



UNIVERSIDADE
FEDERAL DE
ALAGOAS



UNIVERSIDADE FEDERAL DE ALAGOAS
INSTITUTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Dissertação de Mestrado

**Developers' Assumptions on Identifying Bug-introducing
Changes**

Jairo Raphael Moreira Correia de Souza

`jrmcs@ic.ufal.br`

Orientador:

Baldoino Fonseca dos Santos Neto

MACEIÓ, OUTUBRO DE 2019

Jairo Raphael Moreira Correia de Souza

Developers' Assumptions on Identifying Bug-introducing Changes

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Curso de Mestrado em Informática do Instituto de Computação da Universidade Federal de Alagoas.

Orientador:

Baldoino Fonseca dos Santos Neto

Maceió, Outubro de 2019

Catálogo na fonte
Universidade Federal de Alagoas
Biblioteca Central

Bibliotecária: Taciana Sousa dos Santos – CRB-4 – 2062

S729d Souza, Jairo Raphael Moreira Correia de.
Developer's assumptions on identifying bug-introducing changes / Jairo
Raphael Moreira Correia de Souza. – 2019.
72 f. : il. color.

Orientador: Balduino Fonseca dos Santos Neto.
Dissertação (mestrado em Informática) - Universidade Federal de Alagoas.
Instituto de Computação. Maceió, 2019.

Bibliografia: f. 55-65.

Apêndices: f. 66-72.

1. Computação. 2. Software - Desenvolvimento. 3. Software -
Confiabilidade. I. Título.

CDU: 004.05



UNIVERSIDADE FEDERAL DE ALAGOAS/UFAL
Programa de Pós-Graduação em Informática – Ppgi
Instituto de Computação

Campus A. C. Simões BR 104-Norte Km 14 BL 12 Tabuleiro do Martins
Maceió/AL - Brasil CEP: 57.072-970 | Telefone: (082) 3214-1401




Folha de Aprovação

Jairo Raphael Moreira Correia de Souza

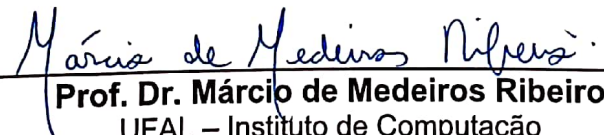
“Developer’s Assumptions on Identifying Bug-introducing Changes”

Dissertação submetida ao corpo docente do Programa de Pós-Graduação em Informática da Universidade Federal de Alagoas e aprovada em 04 de outubro de 2019.


Banca Examinadora:



Prof. Dr. Balduino Fonseca dos Santos Neto
UFAL – Instituto de Computação
Orientador



Prof. Dr. Márcio de Medeiros Ribeiro
UFAL – Instituto de Computação
Examinador Interno



Prof. Dr. Gustavo Henrique Lima Pinto
UFPA – Instituto de Ciências Exatas e Naturais

Agradecimentos

Durante minha jornada no mestrado eu tive a oportunidade de aprender muito mais do que eu imaginava sobre Ciências da Computação. Portanto, primeiramente eu agradeço Deus pela vida, saúde, e oportunidades de aprendizado durante toda a caminhada do do mestrado.

A minha família por todo o apoio e incentivo ao longo dessa caminhada. Também por entender que a educação é a base fundamental do ser humano.

Agradeço ao meu orientador Prof. Dr. Balduino Fonseca dos Santos Neto por toda a paciência, apoio e críticas construtivas durante esse período. Seus ensinamentos foram fundamentais para a minha formação como pesquisador.

Aos membros da banca, os professores Dr. Márcio de Medeiros Ribeiro e ao Dr. Gustavo Henrique de Lima Pinto pela disposição e interesse em contribuir para a melhoria desse estudo.

Ao grupo de pesquisa EASY e seus integrantes que se tornaram colegas de trabalho e pessoas que pretendo ter amizade pelo resto da vida. Sem dúvida, fizeram parte da minha formação. Obrigado pelas discussões, cafezinhos e convivência diária no laboratório.

Agradeço também à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo apoio financeiro para realização da pesquisa. Por último, a todos aqueles que direto ou indiretamente fizeram parte da minha formação: O meu muito obrigado!

Resumo

Durante a revisão de código, desenvolvedores podem realizar premissas que podem guiar o seu processo de revisão de código (por exemplo, os desenvolvedores podem considerar que o código com baixa cobertura de teste tem mais probabilidade de apresentar erros. Assim, eles podem concentrar sua atenção nesse trecho de código). Embora os estudos anteriores tenham investigado as premissas dos desenvolvedores envolvendo diferentes preocupações, nenhum deles concentram-se em analisar as premissas dos desenvolvedores durante a revisão de código para identificar alterações que introduzem bugs. Portanto, nosso estudo tem como objetivo investigar essas premissas, quão relevantes elas são para os desenvolvedores, por que os desenvolvedores colocam mais ou menos relevância para algumas premissas e quais premissas são consenso ou dissenso entre os eles. Para isso, utilizamos a metodologia Q para conduzir nosso estudo contendo 41 desenvolvedores analisando 41 premissas extraídas de diferentes fontes. Os resultados indicam que: (i) as premissas dos desenvolvedores estão relacionadas às características das mudanças, linguagem de programação, experiência e hábitos dos desenvolvedores, práticas organizacionais, manutenção do código, propriedade e testes; (ii) existem cinco pontos de vista comuns entre os desenvolvedores sobre a relevância das premissas; (iii) complexidade e propriedade de código (*code ownership*), tempo de entrega curto, práticas de desenvolvimento e familiaridade com o código são as principais razões para os desenvolvedores colocarem mais e/ou menos relevância em certas premissas; e (iv) premissas envolvendo alto número de funcionalidades (*features*), linguagens de programação e aceitação de solicitações de mudanças (*pull-requests*) apresentam mais consenso entre os desenvolvedores, enquanto que premissas envolvendo commits de *merge*, histórico de software, familiaridade com a linguagem de programação, reutilização de código e propriedade de código apresentam mais dissensos. Essas descobertas são um conhecimento valioso para profissionais e pesquisadores no sentido do aprimoramento de ferramentas e/ou técnicas de revisão de código para identificar mudanças que introduzem bugs.

Palavras-chave: Mudanças que Introduzem Bugs, Revisão de Código, Premissas dos Desenvolvedores, Metodologia Q

Abstract

During the code review, developers can make assumptions that may guide their review process (e. g., developers may consider that code with low test coverage is more likely to introduce bugs. Thus, they can focus their attention on this code). Although studies have investigated developers' assumptions involving different concerns in software engineering, none of those studies focus on analyzing developers' assumptions when reviewing code to identify bug-introducing changes. Our study investigates those assumptions, how relevant they are for developers, why developers put more/less relevance for some assumptions, and which assumptions are consensus/dissensus among groups of developers. We apply the Q-methodology to conduct our study with 41 developers analyzing 41 assumptions extracted from different sources. The results indicate: (i) the developers' assumptions involve concerns related to the programming language, developers' experience and habits, organizational practices as well as code maintenance, ownership and testing; (ii) five common viewpoints among developers regarding the relevance of assumptions; (iii) code complexity and ownership, short delivery time, developers practices and familiarity with the code are the main reasons for developers to put more/less relevance to some assumptions; and (iv) while assumptions involving programming languages, features invocation and pull-request acceptance present more consensus among developers, assumptions involving merge commits, software history, familiarity with the programming language, code reuse and ownership present more dissensus. These findings are valuable knowledge for both practitioners and researchers towards improving state-of-the-art code review tools/techniques to identifying bug-introducing changes.

Keywords: Bug-introducing Changes, Code Review, Developers' Assumptions, Q-Methodology

List of Figures

Figure 1 – Example of a bug-introducing change	5
Figure 2 – Q-Methodology Steps (ZABALA; PASCUAL; XIA, 2016).	6
Figure 3 – How is the rate expertise of participants in different perspectives	13
Figure 4 – Q-Sort structure. The number under parentheses represents the number of assumptions per column.	13
Figure 5 – Steps of our Study	15
Figure 6 – Second step: Classifying assumptions in three levels of agreement. The white card on the top represents the assumption that the participant should drag and drop to one of the colored box.	15
Figure 7 – Third step: Ranking the assumptions in seven agreement levels according to the Q-Sort structure.	16
Figure 8 – Fourth step: Explaining the extreme rated assumptions.	17
Figure 9 – Viewpoint analysis steps and research decisions	18
Figure 10 – <i>Z-scores</i> of each assumption among the viewpoints. They are ordered by the standard deviation of their <i>z-scores</i> for all viewpoints	24

List of Tables

Table 1 – Initial Set of non-rotated Viewpoints	20
Table 2 – Rotated Viewpoints and Flagged Participants	23
Table 3 – Assumptions and viewpoint scores. The highlighted cells in green and red indicate the extreme scores of each factor, whereas an “***” and “*” is used to denote the distinguishing assumptions of each viewpoint indicating significance at $P < .01$ and $P < .05$ respectively. Consensus assumptions are in bold , while dissensions assumptions are in <i>italic</i>	24
Table 4 – Assumptions and Ranks in Viewpoint A	30
Table 5 – Assumptions and Ranks in Viewpoint B	32
Table 6 – Assumptions and Ranks in Viewpoint C	34
Table 7 – Assumptions and Ranks in Viewpoint D	35
Table 8 – Assumptions and Ranks in Viewpoint E	37
Table 9 – Consensus and Dissensus Assumptions	45
Table 10 – List of Assumptions	66
Table 11 – Part of the Initial Data Matrix	67
Table 12 – Correlations between Q-sorts	68

Contents

List of Figures	iv
List of Tables	v
Contents	vi
1 Introduction	1
1.1 Context and Problem	1
1.2 Objectives and Methodological Aspects	2
1.3 Contributions	3
1.4 Thesis Structure	4
2 Background	5
2.1 Bug-introducing Changes	5
2.2 Q-Methodology	6
2.2.1 Defining the Concourse.	6
2.2.2 Defining the Q-Set.	7
2.2.3 Defining the P-Set.	7
2.2.4 Defining the Q-Sort structure.	7
2.2.5 Conducting the Q-Sorting process.	7
2.2.6 Conducting the Viewpoint Analysis.	8
2.2.7 Conducting the Viewpoint Interpretation.	8
2.2.8 Why Q-Methodology?	8
3 Study Design	9
3.1 Research Questions	9
3.2 Gathering Assumptions and Defining the Q-Set	11
3.3 Defining the P-Set	12
3.4 Defining the Q-Sort Structure	12
3.5 Conducting the Q-Sorting process	14
3.6 Conducting the Viewpoint Analysis	17
4 Results and Discussions	28

4.1	RQ1. Which assumptions are behind the developers' beliefs when looking for bug-introducing changes?	28
4.2	RQ2. How relevant are the assumptions for developers?	29
4.2.1	Viewpoint A: No matter the software history, only tests and developers' experience	30
4.2.2	Viewpoint B: Do not Despise Its History Regardless of Your Experience and Ownership	32
4.2.3	Viewpoint C: Have your own testing team	33
4.2.4	Viewpoint D: Be Aware of All Artifacts Impacted by Changes, mainly in Geographically Distributed Teams . .	35
4.2.5	Viewpoint E: Be familiar with the code	37
4.3	RQ3. Why developers (dis)agree with an assumption?	38
4.4	RQ4. Which assumptions are consensus/dissensus among developers's viewpoints?	44
4.4.1	Consensus Assumptions	45
4.4.2	Dissensus Assumptions	46
5	Implications	48
6	Related Work	50
7	Threats to Validity	52
8	Conclusions and Future Works	54
	Bibliography	56
	APPENDIX A Supplementary Material	64
1	Complete List of Assumptions	64
2	Q-Sort Matrix	67
3	Correlation Matrix	68
	APPENDIX B Characterization Questionnaire	68

1 Introduction

In this chapter, we present a summary of the research, starting with the context and problem, connecting them with the objectives and contributions of this work.

1.1 Context and Problem

For many software projects, code review plays a central role in reducing software bugs and improving their overall quality (MCINTOSH et al., 2016; Thongtanunam et al., 2015; An et al., 2018; MANTYLA; LASSENIUS, 2009). During a code review, reviewers must carefully inspect committed changes (patches) aiming to identify as many issues as possible, such as bugs, code improvement, alternative solutions, and more (BACCHELLI; BIRD, 2013; PASCARELLA, 2018). However, even when reviewers perform a careful inspection, they can make assumptions that may lead them to allow (or avoid) changes able to introduce bugs, also known as bug-introducing changes (ŚLIWERSKI; ZIMMERMANN; ZELLER, 2005).

For example, while a developer may consider that code with high test coverage is less likely to introduce bugs (BORLE et al., 2018), other may believe that changes involving a large number of files are more likely to introduce bugs (MACLEOD et al., 2018). Both assumptions may lead reviewers to dedicate more/less attention to review a change and, consequently, directing them to allow/avoid bug-introducing changes. Studies have investigated developers' assumptions, beliefs and perceptions regarding different concerns in software engineering. For instance, (MEYER et al., 2014; MEYER; ZIMMERMANN; FRITZ, 2017) investigate the developers' perceptions about productive work. In (SMITH; BIRD; ZIMMERMANN, 2016), the authors analyze the relation between work practices, beliefs and personality traits. (DEVANBU; ZIMMERMANN; BIRD, 2016) analyze the developers' beliefs in empirical software engineering. More recently (MATTHIES et al., 2019) investigate the developers' perceptions about agile development practices and their usage in Scrum teams.

All of these studies provide contributions towards our knowledge on comprehension of the developer's assumptions and together form an overall understanding of different perspectives. However, none of these studies investigate the developers' assumptions when review

code to identify bug-introducing changes. Furthermore, several empirical studies investigate diverse aspects during the code review to improve software quality (KONONENKO; BAYSAL; GODFREY, 2016; MCINTOSH et al., 2016), but few studies provide a better understanding of developers' assumptions during the code review process. Hence, improving the knowledge of developers' assumptions is necessary to gain a better understanding of this context to be able to improve upon it.

1.2 Objectives and Methodological Aspects

As explained above, understanding the main developer's assumptions on identifying bug-introducing changes is essential to comprehend this subjective context of developers' assumptions and beliefs, in an effort to provide input to researchers and practitioners (reviewers) to improve code review tools, techniques, and provide knowledge to reviewers perform this often complex and mentally demanding task. Based on that, we set the following objectives.

First, this study investigates which assumptions developers make when they review code to identify bug-introducing changes. For instance, our objective is to provide a range of developers' assumptions from different sources. The study also how relevant is each assumption for the developers by analyzing how strong developers agree/disagree with some assumptions. Such analysis may help us to extract common viewpoints among developers about the relevance of the assumptions. Then, we analyze why developers agree/disagree with some assumptions. The objective is to understand the main motivations behind that. Finally, we examine which assumptions are consensus/dissensus among the developers' viewpoints. Consensus assumptions may provide techniques to help similar developers, while dissensus assumptions may provide interesting discussions to researchers and reviewers.

To perform the objectives above mentioned, we involve 41 developers from industry and academia. They analyze 41 assumptions extracted from different sources, for instance, academic papers, in-person interviews with developers, and specialized forums. To evaluate the relevance of the assumptions and extract common viewpoints, we apply the Q-Methodology (BROWN, 1980) that enables us to investigate assumptions in a systematic way from a reduced number of participants. Along with the execution of the study, we also collect comments provided by developers to understand why they put more (or less) rele-

vance to some assumptions.

1.3 Contributions

The main contributions of this study are:

- The developers' assumptions are related to the programming language (for instance, the majority of the developers believe that the introduction of bugs depends on which programming language is used), changes characteristics (for instance, most of the developers believe that changes invoking a high number of features - methods, fields and classes- are more likely to introduce bugs), developer's experience and habits, organizational practices (e. g., pair programming) as well as the code maintenance, ownership and testing;
- We identify five main developers' viewpoints : (**Viewpoint A**) developers loaded into this viewpoint believe that high test coverage and developer's experience play an important role in the reduction of bug-introducing changes. Regardless of the developer's understanding about the software history; (**Viewpoint B**) it is mandatory to understand the software history, focusing on changes that neglect non-functional requirements or involve a large number of files. Regardless of the developer's experience; (**Viewpoint C**) having a testing team is mandatory to avoid bug-introducing changes. Regardless if the developer is familiar with the programming language adopted in a project; (**Viewpoint D**) to avoid bugs, be aware of all artifacts impacted by a change when reviewing it, mainly when we have geographically distributed teams; and (**Viewpoint E**) familiarity with the source code is mandatory to avoid bug-introducing changes regardless of the programming language;
- Most of the developers' reasons to (dis)agree with the assumptions are based on his own experience, code complexity and ownership, short time delivery, development practices, and familiarity with the source code.
- Assumptions involving a number of features invocations, programming languages, and pull-request acceptance produce more consensus among developers than assump-

tions involving merge commits, software history, familiarity with the programming language, code reuse, and ownership;

From this study, we provide valuable information for both researchers and practitioners (code reviewers). Researchers can focus their efforts on specific assumptions aiming to build code review tools/techniques tailored for specific code reviewers. We argue that — if these tools/techniques would be available at hand — code reviewers could focus their efforts on correctly inspecting the code under review, rather than spending cognitive effort and time on several assumptions, speeding up the process of code review to identify bug-introducing changes.

