem is said to be a classification problem[45].

Over the years, researchers have designed a plethora of supervised learning algorithms for generating a hypothesis that approximates a real function, e.g. decision tree, random forests, naive bayes, logistic regression, gradient descent, neural networks. There is not an algorithm that works best in every domain, so multiple learning algorithms must be evaluated when trying to build a classifier.

Explaining the workings of all the supervised algorithms used in this work would be beyond the scope of this document. But, knowing their workings is not required to understand this work because we use them mostly as black boxes.

2.3.2 Tokenization

In NLP, tokenization is the process of splitting text into words and sentences [28]. Considering how sentences in the English language are formed, this seems to be a very simple task at first. To divide sentences, an algorithm could just use the naturally occurring periods, question marks and exclamation points. Likewise, words in a sentence could be divided by just using the spaces between them.

However, a more careful examination will reveal a number of problems with this trivial approach. For instance, text, especially in code reviews, is often fraught with spelling and grammatical errors, e.g. missing spaces between words, misspelled words and missing punctuation. Therefore, a tokenization algorithm that is to be used with code review discussions needs to be designed for handling such errors.

Another example of a problem with this trivial approach would be the multiple meanings of periods in text. Periods are not used solely for marking the end of sentences. In fact, they are also used with completely different meanings in decimal numbers and abbreviations such as "3.14" and "i.e.".

Even the use of whitespace for separating tokens is not as straightforward as it seems.

For example, composite names such as "Rio Grande do Norte" could either be considered as four separate tokens ("Rio", "Grande", "do", "Norte" or as a single token "Rio Grande do Norte". Each application will have to decide on the best approach according to its needs.

Thus, given the complexity of this problem and the different needs of each application, there is a wide variety of tokenization algorithms available.

2.3.3 Lemmatization

Every word has an associated root word called its lemma and the process of converting words to their lemmas is called lemmatization. Concretely, the lemmas for "playing" and "went" are respectively "play" and "go" [28].

Lemmatization is often confused with stemming. Stemming is similar to lemmatization in that it returns the root or stem of the word. However, stemming does not consider the context in which the word appears, nor does it use a dictionary of the language. To illustrate, the stem of "went" is "went", while its lemma is "go".

2.3.4 Bag of Words and N-grams

Since many machine learning algorithms require numerical features, the list of tokens extracted from a text must be quantified somehow. One approach to this is the bag of words model. In this model, each document is represented by a vector containing the frequencies of each word in the document. Each position in the vector corresponds to a token and the size of the vector is equal to the number of different tokens present in the dataset [45].

One of the main drawbacks of the bag of words model is that it discards the order of words in the original document. Consequently, sentences such as "is it ready?" and "it is ready." would be mapped to the same frequency vector. Depending on the desired application, this loss of ordering information can have a significant negative impact on the results.

To preserve some of the ordering, researchers have proposed the n-grams language model [45]. N-grams consist of storing sequences of words of size n, so that the amount of ordering information to preserve can be controlled with n. Concretely, the sentence "it is ready." has three 1-grams ("it", "is", "ready"), two 2-grams ("it is", "is ready") and one 3-gram ("it is ready"). Notice that with an n of size 3, the whole ordering of that small sentence is preserved. Further, some n-grams have special names such as unigrams for 1-grams, bigrams for 2-grams and trigrams for 3-grams.

2.3.5 TF-IDF

Tf-idf is a statistic created to avoid giving too much importance to common but unimportant words such as articles and prepositions, e.g. "the", "a". To this end, it combines two statistics: term frequency and inverse term frequency [35].

Term frequency is the number of times a word appeared in a certain document. It represents the intuitive notion that the more often a word appears in a document, the more important it is.

The inverse document frequency of a word is a statistic inversely proportional to the number of documents in the dataset containing that word. Thus, words that are present in few documents will have a higher inverse-term frequency that words that are present in a lot of documents.

Tf-idf can be formally defined as $tfidf(t) = tf(t, d) * idf(t) = tf(t, d) * logN/df(t)$ where t is a word, d is a document in the dataset, N is the number of documents in the dataset, $df(t)$ is the number of documents containing the word t, tf is the term-frequency and idf is the inverse-term frequency [35].

2.3.6 Performance Metrics

There are four fundamental metrics for evaluating the performance of a binary classifier. A binary classifier outputs either positive or negative for each input it receives. Thus, for a set of instances given to the classifier, we say that [26]:

- TP (true positives) is the number of instances correctly marked as positive by the classifier;
- FP (false positives) is the number of instances wrongly marked as positive by the classifier;
- TN (true negatives) is the number of instances correctly marked as negative by the classifier;

- FN (false negatives) is the number of instances wrongly marked as negative by the classifier.

Using these core metrics as a basis, researchers have proposed many other metrics. Here, we focus on four commonly used metrics: accuracy, precision, recall and f1-score. For the sake of brevity, we also define $P = TP + FN$ as the number of positive instances and $N = TN + FP$ as the number of negative instances.

Accuracy is the proportion of instances that the classifier correctly marked, i.e. $accuracy = (TN + TP)/(N + P)$.

Precision is the proportion of instances that were correctly marked as positive out of all the instances marked as positive, i.e. $precision = (TP)/(FP + TP)$. In a information retrieval context, this would be the proportion of results returned that are actually relevant.

Recall is the proportion of instances that were correctly marked as positive out of all the positive instances, i.e. $recall = (TP)/(FN + TP)$. In a information retrieval context, this would be the proportion of results returned out of all the results in the dataset.

F1-score is the harmonic mean of precision and recall, i.e. $f1score = 2 * precision * recall/(precision + recall)$. This metric is often used when we want to find a classifier with balanced precision and recall instead of high precision and low recall or vice-versa.

Although accuracy is the most straightforward metric and appears to summarize the performance of a classifier well at first, it is not the best choice when dealing with an unbalanced dataset. Suppose a dataset where 10% of the instances are positive and the other 90% are negative. If a classifier was created where any instances given to it were marked as negative, then such a classifier would have an accuracy of 90% on this dataset and would seem to have an excellent performance. However, its precision and recall on this dataset would be 0%.

2.4 Grounded Theory

Grounded theory (GT) is a qualitative research methodology developed by Glaser and Strauss in 1967 which aims to inductively generate theory from data [24]. By following GT methodology, researchers identify codes and categories as they analyze the data and these codes and categories are eventually used to generate a theory describing and explaining the phenomena under study.

Since its inception, multiple variants of GT have been developed such as the Straussian and Constructivist variants, while the original GT is became known as Classic GT. Each variant has different philosophical influences and consequently has its own peculiarities with regards to the methodology although they are very similar. As an example, while classic GT is strict about literature review never being performed before the theory is developed, Straussian GT is more lax in this regard and believes that the use of other literature could be helpful in certain cases [49].

Many studies often claim incorrectly to have used GT when they only used a small subset of it such as the process of open coding. For a study to use GT properly, they must perform a number of its defining activities, e.g. constant comparison, theoretical sampling and memoing [49]. We describe these activities in the next paragraphs in which they are highlighted.

GT can be understood as an iterative process where we constantly move back and forth between coding discussions and analyzing these codes to categorize them and generate a new theory. In GT, there must be a *constant comparison* of data, concepts, categories and memos looking for similarities and differences between them. This is in stark contrast to research methods where analysis is only performed at the end [15].

GT uses *theoretical sampling*, which is different from sampling in quantitative research methods. Whereas a random sampling process defined a priori is the norm for quantitative methods, in GT the researcher purposefully selects new data based on the codes, memos and hypotheses established so far. That is, it is preferable to select new data points that conflict with the current analysis results and that will generate new insights instead of data points that are already explained by the current results. The goal is to achieve *theoretical saturation*, i.e. the point where new data points do not affect the codes, categories and theories generated by the researcher [15].

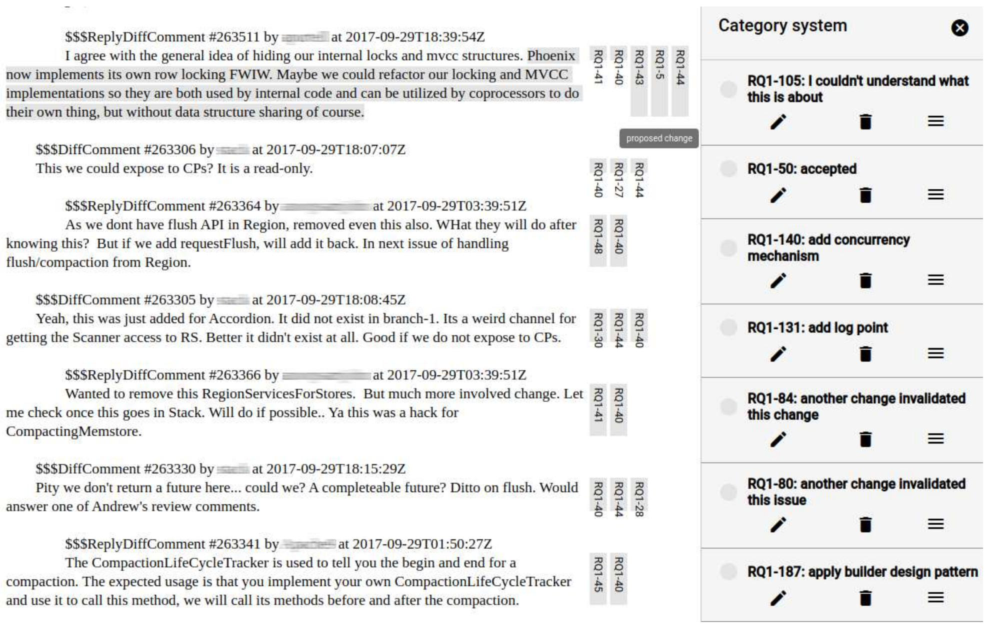
Each piece of data is analyzed according to the processes of *open coding* and *axial coding*. *Open coding* consists of determining concepts that represent what is being said in the data. Concurrently to *open coding*, researchers perform *axial coding* which consists of grouping related lower-level concepts into categories (higher-level categories) and of finding relationships between concepts and categories. Using these two processes, researchers can build a hierarchy of concepts, categories and their relationships that describes and explains the phenomena under study [15].

While *open coding* and *axial coding* are significant activities of GT, their use in isolation is not synonymous to using GT. Another crucial activity of GT is *memoing*. Memos are documents written by the researchers throughout the analysis where they explain the concepts identified, establish relationships between concepts, compare concepts, think about the properties and dimensions of a concept, integrate categories, generate hypotheses, develop theory and so forth. In short, *memoing* consists of documenting the analytical reasoning of the researchers on how the identified lower-level concepts were eventually developed into a theory describing the phenomena under study [15].

A GT study can either limit itself to developing thick and rich description about the phenomena under study or it can go further and develop a theory for explaining it (*theory building*). In order to accomplish that, researchers need to define a *core category* and perform *integration*, which is about relating the categories that were found to this *core category* and establishing the relationships between them, i.e. integrating all the concepts and categories into a unified theory that describes and explains the phenomena under study [15]. After defining a *core category*, the hierarchy of concepts and categories can be seen as a tree where the concepts are the leaf nodes, the categories are internal nodes and the *core category* is the root.

2.4.1 QCAmap

QCAmap is a tool that can be used to support the coding processes of GT [36]. It works well for GT even though it was originally made for another research method, namely, qualitative content analysis. Among its many features, it allows researchers to (1) import textual data, (2) assign concepts to words, phrases and whole paragraphs, (3) group concepts into categories and (4) generate statistics of the coding process.



$$$ReplyDiffComment #263511 by [redacted] at 2017-09-29T18:39:54Z
I agree with the general idea of hiding our internal locks and mvcc structures. Phoenix now implements its own row locking FWIW. Maybe we could refactor our locking and MVCC implementations so they are both used by internal code and can be utilized by coprocessors to do their own thing, but without data structure sharing of course.
RQ1-41 RQ1-40 RQ1-43 RQ1-5 RQ1-44

proposed change

$$$DiffComment #263306 by [redacted] at 2017-09-29T18:07:07Z
This we could expose to CPs? It is a read-only.
RQ1-40 RQ1-27 RQ1-44

$$$ReplyDiffComment #263364 by [redacted] at 2017-09-29T03:39:51Z
As we dont have flush API in Region, removed even this also. WHat they will do after knowing this? But if we add requestFlush, will add it back. In next issue of handling flush/compaction from Region.
RQ1-48 RQ1-40

$$$DiffComment #263305 by [redacted] at 2017-09-29T18:08:45Z
Yeah, this was just added for Accordion. It did not exist in branch-1. Its a weird channel for getting the Scanner access to RS. Better it didn't exist at all. Good if we do not expose to CPs.
RQ1-30 RQ1-44 RQ1-40

$$$ReplyDiffComment #263366 by [redacted] at 2017-09-29T03:39:51Z
Wanted to remove this RegionServicesForStores. But much more involved change. Let me check once this goes in Stack. Will do if possible.. Ya this was a hack for CompactingMemstore.
RQ1-41 RQ1-40

$$$DiffComment #263330 by [redacted] at 2017-09-29T18:15:29Z
Pity we don't return a future here... could we? A completeable future? Ditto on flush. Would answer one of Andrew's review comments.
RQ1-40 RQ1-44 RQ1-28

$$$ReplyDiffComment #263341 by [redacted] at 2017-09-29T01:50:27Z
The CompactionLifeCycleTracker is used to tell you the begin and end for a compaction. The expected usage is that you implement your own CompactionLifeCycleTracker and use it to call this method, we will call its methods before and after the compaction.
RQ1-45 RQ1-40

Category system

RQ1-105: I couldn't understand what this is about

RQ1-50: accepted

RQ1-140: add concurrency mechanism

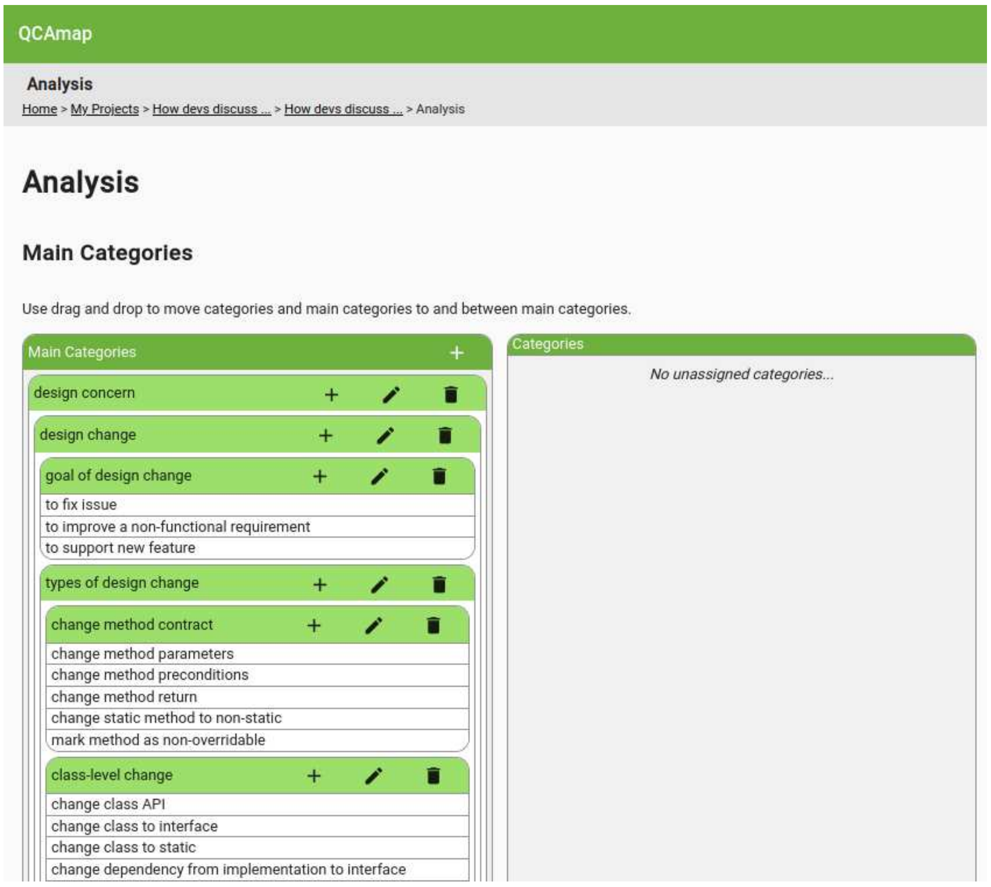
RQ1-131: add log point

RQ1-84: another change invalidated this change

RQ1-80: another change invalidated this issue

RQ1-187: apply builder design pattern

Figure 2.3: QCAmap supports the process of open coding. A passage that we marked with the concept of design change is highlighted in this figure.



QCAmap

Analysis

Home > My Projects > How devs discuss ... > How devs discuss ... > Analysis

Analysis

Main Categories

Use drag and drop to move categories and main categories to and between main categories.

Main Categories

- design concern
  - design change
    - goal of design change
      - to fix issue
      - to improve a non-functional requirement
      - to support new feature
    - types of design change
      - change method contract
        - change method parameters
        - change method preconditions
        - change method return
        - change static method to non-static
        - mark method as non-overridable
      - class-level change
        - change class API
        - change class to interface
        - change class to static
        - change dependency from implementation to interface

Categories

No unassigned categories...

Figure 2.4: QCAmap supports grouping concepts into categories.

Chapter 3

Automatic Identification of Design Discussions in Modern Code Review

Before conducting the characterization study, we needed a dataset of design discussions in MCR. Thus, we saw the opportunity to not only build a public dataset of design discussions in MCR but also to replicate quantitative studies which indicated the presence of substantial design information in MCR.

3.1 Methodology

3.1.1 Research Questions

The goal of this study is to analyze the occurrence of design discussions in MCR discussions and to build a dataset of design discussions using an automatic binary classifier. Hence, we devised the following research questions:

- *RQ1:* How frequently is software design discussed during the process of MCR in an open source software project?
- *RQ2:* How well does our classifier of design discussions perform in terms of precision and recall?

3.1.2 Data Collection

The Apache Software Foundation (ASF), the world's largest open source software (OSS) foundation [19], makes the MCR discussions from its numerous projects publicly available [18]. Due to the wide variety of projects developed by the foundation and the great number of people involved, we suppose that the foundation's data are appropriate for studying design discussions and evaluating the solution proposed in this document.

Code review in Apache's projects are usually performed with the tool Review Board [6]. Using Review Board, users publish code review requests when they want to send changesets to the main code repository and they would like other team members to review these changesets. After a review request is opened, it is possible to comment either on the whole changeset or on specific diff lines in the code. These comments can then be answered by other developers to form discussion threads.

Review Board allows its data to be extracted via a REST API. Therefore, using this API, we extracted 56,397 review requests that happened between 2010-10-25 and 2019-11-29 and collectively contain a total of 386,940 comments.

3.1.3 Data Analysis

After extracting the data, we randomly sampled 1,000 discussions from it and manually identified which were about design. Then, we used this annotated sample to create an automatic classifier capable of recognizing which discussions are about design. Finally, we applied this classifier to the whole dataset of comments extracted from the Apache projects.

Manual Classification of Design Discussions

In order to create an automatic identifier of design discussions using a supervised learning algorithm, a sample of discussions correctly classified as being about design or not is required. So we manually classified a random sample of 1,000 discussions, marking them as either being about design or not.

We had to perform a few preprocessing operations on the data before proceeding with the random sampling of the discussions. These were as follows:

1. Removal of 19,945 (5.2%) review requests which had at most one comment because we observed that this kind of request was usually discarded.

2. Merge of comments made on specific code diffs (DiffComment) with their subsequent answers (ReplyDiffComment) because we observed that reply comments were often brief and impossible to understand when analyzed individually out of their original context.

3. Merge of comments about the whole changeset (Review.body_top and Review.body_bottom) with their associated replies (Review.reply.body_top and Review.reply.body_bottom) for the same reason as the previous item, i.e., they are a thread of comments that tend to lose their meaning when analyzed individually.

4. Removal of 1,505 (0.4%) comments which have more than 32,767 characters because these large comments exceeded the practical limits of the tools we used to work with the data and of the techniques we later used to create an automatic classifier. Also, after examining a few of these 1,505 comments, we observed that they consisted of messages automatically generated by bots, so their removal is unlikely to negatively affect the results of this study.

These preprocessing operations yielded a total of 267,843 discussions belonging to 36,452 review requests. Afterwards, we extracted a random sample of 1,000 discussions from this data. Then, for each discussion, we read it carefully, marked it as being about design or not and assigned one or more concepts to it to create a preliminary categorization of the discussions.

From the 1,000 code review discussions in the sample, 364 (36.4%) were marked as being about design and the remaining 636 (63.6%) as not being about design. Furthermore, Table 3.1 shows our categorization of the discussions in the sample.

Table 3.1: Categories of the 1,000 manually classified code review discussions and their absolute frequencies.

Type	Category	N
Non-Design	Author Acknowledgment	28
Design	Better Solution	135
Non-Design	Bot (automatic message)	83
Design	Bug	63
Non-Design	Change Approval	160
Non-Design	Changeset Partitioning	2
Design	Clarification of Design Decision	82
Design	Code Execution Order	1
Non-Design	Code Formatting	71
Design	Concurrency	5
Design	Configuration	6
Design	Dependency	9
Non-Design	Documentation	113
Design	Duplicated Code	4
Design	Encapsulation	7
Non-Design	Feature Addition (requirements)	3
Non-Design	Generic Question	2
Design	Interface Contract	3
Non-Design	Log Message (message to the user)	15
Design	Log Point (implementation of the log system)	7
Non-Design	Naming	54
Design	Performance	9
Design	Refactoring	31
Non-Design	Rejection	2
Non-Design	Repetition of Previous Comment	19
Non-Design	Review Process	7
Design	Safety	4
Non-Design	Task To Do	10
Design	Test Design	14
Non-Design	Test Requirements	19
Design	Unnecessary Code	32

3.2 Results

3.2.1 Automatic Classifier

After obtaining a set of discussions with labels to indicate whether they contain design information or not, the next step was to build a binary classifier of design discussions using supervised learning techniques and natural language processing techniques.

Automatic Classifier Creation

Succinctly, our automatic classifier was created in three steps: (1) tokenization and lemmatization of the manually classified sample of 1,000 code review discussions; (2) extraction of features appropriate for supervised learning algorithms and (3) training of a supervised learning algorithm.

In the first step, using the spaCy library [1], we began by tokenizing all the code review discussions. Next, from the resulting list of tokens for each discussion, we removed tokens which represent punctuation characters and stop words. Afterwards, we converted each tokens to its lemma when possible and converted them to lowercase.

In the second step, using the scikit-learn library [37], we ran the process of feature extraction on the tokens obtained from the previous step because supervised learning algorithms usually require the inputs to be fixed numerical vectors. First, we converted the list of tokens from each discussion into a bag-of-words model composed of unigrams and bigrams. Then, we normalized the frequencies of these unigrams and bigrams using the tf-idf statistic.

In the third and last step, also using the scikit-learn library [37], we trained the chosen supervised learning algorithm using the bag-of-words of each discussion, which had the normalized frequencies of unigrams and bigrams.

Evaluation of Classification Models

Since there does not exist a supervised learning algorithm that is ideal for all problems, several models must be empirically evaluated before finding one that works satisfactorily. We chose to evaluate seven algorithms for this study: decision tree (DT), random forest (RF), multinomial naive bayes (MNB), linear support vector machine (SVM), gradient boosting classification (GBC), logistic regression (LR) e multi-layer perceptron (MLP).

Table 3.2: Comparison of candidate learning models using their default hyperparameters.

Model	Accuracy	Precision	Recall	F1-score
SVM	80.7% (±5.3%)	71.5% (±8.5%)	78.6% (±3.8%)	0.748 (±0.056)
MLP	80.6% (±6.8%)	74.6% (±12.2%)	71.7% (±8.1%)	0.730 (±0.082)
LR	79.0% (±6.2%)	76.2% (±10.5%)	61.8% (±11.7%)	0.681 (±0.098)
GBC	77.1% (±5.4%)	73.4% (±10.7%)	58.5% (±10.4%)	0.650 (±0.088)
RF	76.9% (±4.4%)	82.8% (±9.1%)	46.4% (±13.9%)	0.591 (±0.113)
DT	73.4% (±1.8%)	69.6% (±9.8%)	50.5% (±21.6%)	0.575 (±0.090)
MNB	74.2% (±5.7%)	86.6% (±15.9%)	34.6% (±12.8%)	0.491 (±0.146)

We used k-fold cross validation with a $k = 5$ to evaluate the algorithms. The main metrics used were precision, recall and F1-score instead of accuracy because the dataset is unbalanced, i.e., it has more negative items than positive ones.

Initially, we trained and compared the seven algorithms with barely any changes to their default hyperparameters. Table 3.2 shows this initial comparison along with the metrics obtained for each algorithm.

After comparing the results of the algorithms, we chose the most promising models, namely: SVM, MLP, LR and MNB. SVM, MLP and LR had the best F1-scores respectively. Despite MNB having a low F1-score, we chose to keep it because it had a very high precision and we were interested in seeing how it would perform with optimized hyperparameters.

After reducing the number of candidate models to four, we performed a search for the best hyperparameters for each of these models according to the F1-score metric. Table 3.3 shows the the results of these models using the best hyperparameters found.

Considering these results, we decided to use the MLP model, which obtained an accuracy of 82.2% (±7.7%), precision of 73.6% (±12.6%), recall of 81.0% (±6.9%) and F1-score of 0.770 (±0.082).

Table 3.3: Comparison of candidate learning models using the best hyperparameters found for them.

Model	Accuracy	Precision	Recall	F1-score
MLP	82.2% (±7.7%)	73.6% (±12.6%)	81.0% (±6.9%)	0.770 (±0.082)
SVM	81.2% (±4.1%)	72.0% (±7.1%)	79.7% (±6.3%)	0.756 (±0.040)
MNB	82.0% (±5.1%)	76.3% (±11.4%)	74.2% (±5.5%)	0.751 (±0.056)
LR	80.8% (±5.9%)	71.9% (±9.3%)	78.3% (±4.7%)	0.749 (±0.065)

Classifier Performance on the Dataset

Having built and evaluated the automatic classifier, we applied it to all the 267,843 code review discussions in dataset. 108,458 (40.5%) discussions were classified as being about design and the remaining 159,385 (59.5%) as not being about design.

3.3 Discussion

The high ratio of design discussions in both the sample (36.4%) and in the whole dataset (40.5%) indicates that design is frequently discussed in MCR.

The ratio of design discussions in the whole dataset (40.5%) was similar to the ratio in the manually classified sample of discussions (36.4%). This suggests that the sample of discussions is representative of the whole dataset.

The ratio of design discussions in this study (40.5%) was considerably higher than previous work. Indeed, Brunet et al. found that about 25% of discussions were about design [9] and Zanaty et al. found that an average of 12% of discussions were about design [58]. We formulated two hypotheses that might explain this difference: (1) the lack of a formal and comprehensive definition of what constitutes a design discussion leads to different researchers classifying discussions differently and (2) different projects and teams discuss design at different rates.

To mitigate the lack of a formal definition of software design, we provided a categorization of what we considered design in this study to make it clear to others what was marked as design or not.

As for the hypothesis that the ratios of design discussions are considerably different from one project to another, we examined the variability of this ratio between different Apache projects to test it (Figure 3.1). There were 27 Apache projects with more than 1,000 discussions and the ratio of design discussions for these ranged from 25.9% to 60.0% and its first, second and third quartiles were 38.0%, 49.0% and 55.4%, respectively. Hence, this provides evidence that design discussions does vary considerably between different teams and projects and this could explain the different results observed in related studies.

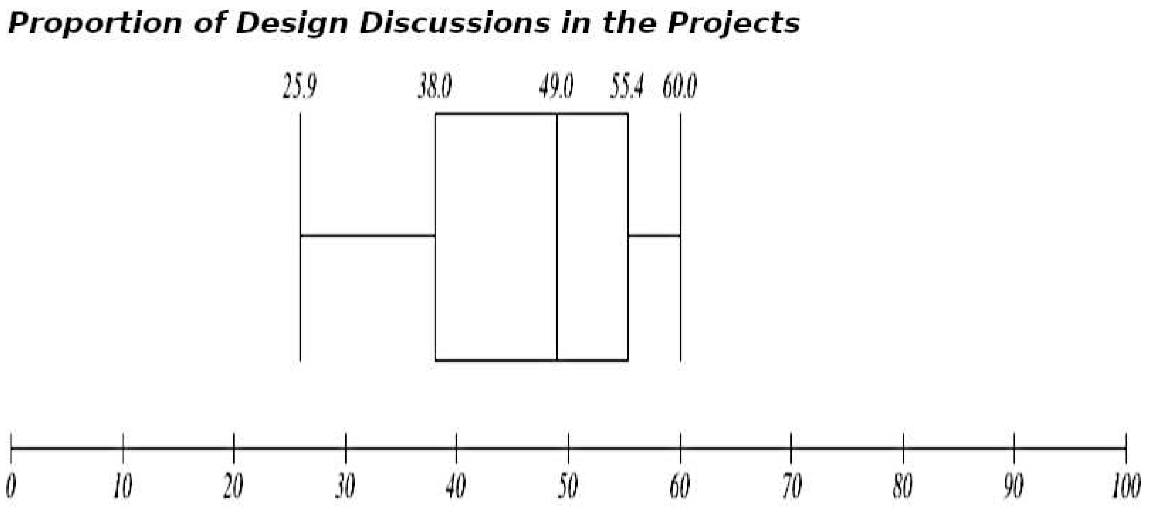


Figure 3.1: A boxplot of the proportion of design discussions in the 27 Apache projects with more than 1,000 discussions.

3.4 Related Work

Brunet et al. used machine learning to develop a binary classifier for identifying design discussions in pull requests, commits and issues. They applied this classifier to 102,122 discussions from 77 projects hosted in GitHub and found that on average (1) 25% of discussions have design information, (2) 26% of developers participated in at least one design discussion, (3) 99% of developers contribute to less than 15% of design discussions and (4) the small set of developers who contribute to most design discussions are the most active committers in the project [9].