1.4 Thesis Structure

The rest of this thesis is structured as follows:

- Chapter 2 describes some background useful for the understanding of this work, including a description of the bug-introducing changes and Q-Methodology.
- Chapter 3 describes the design of our mixed study going over the research questions, the design of our online experiment, and how the data will be analyzed.
- Chapter 4 presents the results and discussions of this study. We start providing a list of developers' assumptions collected from different sources. Then, we extract the developers' viewpoints among these assumptions. Further, we give details of why developers agree/disagree with some assumptions. Finally, we provide which assumptions are consensus/dissensus among the developers' viewpoints.
- Chapter 5 presents the implications of this study.
- Chapter 6 discuss the related works that have been done in the fields of the developers' beliefs, perceptions, personality, and assumptions in Software Engineer. This chapter focuses more on comparing the existing works, contributions, and limitations.
- Chapter 7 details the limitations and threats to validity of this study.
- Chapter 8 presents our conclusion and ideas which could lead to future contributions in the field of Software Engineering.

2 Background

In this chapter we contextualize our work, giving an overview of the bug-introducing changes and presenting the main concepts of the methodology used in this study (Q-Methodology).

2.1 Bug-introducing Changes

During the process of software development and evolution, inevitably, developers perform changes, and these changes may introduce bugs, also known as bug-introducing changes (ŚLIWERSKI; ZIMMERMANN; ZELLER, 2005). Figure 1 demonstrate an example of a bug-introducing change extracted from the Elasticsearch-hadoop project ¹. The left side of the Figure 1 shows the code snippet containing the introduction of the bug reported., while the right side presents the bug fix of an opened bug report related to *null pointer exception* error occurred in the software.

<pre>122 for ((k, v) <- value) { 123 if (shouldKeep(generator.getParentPath(), k.toString())) { 124 generator.writeFieldName(k.toString) 125 - if (value != null) { 126 127 val result = write(schema.valueType, v, generator) 128 if (!result.isSuccessful()) { 129 return handleUnknown(value, generator) 130 } 131 } 132 }</pre> <p>a) Bug-introducing change</p>	<pre>122 for ((k, v) <- value) { 123 if (shouldKeep(generator.getParentPath(), k.toString())) { 124 generator.writeFieldName(k.toString) 125 + if (v == null) { 126 + generator.writeNull() 127 + } else { 128 val result = write(schema.valueType, v, generator) 129 if (!result.isSuccessful()) { 130 return handleUnknown(value, generator) 131 } 132 } 133 }</pre> <p>b) Change containing the fix</p>
---	---

Figure 1 – Example of a bug-introducing change

Bug-introducing changes help developers and reviewers to identify important properties of software bugs such as correlated factors or causalities. For example, we can find detect rules or patterns that are bug-prone in source code (KIM et al., 2006). However, it is important to avoid the introduction of bugs during a change, and code review has the potential to support this process (MCINTOSH et al., 2016).

¹ <<https://github.com/elastic/elasticsearch-hadoop>>

2.2 Q-Methodology

The Q-Methodology is a way to systematically investigate subjectivity, such as opinions, beliefs, behaviors, and attitudes. It combines the strength of qualitative and quantitative research (mixed study) through a methodological bridge between them (STEPHENSON, 1935). The qualitative nature of this methodology emerges from human subjectivity, while the quantitative portion comes from the sophisticated statistical procedures for data analysis (e.g., correlation and viewpoint-analysis) (YANG, 2016). Figure 2 shows the steps necessary for applying the Q-Methodology (ZABALA; PASCUAL; XIA, 2016), which will be presented in the following subsections.

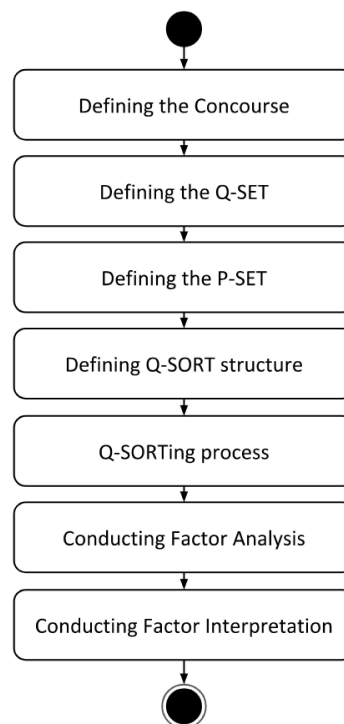


Figure 2 – Q-Methodology Steps (ZABALA; PASCUAL; XIA, 2016).

2.2.1 Defining the Concourse.

The concourse is a set of opinions, assumptions, or beliefs referring to a particular topic. In the context of this research, the concourse is composed of assumptions regarding what is important to inspect when developers review code to identify bug-introducing changes. The assumptions composing the concourse can emerge from different sources, such as academic papers, social media, and in-person interviews. Also, the assumptions should ideally cover

possible points of view about the investigated topic (ZABALA, 2014). To illustrate that, the following assumption “code with high test coverage is less likely to introduce bugs” is a possible element to compose the concourse of this study.

2.2.2 Defining the Q-Set.

The Q-Set is the concourse itself after subject to a refinement in order to express a consistent and coherent set of assumptions. The refinement process can be conducted by applying strategies to remove duplicated or similar assumptions, fixing gaps, among others (PAIGE; MORIN, 2014). The Q-Set has typically between 40 and 80 statements (in our case, assumptions) (BROWN, 1980).

2.2.3 Defining the P-Set.

The P-set consists of a set of participants who will attend the Q-Methodology and should represent a population diversity in the theme of research. The sample of respondents does not need to be large or representative of the population, but must be diverse in a way that is possible to obtain the most diverse range of assumptions about the research topic (ZABALA; PASCUAL; XIA, 2016).

2.2.4 Defining the Q-Sort structure.

The Q-Sort is a grid that represents the distribution of assumptions (Q-Set) answered by a participant. This grid usually follow a quasi-normal distribution as in Figure 4, assuming that fewer assumptions generate strong engagement (BROWN, 1980). With the Q-sort structure defined, the participants (P-Set) are able to make the Q-Sorting task that consists of ranking the assumptions randomly presented according to their agreement or disagreement following the Q-sort structure.

2.2.5 Conducting the Q-Sorting process.

The Q-Sorting process is the phase where the participants of the study (P-Set) rank the assumptions regarding their level of agreement following the Q-Sort structure, according to their perceptions.

2.2.6 Conducting the Viewpoint Analysis.

This step involves a series of statistical procedures aiming to analyze the correlations between variables to reveal the smallest number of viewpoints which can explain the correlations (BROWN, 1980). The participants of the study become the variables of interest since the methodology works by grouping the participants' assumptions (Q-sorts) (WATTS; STENNER, 2012). In our study, a viewpoint represents one common viewpoint among developers about the topic under investigation (WATTS; STENNER, 2012). Hence, two developers that are loaded into the same viewpoint will have a very similar assumption about the topic under investigation.

2.2.7 Conducting the Viewpoint Interpretation.

This last step consists in interpreting the viewpoints resulting from the previous analysis aiming to provide a rationale that explains each identified viewpoint. It involves the production of a series of summarizing reports, each of which explains the viewpoint, assumptions and their ranks (WATTS; STENNER, 2012). By interpreting the viewpoints, it is also possible to find interesting information regarding consensuses and dissensions among viewpoints.

2.2.8 Why Q-Methodology?

Q-Methodology is commonly misinterpreted as a survey, but they are very different methods. Probably, the key distinction is that Q-Methodology aims at discovering what are the diverse viewpoints about a topic under study, in opposition to a survey that aims at having a sample allegedly representing the proportion of those viewpoints in a given population (CARTAXO et al., 2019). As a consequence, Q-Methodology studies neither need large nor random samples. Actually, studies are showing that 40-60 participants are in an effective range (WATTS; STENNER, 2012), distinguishing it from surveys studies that generally requires a larger sample of participants for statistical significance. However, the differences do not finish there, surveys can lead to biases in responses, e.g., some participants might be more favorable than others. On the other side, the Q-sort structure may follow a quasi-normal distribution. So, the participants are required to prioritize their perceptions instead of being mostly positive, neutral, or negative.

3 Study Design

In this chapter, we describe the process that leads to the results of our study. First, we present the research question of our work, and we present the steps of the experiment using the Q-Methodology. Lastly, we perform the data analysis procedure

3.1 Research Questions

During the code review process, developers adopt various assumptions regarding factors that may lead or not to the introduction of bugs. For example, a developer may agree it is mandatory to understand the rationale behind changes to avoid bugs, while another may disagree that large changes (involving a large number of files) are more likely to introduce bugs. As the software evolves, those assumptions tend to become even more diverse, increasing the cognitive load that developers have to deal with when looking for bug-introducing changes. Our study aims to provide qualitative evidence concerning those assumptions. For this purpose, we try to answer four research questions:

The main goal of this research is to provide qualitative evidence concerning those assumptions. For this purpose, we try to answer four research questions:

RQ₁. *Which assumptions are behind the developers' beliefs when looking for bug-introducing changes?*

This research question investigates developers' assumptions when looking for bug-introducing changes. Those assumptions are based on diverse concerns related to software development and maintenance. Particularly, such analysis becomes difficult because it requires to gather concerns from different sources, such as academic papers, informal talks with developers as well as discussions in forums. Our results may shed light on the construction of code review techniques and tools that aims at supporting developers in the identification of bug-introducing changes.

RQ₂. *How relevant are the assumptions for developers?*

After gathering the developers' assumptions, we investigate how strong developers agree or disagree with each assumption. In this case, we could simply analyze the relevance of each assumption from the viewpoint of each developer. However, the results obtained from this cross-check may not be useful since we would obtain a particular developer's viewpoint

instead of a group of developers. Hence, we also investigate the developers' viewpoint on the relevance of a set of assumptions. The recognition of developers' viewpoints may help us to create code review techniques and tools tailored to specific groups of developers, according to their views on assumptions that may lead (or not) to the introduction of bugs.

RQ₃. *Why developers (dis)agree with an assumption?*

Although gathering the assumptions and recognizing the developers' viewpoint may provide evidence that can help us to understand better how developers identify bug-introducing changes, we still do not know why developers believe (or not) in some assumptions. Hence, we ask the developers to comment about why they agree or disagree with some assumptions. From those comments, we investigate the main reasons for developers to believe (or not) in an assumption. The developers' comments may provide practical insights to help developers in the identification of bug-introducing changes.

RQ₄. *Which assumptions are consensus/dissensus among developers' viewpoints?*

Once we recognize the developers' viewpoints, we also analyze the assumptions that are consensus/dissensus between those viewpoints. By identifying the consensus assumptions, researchers may create techniques to help groups of developers sharing the same viewpoint. On the other hand, dissensus assumptions may encourage discussions about conflicts existing among developers.

To perform our study, we follow the Q-Methodology that aims at systematically investigating subjectivity such as opinions, beliefs, behaviors, and attitudes. It combines the strength of qualitative and quantitative research (mixed study) through a methodological bridge between them (STEPHENSON, 1935). The qualitative nature of this methodology emerges from human subjectivity, while the quantitative portion comes from the sophisticated statistical procedures for data analysis (YANG, 2016). The Q-Methodology, together with statistical techniques (e.g., correlation and viewpoint analysis), allows researchers to examine the subjective components of human behavior by employing a systematic and rigorous quantitative procedure (MCKEOWN, 2013). In the next sections, we describe in details the steps performed in our study.

3.2 Gathering Assumptions and Defining the Q-Set

During a code review, developers have different assumptions regarding how to avoid the introduction of bugs. For example, some developers believe that “before accepting a pull-request, it is essential to compile and run the system with the changes to avoid the introduction of bugs”, as discussed in software development forums (RAM, 2018). Other developers believe that “Understanding the rationale behind changes is mandatory to avoid the bugs introduction”, as described in previous studies (TAO et al., 2012).

To gather those assumptions, we extract information from diverse sources: i) we analyze the main assumptions contained in the findings described by 167 academic papers involving diverse aspects on bug-introducing changes; ii) we made informal interviews with developers from academia and industry to find out the main assumptions behind their beliefs regarding bug-introducing changes; and iii) we open discussions threads in programming forums such as *Reddit*², and *Hacker News*³, focusing on discussions involving techniques and tools used by developers to identify bug-introducing changes. As a result, we gathered 90 assumptions from these sources involving different concerns, such as code comprehension sources, developers’ experience and habits, programming language, and maintenance. The complete list of the collected assumptions is available in the Table 10 given in Appendix A.

Although we have obtained a large number of assumptions, many of them present ambiguity and similarity, to mitigate such issues, we selected a subset of assumptions, named *Q-set*, from the original ones as follows. First, three authors analyze the similarity among assumptions to identify the distinguished ones. Next, we involve three experienced researchers to analyze the ambiguity of these assumptions. For each assumption, the researchers answered two questions adapted from a previous study (PAIGE; MORIN, 2014): 1) Is the assumption not ambiguous from the viewpoint of a developer? and 2) Do you have any suggestions to improve this assumption?

In the questions mentioned above of this semi-structured approach, the professionals refined the Q-Set by eliminating ambiguous assumptions. In this case, we only remove the assumptions whose ambiguity is a consensus among the professionals. In the last question, the professionals could suggest writing improvements to the assumption under analysis. After

² <<https://www.reddit.com/r/programming/>>

³ <<https://news.ycombinator.com/>>

these adjustments, we ended up with the 41 assumptions, which meets the recommendations that suggest a Q-Set containing between 40 and 80 items about the topic under investigation (WATTS; STENNER, 2012; BROWN, 1993). In addition, we analyzed and grouped each one of the 41 assumptions on different factors that may influence the introduction of bugs. The final list of assumptions is presented in Table 3.

3.3 Defining the P-Set

In our study, we explore the viewpoints of developers and researchers on assumptions that may influence the introduction of bugs. To widen the diversity of viewpoints in terms of quantity and quality, we invite 72 developers from industry and academia who have had experience in working with bugs, such as, bug reporting and fixing, software development, tests, and code review. We obtain answers from 41 participants.

After completing the Q-Methodology, the participants answer a characterization questionnaire. It contains questions regarding his educational level, previous job, as well as his experience in issues related to bugs, for instance, bug fixing and reporting, code review, software development, and test. Figure 3 presents the percentage of the developers with experience varying from *very poor* to *very high*. We observe that more than 70% of the participants have *high* or *very high* experience in software development and 51% have *high* or *very high* experience in bug fixing. Note also that at least 45% of the participants have *fair* experience in code review, tests and bug reporting.

Although the number of participants involved in our study can be considered small for some conventional analysis, small samples are adequate for Q-Methodology that aims to understand the different viewpoints of reduced groups of participants. Indeed, Q-Methodology is most effective when applied in groups smaller or close to the number of assumptions in the study (WATTS; STENNER, 2012).

3.4 Defining the Q-Sort Structure

To extract the participants' viewpoints, we adopt a Q-Sort structure used in previous studies that use Q-Methodology (KELLY; MOHER; CLIFFORD, 2016; AKHTAR-DANESH; BAUMANN; CORDINGLEY, 2008). The Q-Sort is a grid that represents the distribution of

difference in the relation of distribution effects (BROWN, 1980). The number in parenthesis at the top of each column indicates the number of assumptions that should be put on a column. For instance, the participants have to rank two assumptions as +3, the ones they agree the most.

3.5 Conducting the Q-Sorting process

During the Q-Sorting process, all the participants (P-Set) rank the assumptions according to their agreement or disagreement. Participants completed the online Q-Sorting process using a web tool⁴. Initially, we conducted a pilot study involving ten participants. During the pilot, we encouraged participants to provide feedback regarding the understanding of the tasks to be performed in the study; usability of the tool, including bugs; and the time spent to complete the entire process of Q-Sorting. The participants took about 20 minutes to finish the tasks. None of the participants reported relevant issues, so we decided to consider the answers of these participants in the results of our study. Our Q-sorting process is divided into a 3-phase flow containing five steps, as shown in Figure 5.

Welcome Page: In *phase 1*, we present a welcome page to the participant containing a brief description about the context of the study. At this point, we describe the motivation of our study, and we explain some concepts, i.e., bug-introducing changes. Finally, we provide instructions about estimated time, support contact and a list of supported browsers by our tool;

Q-sorting: Next, in *phase 2*, the participant will do the ranking (Q-sorting) of the assumptions according to their agreement or disagreement. Initially, we ask the participant to drag and drop 41 cards, each containing one assumption of the Q-Set, into three piles — Agree, Disagree and Neutral — as depicted in Figure 6. We present the cards in random order to avoid bias.