Shakiba et al. [46] performed a study like the one by Brunet et al. [9]. Similarly to Brunet et al., they created and evaluated a binary classifier based in machine learning to identify design information in discussions. Differently, they (1) applied the classifier only to commit discussions, (2) used data from projects hosted in GitHub and SourceForge, (3) evaluated more machine learning classification algorithms and (4) only analyzed the performance of the classifier i.e. they did not analyze the design information in those discussions. Thus, their main conclusion was that their classifier based on the random forests algorithm had the best performance with a G-mean of 75.01.

Zanaty et al. performed a mixed methods study of design discussions [58]. First, they manually analyzed a sample of 1,357 comments from OpenStack Nova and 1,460 comments from OpenStack Neutron projects. From this qualitative analysis, they found that an average of 12% of comments have design information and they created a categorization of the design issues discussed in the comments. Second, they built multiple binary classifiers of design discussions and then applied these classifiers to 2,506,308 review comments. Their classifiers had a precision of 59% to 66% and a recall of 70% to 78%. They ultimately conclude that design is not discussed as often as it should be and that most design comments are constructive.

Likewise, Viviani et al. designed and evaluated a binary classifier for finding design points in pull request discussions [52]. To this end, they built a dataset of design points in 10,790 paragraphs from 3 OSS projects. Using this dataset to train the classifier, they found that it had an AUC of 0,87. To further evaluate the generalizability of this classifier to other projects, the authors enlisted 5 Computer Science students to build another dataset with 250 discussion paragraphs from different OSS projects. On this dataset, the classifier had an AUC of 0,81 which suggests that it is generalizable to datasets other than the one used to train it.

In the same manner, Mahadi et al. improved upon Brunet et al.'s design discussion classifier and studied the generalizability of such a classifier to datasets different from the one used to train it [34][33]. Their improved classifier exhibited an AUC of 0.84 but had poor results when applied to different datasets than the training one even when using ULMFiT, which is a technique for improving transfer learning.

3.5 Conclusion

In this study, we wanted to examine the presence of design discussions in MCR and build a classifier and dataset of design discussions that we could use for future studies.

Briefly, the study consisted of: (1) extracting 386,940 comments from 56,397 code reviews of 80 project repositories from the Apache organization, (2) manually identifying in a sample of 1,000 discussions which were about design, (3) creating an automatic classifier capable of recognizing which discussions are about design and (4) applying this classifier to the dataset of extracted comments.

The main contributions of this study were:

- A public dataset of 108,458 design discussions from the Apache Software Foundation that can be used in other studies;
- A free and open source classifier of design discussions trained on this dataset with an F1-score of 0.770 (± 0.082);
- Evidence that design is frequently discussed in MCR;
- Evidence that it is possible to identify design discussions automatically;

Chapter 4

Characterization of Design Discussions in Modern Code Review

So far, most research has focused on quantitative analyses of design in MCR. As a result, there is little evidence on how design is discussed in MCR and what topics of design are discussed because qualitative methodologies are better suited to answer questions related to process and classification. In order to fill this research gap, we conducted a qualitative study using Straussian Grounded Theory to learn more about how developers discuss design in MCR and what topics of design are discussed.

4.1 Methodology

4.1.1 Research Questions

Given the aforementioned knowledge gap in this area, we decided to investigate the following research questions:

- **RQ1:** How do developers discuss design during code review?
- **RQ2:** What topics of design are discussed during code review?

As explained in Section 2.4, Straussian grounded theory (GT) is a qualitative research methodology which can inductively generate theory from data. Moreover, GT is able to not only describe a process but also classify its contents. Thus, we chose the GT methodology for this study because it is well suited for the research questions, i.e. it enables us to simultaneously investigate the process of how developers discuss design during code review and also what topics of design are discussed.

4.1.2 Data Collection

We analyzed public code reviews from OSS projects of the Apache Software Foundation (ASF). Specifically, we used a dataset containing 267,843 review discussions from 80 projects of the ASF. We had previously extracted these discussions from the ASF's publicly available instance of Review Board [18].

We chose this dataset because (1) the ASF develops a wide variety of large projects used in production, (2) of the high number of people involved in these projects and (3) the public and open nature of these code reviews favors the replicability of this study.

4.1.3 Data Analysis

We performed the analysis according to the key aspects of GT which were explained in in Section 2.4.

To begin our analysis, we chose a large code review at random that had a substantial amount of discussion. After analyzing this first code review, we continuously skimmed other code reviews in no particular order and analyzed the ones which we thought would advance the emerging theory following the idea of *theoretical sampling*.

To analyze each selected code review, we used the processes of *open coding* and *axial coding*. QCAmap was the tool used to support the coding processes [36].

When we had difficulty understanding a comment or a discussion, we examined the source code associated with the comment and the metadata for that comment which sometimes indicated whether the comments were accepted or discarded in the end.

Throughout the analysis, we naturally performed the essential GT activities of *constant comparison* and *memoing*. We wrote and stored the memos in Google Docs and later used them for *theory building*.

The author of this thesis and his doctoral advisors conducted periodic reviews to improve the reliability of the results because the author conducted the majority of the coding and memoing. During these reviews, we discussed the concepts, categories, memos and etc that were identified from the data thus far. Furthermore, at the end of the analysis, we compared our results with existing research in the area to evaluate the validity of our results.

4.2 Results

4.2.1 Overview

Following the process of theoretical sampling, we skimmed 180 review requests with design information chosen at random and analyzed in depth 30 of those while incrementally building our theory. We concentrated our effort on these 30 review requests because we deemed them as likely to improve the theory by providing new concepts or conflicting with the intermediary theory.

The 30 review requests analyzed in depth came from 16 Apache Software Foundation software projects. These review requests contain 1936 comments whose total length is 40975 words (approximately 82 pages of 500 words) and which were written by a total of 91 developers.

Our open and axial coding of the 30 review requests identified 174 concepts and 46 categories which have 3330 occurrences in the data. Moreover, developers discussed a total of 409 distinct design concerns, of which 246 were design changes and 163 were design issues. Appendix B shows the full tree of concepts and categories identified during the analysis.

4.2.2 RQ1: How do developers discuss design during code review?

Design review is a significant part of code review and it is conducted in an implicit and unstructured manner during code review. Indeed, information pertaining to design review is mixed in discussions which deal with a range of topics unrelated to software design. To better understand *design review*, we split it into three components: *design concern*, *review action* and *design discussion* (Figure 4.1).

In the figures used to illustrate the model, we used the following notation: (1) a named box denotes a category, (2) names within a box which are prefixed by a plus sign denote

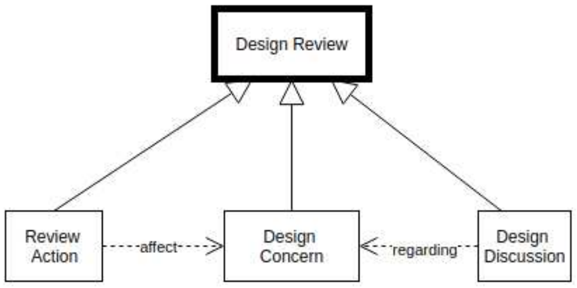


Figure 4.1: A view of the model focusing on the core category: *design review*.

concepts belonging to that category, (3) a regular line with a closed arrow denotes that a category is a subcategory of the category pointed to and (4) a dashed line indicates that one category is related to another.

Design Concern

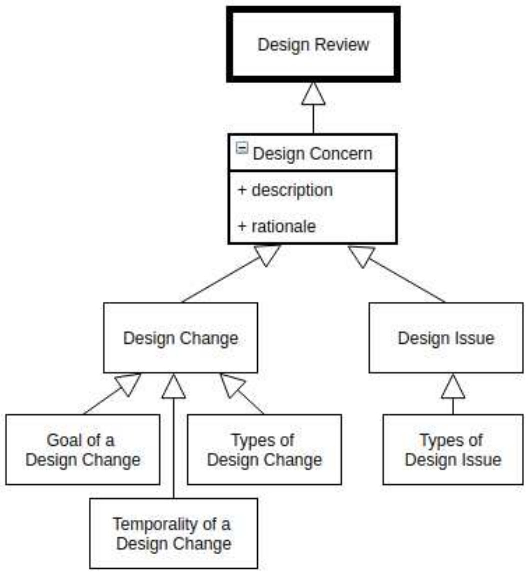


Figure 4.2: A view of the model focusing on *design concern*.

The core of a *design review* is a set of *design concerns* which originate in the changes made by the author of the review request. A *design concern* is an aspect of the software design that is of interest to the developers and it can be either a *design change* or a *design issue* (Figure 4.2).

Observation 1. A design review revolves around a set of design concerns (changes and issues).

A *design change* is a modification to the software design. It has a temporal property in that it can either be a *proposed change* or an already *implemented change*. It also has a goal which can be: (1) *to fix a design issue*, (2) *to improve a non-functional requirement* (e.g. maintainability, performance) or (3) *to support a new feature*.

The following quotation is an example of a *design change*. This change is proposed in order to improve maintainability and it consists of a pull up refactoring where common code would be moved to a common parent abstract class:

> "probably we can make this less painful, maybe having ServerProcedureInterface and TableProcedureInterface extending something like ProcedureQueueObjectInterface or something like that. and then we do getRunQueue with something like procQueueObjectInterface.getQueueId()?" – Review #34130, Comment #134521

A *design issue* is a problem in the software design that impacts the fulfillment of functional or non-functional requirements. Issues can be minor such as *deficient encapsulation* in a class which hampers maintainability or severe such as when the *design doesn't fulfill the functional requirements*.

Here is an example of a *design issue*. The reviewer thinks there might be a *concurrency design issue*:

> "We reuse these instances? Any racing going to happen between threads w/ one setting these data members and another reading?" – Review #10965, Comment #55460

Design issues and *design changes* also share similarities. Both *design issue* and *design change* have an associated *description* (the what) and a *rationale* (the why). For example, in the case of a *design issue*, its *description* explains what the issue is and its *rationale* is the explanation for why it is an issue. Furthermore, there are a plethora of different *types of design change* and *types of design issue* and these are presented in detail in the next section.

Review Action

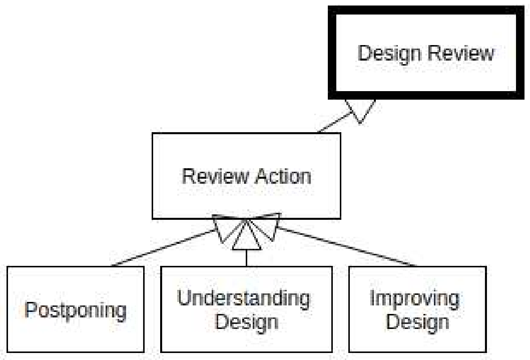


Figure 4.3: A view of the model focusing on *review action*.

Throughout *design review*, developers execute a variety of *review actions*, which fall under one of three categories according to its goal: (1) *improving the design*, (2) *understanding the design* or (3) *postponing* (Figure 4.3).

Observation 2. Developers seek to improve and understand the design during design review.

To *improve the design*, developers *introduce design concerns* during the review and *ask for feedback* for the other participants. The *introduction of design concerns* serves as the starting point for *design discussions* which will discuss the validity of these and what course of action to take in light of them.

To *understand the design*, developers *ask for clarification on design* and *provide clarification on design* to others. They usually want to understand *design concerns* well before submitting their feedback as evidenced by the high frequency of comments asking for clarification. Aside from *design concerns*, they also seek to understand the current design of the software, i.e. the software design as it is before applying the code changes under review. All these clarification requests are not only about the description of design concerns but also their rationale. Next, an example of a developer *asking for clarification on a design change*:

> "What's the purpose of this static factory?" – Review #53297, Comment #224909

Given how large some review requests can become, postponing certain actions is not unusual. Specifically, participants may (1) *ask for time before answering design-related* *questions*, (2) *suggest postponing a discussion* to a later date or (3) postpone implementing a design change even though it has already been agreed upon during the review.

Design Discussion

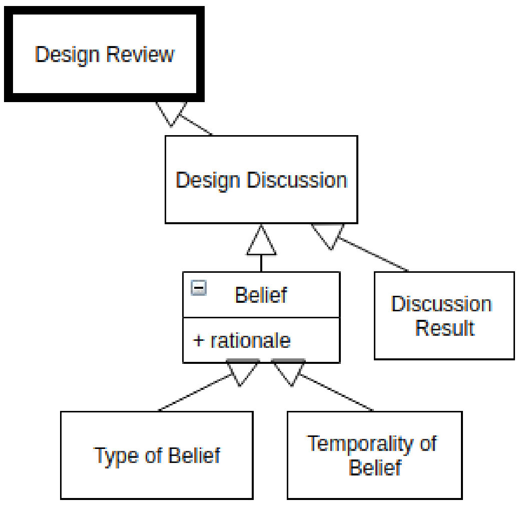


Figure 4.4: A view of the model focusing on *design discussion*.

During the *design review*, developers engage in *design discussions* over the *design concerns* identified. These discussions can be seen as a series of *beliefs* expressed over time by the developers. Moreover, a developer can express multiple *beliefs* regarding a *design concern* because, as he analyzes a design concern and the beliefs of other developers, his beliefs are subject to change and he might produce new ones (Figure 4.4).

Beliefs The *beliefs* regarding the *design concerns* can range from strong agreement to strong disagreement. Therefore, we used a Likert-type scale with 5 values to represent this variation, namely: *strongly agree*, *weakly agree*, *neutral*, *weakly disagree*, *strongly disagree*.

A *belief* can be seen as either a *strong belief* or a *weak belief* according to the language used by the developer. When a developer is not sure of his *belief* or desires to be polite, she tends to frame her *beliefs* as questions or use language constructs that denote uncertainty (Figure 3). By contrast, when a developer is confident in his belief or feels that further discussion is unnecessary, he uses affirmative language as in this example of a developer having a *strongly agree belief* about a *design change*:

"Have two separate methods - addRegistration and updateRegistration" – Review #11700, Comment #44704

Participants often provide explicit rationales for their *beliefs*. For example, they may *strongly agree* that something is indeed a *design issue* because they know of a particular use case where there will be a problem.

Discussion Result By the end of a *design discussion*, there is usually a *discussion result* indicating whether the *design concern* under consideration was *accepted* or *discarded*. Sometimes this result is communicated in the text and other times it is presented by the status of the discussion, i.e. the discussion is marked as dropped or accepted in the review tool.

Observation 3. Developers engage in discussions over design concerns, which are ultimately accepted or rejected.

As the name implies, an *accepted* result means that the *design concern* was accepted by the participants. In particular, it means that either the *design issue* was considered to be valid or the necessity of performing the *design change* was agreed upon.

On the other hand, a *discarded* result can happen for more than one reason. Naturally, a *design concern* may be *rejected* by developers after deliberation. In addition, a *design concern* is also considered *discarded* when the developers agree to *postpone* its associated discussion to another time after the current *design review*. Finally, a *discarded* result can also happen because the corresponding *design concern* was invalidated by another one. For instance, a *design issue* regarding a problem in a certain class can become invalid if a *design change* removes this class from the software.