After defining the piles, we ask the participants to rate the assumptions in seven levels of agreement ranging from -3 (most unlikely) to +3 (most likely), as shown in Figure 7. Each assumption must be associated with only one level. However, the participants can rearrange the assumptions how many times he wishes. Once the participant is satisfied with the distribution, we associate the Q-sort to the participant. What makes the Q-Methodology unique

⁴ <<https://bic.netlify.com>>

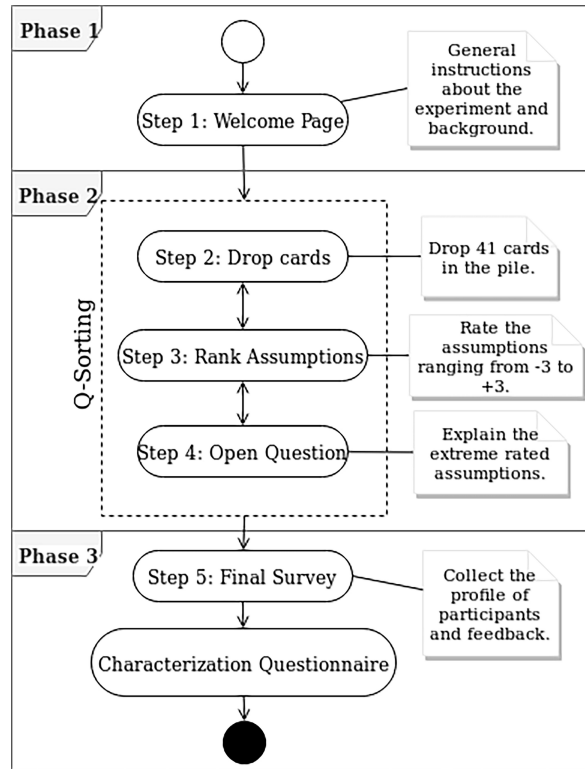


Figure 5 – Steps of our Study

Having a testing team is mandatory to avoid bug-introducing changes.

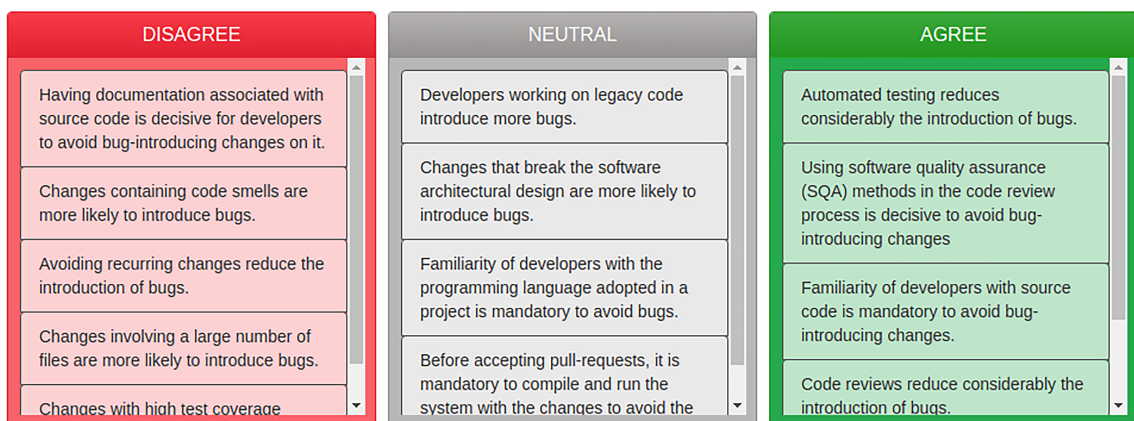


Figure 6 – Second step: Classifying assumptions in three levels of agreement. The white card on the top represents the assumption that the participant should drag and drop to one of the colored box.

is that it forces participants to rank-order each assumption into this grid-based upon their agreement or disagreement. In other words, the sorting procedure forces each participant to examine their viewpoint in a systematic way (BROWN, 1980).

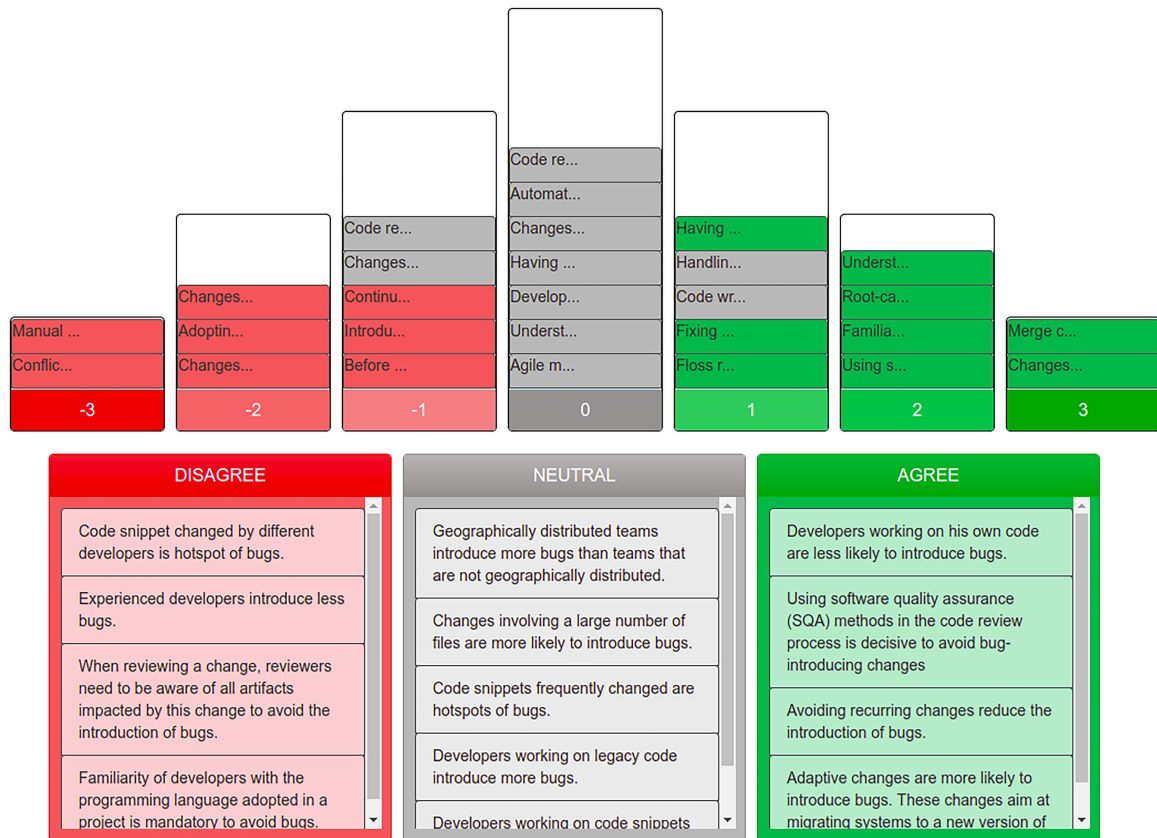


Figure 7 – Third step: Ranking the assumptions in seven agreement levels according to the Q-Sort structure.

Once the participant defines his Q-sort, we ask him to explain why he ranked the assumptions on the extremes (i.e., the two cards took the -3 and piles at the previous step) as shown in Figure 8. Although the participant is only obligated to explain the extreme assumptions, he also can explain the ranking of the other assumptions. The goal is to utilize this qualitative evidence in further analysis during the last step in Q-Methodology (*Viewpoint Interpretation*).

Final Page - After completing the assumptions raking, the participant answers a characterization questionnaire in *phase 3*. It contains questions regarding his educational level, previous job, as well as his experience in issues related to bugs, i.e., bug fixing and reporting, code review, software development, and test. Participants response is only counted in

AGREE +3	
Changes that break the software architectural design are more likely to introduce bugs.	<input style="width: 100%; height: 30px;" type="text"/>
Merge commits introduce more bugs than other commits.	<input style="width: 100%; height: 30px;" type="text"/>
DISAGREE -3	
Conflicting changes introduce more bugs than non-conflicting ones.	<input style="width: 100%; height: 30px;" type="text"/>
Manual tests verify which changes meet user requirements. Therefore, manual testing is mandatory to avoid bugs.	<input style="width: 100%; height: 30px;" type="text"/>

Figure 8 – Fourth step: Explaining the extreme rated assumptions.

the final P-Set if the participant passed through all six steps. The complete characterization questionnaire can be found in the Appendix B.

3.6 Conducting the Viewpoint Analysis

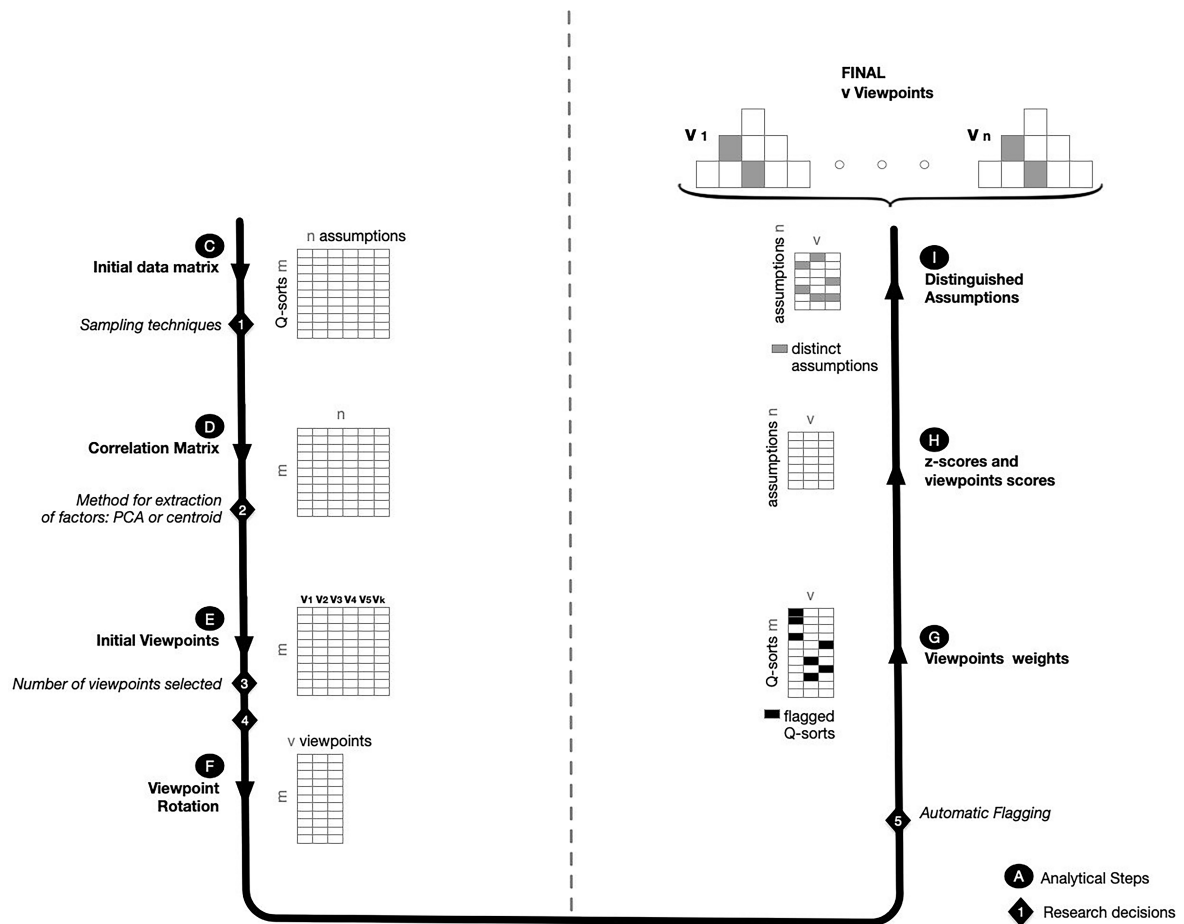
From the Q-Sorts of the 41 participants, we conduct the viewpoint analysis using the Ken-Q Analysis Web Application⁵, which performs the process of viewpoint analysis of Q-Methodology. Figure 9 shows the steps of the viewpoint analysis and specific decisions we have made.

Create a matrix of assumptions and participants. We structure the Q-sort in a two-dimensional matrix (participants \times assumptions). The value of each cell in this matrix is the level of agreement (or disagreement) that the participant attributed to the assumption in the Q-Sort (e.g., +3 or -3). The matrix obtained from the 41 participants and assumptions analyzed in our study is available in the Table 11 given in the Appendix A.

Calculate the correlation matrix. Once we create the matrix of participants and assumptions, we use *Pearson's* correlation coefficient to correlate the participants (i. e., their q-sorts). Even though is recommended the use of *Spearman's* correlation for ordinal data such as that found in Q-sorts (ALBERTS; ANKENMANN, 2001), previous studies indicate that *Pearson's* correlation produces virtually identical results and can be used in when work-

⁵ <<https://shawnbanasick.github.io/ken-q-analysis/>>

Figure 9 – Viewpoint analysis steps and research decisions



ing with Q-data (ALBERTS; ANKENMANN, 2001; BROWN, 1980). As a result, we obtain a full correlation matrix available in the Table 12 given in Appendix A, where a perfect positive correlation is registered as 1 and a perfect negative correlation is -1. This correlation matrix is used as input to extract the initial viewpoints on the next step.

Extracting the initial viewpoints. From the correlation matrix, we extract the initial viewpoints. A viewpoint is the weighted average Q-Sort of a group of participants that assigned similar values to the assumptions, i.e., it represents a hypothetical participant that best represents how those with similar viewpoints would sort the assumptions (ZABALA; PAS-CUAL; XIA, 2016). The intuition behind the viewpoint analysis relies on: Q-sorts which are highly correlated with one another may be considered to have a viewpoint resemblance, those belonging to one viewpoint being highly correlated with one another but uncorrelated with a member of other viewpoints. Viewpoint analysis indicates us how many different viewpoints there are. To systematically produce initial viewpoints, we can apply a data reduction

technique, such as *Centroid Method* (CM) or *Principal Component Analysis* (PCA). These techniques produce a grouping of similar items based on their correlations. While the PCA is used in diverse contexts, the CM is almost exclusively used in Q Methodology. We use the PCA to extract the initial viewpoints because it is the single best mathematical solution and the most popular (RAMLO, 2016). The correlation of each Q-Sort with each viewpoint is given by the *viewpoint loadings*, which range from -1 to +1. The higher the absolute value of loading (i.e., the correlation), the more the Q-sort (which represents a participant) is associated to the viewpoint (ZABALA; PASCUAL; XIA, 2016). Table 1 contains the initial set of *viewpoint loadings* for each of the 41 Q-sorts. Table 1 also describes: (*Eigenvalues*) that are the sum of squared viewpoint loading for each factor, and it measures the level of variance existing in each viewpoint (LARSEN; WARNE, 2010); (*Explained Variance*) is the percentage of total variance calculated for each viewpoint divided by the total of participants. This represents the distribution of the variance existing in each viewpoint; and (*Cumulative Explained Variance*) is the sum of the variance among the viewpoints.