In the following excerpt from the data, the developers decide to *postpone* discussion about a proposed *design change*. They use the issue tracker to organize this future discussion:

"Created HIVE-17622 for followup." – Review #62314, Comment #263796

Example

To illustrate the model, we present an example of a design discussion interpreted with it. This example discussion began with the following comment from one of the developers reviewing the changeset (Review #10965, Comment #41747):

> "Does this have to be on this Interface?" – Reviewer

The reviewer is *introducing a design issue* in that sentence. The use of a question to introduce the issue suggests that the reviewer only *weakly agrees* that this is a *design issue*, i.e. the reviewer is not completely sure of it. Looking at the code associated with this comment, we see that the reviewer is questioning whether a certain method should really be where it was placed. Thus, the *type of design issue* is represented by the concept *wrongly placed method*, which is a part of the more abstract category *placement issue*.

After this first comment, the developer who is the author of the changeset under review replied as follows:

> "why not? This seems the most straightforward place" – Changeset Author

Here, the changeset author is expressing his *belief* on the *design issue* introduced by the reviewer in the previous comment. The tone of his sentence suggests that the *type of belief* is of *weakly disagreement*. The author also presents the *rationale* for this *belief*, i.e. the reason why he disagrees that the fact presented by the reviewer is an issue.

Next, the reviewer responded:

> "I will always pushback adding stuff to this Interface; the more stuff it has to carry, the harder mocking will be.
>
> replayNonceOperation seems particularly exotic, not something I'd think important enough to make it up into this high-level RegionServerServices Interface. I could be wrong." – Reviewer

In this comment, the reviewer is presenting a more detailed *rationale* for his *belief*, i.e. he presents reasons for why the method should not be in that particular place. He still has a *weakly agree belief* that the *design issue* should be accepted. The sentence "I could be wrong" at the end confirms that his *belief* is of not of *strong agreement* but of *weakly agreement*.

Afterwards, the changeset author reacts with this comment:

> "hmm... this seems to be a prescribed way of calling RegionServer parts from Region.

> What would you suggest? I could instead add method like getNonceManager and then call the requisite method on that" – Changeset Author

The first sentence provides a *clarification on the current design*, i.e. it explains the current software design without the changeset under review. Next, the question "What would you suggest?" means that the author is *asking for design change proposals*, which can be seen more abstractly as a *review action* to *improve the design*. Finally, in the last sentence of that comment, the author *introduces a design change* which has *goal* of *fixing the design issue*. This *design change* is a *proposed change* and the *type of this design change* is *create new method*, which falls under the category *create new element*. Moreover, the wording used by the author suggest that he has *weakly agree belief* regarding this *proposed design change*.

The last comment in this design discussion is from the reviewer:

> "I think getting a nonce manager from the Interface is more coarse grained and therefore appropriate for a high-level Interface like this (also means you won't have to change the Interface again if you need more nonce methods). Thanks."
> – Reviewer

The reviewer expresses a *strongly agree belief* regarding the *proposed design change*. In addition to that, the whole discussion is marked as resolved in the review tool. Thus, given this information, we infer that both the *design issue* and the *proposed design change* to fix this issue were *accepted* after this design discussion between the changeset author and the reviewer regarding these two *design concerns*.

4.2.3 RQ2: What topics of design are discussed during code review?

Types of Design Change

We identified 59 concepts representing *types of design change* and we grouped most of them into 11 categories (Table 4.1). We display concepts and categories in the same table because some concepts were not related to any other and consequently were not grouped into a category. Also, for the sake of brevity, we omit concepts which only occurred once.

Change method contract is a category well summarized by its name. It consisted of changes such as *change method parameter*, *change method preconditions* and *change return value*.

Table 4.1: Main concepts and categories of *design changes*. Categories start with uppercase to differentiate them from concepts.

Concept / Category	Occurrences	Reviews With It
Change method contract	19	16
change solution design	9	9
change field mutability	6	6
Class-level change	15	13
Concurrency	4	4
Create new element	15	12
Design pattern	5	4
Exception	14	11
Merge elements	3	3
Move element	20	14
remove code duplication	4	4
Remove unnecessary element	10	10
Traditional refactoring	21	13
Use another element	11	9

The concept of *change solution design* was used to denote context-dependent *design changes* that do not fit into a generic classification of types of *design concerns*. The following quote shows a concrete example of what we mean. In short, in that comment, the developer is talking about the algorithm used to solve a problem in the domain of that application:

> "Will cover this in [URL for issue in JIRA] which will handle per pool validation. I will also add precendence order when multiple actions has to be applied at the same time. I was thinking when KILL and MOVE triggers are violated at the same time, KILL takes precendence. When more than one MOVE trigger is violated at the same time, may be biggest pool (based on resource percent) will be chosen to avoid juggling between pools. May be will make this comparator pluggable." – Review #62314, Comment #265116

We believe such comments are specific to the problem domains of their software and cannot be classified in a straightforward manner in terms of design components common to all applications such as classes, subsystems, modules, methods, relationships between modules, dependencies, etc. Moreover, it is out of the scope of this study to enumerate all the types of design concerns specific to every possible problem domain.

Change field mutability is about altering an immutable field to be mutable or vice-versa.

The category of *class-level change* is similar to *change method contract* and is about changes to the interface of a class and its relationships to other classes. It encompasses modifications such as *change object construction interface*, *change object serialization*, *change relationship multiplicity* and *new relationship between classes*.

We did not delve deeper into design changes related to *concurrency* because they happened sparingly so we believed it would not add much to our study to do so. We identified two concepts: *add concurrency mechanism* and *change concurrency mechanism*. For example, a suggestion for adding a monitor for synchronizing threads was classified as *add concurrency mechanism*.

Create new element is about adding a new element to the software design. Specifically, we placed four actions under this category: *create new class*, *create new field*, *create new method* and *create public interface*.

As intuited, we found discussions about *design patterns* nevertheless they were not frequent. Developers discussed the use of 5 designs patterns in the discussions analyzed: *builder*, *dependency inversion*, *observer*, *singleton* and *visitor*.

The *exception* category consists of changes to how the software handles exceptions.

The *merge elements* category consists of design changes that merged classes or methods which had similar behaviors.

Move element is composed of *design changes* that move responsibilities from one place to another in the software. For example, *moving a class to another module*, *moving methods to another class* and *moving code to another function*. We used the term "code" when we wanted to mean that different elements were moved together e.g. classes, fields and methods.

Remove code duplication is a concept used for a general design change that aims at removing code duplication in the code. This could involve creating a method, a class or something else but it is not clearly specified in the discussion.

The category *remove unnecessary element* is about eliminating unnecessary elements from the software design such as unnecessary fields or methods.

Traditional refactoring are the design changes which involve one of the types of refactorings popularized by Martin Fowler in his seminal book on the subject [20]. We observed the following types of refactorings in the discussions: *encapsulate*, *extract class*, *extract field*, *extract method*, *extract variable*, *inline function*, *pull up code*, *replace conditional with polymorphism*, *replace magic literal* and *replace primitive with object*. Although some move operations (e.g. *move field to another class*) are part of Fowler's catalog of refactorings [20], we decided to assign them to the separate *move element* category because his catalog does not include all concepts we identified such as *move class to another module*.

Here is an example of a refactoring design change of type *pull up code*. The developer has a *weakly agree belief* regarding it and provides the *change rationale* that it would avoid code duplication:

> "Can chainMap and joinChains variables be moved to super class as well? Would avoid lots of duplicate code." – Review #28946, Comment #107937

Use another element is a category for changes that involve replacing a design element with another. The decisions to *replace own implementation with an external dependency*, *use another algorithm*, *use another data structure* or *use another library* were grouped under this

Table 4.2: Main concepts and categories of *design issues*. Categories start with uppercase to differentiate them from concepts.

Concept / Category	Occurrences	Reviews With It
concurrency design issue	8	8
confusing design	4	4
design doesnt fulfill requirements	4	4
Design smell	3	3
Encapsulation	8	7
Exception issue	8	7
Memory	2	2
Method contract issue	7	7
Missing element	5	5
misused design pattern	2	2
Placement issue	8	6
poor performance of design	2	2
Solution design	6	4
Unnecessary element	23	14

category.

Types of Design Issue

We identified 34 concepts representing types of design issue and we grouped most of them into 9 categories (Table 4.2).

Concurrency design issue represents problems with the concurrency strategy in the design. As was the case with design changes, we did not investigate these issues in more detail.

Confusing design is a concept used for denoting that even though the design works it is hard to understand well.

Although the software design might be correct and easy to understand, it might not solve the problem the stakeholders have, i.e. the *design doesn't fulfill the software requirements*.

We identified three types of *design smells*: *flag argument*, *global variable* and *magic literal*. A *flag argument* issue is when a method uses a boolean argument to determine its behavior. *Magic literal* consists of using hard coded numbers in the code instead of well defined named constants.

An *encapsulation* issue happens when a class or module either unnecessarily exposes its contents (*deficient encapsulation*) or when it does not expose a method or field that other classes need (*excessive encapsulation*).

The *exception* category is about problems with *wrong exception handling* or throwing an exception when it shouldn't (*should not throw exception*).

A *memory* design issue happens when there is a *memory leak* or *poor memory management*. We believe that we did not see many *memory leak* issues because most of the analyzed code was written in languages with automatic memory management.

The *method contract issues* we found were: *inconsistent API method contracts*, *wrong method precondition* and *wrong method return*. These issues match well with the *change method contract* types of design change found.

The *missing element* category consists of issues where expected design elements were not present. Specifically, the issues belonging to this category were *missing method*, *missing constructor* and *missing interface*.

Sometimes developers did not use a design pattern appropriately and these issues were marked with the concept *misused design pattern*.

A common type of issue was *placement issue*. These happened when a class was not in the right module (*poorly placed class*), a field was not in the right class (*poorly placed field*) or a method was in the wrong class or module (*wrongly placed method*).

The concept of *poor performance of design* represented an issue where the design was correct but its performance did not match the one expected by stakeholders.

Similarly to *design changes* of type *change solution change* that we explained in the previous section, there were context-dependent *design issues*. The solution design could be either a poor one (*poor solution design*) or just plain wrong (*wrong solution design*).

Unnecessary element was by far the most common design issue. There were cases of *redundant inheritance*, *unnecessary class*, *unnecessary code* (a combination of field, methods and classes), *unnecessary field*, *unnecessary method* and *unnecessary method parameter*.

This suggests that simplifying the design and reusing existing design elements is a major concern of developers.

Observation 4. Developers strive to identify unnecessary design elements during MCR.

Few of the identified design changes and issues could be seen as being related to architectural design. Contrary to our expectations, we did not observe discussions which were clearly about components, relationship between components, architectural patterns and other architectural design topics.

Observation 5. Most of the design concerns discussed during MCR are low-level design (non-architectural design).

Not Design

We identified 45 concepts representing comments that were not about software design and we grouped most of them into 6 categories (Table 4.3). To recap, the purpose of detailing what the non-design discussions were is to provide a clearer idea of what we did not consider to be design in this study.

Some of the concepts above are straightforward. *Code style* are discussions about source code formatting, indentation, spacing and etc. The *documentation* category includes not only *documentation* files but also *code comments* in the source code. *Error message* is exactly as the name implies. *Software requirements* represents discussions about what the software should do, i.e. the specification of features. *Software configuration files* mean configuration files that are not source code and that control the behavior of the software. The *testing* concept was used for all comments relating to test code in the review and software testing concerns.

The bulk of non-design discussions were about *implementation.* We grouped 12 concepts into this category and there were 95 discussions about *implementation.*

To differentiate between design and implementation, we considered method contracts as the boundary between them in this study. That is, we considered discussions about the preconditions, postconditions, return values and similar properties of a method to be design and comments regarding code strictly inside the method as implementation. For example,

Table 4.3: Main concepts and categories of non-design discussions topics. Categories start with uppercase to differentiate them from concepts.

Concept / Category	Occurrences	Reviews With It
code style	19	19
Documentation	42	29
error message	6	6
software requirements	13	13
Implementation	95	28
Logging	11	9
reference to another comment	11	11
reference to info outside this review	19	19
Review workflow	45	27
Social	20	13
software configuration files	4	4
Testing	27	27
Tools	19	14

discussing whether a method should have another parameter would be design but discussing how to optimize a for loop inside this method would be implementation.

We also considered the following concepts to be *implementation*: *implementation bugs*, *code snippets*, *naming of code entities* (e.g. classes, variables, methods and constants), *temporary code* and *use of another programming language*.

Logging consists of discussions about *adding a log point* somewhere in the code and the *log messages*.

It was common to find *references to other comments* in the review request and also *references to information outside the review request*. In particular, developers frequently cited information present in the issue tracker of the project and discussions which happened outside the tool e.g. in person and issue tracker.

Comments about the *review workflow* also comprised a significant chunk of non-design discussions. This includes diverse comments such as *asking the author to close the review request*, *declaring what parts the developer reviewed*, *stating that the dev could not understand a review comment* and approving the review request (*LGTM*).

We used the category *social* to denote comments such as *praising the review*, *praising the author's work* and *greeting the other developers*.

The *tools* category consists of comments by the developers about *build tool configuration* and *version control*, comments generated by build bots (*build tool log*) and comments generated by *review bots*.

Observation 6. Aside from design and implementation, developers frequently discuss documentation, testing and code style during MCR.

4.3 Discussion

4.3.1 Comparison with Existing Literature

One of the tenets of Grounded Theory is that researchers should avoid literature review before the study to avoid bias in the results. Therefore, although we were naturally aware of other research on design information on MCR, we only analyzed these related studies in depth after our study.

We found 13 papers related to design discussions in MCR but most of them are quantitative studies. Indeed, only 3 of them use qualitative research methods. Of these three, two present a categorization of the design information discussed [58][54] and a single one presents a model on how developers discuss design in MCR [53].

Zanaty et al.'s Design Issues (2018)

Zanaty et al. performed a mixed methods study of design discussions [58]. First, they manually analyzed a sample of 1,357 comments from OpenStack Nova and 1,460 comments from OpenStack Neutron projects. From this qualitative analysis, they found that an average of 12% of comments have design information and they created a categorization of the design issues discussed in the comments. Second, they built multiple binary classifiers of design discussions and then applied these classifiers to 2,506,308 review comments. Their classifiers had a precision of 59% to 66% and a recall of 70% to 78%. They ultimately conclude that design is not discussed as often as it should be and that most design comments are constructive.

Like Zanaty et al.'s study, our study also analyzed comments from code reviews of OSS projects and we also created a classification of design issues and a classification of non-design topics. Differently from their study, our use of the GT methodology led to a classification of design issues that is more detailed with multiple hierarchical levels and which integrates a more abstract classification which includes design changes.

All but one of the categories of design issues identified by Zanaty et al. can be seen in our classification. Their categories along with the most similar one in our model are: module coupling (design smell), redundant code (unnecessary code), performance (poor performance of design), side effect (solution design), unnecessary complexity (confusing design), code misplacement (placement issue) and shallow fix. We could not identify a category in our model that directly matches shallow fix as per their definition: "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" [58].

Viviani et al.'s Design Topics (2018)

Viviani et al. manually classified 2,378 paragraphs from 10,790 MCR discussions in order to identify design points [54]. A design point is a new concept they proposed that means "a piece of a discussion relating to a decision about a software system's design that a software development team needs to make." After that manual identification of design points, they applied open coding on 275 paragraphs about design to categorize the design topics that developers had discussed. In the end, they found that approximately 22% of the paragraphs had design information and they produced a categorization of the design topics that developers discuss.