Participant's ID	Viewpoints							
	A	B	C	D	E	F	G	H
#1	0.46	0.09	-0.03	0.29	0.47	0.09	-0.10	-0.17
#2	0.55	-0.04	0.09	-0.15	0.04	0.01	0.34	-0.35
#3	0.57	-0.22	-0.07	-0.11	-0.09	0.30	-0.22	-0.32
#4	0.56	-0.13	-0.04	0.18	0.18	0.00	-0.27	-0.03
#5	0.60	-0.20	-0.26	-0.10	0.34	0.19	-0.20	0.04
#6	0.56	-0.10	0.31	0.42	0.16	-0.10	0.31	-0.09
#7	0.42	0.45	0.10	0.01	-0.27	-0.08	0.32	-0.09
#8	0.33	-0.15	-0.32	-0.17	0.37	0.03	0.14	0.29
#9	0.33	0.16	-0.42	0.16	0.10	0.57	0.15	0.14
#10	0.55	0.03	-0.12	-0.08	-0.33	0.07	-0.08	0.51
#11	0.32	0.28	0.10	0.26	0.46	-0.37	0.05	0.08
#12	0.49	0.03	0.11	-0.41	0.18	0.16	0.29	-0.36
#13	0.62	-0.09	-0.16	-0.08	-0.37	0.13	0.20	-0.23
#14	0.50	0.35	0.07	0.17	-0.16	-0.10	-0.41	-0.08
#15	0.50	0.15	-0.05	-0.13	0.36	-0.27	0.03	-0.10
#16	0.55	0.13	0.23	0.20	0.20	-0.24	-0.29	0.03
#17	0.57	-0.04	0.29	0.14	-0.01	-0.20	0.34	0.28
#18	-0.15	0.33	0.11	0.63	-0.15	0.11	-0.07	-0.26
#19	0.47	-0.17	0.27	-0.42	0.00	-0.12	0.10	0.29
#20	0.34	0.36	0.55	-0.25	-0.25	0.35	0.08	0.20

#21	0.42	0.01	0.20	0.45	-0.29	0.34	-0.04	0.09
#22	0.43	0.23	-0.11	-0.53	-0.05	-0.20	-0.16	-0.09
#23	0.18	0.04	0.66	-0.05	0.41	0.10	-0.06	0.16
#24	0.31	-0.35	0.38	-0.20	-0.03	0.35	-0.15	0.13
#25	0.05	0.06	-0.49	0.48	0.01	0.12	0.39	0.18
#26	0.58	-0.10	0.15	-0.16	-0.12	0.30	0.25	-0.16
#27	0.65	-0.25	0.04	0.26	-0.21	-0.03	0.10	0.23
#28	0.45	0.38	-0.19	-0.10	-0.07	0.29	-0.20	-0.16
#29	0.44	0.02	-0.32	0.06	-0.40	-0.33	-0.27	0.07
#30	0.56	0.22	-0.32	-0.16	-0.21	0.01	0.08	0.23
#31	0.29	-0.70	0.26	0.27	-0.28	0.02	0.01	-0.11
#32	0.53	0.26	0.03	-0.05	-0.09	0.06	-0.40	0.09
#33	0.65	0.26	-0.05	0.07	0.05	0.05	-0.26	-0.21
#34	0.31	-0.73	-0.08	0.21	-0.09	-0.02	-0.25	-0.04
#35	0.37	0.26	0.03	-0.03	0.01	-0.21	-0.09	0.20
#36	0.25	-0.13	-0.32	0.00	0.55	0.30	0.06	0.12
#37	0.59	-0.31	-0.29	-0.20	0.04	-0.33	0.02	-0.12
#38	0.34	-0.10	0.53	0.09	0.11	-0.14	-0.01	-0.03
#39	0.40	0.45	-0.07	0.08	-0.17	-0.25	0.32	-0.11
#40	0.65	0.02	-0.20	0.23	0.15	0.02	0.07	0.07
#41	-0.48	0.44	0.19	0.03	0.21	0.42	-0.11	0.09

Table 1 – Initial Set of non-rotated Viewpoints

Conducting the viewpoint rotation. Although several viewpoints can be extracted from the Q-Sorts correlation matrix, few viewpoints are capable of explaining the most variance of the matrix. The rotation of viewpoints aims at identifying these viewpoints. In practice, the rotation enables us to identify *viewpoints loadings* higher than the initial ones. As a result, we obtain an increase in the strength of the correlations between some participants (Q-Sorts) and viewpoints. Before rotating the viewpoints, we need to define which ones should be rotated. To do that, the *Q-Methodology* suggests some criteria to determine how many viewpoints to retain:

- The most common method of choosing the number of viewpoints to retain is named *Guttman Rule* (GUTTMAN, 1954) that requires to retain viewpoints whose eigenvalues are higher than 1. This threshold is based on a requirement that a retained viewpoint with *eigenvalue* is lower than 1 explain less variance than a single Q-Sort (participant) (LARSEN; WARNE, 2010);

- The cumulative explained variance of each viewpoint should be above 40%, which is the proportion of the assumptions that are explained by the viewpoints (KELLY; MOHER; CLIFFORD, 2016; WATTS; STENNER, 2012; ZABALA; PASCUAL; XIA, 2016).

We observe that all the viewpoints have an eigenvalue above 1. The *Viewpoint H* has the lowest eigenvalue equal to 1.52. On the other hand, only when we consider at least four viewpoints, we obtain a cumulative explained variance higher than 40%. Note that we obtain a *cumulative explained variance* of 42 when we consider the viewpoints A, B, C, and D. Then, the application of these constraints leads us to retain four or more viewpoints. Although the constraints have reduced our scope of viewpoints, we still have to choose to rotate between four and eight viewpoints. At this point, the Q-Methodology supports the use of qualitative criteria. This leaves the researcher free to consider any solution they consider theoretically informative (BROWN, 1993; WATTS; STENNER, 2012). After analyzing the available solutions, we observe the solution containing only four viewpoints is composed of many distinguishing assumptions.

On the other hand, the solutions containing seven or eight viewpoints are composed of few distinguishing assumptions. As we shall see in more detail in the next steps, a distinguishing assumption is an assumption ranked in a viewpoint that significantly differs from its rank in all other viewpoints. Therefore, we removed solutions with a high or low number of distinguishing assumptions because they make it more difficult to identify the reasoning behind the participants loaded in each viewpoint. Both solutions with five and six viewpoints presented a reasonable number of distinguishing assumptions. However, we selected a solution with five viewpoints to be rotated because we perceived that the participants' reasoning presented more coherence. Although we have selected only one solution to perform the remaining steps of the Q-Methodology, all the data and solutions are available in the online material (SOUZA, 2019). The web tool *Ken-Q Analysis* can be used to investigate the results provided by the discarded solutions.

Once we select five viewpoints to be rotated, we can proceed with the rotation by using mathematically optimal, such as *varimax*, or manual (judgemental) techniques (BROWN, 1993). The choice will depend on the nature of the data and upon the aims of the investigator to reveal the most theoretically informative solution (BROWN, 1980). We use *varimax*

rotation, which statistically positions the viewpoints to cover the maximum amount of variance, and ensures that each Q-Sort has a high factor loading to only one factor. We prefer a mathematical rotation since it makes theoretical sense for us to pursue a rotated solution which maximizes the amount of variance explained by the extracted viewpoints (WATTS; STENNER, 2012).

Conducting the flagging. After rotating the viewpoints, we flag the participants with significant loading, i. e., the participants (Q-Sorts) most representatives for each viewpoint. In practice, when flagging, one aims at maximizing differences between viewpoints (ZABALA; PASCUAL; XIA, 2016). To do that, we run the automatic flagging, which is based on two criteria: (i) the loading value should be significantly high to 1.96 times to the standard error of a zero-order loading (in our case, for a p-value < .05); and (ii) the square loading for a viewpoint should be higher than the sum of the squared loadings for all other viewpoints (BROWN, 1980). As a result of the rotation and flagging, we obtain Table 2, which describes the loading values between the participants and the five rotated viewpoints. We present the significant loading values in **blue**. Note that 29 of the 41 participants present a significant loading (70% of the participants), and none participant was loaded significantly in two or more viewpoints. Finally, we also describe the number of participants loaded in each viewpoint.

Calculating the *z-scores* and viewpoint scores. In previous steps, the Q-Methodology focuses on analyzing the correlation between the participants and viewpoints. At this point, our focus is to investigate the relation between the assumptions and viewpoints. We begin by analyzing the ranking of assumptions within each viewpoint through the use of *z-scores* and viewpoint scores. In particular, this ranking indicates the assumption's relative position within the viewpoint. For each assumption and viewpoint, we measure the *z-score* by calculating the weighted average of the ranks (i. e., -3 to +3) attributed to the assumption by the flagged participants (Q-sorts) loaded into the viewpoint (ZABALA, 2014). Figure 10 shows the *z-scores* of the assumptions analyzed in our study. The y-axis represents the assumptions ID, and the x-axis describes the *z-scores* in an interval between -3 to +3. Remember this interval is the same adopted in our grid structure to represent a Q-sort. The legend on the right corner of the figure describes the symbols and colors used to represent each viewpoint.

From the *z-scores*, we calculate the viewpoint score of each assumption by rounding its *z-*

Participant's ID	Viewpoint A	Viewpoint B	Viewpoint C	Viewpoint D	Viewpoint E
# Defining Participants	11	5	6	3	4
#1	0.1317	0.0107	0.5681	-0.1233	0.4178
#3	0.2774	0.4562	0.0694	0.1855	0.2656
#5	0.1795	0.2834	0.1791	0.0777	0.6583
#6	0.1217	0.3369	0.705	0.0325	0.0615
#7	0.6357	-0.0141	0.1688	0.0635	-0.1584
#8	0.0502	0.0984	0.0193	0.0271	0.6174
#10	0.5261	0.3814	-0.0179	0.0843	0.0697
#11	0.1502	-0.2067	0.5685	-0.0767	0.2645
#14	0.5653	0.0802	0.3127	-0.0331	-0.0587
#15	0.2957	-0.0486	0.2827	0.1655	0.4694
#16	0.2895	0.1099	0.567	0.1059	0.1503
#19	0.1914	0.2443	0.0738	0.6049	0.1826
#22	0.5198	-0.0323	-0.1893	0.3721	0.2967
#23	-0.1261	-0.1822	0.5784	0.5051	0.0037
#24	-0.0618	0.347	0.1602	0.5016	0.0093
#25	0.0906	0.1043	0.0555	-0.6533	0.1471
#27	0.2909	0.6312	0.3264	0.0012	0.056
#28	0.6001	-0.0356	0.0435	-0.0086	0.1906
#29	0.4943	0.4036	-0.1238	-0.1718	0.0464
#30	0.6518	0.2002	-0.0959	-0.0099	0.2542
#31	-0.2335	0.8025	0.2123	0.119	-0.1667
#32	0.5351	0.0945	0.2009	0.1321	0.0977
#33	0.5544	0.1235	0.3337	0.0176	0.2434
#34	-0.2451	0.7627	0.0833	-0.0331	0.1654
#35	0.3941	-0.0215	0.1871	0.0861	0.0942
#36	-0.0821	-0.0021	0.1612	-0.1021	0.662
#38	-0.0017	0.1673	0.5089	0.3641	-0.0824
#39	0.6106	-0.0327	0.1546	-0.0946	-0.0242
#41	-0.1431	-0.6514	0.0554	-0.0567	-0.23

Table 2 – Rotated Viewpoints and Flagged Participants

score to the closest rank within the range used in the grid (i. e., -3 to +3) (ZABALA, 2014). As a result, we obtain Table 3 that describes the viewpoint score between the assumption and viewpoint. In practice, the viewpoint scores enable us to perceive how strong developers loaded in a viewpoint agree (or disagree) with a specific assumption. For example, developers loaded in Viewpoint E strongly agree (+3) that “Understanding the rationale behind changes is mandatory to avoid the bugs introduction” (Assumption #1).

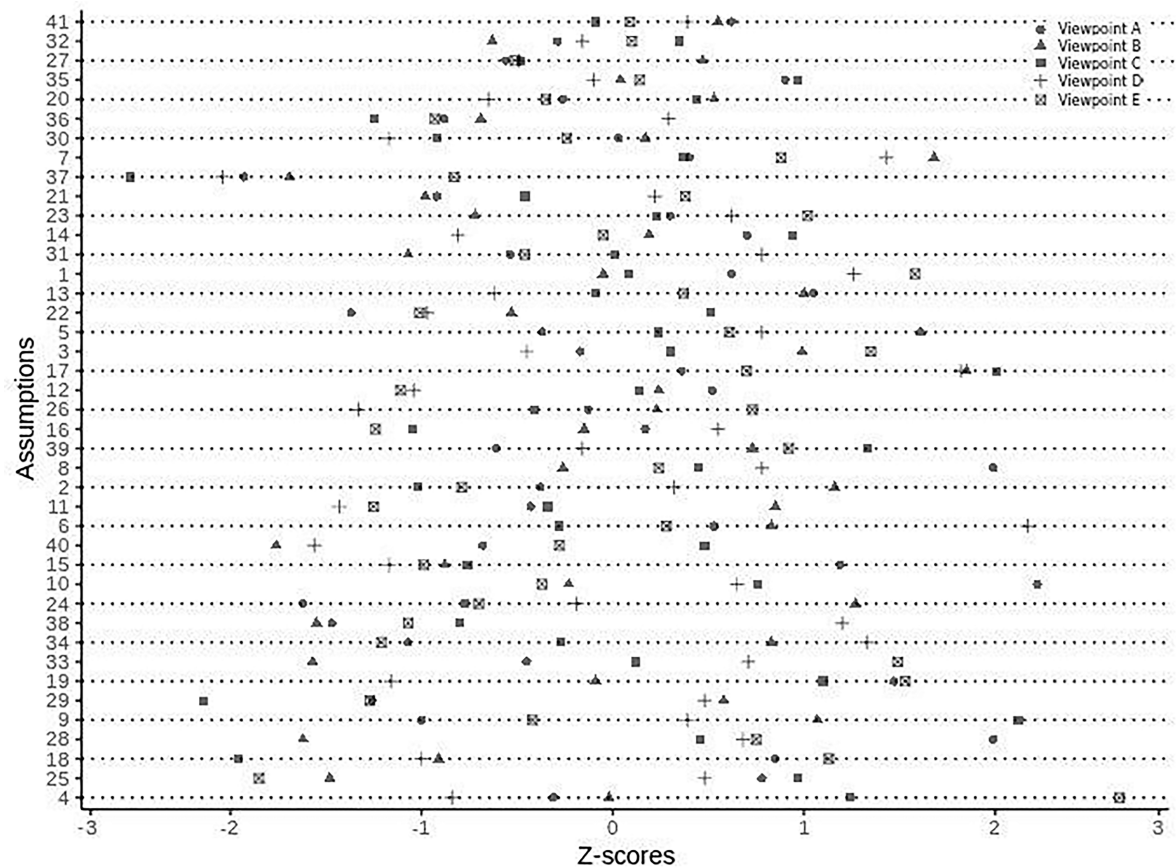


Figure 10 – Z-scores of each assumption among the viewpoints. They are ordered by the standard deviation of their z-scores for all viewpoints

Table 3 – Assumptions and viewpoint scores. The highlighted cells in **green** and **red** indicate the extreme scores of each factor, whereas an “**” and “*” is used to denote the distinguishing assumptions of each viewpoint indicating significance at $P < .01$ and $P < .05$ respectively. Consensus assumptions are in **bold**, while dissensions assumptions are in *italic*

#	Assumptions	Viewpoints Scores				
		A	B	C	D	E
1	Understanding the rationale behind changes is mandatory to avoid the bugs introduction. (RAM, 2018)	+1*	0	0	+2	+3
2	Changes that neglect non-functional requirements are more likely to introduce bugs. (CLELAND-HUANG et al., 2005)	0	+2*	-2	0*	-1
3	Having documentation associated with source code is decisive for developers to avoid bug-introducing changes on it. (MACLEOD et al., 2018; LINARES-VÁSQUEZ et al., 2015)	0	+1	0	0	+2**
4	Familiarity of developers with source code is mandatory to avoid bug-introducing changes. (FRITZ et al., 2010; FRITZ et al., 2014)	0	0	+2**	-1	+3

5	Changes involving a large number of files are more likely to introduce bugs. (MACLEOD et al., 2018; KONONENKO et al., 2015)	0*	+2*	0	+2	+1
6	When reviewing a change, reviewers need to be aware of all artifacts impacted by this change to avoid the introduction of bugs. (DEMUTH et al., 2016)	+1	+1	-1	+3**	0
7	Changes invoking a high number of features (methods, fields, classes) are more likely to introduce bugs. (DAO; ZHANG; MENG, 2017)	+1	+3	+1	+2	+1
8	Changes with high test coverage introduce less bugs. (BORLE et al., 2018)	+2**	-1	+1	+1	0
9	Having a testing team is mandatory to avoid bug-introducing changes. (PRECHELT; SCHMEISKY; ZIERIS, 2016)	-2*	+2	+3**	0	0*
10	Automated testing reduces considerably the introduction of bugs.	+3**	0	+1	+1	0
11	Manual tests verify which changes meet user requirements. Therefore, manual testing is mandatory to avoid bugs. (BOYAPATI; KHURSHID; MARINOV, 2002)	-1	+1**	-1	-2	-2
12	Continuous integration is mandatory to avoid bug-introducing changes.	+1	0	0	-1	-2
13	Using software quality assurance (SQA) methods in the code review process is decisive to avoid bug-introducing changes.	+2	+2	0	-1	+1
14	Changes that break the software architectural design are more likely to introduce bugs. (SOUSA et al., 2018)	+1	0	+1	-1*	0
15	Changes containing code smells are more likely to introduce bugs. (SAE-LIM; HAYASHI; SAEKI, 2017)	+2**	-1	-1	-2	-1
16	Avoiding recurring changes reduce the introduction of bugs. (MARTINEZ; DUCHIEN; MONPERRUS, 2013; OLIVA; GEROSA, 2015)	0	0	-2	+1	-2
17	Before accepting pull-requests, it is mandatory to compile and run the system with the changes to avoid the introduction of bugs.	0	+3	+3	+3	+1
18	<i>Familiarity of developers with the programming language adopted in a project is mandatory to avoid bugs.</i>	+1	-1	-2**	-1	+2
19	Code reviews reduce considerably the introduction of bugs. (KONONENKO et al., 2015)	+2	0**	+2	-2**	+2
20	Developers working on legacy code introduce more bugs.	0	+1	+1	-1	0
21	Adopting pair programming is decisive to avoid bug-introducing changes.	-1	-1	-1	0	+1
22	Agile methods aim at reducing the delivery lifecycle. Therefore, when adopting these methods, developers are more likely to perform bug-introducing changes. (HODA; NOBLE, 2017)	-2	-1	+1**	-1	-1
23	Code snippets frequently changed are hotspots of bugs. (PADHYE; MANI; SINHA, 2014; MCINTOSH; KAMEI, 2018)	0	-1**	0	+1	+2
24	<i>Understanding the software history is mandatory for developers to avoid the introduction of bugs. (HATA; MIZUNO; KIKUNO, 2012; COUTO et al., 2014; Kim; Whitehead, Jr.; Zhang, 2008; ZIMMERMANN; NAGAPPAN; ZELLER, 2008)</i>	-3**	+2**	-1	0	-1
25	<i>Code reuse avoids the introduction of bugs. (LI et al., 2006)</i>	+1	-2	+2	+1	-3
26	Handling the exceptions associated with a change is mandatory to avoid the introduction of bugs. (EBERT; CASTOR, 2013)	0	0**	-1	-2**	1

27	Using static analysis tools is decisive to avoid bug-introducing changes. (JOHNSON et al., 2013)	-1	0	-1	-1	-1
28	Experienced developers introduce less bugs. (RAHMAN; DEVANBU, 2011)	+3**	-2**	+1	+1	+1
29	Fixing bugs is riskier (more likely to introduce bugs) than adding new features. (ZHANG; GONG; VERSTEEG, 2013; OHIRA et al., 2012)	-2	+1	-3**	0	-3
30	Floss refactorings tend to introduce bugs. This refactoring consists of refactoring the code together with non-structural changes as a means to reach other goals, such as adding features or removing bugs.(CEDRIM et al., 2017)	0	0	-2	-2	0
31	Root-canal refactorings are less prone to introduce bugs. This refactoring is used for strictly improving the source code structure and consists of pure refactoring.(CEDRIM et al., 2017)	-1	-2	0	+1*	-1
32	Adaptive changes are more likely to introduce bugs. These changes aim at migrating systems to a new version of a third party API. (MEQDADI et al., 2013)	0	-1	0	0	0
33	<i>Developers working on his own code are less likely to introduce bugs (TUFANO et al., 2017; RAHMAN; DEVANBU, 2011; BIRD et al., 2011)</i>	-1*	-2**	0	+1	+2*
34	<i>Merge commits introduce more bugs than other commits. (LESSENICH et al., 2017; DEVANBU; ZIMMERMANN; BIRD, 2016)</i>	-2	+1	0**	+2	-2
35	Conflicting changes introduce more bugs than non-conflicting ones.	+2	0	+2	0	0
36	Developers working on code snippets containing third-party APIs tend to introduce bugs. (MEQDADI et al., 2013)	-1	-1	-2	0**	-1
37	Introduction of bugs depends on which programming language is used.	-3	-3	-3	-3	-1**
38	Geographically distributed teams introduce more bugs than teams that are not geographically distributed. (BIRD et al., 2009; DEVANBU; ZIMMERMANN; BIRD, 2016)	-2	-2	-1	+2**	-2
39	Developer's specific experience in the project is more decisive to avoid the introduction of bugs than overall general experience in programming. (SPADINI et al., 2018; DEVANBU; ZIMMERMANN; BIRD, 2016)	-1	+1	+2	0	+1
40	Code written in a language with static typing (e.g., C#) introduces fewer bugs than code written in a language with dynamic typing (e.g., Python). (HANENBERG, 2010a; RAY et al., 2017)	-1	-3	+1*	-3	0
41	Code snippet changed by different developers is hotspot of bugs.	+1	+1	0	0	0

Identifying the distinguishing assumptions. We identify the assumptions that distinguish viewpoints and those that are consensus based on whether an assumption's *z-scores* across viewpoints are statistically different. An assumption is distinguishing for a viewpoint if it ranks in a position that significantly differs from all other viewpoints. The threshold for

a difference to be considered significant is calculated based on the *Standard Errors for Differences* (SED) with the level of confidence below 0.05, as used in previous studies (KELLY; MOHER; CLIFFORD, 2016). If the difference in *z-scores* is larger than the threshold, then the assumption distinguishes one viewpoint from all another. When none of the differences between the viewpoints is significant, then is considered a consensus (ZABALA; PASCUAL; XIA, 2016). For example, the assumptions presented on top of Figure 10 (e.g #41, #32) represents a consensus assumption. Independently of the viewpoints, developers tend to agree with this assumption due to their *z-scores* values are similar, around zero. On the other hand, the assumptions showed at the bottom of Figure 10 represents the distinguishing assumptions. Note that in the assumption #25, the developers loaded in Viewpoints A, C, and D agree with this assumption, while developers loaded in the other viewpoints (B and E) disagree with them.

4 Results and Discussions

In this chapter, we describe and discuss the main results to respond the research questions analyzed in our study.

4.1 RQ1. Which assumptions are behind the developers' beliefs when looking for bug-introducing changes?

To answer this research question, we gather several assumptions from academic papers, social network, and informal talks. As a result, we obtain the 41 assumptions described in Table 3. We observe those assumptions are related to a diversity of factors that may lead or not to the introduction of bugs.

Comprehension. The assumptions #1, #4, #18, and #24 involve aspects related to understanding and familiarity. While the assumptions #1 and #24 are related to the understanding of changes and software history, the #4 and #18 address the familiarity of developers with source code and programming language.

Experience. Both the assumptions #28 and #39 address developer's experience. While #28 considers a general experience, #39 focuses on the experience in a specific project.

Habits. Six assumptions involve factors related to the developer's habits during software development. For instance, the assumptions #16-19 and #25-27 address the developers' habits of reusing (#25) or reviewing (#19) code, avoiding recurring changes (#16), handling exceptions (#26), using static analysis tools (#27), or even running the systems with the changes before accepting a pull-request (#17).

Programming Language. Both the assumptions #37 and #40 address factors related to programming language. While #37 addresses the influence of the programming language on bug-introducing changes, #40 focuses on the difference between languages with static and dynamic typing.

Maintenance. Eleven assumptions (#2-3, #12, #14-15, #20, #29-32, and #34) address factors related to software maintenance tasks. In particular, the assumption #15 focuses on code smells and #30-31 address refactoring. Two assumptions involve functional (#3) and non-functional requirements (#2). We also have assumptions involving architectural design (#14), legacy code (#20), software evolution (#32), and continuous integration (#12). Finally,

the assumptions #29 and #34 address factors more related to the routine of developers, such as, fixing bugs or adding new features (#29) and merging commits (#34).

Organizational. The assumptions also address factors related to organizational practices, such as, using SQA - Software Quality Assurance - (#13) and agile methods (#22), adopting pair programming (#21) as well as adopting geographically distributed teams (#38).

Ownership. The assumptions #23, #33, #36 and #41 address factors related to the code ownership. In particular, they focus on analyzing whether developers are working on his own code (#33), if the code is frequently changed (#23) or even if it is changed by different developers #41, or it contains third-party APIs (#36).

Changes. The assumptions also involve factors related to the characteristics and impact of a change. In this case, we analyze if a change involves a high number of features, files, classes, methods and fields (#5 and #7), if it is conflicting (#35) or even if the developers are aware of the artifacts impacted by a change (#6).

Tests. Four assumptions (#8-11) address the influence of the use of tests on bug-introducing changes, focusing on test coverage (#8), testing team (#9), automated testing (#10), and manual tests (#11).

Even though one might associate other factors to the assumptions, our intention is to highlight the diversity of factors involved in the assumptions gathered from the sources analyzed in our study.

Summary of RQ₁. The results present 41 assumptions involving a diversity of beliefs related to comprehension, experience, habits, programming language, maintenance, organizational, ownership, changes, and tests.

4.2 RQ₂. How relevant are the assumptions for developers?

In the previous research question, we analyze the assumptions behind the developers' beliefs. Now, RQ₂ intends to investigate developers' viewpoints on the relevance of the assumptions. By following the Q-Methodology, we obtain five viewpoints (named A, B, C, D and E), as described in Chapter 3. Next, we adopt the procedure described in a previous study (WATTS; STENNER, 2012) to interpret each viewpoint. Initially, three authors worked separately to interpret the viewpoints. Then, they work together to produce the fi-

nal interpretation of the viewpoints A, B, C, D, and E. For each viewpoint, we describe the developers background and experience loaded into the viewpoint, and investigate the main reasons for developers (dis)agree with each assumption. We also present the assumptions scores of each viewpoint. For instance, (A10: +3), which means that for the viewpoint under analysis, assumption number 10 is ranked as +3.

4.2.1 Viewpoint A: No matter the software history, only tests and developers' experience

Eleven out of 41 participants significantly loaded into this viewpoint, which corresponds to 26% of all participants. Most of the participants (64%) are from industry and at least 45% of them have high experience in software development, tests, bug fixing and code review. Table 4 describes the assumptions statistically related to the Viewpoint A and the ranks attributed by the developers loaded into this viewpoint.

#	Viewpoint A - Assumptions	Rank
10	Automated testing reduces considerably the introduction of bugs.	+3
28	Experienced developers introduce less bugs.	+3
8	Changes with high test coverage introduce less bugs.	+2
15	Changes containing code smells are more likely to introduce bugs.	+2
1	Understanding the rationale behind changes is mandatory to avoid the bugs introduction.	+1
5	Changes involving a large number of files are more likely to introduce bugs.	0
33	Developers working on his own code are less likely to introduce bugs.	-1
9	Having a testing team is mandatory to avoid bug-introducing changes.	-2
24	Understanding the software history is mandatory for developers to avoid the introduction of bugs.	-3

Table 4 – Assumptions and Ranks in Viewpoint A

Viewpoint interpretation: When we aim at reducing the introduction of bugs, the high test coverage (A08) plays an important role in such reduction, mainly if the tests are performed by experienced developers (A28) in an automatic way (A03). Regardless of the developer's understanding of software history (A24).

Agree. The developers loaded into this viewpoint (*Developers A*) strongly agree that “Experienced developers introduce less bugs” (A28: +3). This belief reinforces a previous study (RAHMAN; DEVANBU, 2011) that indicates developers' experience plays an important role in the reduction of bugs. Similarly, the developers strongly agree that “automated

testing reduces considerably the introduction of bugs" (A10: +3). Only in this viewpoint, the developers strongly agree with both assumptions A28 and A10, indicating they put substantial faith in the influence of the developer's experience and tests in the reduction of bugs. Still regarding tests, Developer A also agree that "changes with high test coverage introduce less bugs" (A08: +2), introducing some questions to previous study (KOCHHAR et al., 2017) that has not found any statistically significant relation between code coverage and bugs. We also observe that Developer A are concerned with source code quality. In particular, they believe in the influence of code smells in the increase of bugs (A15: +2), reinforcing a study (KHOMH et al., 2011) that suggests the existence of code smells is an indicator of fault-proneness. In the end, the developers agree it is mandatory to comprehend the rationale behind the changes to reduce the introduction of bugs (A1: +1), confirming a previous study (TAO et al., 2012) that found that the rationale of a change is the most important information for change understanding.

Disagree. Although Developer A defend the importance of understanding the rationale behind a change, they strongly disagree that it is mandatory to understand software history (A24: -3). Only in Viewpoint A, developers present this strong disagreement regarding software history. Note also that even though Developer A put substantial faith on tests to avoid bug-introducing changes, they disagree it is mandatory to have a testing team (A09: -2). This belief reinforces the findings of studies (WHITTAKER; ARBON; CAROLLO, 2012; FEITELSON; FRACHTENBERG; BECK, 2013) indicating that companies, such as *Google* and *Facebook*, obtained some benefits when they decided not having a testing team. For example, those companies could decrease the time-to-deployment, and they perceived that developers felt more committed to producing quality code. Finally, there is a contradiction in the literature regarding the influence of developers work on his own code or not. While a recent study, Tufano et al. (2017) suggest that source code in which different developers are involved are more related to bugs, another study (RAHMAN; DEVANBU, 2011) indicate that buggy code is more strongly associated with a single developer's contribution. Developer A reinforce this study (RAHMAN; DEVANBU, 2011) by disagreeing that developers working on his own code may reduce the introduction of bugs (A33: -1).

Neutral. In an existing study, Śliwerski, Zimmermann and Zeller (2005) has found that bug-introducing changes are roughly three times larger (involving a high number of files)

than other changes. However, *Developer A* do not present a position regarding this issue (A5: 0).

4.2.2 **Viewpoint B:** Do not Despise Its History Regardless of Your Experience and Ownership

Five out of 41 participants significantly loaded on this viewpoint, which corresponds to 12% of all participants. Most of the participants (80%) work in the industry and have high experience in software development and bug fixing. Table 5 describes the assumptions and ranks attributed by developers related to the *Viewpoint B*.

#	Viewpoint B - Assumptions	Rank
24	Understanding the software history is mandatory for developers to avoid the introduction of bugs.	+2
5	Changes involving a large number of files are more likely to introduce bugs.	+2
2	Changes that neglect non-functional requirements are more likely to introduce bugs.	+2
11	Manual tests verify which changes meet user requirements. Therefore, manual testing is mandatory to avoid bugs.	+1
27	Using static analysis tools is decisive to avoid bug-introducing changes.	0
19	Code reviews reduce considerably the introduction of bugs.	0
23	Code snippets frequently changed are hotspots of bugs.	-1
33	Developers working on his own code are less likely to introduce bugs.	-2
28	Experienced developers introduce less bugs.	-2

Table 5 – Assumptions and Ranks in Viewpoint B

Viewpoint interpretation: To avoid bugs, it is mandatory to understand the software history (A24), focusing on changes that neglect non-functional requirements (A2) or involve a large number of files (A5). Regardless of the developer's experience or if he is working on his own code (A33).

Agree. The developers loaded in this viewpoint (*Developers B*) agree with “understanding the software history is mandatory for developers to avoid the introduction of bugs” (A24:+2), reinforcing existing studies (HATA; MIZUNO; KIKUNO, 2012; COUTO et al., 2014; Kim; Whitehead, Jr.; Zhang, 2008) that suggest the software history as a promising way to predict bugs. Regarding the scope of the software history, developers concern with the changes involving a large number of files (A05: +2) or neglecting non-functional requirements (A02: +2). Both concerns have already been evidenced in previous studies (TU-

FANO et al., 2017; CLELAND-HUANG et al., 2005). While a study (TUFANO et al., 2017) present a statistically significant relation between large changes (involving a high number of files) and bugs, another study (CLELAND-HUANG et al., 2005) suggest that non-functional requirements are a root cause of bugs in most of the cases. Besides concerning with non-functional requirements, *Developer B* also concern with user requirements. In particular, they believe that manual testing (i. e., tests that verify which changes meet user requirements) is mandatory to avoid bugs (A11: +1). This belief introduce some questions to a previous study (WHITTAKER; ARBON; CAROLLO, 2012) that discourages manual testing since it is very expensive to be performed and turned into a bottleneck.

Disagree. *Developer B* disagree that “experienced developers introduce less bugs” (A28: -2), reinforcing previous studies (TUFANO et al., 2017; MOCKUS, 2010) that indicate more experienced developers are usually the authors of bug-introducing changes because they perform more complex tasks. Similarly, *Developers B* disagree that “developers working on his own code are less likely to introduce bugs” (S33: -2). As discussed in the previous chapter, this disagreement reinforces a recent study (TUFANO et al., 2017) but also contradicts another one (RAHMAN; DEVANBU, 2011). *Developer B* also disagree that “code snippets frequently changes are hot-spots of bugs” (A23: -1), introducing some questions to a recent study (MONDAL; ROY; SCHNEIDER, 2017) that indicates more recently changed code have a higher possibility of containing bugs.

Neutral. Even though one might argue that code review and the use of static analysis tools to avoid bugs, *Developer B* have a neutral position regarding these issues.

4.2.3 Viewpoint C: Have your own testing team

Six out of 41 participants significantly loaded on this viewpoint, which corresponds to 15% of all participants. Most of the participants (83%) are from industry, and at least 50% of them have high experience in software development, code review, and bug fixing. Table 6 describes the assumptions and ranks attributed by developers related to the *Viewpoint C*.

Viewpoint interpretation: Having a testing team (A09) is mandatory to avoid bug-introducing changes. Regardless if a developer focuses on fixing bugs (A18) or he is familiar with the programming language adopted in a project (A18).

Agree. Developers loaded in this viewpoint (*Developers C*) strongly agree “Having

#	Viewpoint C - Assumptions	Rank
9	Having a testing team is mandatory to avoid bug-introducing changes.	+3
4	Familiarity of developers with source code is mandatory to avoid bug-introducing changes.	+2
22	Agile methods aim at reducing the delivery lifecycle. Therefore, when adopting these methods, developers are more likely to perform bug-introducing changes.	+1
40	Code written in a language with static typing (e.g., C#) introduces fewer bugs than code written in a language with dynamic typing (e.g., Python).	+1
34	Merge commits introduce more bugs than other commits.	0
18	Familiarity of developers with the programming language adopted in a project is mandatory to avoid bugs.	-2
29	Fixing bugs is riskier (more likely to introduce bugs) than adding new features.	-3

Table 6 – Assumptions and Ranks in Viewpoint C

a testing team is mandatory to avoid bug-introducing changes" (A09:+3), contradicting previous studies (FEITELSON; FRACHTENBERG; BECK, 2013; WHITTAKER; ARBON; CAROLLO, 2012; PRECHELT; SCHMEISKY; ZIERIS, 2016) that shows a reduction of bugs when companies do not adopt a testing team. They also believe familiarity with source code is mandatory to avoid bug-introducing changes (A04:+2). This belief reinforces a previous study (RAHMAN; DEVANBU, 2011) that indicates code snippets suffering interference from different developers are more difficult to be familiar with and, consequently, developers tend to introduce more bugs. Moreover, *Developer C* present a concern with the typing (static or dynamic) supported by programming language. They believe languages with static typing (e. g. C#) are less susceptible to bugs, reinforcing the discussions of developers in specialized forums (HANENBERG, 2010b; RAY et al., 2017). In a previous study, Verner, and Babar (2004) indicate that one of the main benefits of agile methods is to reduce the delivery lifecycle (Verner; ; Babar, 2004). However, *Developer C* believe it is needed to be more careful on speeding up the software development process since it may lead developers to introduce more bugs.