Similarly to Viviani et al., we also analyzed comments from code reviews of OSS projects and created a classification of design information. Unlike their study, we produced a more comprehensive classification by using GT and a larger dataset and the definition of design we used seems to be considerably different.

At first, most of their categories of design information can be readily seen in our model. These along with the most similar one in our model are: code (implementation), maintainability, testing (testing), robustness (exception), performance (poor performance of design), configuration (software configuration files), documentation (documentation) and clarification (understanding design).

However, we did not consider most of their categories of design information to be design in our study. Specifically, according to the definition of design we used, we regarded code (implementation) as separate from design, maintainability was seen as one of the possible goals of a design change (to improve a non functional requirement) instead of a type of design concern, testing was deemed to not be design, configuration was not considered design and clarification are actions the developers take during design review to better understand the design concerns.

Viviani et al.'s Model of Design Discussions (2018)

Viviani et al. analyzed 3 pull requests from Rails, an OSS project, in order "to better understand the form and content of how developers discuss design" [53]. Using a process like open coding, they interpreted design discussions as revolving around finding candidate solutions to design questions. Furthermore, developers support or reject the candidate solutions presented and provide rationales for their opinions.

The model of design discussions proposed by Viviani et al. is compatible with the one in this study. Their concept of design questions and candidate solutions can be mapped respectively to design issues and proposed design changes in our model. Likewise, the process of supporting and rejecting candidate solutions according to rationales can be represented by the concept of belief in our model.

The similarity between our model and the one from Viviani et al. suggests that our results correctly represent design discussions and that they are generalizable to other organizations due to the different contexts.

Differently from the model of Viviani et al., our analysis of a larger dataset enabled us to produce a richer model with additional concepts. For instance, our model captures review actions performed by developers in the discussions such as asking and providing clarification on design concerns and we delved deeper into design changes by specifying their goals, types and temporality.

4.3.2 Differentiating between Requirements, Design and Implementation

One of the challenges of this study has been to determine if a discussion is about requirements, design or implementation. This happens primarily because we are analyzing discussions happening at the level of code so that most comments are associated with lines of code. But, even if a comment is associated with lines of code, the underlying subject of the discussion may be software requirements or software design instead of implementation.

Differentiating between design and implementation was the hardest. The reason for this is that there is not a clear and unambiguous definition of software design that is widely accepted by researchers and practitioners even though design is a fundamental part of most software processes. Despite adopting the definition proposed by Ralph and Wand[43] which aims to solve this problem, we found that it does not provide a clear separation between design and implementation. Concretely, we can perceive this difficulty by trying to answer questions such as: (1) is a discussion regarding removing a method from a class a design or an implementation discussion? (2) is the choice of a data structure a design or an implementation decision?

We mitigated the problem of differentiating between design and implementation by clearly specifying what we considered design with our classification of types of design issue and design change and by deliberately adopting method contracts as the boundary between design and implementation. In this manner, researchers and practitioners can determine exactly what we considered as design. Nevertheless, future research is needed to address this gap so that research papers on design from different organizations can be combined seamlessly into a coherent body of work.

Although discerning between requirements and design was sometimes hard, it was usually surmountable by spending more time to understand the discussion well. According to Ralph and Wand's definition of design, design is basically a specification of an object which aims to satisfy a set of requirements. The following quote shows an example of a comment which may seem to be about design at first. More specifically, it looks as if the discussion revolves around what the best API for users would be. However, on closer inspection, the developer is suggesting a change in the behavior of the 'rename' method which is provided for users of this library. We concluded, therefore, that this discussion is about software requirements, in particular the specification of a feature required by the users:

> "While POSIX guarantees that 'rename' e.g., does not see inconsistent state, there is nothing preventing 'to' from being deleted once we execute the conditional code here.
>
> Since it is hard to know what semantics users expect _in general_, it might make more sense to not add the sync behavior to 'rename', but to e.g., ask users to perform 'fsync' themself [sic]." – Review #69009, Comment #294037

4.3.3 Limitations

First, we have the limitations common to qualitative studies using Straussian grounded theory. Principally, it is possible that the researchers did not have sufficient theoretical sensitivity as required by grounded theory and failed to perceive significant concepts, categories and relationships. Moreover, the time consuming nature of qualitative studies means that only a small subset of the data available could be analyzed in comparison to quantitative studies.

Another threat to validity is that we could have misunderstood grounded theory and misapplied it which is a common occurrence according to Stol et al [49]. We manage this threat by describing in detail how we applied grounded theory during this study in the Methodology section so that other researchers can evaluate it.

Although a single researcher performed most of the open coding process, the analysis was periodically reviewed by two other more experienced software engineering researchers to strengthen its reliability. Furthermore, Straussian grounded theory methodology assumes that multiple interpretations can be extracted from the same data due to the richness of qualitative data and the fact that "different analysts focus on different aspects of data, interpret things differently and identify different meanings" [15].

Notably, at present, it is uncertain how generalizable the results from this study are outside the Apache organization. Given our personal experience with code review and that our results were compatible with the ones obtained by Viviani et al. studying pull requests from an OSS project outside the Apache organization, we believe that our model applies to other contexts but further research is needed to evaluate this.

4.4 Conclusion

Modern code review (MCR) is a promising source of design information which is currently underutilized due to the difficulty of manually extracting such information and the absence of automatic tools for that. In order to further research on how to facilitate the extraction of design information, it is necessary to learn how developers discuss design in code review and what topics of design are discussed.

Given the dearth of qualitative studies on this topic, we conducted a qualitative study using Straussian Grounded Theory (GT) to characterize how developers discuss design during code reviews and what topics of design are discussed in the reviews.

Accordingly, this study resulted in a model of how design is reviewed in MCR and a classification of types of design information discussed in code review. The results of this study are not only compatible with previously published literature but also presents new concepts. In addition, we presented the major challenges and limitations of this study in order to help future research in this area.

Chapter 5

Conclusion

In this thesis, our goal was to investigate Modern Code Review (MCR) and the design discussions that occur as part of MCR. To this end, we conducted a characterization of design discussions in MCR using a dataset of design discussions from the Apache Software Foundation.

5.1 Contributions

Our major contributions were in summary:

- **Model of design discussions in MCR:** a model of how developers discuss design during MCR;
- **Classification of design in MCR:** a classification of types of design issues and design changes discussed in MCR;
- **Dataset of design discussions:** a public dataset of 108,458 design discussions from the Apache Software Foundation that can be used in other studies.

Importantly, our characterization study of design discussions in MCR resulted in the following key findings:

1. A design review within MCR revolves around a set of design concerns (changes and issues);

2. Developers seek to improve and understand the design during design review within MCR;
3. Developers engage in discussions over design concerns, which are ultimately accepted or rejected;
4. Developers strive to identify unnecessary design elements during MCR.
5. Most of the design concerns discussed during MCR are low-level design (non-architectural design);
6. Aside from design and implementation, developers frequently discuss documentation, testing and code style during MCR.

5.2 Future work

There are a number of opportunities for future work in the area of design discussions in MCR which can be helped by this work.

Future studies could be made to improve upon our characterization study of design in MCR. The immediate next step would be to conduct further studies to validate the theory of how developers discuss design in MCR. For instance, a survey could be conducted with developers to evaluate if they agree with it. A replication study could also be conducted on a different dataset to evaluate the generalizability of the theory to other software development contexts. In particular, we have not analyzed review discussions from closed source software projects in our study. Another possibility is a large study focused solely on producing a more comprehensive classification of types of design information in MCR.

As for the problem of automatic classifying design discussions in MCR, we believe that the next logical step would be to investigate multiclass classifiers capable of identifying what type of design issue or change is being discussed. We plan to leverage the results of our characterization study of design in MCR to research multiclass classifiers of design discussions.

Due to the rapid advances in the field of natural language processing, it would be interesting to examine the viability of automatic summarization of design information in MCR because such a tool would probably be of great help to practitioners.

Regarding secondary studies in the area of MCR, an update of our systematic mapping would be interesting given that it only covers papers published up to early 2018. Also, a systematic review of binary classifiers of design discussions would also be valuable because there are several papers proposing and evaluating such classifiers as of today.

Finally, the whole dataset used in this thesis is available to aid future research and to encourage replication at: `https://sites.google.com/view/ch-design-discuss-thesis/`

This work was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

Bibliography

[1] Explosion AI. spaCy · Industrial-strength Natural Language Processing in Python.

[2] Alberto Bacchelli and Christian Bird. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 712–721, Piscataway, NJ, USA, 2013. IEEE Press.

[3] Deepika Badampudi, Ricardo Britto, and Michael Unterkalmsteiner. Modern code reviews - Preliminary results of a systematic mapping study. In *Proceedings of the Evaluation and Assessment on Software Engineering*, pages 340–345, Copenhagen Denmark, April 2019. ACM.

[4] Sebastian Barney, Kai Petersen, Mikael Svahnberg, Aybüke Aurum, and Hamish Barney. Software quality trade-offs: A systematic map. 54(7):651–662.

[5] Gabriele Bavota and Barbara Russo. Four eyes are better than two: On the impact of code reviews on software quality. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 81–90. IEEE, 2015.

[6] Inc. Beanbag. Review Board. `https://www.reviewboard.org/`.

[7] 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.

[8] Pierre Bourque and Richard E. Fairley, editors. *SWEBOK: Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, Los Alamitos, CA, version 3.0 edition, 2014.

[9] 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*, MSR 2014, pages 340–343, New York, NY, USA, 2014. ACM. event-place: Hyderabad, India.

[10] D. Budgen. *Software Design*. International computer science series. Pearson/Addison-Wesley, 2003.

[11] David Budgen, Mark Turner, Pearl Brereton, and Barbara Kitchenham. Using mapping studies in software engineering. In *Proceedings of PPIG*, volume 8, pages 195–204. Lancaster University.

[12] M. Ciolkowski, O. Laitenberger, and S. Biffl. Software reviews: The state of the practice. *IEEE Software*, 20(6):46–51, November 2003.

[13] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. SEI Series in Software Engineering. Pearson Education, 2010.

[14] Flavia Coelho, Tiago Massoni, and Everton L. G. Alves. Refactoring-aware Code Review: A Systematic Mapping Study. In *Proceedings of the 3rd International Workshop on Refactoring*, IWOR '19, pages 63–66, Piscataway, NJ, USA, 2019. IEEE Press. event-place: Montreal, Quebec, Canada.

[15] Juliet Corbin and Anselm Strauss. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Sage Publications, Inc, 3 edition.

[16] Frank Elberzhager, Jürgen Münch, and Vi Tran Ngoc Nha. A systematic mapping study on the combination of static and dynamic quality assurance techniques. 54(1):1–15.

[17] M.E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[18] Apache Software Foundation. All Review Requests | Review Board. `https://reviews.apache.org/r/`.

[19] Apache Software Foundation. The Apache Software Foundation. https://www.apache.org/.

[20] Martin Fowler. *Refactoring*. Addison-Wesley Signature Series (Fowler). Addison-Wesley, Boston, MA, 2 edition, 2018.

[21] Victor Freire. *Automatic Decomposition of Code Review Changesets in Open Source Software Projects*. PhD thesis, Universidade Federal de Campina Grande, Campina Grande, September 2016.

[22] Gerrit team. Gerrit. https://www.gerritcodereview.com/. [Online; accessed 01-Jan-2021].

[23] GitHub. Github's features. https://github.com/features. [Online; accessed 01-Jan-2021].

[24] Barney G. Glaser and Anselm L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine de Gruyter, New York, NY, 1967.

[25] Google LLC. Google scholar. https://scholar.google.com/citations?view_op=top_venues&hl=en&vq=eng_softwaresystems. [Online; accessed 29-Aug-2019].

[26] A. Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2017.

[27] L. Harjumaa, I. Tervonen, and A. Huttunen. Peer Reviews in Real Life - Motivators and Demotivators. pages 29–36. IEEE, 2005.

[28] D. Jurafsky and J.H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall Series in Artifi. Pearson Prentice Hall, 2009.

[29] B. Kitchenham, P. Brereton, and D. Budgen. Mapping study completeness and reliability - a case study. In *16th International Conference on Evaluation Assessment in Software Engineering (EASE 2012)*, pages 126–135.

[30] Barbara Kitchenham, Pearl Brereton, and David Budgen. The educational value of mapping studies of software engineering literature. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 589–598. ACM. event-place: Cape Town, South Africa.

[31] Barbara A. Kitchenham, David Budgen, and O. Pearl Brereton. Using mapping studies as the basis for further research–a participant-observer case study. 53(6):638–651.

[32] S. MacDonell, M. Shepperd, B. Kitchenham, and E. Mendes. How reliable are systematic reviews in empirical software engineering? 36(5):676–687.

[33] Alvi Mahadi. *Conclusion stability for natural language based mining of design discussions*. Thesis, 2021. Accepted: 2021-02-12T05:11:37Z.

[34] Alvi Mahadi, Karan Tongay, and Neil A. Ernst. Cross-Dataset Design Discussion Mining. *arXiv:2001.01424 [cs]*, January 2020. arXiv: 2001.01424.

[35] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. Scoring, term weighting, and the vector space model. In *Introduction to Information Retrieval*, pages 100–123. Cambridge University Press, 2008.

[36] Mayring, Philipp and Fenzl, Thomas. Qcamap. `https://www.qcamap.org/`. [Online; accessed 10-Aug-2019].

[37] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[38] Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[39] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. Systematic mapping studies in software engineering. In *12th International Conference on Evaluation and Assessment in Software Engineering*, volume 17, page 1.

[40] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. 64:1–18.

[41] Phacility, Inc. Phabricator. `https://www.phacility.com/phabricator/`. [Online; accessed 01-Jan-2021].

[42] Roger S. Pressman. *Software engineering: a practitioner's approach.* McGraw-Hill series in computer science. McGraw Hill, Boston, Mass, 5th ed edition, 2000.

[43] Paul Ralph and Yair Wand. A Proposal for a Formal Definition of the Design Concept. In Kalle Lyytinen, Pericles Loucopoulos, John Mylopoulos, and Bill Robinson, editors, *Design Requirements Engineering: A Ten-Year Perspective*, Lecture Notes in Business Information Processing, pages 103–136, Berlin, Heidelberg, 2009. Springer.

[44] N. Rozanski and E. Woods. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley, 2012.

[45] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 3 edition, 2010.

[46] Abbas Shakiba, Robert Green, and Robert Dyer. FourD: Do Developers Discuss Design? Revisited. In *Proceedings of the 2Nd International Workshop on Software Analytics*, SWAN 2016, pages 43–46, New York, NY, USA, 2016. ACM. event-place: Seattle, WA, USA.

[47] F. Shull and C. Seaman. Inspecting the History of Inspections: An Example of Evidence-Based Technology Diffusion. *IEEE Software*, 25(1):88–90, January 2008.

[48] I. Sommerville. *Software Engineering*. Always learning. Pearson, 2016.

[49] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. Grounded theory in software engineering research: a critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 120–131, Austin, Texas, May 2016. Association for Computing Machinery.

[50] A. Sutherland and G. Venolia. Can peer code reviews be exploited for later information needs? In *2009 31st International Conference on Software Engineering - Companion Volume*, pages 259–262, May 2009.

[51] A. Tahir and S. G. MacDonell. A systematic mapping study on dynamic metrics and software quality. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 326–335.