Disagree. However, *Developer C* strongly disagree “fixings bugs is more likely to introduce bugs than adding new features" (A:29, -3). This belief contradicts a study (YIN et al., 2011) that indicates developers may introduce more bugs when fixing existing ones than adding new features. Similarly, the developers also disagree that familiarity with the programming language adopted in a project is mandatory to avoid bugs (A18: -2). This belief contradicts some discussions in specialized forums where participants argue a high

influence of the programming language in the reduction of bugs (HANENBERG, 2010b; RAY et al., 2017).

Neutral. Merge commits is a crucial moment along with the software development since different parts of the systems are integrated and conflicts can arise. Thus, one might expect that merge commits may lead developers to introduce more bugs than other commits. However, *Developer C* has a neutral position regarding this issue.

4.2.4 **Viewpoint D: Be Aware of All Artifacts Impacted by Changes, mainly in Geographically Distributed Teams**

Three out of 41 participants significantly loaded on this viewpoint, which corresponds to 7% of all participants. All participants are from industry, and at least 33% of them have high experience in software development, bug reporting, and bug fixing.

#	Viewpoint D - Assumptions	Rank
6	When reviewing a change, reviewers need to be aware of all artifacts impacted by this change to avoid the introduction of bugs.	+3
38	Geographically distributed teams introduce more bugs than teams that are not geographically distributed.	+2
31	Root-canal refactorings are less prone to introduce bugs. This refactoring is used for strictly improving the source code structure and consists of pure refactoring.	+1
36	Developers working on code snippets containing third-party APIs tend to introduce bugs.	0
2	Changes that neglect non-functional requirements are more likely to introduce bugs.	0
14	Changes that break the software architectural design are more likely to introduce bugs.	-1
19	Code reviews reduce considerably the introduction of bugs.	-2
26	Handling the exceptions associated with a change is mandatory to avoid the introduction of bugs.	-2

Table 7 – Assumptions and Ranks in Viewpoint D

Viewpoint interpretation: To avoid bugs, be aware of all artifacts impacted by a change when reviewing it, mainly when we have geographically distributed teams (A38). It does not matter if code review (A19) and exceptions handling (A26) policies have been applied in the project.

Agree. Developers loaded in this viewpoint (*Developers D*) strongly agree it is needed to be aware of all artifacts impacted by a change when reviewing it to avoid the

introduction of bugs (A6: +3). This belief reinforces existing studies (MISIRLI; SHIHAB; KAMEI, 2016) that use change impact analysis to localize bugs. The developers also agree that “*distributed teams introduce more bugs than teams that are not geographically distributed*” (A38: +2). This belief complements the results of previous study (DEVANBU; ZIMMERMANN; BIRD, 2016) that indicates geographic distribution has a measurable effect on software quality, but it not identify a consistent effect since sometimes it was good, and sometimes bad. Still regarding software quality, Developers D believe that “*root-canal refactorings are less prone to introduce bugs*” (A31: +1), introducing some questions to previous study (BAVOTA et al., 2012) that suggests refactorings tend to induce bugs very frequently.

Disagree. Bacchelli and Bird (2013) suggest that the main motivation for developers to review code is to find bugs, Developers D disagree that code reviewing reduces considerably the introduction of bugs (A19: -2). Similarly, Developer D disagree it is mandatory to handle exceptions in order to avoid bugs (A26: -2), reinforcing previous studies (EBERT; CASTOR; SEREBRENIK, 2015; OLIVEIRA et al., 2018) that suggests better testing for exception handling bugs since they are ignored by developers less often than other bugs. Developer D also disagree that changes breaking the software architectural design are more likely to induce bugs (A14:-1). This belief introduces some questions to existing studies that suggest when the software architectural design is compromised by poor or hasted design choices, the architecture is often subject to different architectural problems or anomalies, which may lead to bugs (Kourosfar et al., 2015; FONTANA et al., 2016).

Neutral. Finally, it is known that the reuse of third-party APIs has several benefits to software development, mainly in terms of time-consuming. On the other hand, Linares-Vásquez et al. (2013) also indicate that the fault-proneness of APIs have been the main cause of applications failures. The belief of Developer D reinforces these contradictions since they adopt a neutral view regarding the impact of APIs on the introduction of bugs (A36: 0). Moreover, even though Cleland-Huang et al. (2005) suggests that changes neglecting non-functional requirements are the root cause of bugs, Developer D do not have a definitive view about this issue (A2: 0).

4.2.5 Viewpoint E: Be familiar with the code

Four out of 41 participants significantly loaded on this viewpoint, which corresponds to 9% of all participants. All participants are from industry, and at least 50% of them have high experience in software development and bug fixing.

#	Viewpoint E - Assumptions	Rank
4	Familiarity of developers with source code is mandatory to avoid bug-introducing changes.	+3
33	Developers working on his own code are less likely to introduce bugs.	+2
9	Having a testing team is mandatory to avoid bug-introducing changes.	0
37	Introduction of bugs depends on which programming language is used.	-1

Table 8 – Assumptions and Ranks in Viewpoint E

Viewpoint interpretation: Familiarity with the source code is mandatory to avoid bug-introducing changes (A4;A33) regardless of the programming language (A37).

Agree. Developers loaded into this viewpoint (*Developers E*) strongly agree that “familiarity of developers with source code is mandatory to avoid bug-introducing changes” (A4:+3), reinforcing but also contradicting existing studies (HANENBERG, 2010b; RAY et al., 2017; MOCKUS, 2010). Rahman and Devanbu (2011) suggests that the higher the developers’ experience in a code, the more they know the code and, consequently, they introduce less bugs. On the other hand, studies (TUFANO et al., 2017; MOCKUS, 2010) also suggest that developers tend to perform slightly larger (and likely more complex) changes when they have higher experience on the code source and, consequently, they induce more bugs (MOCKUS, 2010). Similarly, *Developers E* also agree that “developers working on his own code are less likely to introduce bugs”, reinforcing a study (BIRD et al., 2011) that indicates code maintained by single developers are strongly associated with less bugs.

Disagree. We observe that *Developer E* disagree that the introduction of bugs depends on which programming language (A37:-1). This belief introduces some questions to a study (HANENBERG, 2010c) that indicates typed languages, in general, are less fault-proneness than the dynamic types.

Neutral. Previous studies (FEITELSON; FRACHTENBERG; BECK, 2013; WHITTAKER; ARBON; CAROLLO, 2012) involving large companies (e. g., Google and Facebook) discourage the adoption of testing team, *Developer E* assume a neutral

position regarding this issue (A09: 0).

Summary of RQ₂. We identify five developers' viewpoints regarding a diversity of assumptions. While some developers strongly agree that using automated testing (Viewpoint A), having a testing team (Viewpoint C), being aware of impacted artifacts (Viewpoint D), and being familiar with source code (Viewpoint E) are mandatory to avoid bugs, other developers strongly disagree with the relevance of understanding the software history to avoid the introduction of bugs (Viewpoint A).

4.3 RQ₃. Why developers (dis)agree with an assumption?

After gathering the assumptions and identifying the developers' viewpoints, RQ₃ investigates the main reasons for developers (dis)agree with each assumption. To answer this research question, we ask developers to comment about the motivation for them (dis)agree with a specific assumption. Based on those comments, we discuss why developers agree or disagree with the assumptions. We translated and lightly edited some relevant comments to provide qualitative evidence to reflect the developers' viewpoints.

Viewpoint A. As previously discussed, Developers A believe that experienced developers (A28), automated testing (A10), high test coverage (A08), and understanding the rationale behind changes (A01) may reduce the introduction of bugs. Moreover, they also believe that code smells may lead to bug-introducing changes (A15). The comments of these developers provide some explanations regarding those beliefs. For instance, developers #14,32,33,41 argue that developers' experience avoids the repetition of mistakes already led to buggy code in the past. Regarding automated testing, developer #39 argues that tests should be cheap and easy to run repeatedly on every single change. Still regarding tests, developer #30 presents a reasonable explanation on the influence of test coverage to reduce bug-introduce changes. He thinks if developers test most of (or all) the branches in a particular method with different inputs, they can cover most of the corner cases. In this way, the bug-proneness of a new change is reduced. Finally, developer #5 argues that understanding the code is essential to do not make something wrong. Some comments (translated and lightly edited) reflecting these beliefs include:

“I believe that developers that already have saw/wrote buggy code and learned

from their mistakes are less likely to introduce bugs..." (#14)

"This should be self-evident: tests should be cheap and easy to run again and again on every single change." (#39)

"I think if we test most of (or all) the branches in a particular method with different inputs, we can cover most of the corner cases. In this way, the bug-proneness of the new change is reduced." (#30)

"If you do not understand what you are doing with code, the chances to do something wrong are higher. After all, you do not know what is going on." (#5)

On the other hand, Developer A disagree that working on his own code (A33), having a testing team (A09) or understanding the software history is mandatory to avoid bugs (A24). Developer #10 argues that developers tend to be more sloppy when working alone or on his own code. Developer #21 also explains that when we have a single developer, he tends to get used to the code and, he tends to test only what he believes it is not broken – as a consequence, increasing the changes to introduce bugs. Regarding having a testing team, developer #30 argues that a developer is able to write code and test it. Finally, developer #4,14 argue that even though the software history had bugs in the past, this does not decrease the chances of introducing a new bug.

"Mostly, developers tend to be more sloppy when working alone or on his own code" (#10)

"When we develop we tend to get used to the code we produce and we are tempted to just test what we are sure is not broken. So, the chance to introduce bug is big." (#21)

"I think the same developer who wrote the code can write good enough tests to avoid bug-introducing changes. I worked in some teams where the developer is also responsible for writing the test code. After this, a code review session is mandatory (with, at least one different developer). After the code review, we usually end-up with code of good quality and with low chance of introducing bugs. In this way, I think a testing team is not mandatory." (#30)

“the history may show some past errors, but it by itself does not decrease the chances of a bug.” (#4)

“It does not make sense, because errors can continue to happen even understanding the software history.” (#14)

Viewpoint B. Developer B agree that understanding the software history (A24), applying manual tests (A11), avoiding large (A05), and negligent changes (A02) may reduce the introduction of bugs. Developer #34 argues that after understanding the software history developers may identify the parts most likely to have bugs. Moreover, developer #3 argues that manual tests are important and complement automated ones. Regarding large changes, developers #30,34,10,27,37 argue that a large number of files require more attention when performing tests. Still regarding changes, developer #3 argues that neglecting non-functional requirements are the main cause of anomalies at run-time.

“When I understand the history of software, it is possible to identify possible locations that are most likely to introduce bugs and thereby avoid them.”(#34)

“In my opinion, manual testing is important and complements automated testing. In development, we must not forget to create automated tests and also do manual tests to avoid the introduction of bugs.”(#3)

“When a developer changes several files in a single commit, the odds of introducing bugs are higher. I think it is harder to test all the changes (manually or automatically) if we have a lot of them...”(#30)

“Because it’s harder to check every single feature added/removed and it’s effect on the others.”(#10)

“Neglecting non-functional requirements are the main cause to anomalies at run-time.”(#3)

Developers disagree that experienced developers (A28), avoiding code snippets frequently changed, and working on his own code (A33) may avoid bugs (A23). Developer #34 argues that experienced developers tend to works in critical modules of the software, then he tends to introduce more bugs. Developer #11 explains that code snippets frequently

changed are the most reviewed code in the system and, therefore, it is less likely to remain bugs. Developer #27 argues that developers working on his own code may not be aware of mistakes, which may be recognized from outside developers.

“A senior developer who works on the most critical or complex modules of the system can insert more bugs than a junior developer working only with interfaces or simpler parts.” (#34)

“These code snippets are the most reviewed code in the system. Therefore it is less likely to remain buggy for a long period of time.”(#11)

“The fact that a developer is working on his own code may cause the introduction of bugs since he may not be aware of mistakes that he is doing. Developers from outside may have a different view from the original developer and find some bugs.”(#27)

Viewpoint C. Developer C agree that having a testing team (A09), familiarity with the source code (A04), avoiding agile methods (A22) and using static typing languages (A40) may reduce bug-introducing changes. Developers #4,11,16 argue that the testing team is focused on identifying bugs. Moreover, developer #17 suggests that a testing team tends to be less biased when testing the source code since their members did not write the code. Regarding the familiarity with the source code, developer #38 argues the familiarity of the developers with the source code provides a better sense where they can change without introducing bugs. Developer #16 provides comments that help to explain the agreement regarding the agile methods and static typing languages. He argues that shorter delivery cycles in the agile methods produce a pressure behind the developers to provide fast deliveries on the software development at the industry, and this hurry to complete the tasks may reduce the software quality. Moreover, he also argues that dynamic typing is more sensitive to introduce bugs by the developer, e.g. when values of different types are assigned to the same variable. Therefore, more effort is required for the tester, and this type of error may remain ignored.

“A test team tends to be less biased when testing the source code since they did not write the code” (#17)

“The familiarity of the developers with the source code gives a better sense where they can change, avoiding the introduction of bugs” (#38)

“Unintentionally, a commercial environment with shorter delivery cycles produce pressure under the developers to deliver fast. Certainly, only adopting agile methods will not avoid the introduction of bugs, but in high-pressure environments, the hurry ends up interfering in the software quality.” (#16)

“Dynamic typing codes are more sensitive to programmer errors. When values of different types are assigned to the same variable, the test cases require more effort for the tester, and eventually, this type of error goes unnoticed.” (#16)

Although *Developer C* agree with the assumptions A4, A9, A22 and A40, they disagree that familiarity with the programming language (A18) may reduce bugs and to fix them is riskier than adding new features (A29). *Developer #16* explains that independent of programming language, many bugs are due to flaws in programming or business logic. Regarding fixing bugs or adding new features, developers #11,6 argue that when a developer has to fix a bug, he will focus on the primary objective that is to prevent bugs, and probably this will not happen if the code is correctly tested.

“Test teams have focused on discovering flaws. They test the entire coverage area of the changed code, so they are key to detecting bugs” (#16)

“The whole point of fixing bugs is to prevent faulty code. When one adds new features, the focus is to deliver the feature, and some change can be made with buggy code.” (#11)

Viewpoint D. *Developer D* agree that being aware of all artifacts impacted by a change (A06), working on teams not geographically distributed (A38) and applying root-cause refactorings may reduce bugs (A31). *Developer #24* argues that reviewers need to be able to identify any anomaly inserted in the artifacts involved. Regarding geographically distributed teams, *developer #19* argues that this assumption depends a lot on the team’s management. He explains that when the team leader is weak, even if it is geographically in the same place, there may be a higher number of bugs introduced when compared to a team with proper supervision that works geographically separate. However, the last case requires

a lot more management effort. Finally, developer #19 explains that the time zone differences are an obstacle for easy and effective communication.

“Reviewers will be able to identify any anomaly inserted in the artifacts involved.” (#24)

“This factor depends a lot on the teams’ management. When team management is weak, even if it is geographically in the same place, there may be an even higher number of bugs inserted when compared to a time with good management that works geographically separate. However, the second case requires a lot more management effort.” (#19)

Developer D disagree that handling the exceptions (A26), reviewing code (A19) and changes breaking the architectural design (A14) may reduce bugs. Developer #24 argues that even though not all exceptions should be handled, he is aware that correct exception handling (through testing) may prevent the introduction of bugs. Regarding code review, developer #26 argues that reviewers are more focused on code quality (such as bad smells), not necessarily on avoiding bugs.

“Not all exceptions should be handled by the developer (runtime exceptions). Besides, I am aware of studies that prove that correct exception handling (through testing) prevents the propagation of bugs in the system.” (#24)

“During the code review, the reviewers are more alert about the quality of the code (conventions, bad smells, architecture, and tests) than checking if the algorithm is implemented correctly or if the change will impact an existing feature. However, having reviewers forces the use of good practices to the project, which in the future will contribute to reducing the introduction of bugs during maintenance.”(#26)

Developer E agree that familiarity with source code (A04) and working on his own code may reduce bugs (A33). In particular, the developer #36 argues that the more you know (or you are familiar with) a source code, the more you control it, and this is crucial to avoid bugs.

“I have worked on projects aiming to evolve systems that I was unfamiliar. My overall feeling is that more I knew about the system, better I was able to make (close to) bug-free code.”(#36)

On the other hand, Developer E disagree that the introduction of bugs depends on which programming language is used (A37). Developers #16,#23,#37 argue that bugs exist in every language, while developer #40 complement this arguing that the introduction of bugs depends more of developers’ skills.

“The introduction of bugs is independent of the programming language.” (#16)

“Bugs exist in every language, it depends more on the skills of the developer than the language itself.” (#40)

Summary of RQ₃. Most the developer’s reasons to (dis)agree with the assumptions are based on his own experience, code complexity, short time delivery, development practices, management efforts, and familiarity with the source code.