[52] Giovanni Viviani, Michalis Famelis, Xin Xia, Calahan Janik-Jones, and Gail C. Murphy. Locating Latent Design Information in Developer Discussions: A Study on Pull Requests. *IEEE Transactions on Software Engineering*, pages 1–1, 2019. Conference Name: IEEE Transactions on Software Engineering.

[53] Giovanni Viviani, Calahan Janik-Jones, Michalis Famelis, and Gail C. Murphy. The structure of software design discussions. pages 104–107. ACM, May 2018.

[54] Giovanni Viviani, Calahan Janik-Jones, Michalis Famelis, Xin Xia, and Gail C. Murphy. What Design Topics Do Developers Discuss? In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, pages 328–331, New York, NY, USA, 2018. ACM. event-place: Gothenburg, Sweden.

[55] Roel Wieringa, Neil Maiden, Nancy Mead, and Colette Rolland. Requirements engineering paper classification and evaluation criteria: a proposal and a discussion. 11(1):102–107.

[56] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. pages 1–10. ACM Press.

[57] Claes Wohlin, Per Runeson, Paulo Anselmo da Mota Silveira Neto, Emelie Engström, Ivan do Carmo Machado, and Eduardo Santana de Almeida. On the reliability of mapping studies in software engineering. 86(10):2594–2610.

[58] Farida El Zanaty, Toshiki Hirao, Shane McIntosh, Akinori Ihara, and Kenichi Matsumoto. An Empirical Study of Design Discussions in Code Review. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering*

and Measurement, ESEM ’18, pages 11:1–11:10, New York, NY, USA, 2018. ACM. event-place: Oulu, Finland.

Appendix A

A Systematic Mapping of Modern Code Review

Although several papers had been published on MCR by 2018, there was not any systematic mapping available on the field at the time. Thus, we decided to perform a systematic mapping not only to fill this research gap but also to evaluate the relevance of our thesis topic.

A.1 Methodology

We used the systematic mapping (SM) methodology described in the guidelines of Petersen et al. [39][40], which will be described in more detail in the following subsections. Briefly, first, we created the protocol for the systematic mapping specifying the whole process, e.g. the goal, the research questions, the search strategy and the inclusion and exclusion criteria. Second, we searched for studies about MCR using electronic data sources and the snowball search method. During this search, we evaluated the potential papers against the inclusion and exclusion criteria. Finally, we examined each of the relevant papers to: (1) create a spreadsheet with their metadata and topic independent classification and (2) create a topic dependent classification by the process of keywording the abstracts.

A.1.1 Research Questions

The goal of this study is to classify and summarize the published research on MCR in order to provide researchers with an overview of the field and help them identify opportunities for further studies. With this purpose, we established the following research questions:

- RQ1: Where and when were MCR studies published?
- RQ2: What researchers and organizations are most active in MCR research?
- RQ3: What are the most investigated MCR topics?
- RQ4: What types of research methods are used in MCR research?
- RQ5: What types of contributions are made in MCR research?
- RQ6: Which context is more used in MCR studies: open source or closed source?
- RQ7: What are the most referenced (influential) papers?

A.1.2 Study Search

We used multiple data sources and combined the results of two different search strategies to maximize the probability of finding all relevant studies.

Electronic Data Source Search

We performed searches for MCR papers in multiple websites. Specifically, we used four publisher websites (ACM Digital Library, IEEE Xplore Digital Library, ScienceDirect and SpringerLink) and one search engine (Google Scholar).

For these searches, we used the query: "code review". We had to use the general term "code review" instead of "modern code review" because some papers do not use the term "modern code review" even though they are about MCR.

Snowball Search

The snowball search procedure consists of iteratively examining the backwards and forward references of relevant papers until they have all been examined. For a given study, the backward references are the papers it cites, i.e. the ones listed in its References section and the forward references are the papers which cite the study [56]. Google Scholar was used to identify the papers which cite a given study.

We used 7 popular papers about MCR as the start set and then iteratively examined the backward and forward references of these papers and of the subsequent ones that we found. These 7 papers are highly cited and they had been published between 2008 and 2016 in mostly different venues. According to Wohlin, selecting a good start set is still an open research problem but it should contain diverse and highly cited papers to minimize the chance of missing relevant papers [56].

A.1.3 Study Selection

During the search for studies about MCR, we evaluated the potential papers against the inclusion and exclusion criteria. For each paper, we read the abstract and, when necessary, the full paper to determine if it matched the criteria set in the protocol of the study.

Inclusion Criteria

- Study is about Modern Code Review (MCR) as defined by Bachelli and Bird: "review that is (1) informal (in contrast to Fagan-style), (2) tool-based, and that (3) occurs regularly in practice nowadays, for example at companies such as Microsoft, Google, Facebook, and in other companies and OSS projects" [2]. It is about the practice of MCR or a tool or technique for supporting the activity.

- Study was peer-reviewed, i.e. conference papers, journal papers and chapters from peer-reviewed books.

- Study was written in English and is available in full-text.

Exclusion Criteria

- Paper is not about MCR (e.g., Fagan's inspections, peer review of school assignments).
- Paper superficially mentions MCR but does not present a clear contribution to the field.
- Secondary studies (e.g., systematic literature reviews and surveys).
- Studies in short formats (e.g., tutorial papers, long abstract papers and poster papers).
- Shorter versions of the same study will not be considered (e.g. if a study is first published as a conference paper and later an extended version of it is published in a journal, only the journal version will be considered).
- Non-research material (e.g. magazine articles and research proposals).

A.1.4 Data Extraction

For each selected study, we extracted its metadata and classified it according to two classification schemes: a topic independent one (applicable to other fields of study) and a topic dependent one (exclusive to Modern Code Review).

Metadata

We filled a spreadsheet containing the following metadata columns for each paper: title, author, affiliations, year, publication type, venue title, venue abbreviation, number of citations.

Topic Independent Classification

A topic independent classification is a general classification scheme that can be applied to several different fields of study, i.e. it is not specific to Modern Code Review. In this study, we classified papers according to research method, research type, contribution type and context (whether the software examined had an open or closed source license) [40]. We recorded this classification in the same spreadsheet for the metadata of the papers.

Topic Dependent Classification

A topic dependent classification is a custom classification scheme created specifically for a certain topic. We created a topic dependent classification scheme for MCR by using a method similar to open coding on the abstracts of the papers, which Petersen et al. [40] called keywording. We read each of the abstracts and assigned concepts that summarized what a given paper is about. These concepts were later grouped into categories to summarize the topics studied in MCR research even further. For example, one of the abstracts we found had the following passage: "We performed a study that looked at the code review practices of software product teams at Microsoft" [50]. From this passage, we identified the concept *the process of code review* and this concept was later grouped into the more abstract category *Review Process* alongside concepts such as *classification of code review processes*.

The process of creating this classification was supported by QCAmap, a free web-based tool for qualitative content analysis [36].

A.2 Results

We initially found 164 studies using electronic data search and 211 studies using snowball search. After combining the results and removing duplicates, there were 243 studies left. During the keywording process, when analyzing the studies in more depth, we noticed there were still some irrelevant studies. After removing these, we finally had a total of 177 papers which fit our inclusion and exclusion criteria. These 177 papers are used to answer the research questions of this study.

A.2.1 RQ1: Where and when were MCR studies published?

For this research question, we want to know the venues where MCR studies are being published in and the number of MCR studies published throughout the years.

Venues

Papers about MCR have been published in a total of 95 venues. Table A.1 shows the 12 venues with most papers about MCR. These 12 most popular venues are focused on the field

Table A.1: The top 12 most popular venues in MCR research.

#	Venue	Papers
1	MSR	17
2	ICSE	12
3	ICSME	9
4	APSEC	7
5	FSE	7
6	SANER	6
7	ESEM	5
8	ASE	4
9	Empir Software Eng	4
10	CHASE	3
11	ICSE-C	3
12	OSS	3

of Software Engineering except for MSR, which is about mining and analyzing data from software repositories.

Few of the venues found are commonly targeted for publication of MCR research. More specifically, only 26 (27.4%) venues had more than one paper about MCR published in them and only 12 (12.6%) venues had more than two papers about MCR.

Only 32 (18.1%) of the 177 papers have been published in journals. This indicates that research in the field of MCR is still far from comprehensive and there is ample opportunity for future research.

Years

The first paper about MCR was published in 2002 and a total of 177 MCR papers have been published between the years 2002 and 2018.

There has been significant growth and interest in the field of MCR according to the histogram of MCR papers over the years (Figure A.1). Since 2006, the number of papers published about MCR has never decreased from one year to another and since 2011, the number of papers published about MCR has been growing year after year.

We omit the year 2018 from the histogram to avoid the erroneous impression that there was a sudden drop in the number of papers in that year. Very few papers from 2018 are present in the dataset, because the study selection phase started in July/2017 and ended in April/2018.

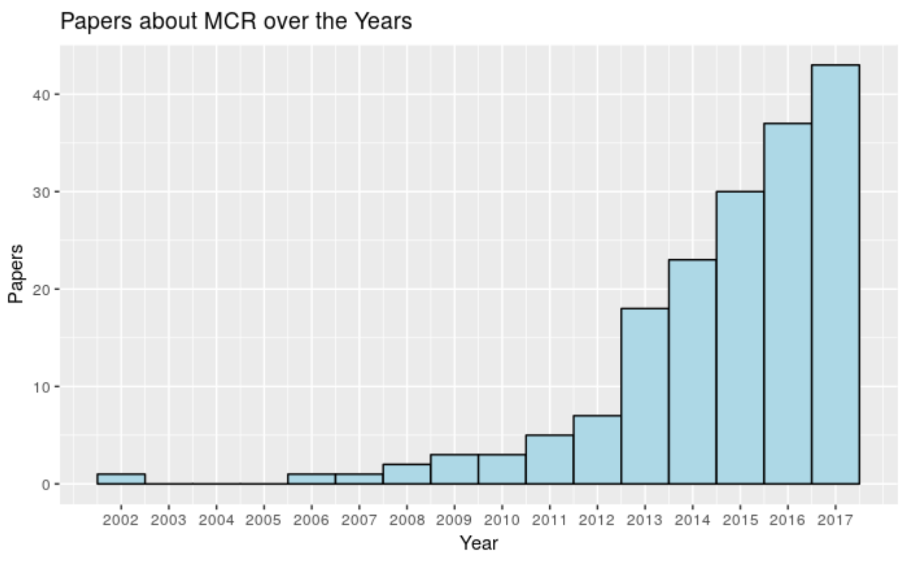


Figure A.1: Histogram showing the distribution of papers over the years.

A.2.2 RQ2: What researchers and organizations are most active in MCR research?

Organizations

152 organizations have published papers in the field of MCR. Table A.2 shows the 10 organizations who authored the most papers about MCR.

Few organizations are responsible for most of MCR research. 61 (34.5%) of the papers have been published by the top 8 (5.0%) organizations and 106 (59.9%) of the papers have been published by the top 30 (20.0%) organizations.

Most organizations are not frequent contributors to research on MCR. 98 (64.5%) of the 152 organizations only published a single paper on MCR.

Table A.2: The top 10 most active organizations in MCR research.

#	Organization	Papers
1	Nara Institute of Science and Technology	16
2	Microsoft	13
3	Delft University of Technology	7
4	Osaka University	7
5	University of Alabama	7
6	Bitergia	6
7	Leibniz Universität Hannover	6
8	National University of Defense Technology	6
9	Polytechnique Montréal	6
10	Universidad Rey Juan Carlos	6

Researchers

A total of 378 researchers have published papers in the field of MCR. Table A.3 shows the top 10 researchers by activity in the field.

Similarly to the distribution of organizations, few researchers are responsible for most of MCR research. 72 (40.7%) of the papers have been published by the top 19 (5.0%) researchers and 124 (70.1%) of the papers have been published by the top 76 (20.0%) researchers.

Likewise, most researchers are not frequent contributors to research on MCR. 277 (73.3%) of the 378 researchers only worked on a single paper about MCR.

The most active researchers are not necessarily the most popular ones. Table A.4 shows the top 10 most referenced researchers on MCR. Indeed, only 3 researchers are among the top 10 most active and the top 10 most referenced authors.

A.2.3 RQ3: What are the most investigated MCR topics?

To answer this research question, we created a topic dependent classification scheme as detailed in Section A.1.4. We identified a total of 148 concepts (Table A.5), which we

Table A.3: The top 10 most active researchers in MCR research.

#	Researcher	Papers
1	Iida, H.	12
2	Kula, R. G.	10
3	Bird, C.	9
4	Bosu, A.	8
5	Yoshida, N.	8
6	Bacchelli, A.	7
7	Thongtanunam, P.	7
8	Wang, H.	7
9	Yin, G.	7
10	Yu, Yue	7

Table A.4: The top 10 most referenced authors in MCR research.

#	Researcher	Papers
1	Bird, C.	823
2	Rigby, P. C.	546
3	Bacchelli, A.	535
4	Hassan, A. E.	368
5	German, D. M.	366
6	Storey, M. A.	366
7	Adams, B.	346
8	Iida, H.	333
9	Kamei, Y.	262
10	McIntosh, S.	250

Table A.5: The top 10 most investigated concepts in MCR research.

#	Concept	Frequency in Papers
1	the process of code review	28.8%
2	tool support for code review	24.9%
3	GitHub	16.9%
4	reviewer recommendation	12.4%
5	pull requests	11.9%
6	outcomes of code review	10.7%
7	review participation	10.7%
8	machine learning	10.7%
9	understanding of the code under review	10.7%
10	recommendations for code review	9.0%

grouped into 24 categories (Table A.6). The terms *review* and *code review* in the names of the concepts mean *modern code review*. Most of the categories are easily understood by their names, so we only detail some of them in the next paragraphs.

The most popular category is *Review Process* (Table A.6). Over 50% of the papers about MCR focus on the process of MCR, e.g. they analyze the practices used in real world projects, classify different MCR processes or recommend best practices for MCR.

The second most popular category is *Techniques*. When a paper is associated with this category, it means that the paper studies the application of some technique to MCR. The most common techniques being applied to MCR are: *machine learning* (10.7%), *program analysis* (8.5%), *social network analysis* (5.6%) and *natural language processing* (5.6%). It is not surprising that machine learning ranks so high, since it has been frequently applied to widely different fields in recent years.

The third most popular category is *Open Source*. Papers in this category can be about one of the following three topics: (1) *analyze data from GitHub* (16.9%), (2) *analyze pull requests* (11.9%) or (3) *comparison of MCR in open source projects with MCR in closed source software projects* (3.4%).

Table A.6: The categories of concepts investigated in MCR research.

#	Category	Frequency in Papers
1	Review Process	50.8%
2	Techniques	49.7%
3	Open Source	32.2%
4	Review Tools	32.2%
5	Process Metrics	30.5%
6	Non-technical Factors of Code Review	20.3%
7	Understanding	20.3%
8	Improving Effectiveness	18.1%
9	Impact of Code Review	17.5%
10	Reviewer Metrics	17.5%
11	Automation of Review Tasks	14.7%
12	Code Quality	14.7%
13	Review Repository Mining	13.6%
14	Review Comments	10.2%
15	Changeset Metrics	9.0%
16	File Metrics	9.0%
17	Review Adoption	7.9%
18	Security	7.3%
19	Comparison with Other QA Activities	4.0%
20	Architecture/Design	3.4%
21	Knowledge Management	2.8%
22	MCR in Different Contexts	2.8%
23	Author Metrics	2.8%
24	Code Ownership	2.3%

The fourth most popular category, *Review Tools*, is about tools for supporting MCR. Concepts within this category include the limitations of MCR tools and comparison of different MCR tools.