4.4 RQ₄. Which assumptions are consensus/dissensus among developers’s viewpoints?

Although developers have different viewpoints on the relevance of assumptions, it is expected they have some views in common regarding the relevance of the assumptions. RQ₄ intends to investigate the assumptions that are consensus or dissensus among the developers’ viewpoints. Table 9 presents the assumptions that are consensus (#A7, #A17 and #A37) or dissensus (#A18, #A24, #A25, #A33 and #A34) among the developers’ viewpoints. We also describe the assumptions scores associated to each viewpoint. In this chapter, we present the assumptions ranks in a way to highlight the difference between how the viewpoints under analysis rank an assumption, compared to the others. For instance, [A33: -1*, -2**, 0, +1, +2*], which means that we are analyzing assumption 33 and its rank is -1 in viewpoint A, while in viewpoint B its rank is -2, in viewpoint C is 0, in viewpoint D is +1 and in viewpoint E is +2. The asterisk means that an assumption rank is distinguishing in that viewpoint, whereas an asterisk “**” and “*” denote the significance at $P < .01$ and $P < .05$ respectively.

#	Assumptions	Viewpoint Scores				
		A	B	C	D	E
Consensus						
7	Changes invoking a high number of features (methods, fields, classes) are more likely to introduce bugs.	+1	+3	+1	+2	+1
17	Before accepting pull-requests, it is mandatory to compile and run the system with the changes to avoid the introduction of bugs.	0	+3	+3	+3	+1
37	Introduction of bugs depends on which programming language is used.	-3	-3	-3	-3	-1
Dissensus						
18	Familiarity of developers with the programming language adopted in a project is mandatory to avoid bugs.	+1	-1	-2**	-1	+2
24	Understanding the software history is mandatory for developers to avoid the introduction of bugs.	-3**	+2**	-1	0	-1
25	Code reuse avoids the introduction of bugs.	+1	-2	+2	+1	-3
33	Developers working on his own code are less likely to introduce bugs.	-1*	-2**	0	+1	+2*
34	Merge commits introduce more bugs than other commits.	-2	+1	0**	+2	-2

Table 9 – Consensus and Dissensus Assumptions

4.4.1 Consensus Assumptions

Regardless of the different developers' viewpoints, we observe three assumptions A7, A17, and A37 in consensus.

A7: Be Careful with High Number of Features. In all the viewpoints (A-E), we have a consensus regarding the assumption "Changes invoking a high number of features (methods, fields, classes) are more likely to introduce bugs" [A7: +1, +3, +1, +2, +1]. The developers argue that the larger the number of invoked features, the more difficult and complex to track the impact of the changes. As a consequence, making it more likely to introduce bugs.

A17: Compile and run the system with the changes. In most of the developers' viewpoints, there is a strong agreement consensus with the assumption "Before accepting pull-requests, it is mandatory to compile and run the system with the changes to avoid the introduction of bugs" [A17: 0, +3, +3, +3, +1]. Developers believe it is fundamental or even rather obvious the need to compile and run the system before committing changes to avoid conflicts or identify even the simplest of the bugs.

A37: Does not Matter the Programming Language. Differently from previous assumptions, developers present a disagreement consensus. Majority of them strongly disagree that "Introduction of bugs depends on which programming language is used" [A37: -3, -3, -3,

-3, -1]. Developers argue that bugs exist in every programming language, and the introduction of bugs depends more on the knowledge or skills of a developer than the programming language itself.

4.4.2 Dissensus Assumptions

There is no concordance among the developers' viewpoints in the assumptions A18, A24, A25, A33, and A34.

A18: Familiarity with the programming language. This assumption presents dissensus according to the developer's viewpoints scores. They have dissensus to believe that "Familiarity of developers with the programming language adopted in a project is mandatory to avoid bugs" [S18: +1, -1, -2, -1, +2]. While developers loaded in Viewpoint A and E agree with this assumption, developers loaded in Viewpoint B-D disagree with it. For instance, developers loaded in Viewpoints A and E argues that understanding the specificities of the programming language decreases the risk of bugs. On the other hand, developers loaded in the remaining viewpoints argue that most of the bugs are related to the programming or business logic than the programming language used.

A24: Understanding software history. Similarly to the previous assumption, the developer's viewpoints have different beliefs among this assumption. They have dissensions upon: "Understanding the software history is mandatory for developers to avoid the introduction of bugs" [A24: -3, +2, -1, 0, -1]. While developers loaded in Viewpoint B agrees with this assumption, the developers loaded in Viewpoint A, C, and E disagree with it. Developers B argues that when understanding a software history, it is possible to identify hot-spots that are the more prone to introduce bugs. On the other hand, Developers A, C, and E argue that there is not any relation between software history and bugs. They also comment that even though the understanding of the software history may show some errors, itself do not decrease the introduction of bugs.

A25: Code reuse. Regarding the assumptions "Code reuse avoids the introduction of bugs" [A25: +1, -2, +2, +1, -3], while developers loaded in Viewpoints A, C, and D agree with it, developers loaded in the remaining viewpoints disagree with this assumption. Developers A, C, and D argue that reused code is already highly tested and mature, then it is more "safer" than writing new code (and supposedly bug-free). On the other hand, Developers B

and E comment that to reuse code does not mean the code is correct. The intuition behind this comments relies on the fact that “copy + paste” from *StackOverflow* is not exactly a good practice since the code may not be complete or correct. Moreover, those developers also explain that when reusing code containing bugs or smell, you are only spreading it.

A33: Code ownership. The developer’s viewpoints have dissensus that “Developers working on his own code are less likely to introduce bugs” [A33: -1, -2, 0, +1, +2]. Developers D and E believe that code ownership is an important aspect to prevent the introduction of bugs since it is expected that developers who have worked in the code have better control and knowledge about it. On the other hand, Developers A and B do not believe in code ownership. They argue developers tend to be more sloppy, and, consequently, may not be aware of his own mistakes during the software development.

A34: Aware with merge commits. Similarly to the previous assumption, the developer’s viewpoints have dissensus that “Merge commits introduce more bugs than other commits” [A34:-2, +1, 0, +2, -2]. Developers A and E disagree that merge commits are more likely to introduce bugs. They argue that merge commits are generally fairly straightforward and more accessible to resolve them. On the other hand, Developers B and D agree with the assumption #34. They describe three main reasons: (i) usually, the developers that perform the merge is not the same that created the conflict; (ii) merge commits are harder to inspect than other commits since merge commits involve more files to be inspected; and (iii) merge is a result of development, and bugs can already be introduced before.

<p>Summary of RQ₄. The results indicate that assumptions involving amount of features, system compilation and execution, and programming language produce more consensus among developers than assumptions involving familiarity, software history, reuse, ownership, and merge commits.</p>
--

5 Implications

In this chapter, we discuss some of the implications of our study. The assumptions, viewpoints, consensus/dissensus as well as the comments provided by the participants provide valuable knowledge for researchers and practitioners to improve the process of code review as stated in Chapter 4.

The assumptions, viewpoints, consensus/dissensus, as well as the comments provided by the participants, provide valuable knowledge for researchers and practitioners to improve the process of code review.

Assumptions. Existing studies have focused their researches only on technical issues, such as the influence of the number of files (BAUM; SCHNEIDER; BACCHELLI, 2019) or tests (BORLE et al., 2018) on bug-introducing changes. The assumptions collected in our study can be useful for researchers to further investigate the issues involved in a code review. For instance, researchers can explore the relationship between those assumptions and technical aspects, e. g., how developers have used code review tools/techniques to support assumptions that they strongly agree with. Furthermore, they can investigate these assumptions in future qualitative researches just as previous studies (MATTHIES et al., 2019; TAO et al., 2012; DEVANBU; ZIMMERMANN; BIRD, 2016) helped us to construct our set of assumptions.

Viewpoints. The five viewpoints extracted in our study reflect common perceptions of groups of developers. This way, those viewpoints can support researchers to build code review tools/techniques tailored to a specific group. For example, the developers loaded into Viewpoint A believe that no matter the software history, only tests and developers' experience. Thus, we can create tools/techniques that aid the reviewers to analyze the tests performed on the changes under review as well as the experience of the developers responsible for those changes. This way, we can speed up the process of code review. Moreover, those viewpoints reinforce or introduce some questions to previous studies, thus paving the way for improving state-of-the-art techniques to code review.

Consensus/Dissensus. Besides extracting the viewpoints, we also analyze the consensus and dissensus among those viewpoints. By identifying the consensus, we shed light on the creation of code review tools/techniques that can support common perceptions among

groups of developers. For instance, the groups loaded into the Viewpoints A-E have a consensus regarding the assumption "Changes invoking a high number of features (methods, fields, classes) are more likely to introduce bugs". Thus, researchers can focus their efforts to create tools/techniques that make easier reviewers analyze the invoked features during a code review, speeding up the process of code review performed by those groups. On the other hand, the identification of dissensus assumptions reinforces the need for building tools tailored to specific viewpoints. For example, while groups of developers loaded into Viewpoint B and E disagree that "code reuse avoids the introduction of bugs", groups loaded into Viewpoint A, C and E agree with this assumption. Thus, code review tools that support the analysis of code reuse may not be useful for the groups loaded into Viewpoint B and E.

Comments. The participants of our study provide several comments to explain the reasons for agreeing/disagreeing with an assumption. Those comments reveal practical insights that should be considered during the process of code review to identifying bug-introducing changes. For example, the comments of the developers #26 and #17 argue the need of having reviewers and testers teams to follow good practices and avoid biases during code review.

6 Related Work

This chapter includes the related work performed on developers' beliefs, perceptions, personality, and assumptions related to diverse contexts in software engineering, which provided inspiration for our study.

Previous studies have investigated about the developers' beliefs, perceptions, personality, and assumptions regarding different concerns in software engineering. For instance, Meyer et al. (2014) investigate the developers' perceptions about productive and unproductive work through a survey and observational study. The results indicate that interruptions or context switches impact productivity significantly. In another study, they investigate the variation in productivity perceptions based on a survey with 413 developers (MEYER; ZIMMERMANN; FRITZ, 2017). They also identify six groups of developers with similar perceptions of productivity regarding the impact of aspects social, lone, focused, balanced, leading, and goal-oriented developers. Similarly to our study, these studies (MEYER et al., 2014; MEYER; ZIMMERMANN; FRITZ, 2017) aims at identifying common perceptions among developers. However, they focus on aspects that may impact productivity instead of concerns involved in the code review process to identify bug-introducing changes. In addition, they do not focus on identifying consensus/dissensus among developers' viewpoints.

Smith, Bird and Zimmermann (2016) investigate the relations between work practices, beliefs, and personality traits, involving 797 software engineers of Microsoft. The results suggest some differences between engineers, developers and, managers in terms of five personality domains (named OCEAN - openness, conscientiousness, extraversion, agreeableness, and neuroticism). For instance, developers who chose to build tools were more open, conscientious, extraverted, and less neurotic. Engineers who agree with the assumption "Agile development is awesome" were more extroverted and less neurotic. Managers were more conscientious and extraverted. On the other hand, there was no personality difference between developers and testers, introducing some questions to previous research. They present interesting insights about the developers' beliefs, work practices, and personality. However, the authors do not explore comparative results with the empirical evidence of these findings.

In a recent study, Matthies et al. (2019) investigate the perceptions of agile development practices and their usage in Scrum teams. The authors collect perceptions of 42 students

through surveys. The results indicate that the Scrum methodology impacts the students' views of employed development practices. In fact, most of the students attributed to the agile manifest the main reason to use the version control system according to agile ideas. The main limitation of this work is to collect perceptions only from students of a software engineering course. As a consequence, those perceptions may not reflect the perceptions of professional developers.

Devanbu, Zimmermann and Bird (2016) investigate the developers' beliefs in empirical software engineering. They gather a set of claims, mostly from literature, aiming to investigate the accordance between the empirical evidence and developers' beliefs. The results indicate that belief is mainly based on personal experience, rather than on findings in empirical research. The results also suggest that actual evidence in a project has not a strict relation with the beliefs. The beliefs collected in this study serve as a basis for our study. While they investigate developers' beliefs regarding empirical software engineering evidence, our study focuses on analyzing the developers' beliefs when reviewing code to identify bug-introducing changes. To do that, we edit the beliefs collected in their study (DEVANBU; ZIMMERMANN; BIRD, 2016) to focus on bug-introducing changes. Moreover, we extend the number of beliefs to be analyzed, involving 41 beliefs instead of only 16 as described by the authors (DEVANBU; ZIMMERMANN; BIRD, 2016).

The studies aforementioned are concerned with viewpoints of a group of people – usually developers – regarding diverse topics, but none of these studies focused on analyzing developers' viewpoints on identifying bug-introducing changes. Besides, the majority of them applied either qualitative (surveys) and quantitative approaches to explore subjectivity. Brown (1993) critiques this dichotomy, advocating in favor of Q-Methodology as a possible alternative. In our recent study (CARTAXO et al., 2019), we use the Q-Methodology that transcends this argument because Q is neither entirely qualitative or fully quantitative. By combining the strengths of both qualitative and quantitative research, Q-Methodology allows for the simultaneous study of objective and subjective issues to determine an individual's viewpoints (CROSS, 2005).

7 Threats to Validity

In this chapter, we present the threats to validity by following the (WOHLIN et al., 2012) validity criteria.

Construct Validity. One might consider that 41 assumptions may not be enough to support a broad understanding of the developers' viewpoints regarding bug-introducing changes. However, we select assumptions involving a diversity of concerns related to different aspects inherent to software development. In general, there are 40 to 80 assumptions used in Q-Methodology studies (WATTS; STENNER, 2012; CARTAXO et al., 2019). Still regarding the assumptions, they may present ambiguity and similarity among them. Three professionals followed the guideline adopted in previous study (PAIGE; MORIN, 2014) to identify ambiguous and similar assumptions. As a result of this analysis, we remove ambiguous assumptions and consider only one among the similar ones. Another threat is related to the small number of participants (P-Set) involved in our study. However, our P-Set is near the range considered as ideal in a Q-Methodology study: between 40 and 60 participants (WATTS; STENNER, 2012). Moreover, our P-Set is in accordance with our previous work (CARTAXO et al., 2019) that analyzes the developers' viewpoints regarding systematic reviews. Even aware of these facts, we mitigate this threat by involving developers with practical experience in issues related to bugs, as shown in Figure 3. Even so, one might argue that quasi-normal distribution is not appropriated to capture the participant's perceptions regarding the assumptions asked (e.g., one participant could have a more overall negative perception regarding code complexity). Therefore, this participant's perception (i.e., the Q-Sort), would more likely follow a negatively skewed curve. Although we concur, we believe the mitigation plan here is to have a comprehensive set of assumptions that cover a wide range of perceptions. Therefore, it could be hardly the case of participants do not agree, or agreeing to every assumption since many of them are nearly the opposite.

Conclusion validity. To perform our study, we use techniques, such as, Pearson's correlation, PCA, *Guttman Rule Guttman*, and cumulative explained variance, which have a high influence in the findings of our study. However, those techniques have been used by existing studies (CARTAXO et al., 2019; KELLY; MOHER; CLIFFORD, 2016) involving Q-Methodology. Moreover, we select those techniques by considering mathematically and

optimal choices instead of subjectivity measures.

Another threat to be considered is the reduction of viewpoints analyzed. Initially, we obtain eight viewpoints, but we analyze only five. In this case, we count with the Q-Methodology that support us to use qualitative criteria to make such reduction, as described in the Chapter 3. Researchers interested on investigating other solutions can analyze all the viewpoints available the online material (SOUZA, 2019).

Internal validity. Each developer is responsible for ranking 41 assumptions related to a variety of concerns. This number of assumptions may imply that some developers rank some assumptions without paying the required attention. To mitigate this threat, we allow the developers to interrupt and resume their analysis. Thus, they could do a subset of the analysis up to the point they feel tired. Later, they could resume the analysis when they feel prepared to continue.

External Validity. Although we have identified different viewpoints, they may not reflect the developers' beliefs in a broader way. However, Q-Methodology does not attempt to make a claim of universal relevance or to represent the views of a larger sample (BROWN, 1993). Moreover, the developers' comments to explain why they (dis)agree with an assumption represent complementary evidence, requiring further investigation to understand the developers' reasons to agree or not with an assumption.

8 Conclusions and Future Works

This chapter will draw conclusions based on the results, discussion, and contribution of our work. Lastly, some ideas for future work will be discussed.

We presented a study aimed at investigating which assumptions developers make when they review code to identify bug-introducing changes and developers' viewpoints regarding those assumptions. Moreover, we analyzed why a developer (dis)agrees with some assumptions and which ones are consensus/dissensus among developers. We made a study involving 41 developers analyzing 41 assumptions involving a diversity of concerns, such as comprehension, developer's experience and habits, programming language, maintenance, organizational, ownership, changes, and tests.

We identified five main developers' viewpoints. Some developers agreed that using automated testing (Viewpoint A), having a testing team (Viewpoint C), being aware of impacted artifacts (Viewpoint D), and being familiar with source code (Viewpoint E) are mandatory to avoid bugs. On the other hand, other developers strongly disagreed with the relevance of understanding the software history to avoid the introduction of bugs (Viewpoint A).

After analyzing the developers' comments, we observed that most of the developers' reasons to (dis)agree with the assumptions are based on his own experience, code complexity, short time delivery, development practices, management efforts and familiarity with the source code. Finally, we also observed that assumptions involving programming language, system compilation and execution, and a high number of features produce more consensus among developers than assumptions involving familiarity, software history, reuse, ownership and merge commits.