The fifth most popular category, *Process Metrics*, encompasses metrics of the process of MCR. For example, the concept *time taken to review* is part of this category and it is a metric of how much time it takes for developers to review a changeset. Other examples of concepts within this category include *review cost* and *number of reviews that find defects*.

The sixth most popular category, *Non-technical Factors of Code Review*, is about the human and social factors that affect code review such as conflicts between developers, how developers perceive one another and the sentiments embedded in review comments.

A.2.4 RQ4: What types of research methods are used in MCR research?

Research Methods

The six research methods described in the guidelines of Petersen et al. [40] have been used in the field of MCR. These are, in order of popularity: case study, experiment, survey, prototyping, mathematical analysis and simulation.

The most popular research method is the *case study* which has been used in 98 (55.4%) of the papers (Figure A.2). The second most popular method is the *experiment* which has been used in 40 (22.6%) of the papers. The *experiment* method is less than half as common as the *case study* method.

The sum of the papers in Figure A.2 is larger than the total number of papers because some papers used multiple research methods.

Research Types

We applied the classification of research types proposed by Wieringa et al. [55], according to which there are six possible types: evaluation research, validation research, solution proposal, experience paper, opinion paper and philosophical paper. Both *evaluation research* and *validation research* denote research which empirically validates a solution, however *evaluation research* is used when the solution is validated in a real world context and *val-*

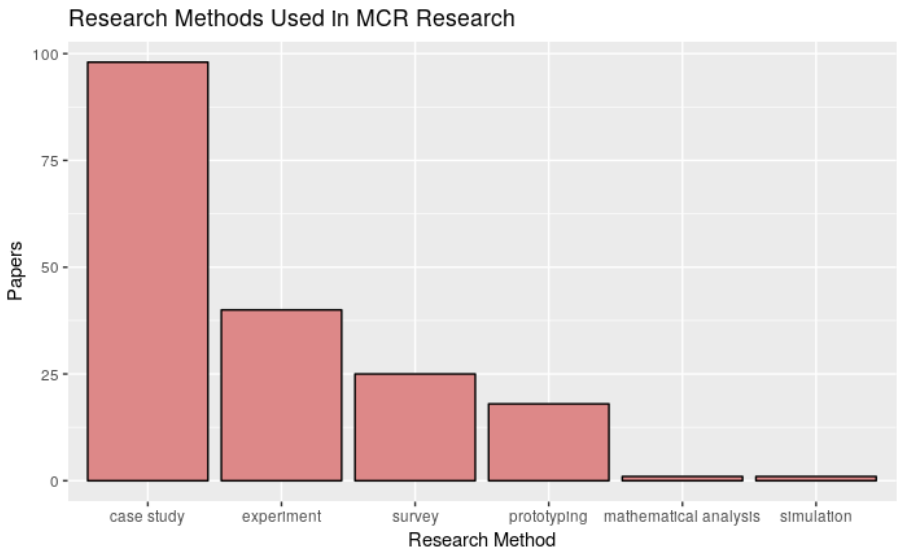


Figure A.2: Bar plot comparing the usage of different research methods in MCR research.

idation research for when the solution is validated in a laboratory setting [40]. A *solution proposal* is a study that presents a solution but only provides superficial validation such as small toy example. The meaning of the other research types can be inferred from their names.

All six types of research have been performed in the field of MCR. The most popular research type is by far the *evaluation research* which has been used in 77.4% of the papers (Figure A.3). The second most popular method is the *solution proposal* which has been used in 9.04% of the papers.

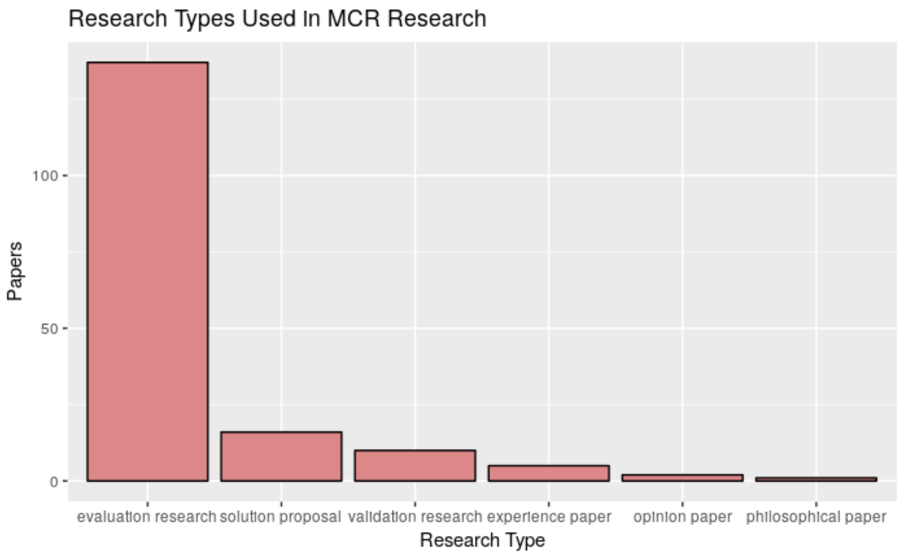


Figure A.3: Bar plot comparing the usage of different research types in MCR research.

A.2.5 RQ5: What types of contributions are made in MCR research?

The contribution type is the type of object being studied by a paper. Petersen et al. classified the contributions from papers in five types: process, method, tool, model and metric [40]. We defined a sixth type called *dataset* after observing a number of papers whose main focus were valuable datasets that the authors had produced for research.

The most popular contribution type is *process* which has been used in 71 (40.1%) of the 177 papers (Figure A.4). The second and third most popular contribution types are, respectively, *method* which has been used in 52 (29.4%) of the 177 papers and *tool* which has been used in 28 (15.8%) of the 177 papers.

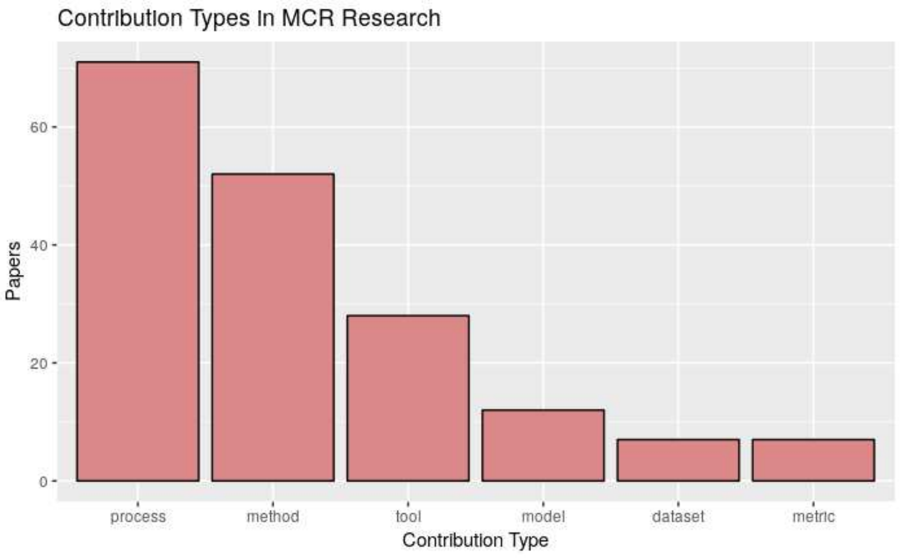


Figure A.4: Bar plot comparing the usage of different contribution types in MCR research.

A.2.6 RQ6: Which context is more used in MCR studies: open source or closed source?

For this research question, we classified the context of an MCR study as being either open source software (OSS) or closed source software (CS) and we want to know whether MCR research tends to focus on one context more than the other.

Open source software is the most popular context for MCR research by far, having been studied in 144 (81.4%) of the 177 papers, while the context of closed source software (CS) was studied in just 40 (22.6%) of the 177 papers (Figure A.5). 9 papers have analyzed both OSS and CS contexts.

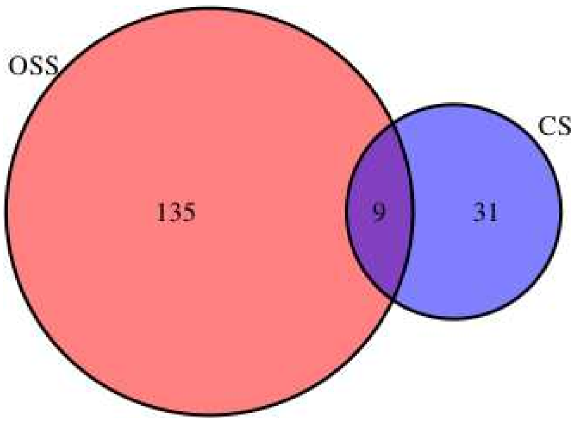


Figure A.5: Venn diagram comparing the number of papers analyzing MCR in Open Source Software (OSS) projects and Closed Source Software (CS) projects.

Most research on MCR in a closed source context is done by a few organizations. Of the 54 organizations who have published more than one paper about MCR, only 8 organizations have authored more than one paper about MCR within a closed source software context. Furthermore, these 8 organizations have published 32 (80.0%) of the papers with a closed source context.

A.2.7 RQ7: What are the most referenced (influential) papers?

For this research question, we analyzed the most cited papers about MCR. We wanted to know what they are about and what they have in common.

Table A.8 shows the top 10 most referenced papers on MCR along with how much they were cited according to Google Scholar in March 2019.

Six of the most referenced papers were written by the same scientific collaboration network (Papers 1, 2, 4, 5, 6, 8 in Table A.8). Considering a graph where the 10 most referenced papers are the vertices and the edges indicate which papers have at least one author in common, then these six papers would form a connected component and the remaining papers would not be linked to any other.

Two organizations have collaborated with more than one of the 10 most referenced papers. Papers 1, 4, 6, 7 have at least one author affiliated with Microsoft and papers 2 and 5

Table A.7: The 8 organizations who published multiple papers about MCR within a closed source context and are responsible for 80.0% of the research with this context.

#	Organization	Papers	OSS	CS
1	Microsoft	13	6	10
2	Leibniz Universitat Hannover	6	0	6
3	Vienna University of Technology	4	0	4
4	University of Saskatchewan	5	3	3
5	University of Mannheim	3	0	3
6	Kyushu University	3	2	2
7	University of Alabama	7	6	2
8	North Carolina State University	4	2	2

Table A.8: The top 10 most referenced papers in MCR research.

#	Researcher	Papers
1	Expectations, Outcomes, and Challenges of Modern C...	323
2	Open Source Software Peer Review Practices: A Case...	176
3	A large-scale empirical study of just-in-time qual...	162
4	Learning Natural Coding Conventions	145
5	Understanding broadcast based peer review on open...	139
6	Convergent Contemporary Software Peer Review Pract...	136
7	How Do Software Engineers Understand Code Changes?...	103
8	Modern Code Reviews in Open-source Projects: Which...	99
9	Reducing Human Effort and Improving Quality in Pee...	96
10	Collaboration, peer review and open source softwar...	88

have at least one author affiliated with the University of Victoria.

The most referenced papers were all published in high impact venues. These venues, along with their h5-indexes, are: ACM/IEEE International Conference on Software Engineering (75), ACM SIGSOFT International Symposium on Foundations of Software Engineering (51), IEEE Transactions on Software Engineering (48), Mining Software Repositories (38) and Information Economics and Policy (19) [25].

The distribution of research methods and types of the 10 most cited papers is very similar to the distribution in the general population of MCR papers. As research method, most papers use *case study* (5 of the 10 papers) and, as research type, the papers are almost all classified as *evaluation research* (9 of the 10 papers).

The most predominant contribution type of these most cited papers is *process*. 6 of the 10 papers studied the process of MCR, 2 contributed models for MCR and the remaining 2 studied methods to support MCR.

Regarding the context of the papers, 5 papers analyze an open source context exclusively, 3 papers studying a closed source context exclusively and 2 analyzed both open source and closed source contexts. It is noteworthy that of the 9 papers in the whole dataset which analyze both open and closed contexts, 2 are part of the top 10 most cited papers. These two papers consists of studies which analyzed a large amount of data from multiple and diverse projects.

According to the topic dependent classification, most of the top cited papers study: the process of MCR (6 papers), the impact of MCR on the software project (4 papers), tools for supporting MCR (4 papers), the application of a certain technique to MCR (3 papers), how to automate review tasks (2 papers), the effect of MCR on code quality (2 papers), how to improve the effectiveness of MCR (2 papers), the non-technical factors of MCR (2 papers), MCR in open source software versus MCR in closed source software (2 papers) and understanding of the code under review (2 papers).

A.3 Discussion

In this section, we summarize the main findings of this study and their implications for researchers.

A.3.1 Main Findings

The field of Modern Code Review is growing fast. A total of 177 papers about MCR have been published since 2002 and research on MCR has been growing steadily every year since 2011. Also, most papers so far have been published in conferences and workshops, which suggests that the field is still in its infancy. Overall, this indicates that the field of MCR is promising and relevant and that there is ample opportunity for future research.

Several researchers and organizations are involved in MCR research, however most are infrequent contributors. Of the 378 researchers and 178 organizations involved in MCR research, 73.3% of the researchers and 64.5% of the organizations have participated in just a single paper about MCR.

Researchers are studying a wide variety of topics within MCR. Our topic dependent classification groups MCR papers according to 148 concepts and 24 categories which were extracted from the abstract of the papers. The most investigated topics are the process of MCR, the application of specific techniques (e.g. machine learning and natural language processing) to parts of the MCR process, the use of MCR in OSS projects and tools for supporting the activities of MCR.

Almost 80% of the research provides empirical validation of their results in a real world setting. This is a very positive finding because it indicates that the results of MCR research tend to be reliable and grounded in real world data.

Most papers study the process of MCR (40.1%). This is a positive finding because it indicates that the investigation of possible tools and methods for supporting MCR is well grounded in research of the process of MCR and its outcomes and challenges.

Although process is the most frequent contribution type for papers, papers studying methods and tools are also common. When counting together papers that study methods and tools for supporting MCR, they account for 45.2% of all MCR papers, which is more than papers studying the process of MCR. This is evidence that research in this field is not only concerned with understanding MCR but also with improving it.

MCR is considerably more studied in an OSS context than in a closed source software (CS) context. Only 22.6% of the papers studied MCR within an OSS context. Moreover, we have identified that 8 organizations are responsible for publishing 80.0% of the papers using a CS context. We hypothesize that this happens because data from the review process of OSS projects is often readily available online unlike CS projects.

The most cited MCR papers usually describe and characterize the process of MCR and were published in high impact venues. It is likely that this happens because papers about MCR usually cite at least one paper which describes and explains the process of MCR for purposes of context and motivation. Since these studies about the process of MCR were published in popular venues and are well written and reviewed, they are usually cited in papers about MCR.

A.3.2 Implications for Researchers

Researchers interested in MCR should focus their attention on the 12 venues listed in A.1, given that they are the only ones which had at least 3 papers about MCR published at the time of the data collection phase.

MCR researchers should follow the most active and popular organizations and researchers (Section A.2.2) in order to keep up to date with new developments in the field of MCR. Also, our list of researchers and organizations involved in MCR may also be useful for identifying potential collaborators.

Our topic dependent classification (Section A.2.3) allows the researcher to quickly identify MCR papers of interest to him. For instance, a researcher interested in code understanding during the process of code review can retrieve from the classification which papers contain the concept *understanding of the code under review*. Thus, our classification facilitates literature review.

Considering the disproportionate amount of papers studying MCR solely within an OSS context (77.4%), researchers would do well to focus on a closed source software context in future studies in order to confirm that the findings are generalizable to that context. In particular, we believe that a paper publishing a dataset of MCR data from CS projects would be very useful, since there were only 7 datasets papers published in the field and they were all built from OSS projects.

MCR researchers will likely find useful to be familiar with the papers which we identified as the most referenced in the field (A.8). The majority of these papers talk about the process of MCR, so they should be relevant for most research in the field.

A.3.3 Threats to Validity

As with any research study, there are many possible threats to the validity, but we tried to mitigate them as far as possible.

To begin with, only one author selected the studies for analysis, so the selection might be biased. That is, although we have clearly defined inclusion and exclusion criteria to avoid this, it is still a possibility that irrelevant studies were deemed to fit the criteria and be about MCR and that relevant studies were wrongly considered to not fit the criteria.

Further, even though we used two search strategies in a systematic way, some relevant studies may not have been found. It is not impossible that there exists relevant studies about MCR which were not cited by any other in our dataset and which did not match the keywords we used in the electronic data search.

For the data extraction phase, in general, the possible threats to validity are that we wrongly filled entries in the data extraction spreadsheet or that we misclassified papers.

For the metadata extraction of the selected studies, we manually checked that each field was correct in order to avoid errors.

As for the topic independent classification process, it is possible that some papers were misclassified since we do not read each paper fully in a conventional systematic mapping study. In particular, the distinction between papers contributing methods and papers contributing tools is not always clear (Section A.2.5). For instance, sometimes, the authors of a given paper state that their main contribution is a tool, but this tool is actually a prototype implementation of their new method for supporting code review. Therefore, it is possible some papers were misclassified as mainly contributing a tool instead of a method and vice-versa.

The topic dependent classification process is vulnerable to a number of problems, e.g. we might have missed important concepts, might have given poor names to concepts, might have conceived poor categories and so forth. That is because the process is based on open coding from grounded theory [40], which is a qualitative process whose results vary according to the interpretation, skill and background of the researchers involved [15].

A.4 Related Work

In this section, we describe two other systematic mappings (SM) related to MCR. Because there are only two directly related studies, we also briefly summarize in this section: studies about the use of SM studies in Software Engineering and studies reporting SM in the more general field of Software Quality which encompasses MCR.

A.4.1 Systematic Mappings in Modern Code Review

When we conducted our study, there were not any published systematic mappings related to MCR. However, before we published our study, researchers published two systematic mappings on the field.

Coelho et al. conducted a systematic mapping similar to ours but with a narrower scope. Specifically, while we analyzed papers about MCR, they focused on papers with solutions for MCR which consider the presence of refactorings in changesets. They found 13 papers matching their search criteria. Subsequently, they created a topic independent classification (question, results and validation) and a topic dependent classification for these papers [14].

Badampudi et al. reported the preliminary results of their systematic mapping on MCR [3]. They performed an electronic data source search and found 177 papers about MCR. Next, they analyzed those papers according to the topics they discuss, i.e. they created a topic dependent classification for these papers. Although their classification has topics which are similar to ours, we believe our classification is more extensive with more concepts. We also analyzed the papers according to multiple other dimensions such as venue and research method.

A.4.2 Systematic Mappings in Software Engineering

Petersen et. al presents a guide on how to perform a SM in Software Engineering and also compares and contrasts SM with systematic review (SR) by analyzing ten published systematic reviews [39]. In 2015, Petersen et al. notices that the aforementioned guidelines from 2008 could be improved and as a result, they perform a SM of existing SM studies in Software Engineering to determine how SM studies are being conducted in practice. With this information, they propose new and updated guidelines for performing a SM study [40].

Kitchenham et al. analyzed five studies which were based on preceding SM studies and found that research based on an earlier SM study had a number of advantages such as faster completion time and easier understanding of the literature [31]. Budgen et al. examined six mapping studies in Software Engineering and found that researchers often had difficulty with classifying the selected studies and with data extraction due to poor reporting in the studies [11].

Kitchenham et al. conducted a case study with six students and found that SM studies are useful for undergraduate and postgraduate students because they teach research skills and provide an overview of a research field [30].

Wohlin et al., MacDonell et al. and Kitchenham et al. compared SM studies which addressed the same topic and obtained mixed results on how similar the results of the SM studies are and how much overlap in selected studies they had [57][32][29].

A.4.3 Systematic Mappings in Software Quality

Barney et al. conducted a SM study on the topic of software quality trade-offs. As a major result, they found that only 28% of the selected studies provide empirical validation [4]. In sharp contrast, we found that almost 80% of the studies provide empirical validation with real world data in the field of MCR.

Tahir et al. did a SM study in the area of dynamic metrics of software quality and found that most studies at the time of the study focused on complexity and maintainability metrics. Also, 63% of the studies provided empirical validation to their claims [51].

Elberzhager et al. conducted a SM study on methods that combine static and dynamic quality assurance techniques. They found evidence of growing interest in the field and that nearly 50% of the studies provide empirical validation [16].

A.5 Conclusion

Given the popularity of MCR and the lack of secondary studies in the field in 2017, we conducted a systematic mapping study of MCR, where we identified 177 studies about MCR and classified them according to not only an existing classification scheme for research studies in general but also to a new classification scheme we derived from the papers themselves.

From our analysis and classification of the papers, we obtained interesting insights and also provided recommendations for future research in the area. We mainly discovered that: (1) the field of MCR is growing fast and steadily since 2011, (2) a wide variety of topics within MCR are being studied, (3) close to 80% of the studies provide empirical validation, (4) most papers study the process of MCR and such papers are usually the most cited and (5) there are comparatively few studies using a closed source software context.

However, our greatest contribution in this work is the map of the published peer-reviewed research on MCR. This map provides multiple benefits to researchers such as enabling them to easily find the published research relevant to their current work in MCR and visualizing the topics within MCR studied so far.

As potential future work, we believe that more secondary studies could be done in MCR. In particular, it would be interesting to conduct more in depth studies such as systematic reviews focusing on more specific topics within MCR. For example, systematic reviews about the process of MCR are promising given the high number of primary studies examining this topic.

Finally, the whole dataset containing the list of selected studies, their metadata and the classification can be found at: `https://sites.google.com/view/ch-design-discuss-thesis/`

Appendix B

Tree of Concepts and Categories

The following table shows the whole tree of concepts and categories identified during the characterization study of design discussions in MCR described in Chapter 4. The table also includes the absolute and relative number of occurrences of each concept and category.

Regarding the notation used, the categories to the left encompass the categories and concepts to the right. For example, the concept *to fix issue* belongs to the category *Goal of design change*, which belongs to the category *Design change*, which belongs to *Design concern*. Note that the top-most category *Design review*, which encompasses all other categories, was omitted due to print-size constraints.

Category				**Concept**	**N**	**%**
Design concern					1160	34.83%
	Design change				689	20.69%
				change description	1	0.03%
				change rationale	22	0.66%
				implemented change	40	1.20%
				proposed change	334	10.03%
		Goal of design change			77	2.31%
				to fix issue	42	1.26%
				to improve a non-functional requirement	31	0.93%
				to support new feature	4	0.12%
		Types of design change			215	6.46%
				change memory management	2	0.06%
				change solution design	19	0.57%
				change to a stateless design	1	0.03%
				change field mutability	8	0.24%
				extract library	1	0.03%
				remove code duplication	4	0.12%
				removed code with design issues	1	0.03%
			Change method contract		27	0.81%
				change method parameters	8	0.24%
				change method preconditions	7	0.21%
				change method return	7	0.21%
				change static method to non-static	3	0.09%
				mark method as non-overridable	2	0.06%
			Class-level change		20	0.60%
				change class API	1	0.03%
				change class to interface	1	0.03%
				change class to static	4	0.12%
				change dependency from impl to interface	2	0.06%
				change object construction interface	5	0.15%
				change object serialization	2	0.06%
				change relationship multiplicity	1	0.03%
				new relationship between classes	2	0.06%
				remove from public API	2	0.06%
			Concurrency		5	0.15%
				add concurrency mechanism	1	0.03%
				change concurrency mechanism	4	0.12%
			Create new element		18	0.54%
				create new class	6	0.18%
				create new field	1	0.03%
				create new method	10	0.30%
				create public interface	1	0.03%
			Design pattern		5	0.15%
				apply builder design pattern	1	0.03%
				apply dependency inversion pattern	1	0.03%
				apply observer pattern	1	0.03%
				apply singleton pattern	1	0.03%
				apply visitor design pattern	1	0.03%
			Exception		21	0.63%
				change exception handling	16	0.48%
				change thrown exception	5	0.15%
			Merge elements		3	0.09%
				merge classes	1	0.03%
				merge methods	2	0.06%
			Move element		20	0.60%
				move class to another module	2	0.06%

				move code to another class	7	0.21%
				move code to another function	6	0.18%
				move code to another module	1	0.03%
				move field to another class	2	0.06%
				move methods to another class	2	0.06%
			Remove unnecessary element		11	0.33%
				remove unnecessary code	7	0.21%
				remove unnecessary field	2	0.06%
				remove unnecessary method	2	0.06%
			Traditional refactorings		33	0.99%
				encapsulate	2	0.06%
				extract class	3	0.09%
				extract field	1	0.03%
				extract method	11	0.33%
				extract variable	1	0.03%
				inline function	1	0.03%
				pull up code	4	0.12%
				replace conditional with polymorphism	1	0.03%
				replace magic literal	6	0.18%
				replace primitive with object	3	0.09%
			Use another element		16	0.48%
				replace own impl with an external dependency	1	0.03%
				use another algorithm	3	0.09%
				use another data structure	9	0.27%
				use another library	3	0.09%
	Design issue				471	14.14%
				issue	306	9.19%
				issue rationale	19	0.57%
		Types of design issue			146	4.38%
				class should be more generic	1	0.03%
				concurrency design issue	15	0.45%
				confusing design	5	0.15%
				design doesnt fulfill the software requirements	7	0.21%
				misused design pattern	3	0.09%
				new design causes regression	1	0.03%
				not reusing existing code	1	0.03%
				poor performance of design	3	0.09%
			Design smell		3	0.09%
				flag argument	1	0.03%
				global variable	1	0.03%
				magic literal	1	0.03%
			Encapsulation		18	0.54%
				deficient encapsulation	16	0.48%
				excessive encapsulation	2	0.06%
			Exception issue		11	0.33%
				should not throw exception	1	0.03%
				wrong exception handling	10	0.30%
			Memory		2	0.06%
				memory leak	1	0.03%
				poor memory management	1	0.03%
			Method contract issue		11	0.33%
				inconsistent API method contracts	1	0.03%
				wrong method precondition	1	0.03%
				wrong method return	9	0.27%
			Missing element		14	0.42%
				incomplete interface (missing methods)	12	0.36%
				missing constructor	1	0.03%

				missing interface	1	0.03%
			Placement issue		9	0.27%
				poorly placed class	1	0.03%
				poorly placed field	2	0.06%
				wrongly placed method	6	0.18%
			Solution		9	0.27%
				poor solution design	3	0.09%
				wrong solution design	6	0.18%
			Unnecessary element		33	0.99%
				redundant inheritance	2	0.06%
				unnecessary class	5	0.15%
				unnecessary code	7	0.21%
				unnecessary field	4	0.12%
				unnecessary method	11	0.33%
				unnecessary method parameter	4	0.12%
Not design					901	27.06%
				assertion	4	0.12%
				code style	75	2.25%
				duplicated discussion comment	58	1.74%
				error message	19	0.57%
				feature specification (software requirements)	26	0.78%
				reference to a specific dev	2	0.06%
				reference to another comment	19	0.57%
				reference to infor outside this review request	46	1.38%
				software configuration files	8	0.24%
				testing	87	2.61%
	Documentation				147	4.41%
				code comment	97	2.91%
				documentation	50	1.50%
	Logging				18	0.54%
				add log point	6	0.18%
				log message	12	0.36%
	Implementation				237	7.12%
				change method implementation	4	0.12%
				code snippet	26	0.78%
				implementation bug	44	1.32%
				local variable initialization	10	0.30%
				local variable type	5	0.15%
				missing method calls in method impl	1	0.03%
				move statements within method	12	0.36%
				naming in code	82	2.46%
				optimize method implementation	4	0.12%
				replace code with calls to existing methods	2	0.06%
				simplify method implementation	35	1.05%
				temporary code	4	0.12%
				unnecessary line of code	7	0.21%
				use another programming language	1	0.03%
	Review workflow				69	2.07%
				ask author to keep working on the changeset	2	0.06%
				ask devs to review the patch	1	0.03%
				ask if review was abandoned	1	0.03%
				ask author to close finished issues	1	0.03%
				ask author to close review request	1	0.03%
				declaration of what the dev reviewed	11	0.33%
				dev couldnt understand the review comment	1	0.03%
				lgtm	24	0.72%
				move part of the changes to another review	6	0.18%

				summary of the review comments written	21	0.63%
	Social				42	1.26%
				praise the author's work	27	0.81%
				praise the review	1	0.03%
				social	14	0.42%
	Tools				44	1.32%
				build tool configuration	6	0.18%
				build tool log	2	0.06%
				review bot	19	0.57%
				version control	17	0.51%
Review actions					595	17.87%
	Improving design				422	12.67%
				ask for design change proposals	4	0.12%
				ask for feedback	9	0.27%
		Introduce design concern			409	12.28%
				introduce change	246	7.39%
				introduce issue	163	4.89%
	Postponing				18	0.54%
				ask for time before answering	7	0.21%
				postponed a design change	8	0.24%
				suggest postponing discussion	3	0.09%
	Understanding design				155	4.65%
				paraphrase belief to confirm understanding	2	0.06%
				understood clarification	6	0.18%
		Ask for clarification on design			76	2.28%
				ask for clarification on current design	7	0.21%
				ask for clarification on design change	63	1.89%
				ask for clarification on design issue	6	0.18%
		Provide clarification on design			71	2.13%
				clarification on change description	33	0.99%
				clarification on change rationale	24	0.72%
				clarification on current design	8	0.24%
				clarification on issue description	6	0.18%
Design discussion					674	20.24%
	Discussion result				157	4.71%
				accepted	79	2.37%
		Discarded			78	2.34%
				postponed discussion	17	0.51%
				rejected	53	1.59%
			Design concern was invalidated		8	0.24%
				another change invalidated this change	6	0.18%
				another change invalidated this issue	2	0.06%
	Belief				517	15.53%
				belief rationale	20	0.60%
		Types of belief			497	14.92%
				neutral belief	7	0.21%
				strongly agree belief	171	5.14%
				strongly disagree belief	37	1.11%
				weakly agree belief	247	7.42%
				weakly disagree belief	35	1.05%
					3330	100.00%