To the best of our knowledge, this is the first study to investigate developers' assumptions on identifying bug-introducing changes. Besides our findings, this study also contributes to guiding the implementation of future code review tools and techniques to assist developers to identify the introduction of bugs during the code review process. So, we call on researchers and practitioners (reviewers) to expand and improve upon work conducted in this study, either in the assumptions discussed in this work or others. In order to improve the relevance of research in the field and the quality of techniques, and produced code review tools in the community, it is necessary to gain a better understanding of developers assumptions during

software development and code review related tasks. As future work, we pretend to expand our study investigating if the assumptions collected from our study and developers' beliefs extracted from viewpoints occurs in open source repositories (i.e., in practice).

Bibliography

AKHTAR-DANESH, N.; BAUMANN, A.; CORDINGLEY, L. Q-methodology in nursing research: A promising method for the study of subjectivity. *Western Journal of Nursing Research*, v. 30, n. 6, p. 759–773, 2008. PMID: 18337548. Disponível em: <<https://doi.org/10.1177/0193945907312979>>.

ALBERTS, K. S.; ANKENMANN, B. Simulating pearson's and spearman's correlations in q-sorts using excel: A simulation proof of a widely believed result. *Social Science Computer Review*, v. 19, n. 2, p. 221–226, 2001. Disponível em: <<https://doi.org/10.1177/089443930101900208>>.

An, L. et al. Why did this reviewed code crash? an empirical study of mozilla firefox. In: *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. [S.l.: s.n.], 2018. p. 396–405. ISSN 2640-0715.

BACCHELLI, A.; BIRD, C. Expectations, outcomes, and challenges of modern code review. In: *Proceedings of the 2013 International Conference on Software Engineering*. Piscataway, N J, USA: IEEE Press, 2013. (ICSE '13), p. 712–721. ISBN 978-1-4673-3076-3. Disponível em: <<http://dl.acm.org/citation.cfm?id=2486788.2486882>>.

BAUM, T.; SCHNEIDER, K.; BACCHELLI, A. *Associating working memory capacity and code change ordering with code review performance*. [S.l.]: Empirical Software Engineering, 2019. ISBN 1066401896.

BAVOTA, G. et al. When does a refactoring induce bugs? an empirical study. In: *Proceedings of the 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. Washington, DC, USA: IEEE Computer Society, 2012. (SCAM '12), p. 104–113. ISBN 978-0-7695-4783-1.

BIRD, C. et al. Does distributed development affect software quality? an empirical case study of windows vista. In: *2009 IEEE 31st International Conference on Software Engineering*. [S.l.: s.n.], 2009. p. 518–528. ISSN 0270-5257.

BIRD, C. et al. Don't touch my code!: Examining the effects of ownership on software quality. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. New York, NY, USA: ACM, 2011. (ESEC/FSE '11), p. 4–14. ISBN 978-1-4503-0443-6. Disponível em: <<http://doi.acm.org/10.1145/2025113.2025119>>.

BORLE, N. C. et al. Analyzing the effects of test driven development in github. In: *Proceedings of the 40th International Conference on Software Engineering*. New York, NY, USA: ACM, 2018. (ICSE '18), p. 1062–1062. ISBN 978-1-4503-5638-1. Disponível em: <<http://doi.acm.org/10.1145/3180155.3182535>>.

BOYAPATI, C.; KHURSHID, S.; MARINOV, D. Korat: Automated testing based on java predicates. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 27, n. 4, p. 123–133, jul. 2002. ISSN 0163-5948. Disponível em: <<http://doi.acm.org/10.1145/566171.566191>>.

BROWN, S. R. *Political subjectivity: Applications of Q methodology in political science*. [S.l.]: Yale University Press, 1980.

BROWN, S. R. A primer on q methodology. *Operant subjectivity*, v. 16, n. 3/4, p. 91–138, 1993.

CARTAXO, B. et al. Software engineering research community viewpoints on rapid reviews. In: *Proceedings of the 13th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. [S.l.: s.n.], 2019. (ESEM '19).

CEDRIM, D. et al. Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: ACM, 2017. (ESEC/FSE 2017), p. 465–475. ISBN 978-1-4503-5105-8. Disponível em: <<http://doi.acm.org/10.1145/3106237.3106259>>.

CLELAND-HUANG, J. et al. Goal-centric traceability for managing non-functional requirements. In: *Proceedings of the 27th International Conference on Software Engineering*. New York, NY, USA: ACM, 2005. (ICSE '05), p. 362–371. ISBN 1-58113-963-2. Disponível em: <<http://doi.acm.org/10.1145/1062455.1062525>>.

COUTO, C. et al. Predicting software defects with causality tests. *Journal of Systems and Software*, v. 93, p. 24 – 41, 2014. ISSN 0164-1212. Disponível em: <<http://www.sciencedirect.com/science/article/pii/S0164121214000351>>.

CROSS, R. Exploring attitudes: The case for q methodology. *Health education research*, v. 20, p. 206–13, 05 2005.

DAO, T.; ZHANG, L.; MENG, N. How does execution information help with information-retrieval based bug localization? In: *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. [S.l.: s.n.], 2017. p. 241–250.

DEMUTH, A. et al. Introducing traceability and consistency checking for change impact analysis across engineering tools in an automation solution company: An experience report. In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.: s.n.], 2016. p. 529–538.

DEVANBU, P.; ZIMMERMANN, T.; BIRD, C. Belief & evidence in empirical software engineering. In: *Proceedings of the 38th International Conference on Software Engineering*. New York, NY, USA: ACM, 2016. (ICSE '16), p. 108–119. ISBN 978-1-4503-3900-1. Disponível em: <<http://doi.acm.org/10.1145/2884781.2884812>>.

EBERT, F.; CASTOR, F. A study on developers' perceptions about exception handling bugs. In: *2013 IEEE International Conference on Software Maintenance*. [S.l.: s.n.], 2013. p. 448–451. ISSN 1063-6773.

EBERT, F.; CASTOR, F.; SEREBRENIK, A. An exploratory study on exception handling bugs in java programs. *Journal of Systems and Software*, v. 106, p. 82 – 101, 2015. ISSN 0164-1212. Disponível em: <<http://www.sciencedirect.com/science/article/pii/S0164121215000862>>.

FEITELSON, D. G.; FRACHTENBERG, E.; BECK, K. L. Development and deployment at facebook. *IEEE Internet Computing*, v. 17, n. 4, p. 8–17, July 2013. ISSN 1089-7801.

FONTANA, F. A. et al. On evaluating the impact of the refactoring of architectural problems on software quality. In: *Proceedings of the Scientific Workshop Proceedings of XP2016*. New York, NY, USA: ACM, 2016. (XP '16 Workshops), p. 21:1–21:8. ISBN 978-1-4503-4134-9. Disponível em: <<http://doi.acm.org/10.1145/2962695.2962716>>.

FRITZ, T. et al. Degree-of-knowledge: Modeling a developer's knowledge of code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, v. 23, 03 2014.

FRITZ, T. et al. A degree-of-knowledge model to capture source code familiarity. In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. New York, NY, USA: ACM, 2010. (ICSE '10), p. 385–394. ISBN 978-1-60558-719-6. Disponível em: <<http://doi.acm.org/10.1145/1806799.1806856>>.

GUTTMAN, L. Some necessary conditions for common-factor analysis. *Psychometrika*, v. 19, n. 2, p. 149–161, Jun 1954. ISSN 1860-0980. Disponível em: <<https://doi.org/10.1007/BF02289162>>.

HANENBERG, S. Doubts about the positive impact of static type systems on programming tasks in single developer projects - an empirical study. In: *Proceedings of the 24th European Conference on Object-oriented Programming*. Berlin, Heidelberg: Springer-Verlag, 2010. (ECOOP'10), p. 300–303. ISBN 3-642-14106-4, 978-3-642-14106-5. Disponível em: <<http://dl.acm.org/citation.cfm?id=1883978.1883998>>.

HANENBERG, S. Doubts about the positive impact of static type systems on programming tasks in single developer projects - an empirical study. In: *Proceedings of the 24th European Conference on Object-oriented Programming*. Berlin, Heidelberg: Springer-Verlag, 2010. (ECOOP'10), p. 300–303. ISBN 3-642-14106-4, 978-3-642-14106-5. Disponível em: <<http://dl.acm.org/citation.cfm?id=1883978.1883998>>.

HANENBERG, S. An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. New York, NY, USA: ACM, 2010. (OOPSLA '10), p. 22–35. ISBN 978-1-4503-0203-6. Disponível em: <<http://doi.acm.org/10.1145/1869459.1869462>>.

HATA, H.; MIZUNO, O.; KIKUNO, T. Bug prediction based on fine-grained module histories. In: *Proceedings of the 34th International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2012. (ICSE '12), p. 200–210. ISBN 978-1-4673-1067-3. Disponível em: <<http://dl.acm.org/citation.cfm?id=2337223.2337247>>.

HODA, R.; NOBLE, J. Becoming agile: A grounded theory of agile transitions in practice. In: *Proceedings of the 39th International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2017. (ICSE '17), p. 141–151. ISBN 978-1-5386-3868-2. Disponível em: <<https://doi.org/10.1109/ICSE.2017.21>>.

JOHNSON, B. et al. Why don't software developers use static analysis tools to find bugs? In: *Proceedings of the 2013 International Conference on Software Engineering*.

Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 672–681. ISBN 978-1-4673-3076-3. Disponível em: <<http://dl.acm.org/citation.cfm?id=2486788.2486877>>.

KELLY, S. E.; MOHER, D.; CLIFFORD, T. J. Expediting evidence synthesis for healthcare decision-making: exploring attitudes and perceptions towards rapid reviews using q methodology. *PeerJ*, PeerJ Inc., v. 4, p. e2522, 2016.

KHOMH, F. et al. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, v. 17, n. 3, p. 243–275, ago. 2011. ISSN 1382-3256.

Kim, S.; Whitehead, Jr., E. J.; Zhang, Y. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, v. 34, n. 2, p. 181–196, March 2008. ISSN 0098-5589.

KIM, S. et al. Automatic identification of bug-introducing changes. In: *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2006. (ASE '06), p. 81–90. ISBN 0-7695-2579-2. Disponível em: <<http://dx.doi.org/10.1109/ASE.2006.23>>.

KOCHHAR, P. S. et al. Code coverage and postrelease defects: A large-scale study on open source projects. *IEEE Transactions on Reliability*, v. 66, n. 4, p. 1213–1228, Dec 2017. ISSN 0018-9529.

KONONENKO, O.; BAYSAL, O.; GODFREY, M. W. Code review quality: How developers see it. In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2016. p. 1028–1038. ISSN 1558-1225.

KONONENKO, O. et al. Investigating code review quality: Do people and participation matter? In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.: s.n.], 2015. p. 111–120.

Kourosfar, E. et al. A study on the role of software architecture in the evolution and quality of software. In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. [S.l.: s.n.], 2015. p. 246–257. ISSN 2160-1852.

LARSEN, R.; WARNE, R. T. Estimating confidence intervals for eigenvalues in exploratory factor analysis. *Behavior Research Methods*, v. 42, n. 3, p. 871–876, Aug 2010. Disponível em: <<https://doi.org/10.3758/BRM.42.3.871>>.

LESSENICH, O. et al. Renaming and shifted code in structured merging: Looking ahead for precision and performance. In: *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2017. (ASE 2017), p. 543–553. ISBN 978-1-5386-2684-9. Disponível em: <<http://dl.acm.org/citation.cfm?id=3155562.3155631>>.

LI, Z. et al. Cp-miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, v. 32, n. 3, p. 176–192, March 2006. ISSN 0098-5589.

LINARES-VÁSQUEZ, M. et al. Api change and fault proneness: A threat to the success of android apps. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: ACM, 2013. (ESEC/FSE 2013), p. 477–487. ISBN 978-1-4503-2237-9. Disponível em: <<http://doi.acm.org/10.1145/2491411.2491428>>.

LINARES-VÁSQUEZ, M. et al. How do developers document database usages in source code? (n). In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.: s.n.], 2015. p. 36–41.

MACLEOD, L. et al. Code reviewing in the trenches: Challenges and best practices. *IEEE Software*, v. 35, n. 4, p. 34–42, July 2018. ISSN 0740-7459.

MANTYLA, M. V.; LASSENIUS, C. What types of defects are really discovered in code reviews? *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 35, n. 3, p. 430–448, maio 2009. ISSN 0098-5589. Disponível em: <<http://dx.doi.org/10.1109/TSE.2008.71>>.

MARTINEZ, M.; DUCHIEN, L.; MONPERRUS, M. Automatically extracting instances of code change patterns with ast analysis. In: *2013 IEEE International Conference on Software Maintenance*. [S.l.: s.n.], 2013. p. 388–391. ISSN 1063-6773.

MATTHIES, C. et al. Attitudes, beliefs, and development data concerning agile software development practices. In: *Proceedings of the 41st International Conference on Software Engineering: Software Engineering Education and Training*. Piscataway, NJ, USA: IEEE Press, 2019. (ICSE-SEET '19), p. 158–169. Disponível em: <<https://doi.org/10.1109/ICSE-SEET.2019.00025>>.

MCINTOSH, S.; KAMEI, Y. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. *IEEE Transactions on Software Engineering*, v. 44, n. 5, p. 412–428, May 2018. ISSN 0098-5589.

MCINTOSH, S. et al. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, v. 21, n. 5, p. 2146–2189, Oct 2016. ISSN 1573-7616. Disponível em: <<https://doi.org/10.1007/s10664-015-9381-9>>.

MCKEOWN, B. F. *Q Methodology (Quantitative Applications in the Social Sciences)*. SAGE Publications, Inc, 2013. ISBN 1452242194. Disponível em: <<https://www.amazon.com/Methodology-Quantitative-Applications-Social-Sciences/dp/1452242194?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbóri05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=1452242194>>.

MEQDADI, O. et al. Towards understanding large-scale adaptive changes from version histories. In: *Proceedings of the 2013 IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2013. (ICSM '13), p. 416–419. ISBN 978-0-7695-4981-1. Disponível em: <<http://dx.doi.org/10.1109/ICSM.2013.61>>.

MEYER, A. N. et al. Software developers' perceptions of productivity. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2014. (FSE 2014), p. 19–29. ISBN 978-1-4503-3056-5. Disponível em: <<http://doi.acm.org/10.1145/2635868.2635892>>.

MEYER, A. N.; ZIMMERMANN, T.; FRITZ, T. Characterizing Software Developers by Perceptions of Productivity. *International Symposium on Empirical Software Engineering and Measurement*, v. 2017-Novem, n. November, p. 105–110, 2017. ISSN 19493789.

MISIRLI, A. T.; SHIHAB, E.; KAMEI, Y. Studying high impact fix-inducing changes. *Empirical Software Engineering*, v. 21, n. 2, p. 605–641, Apr 2016. ISSN 1573-7616. Disponível em: <<https://doi.org/10.1007/s10664-015-9370-z>>.

MOCKUS, A. Organizational volatility and its effects on software defects. In: *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2010. (FSE '10), p. 117–126. ISBN 978-1-60558-791-2. Disponível em: <<http://doi.acm.org/10.1145/1882291.1882311>>.

MONDAL, M.; ROY, C. K.; SCHNEIDER, K. A. Identifying code clones having high possibilities of containing bugs. In: *Proceedings of the 25th International Conference on Program Comprehension*. Piscataway, NJ, USA: IEEE Press, 2017. (ICPC '17), p. 99–109. ISBN 978-1-5386-0535-6. Disponível em: <<https://doi.org/10.1109/ICPC.2017.31>>.

OHIRA, M. et al. The impact of bug management patterns on bug fixing: A case study of eclipse projects. In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. [S.l.: s.n.], 2012. p. 264–273. ISSN 1063-6773.

OLIVA, G. A.; GEROSA, M. A. Chapter 11 - change coupling between software artifacts: Learning from past changes. In: BIRD, C.; MENZIES, T.; ZIMMERMANN, T. (Ed.). *The Art and Science of Analyzing Software Data*. Boston: Morgan Kaufmann, 2015. p. 285 – 323. ISBN 978-0-12-411519-4. Disponível em: <<http://www.sciencedirect.com/science/article/pii/B9780124115194000112>>.

OLIVEIRA, J. et al. Do android developers neglect error handling? a maintenance-centric study on the relationship between android abstractions and uncaught exceptions. *Journal of Systems and Software*, v. 136, p. 1 – 18, 2018. ISSN 0164-1212. Disponível em: <<http://www.sciencedirect.com/science/article/pii/S0164121217302558>>.

PADHYE, R.; MANI, S. K. K.; SINHA, V. Needfeed: Taming change notifications by modeling code relevance. *ASE 2014 - Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, p. 665–675, 09 2014.

PAIGE, J.; MORIN, K. Q-sample construction: A critical step for a q-methodological study. *Western journal of nursing research*, v. 38, 08 2014.

PASCARELLA, L. Information needs in contemporary code review. In: *CSCW 2018*. Washington, DC, USA: IEEE Computer Society, 2018. (CSCW '18).

PRECHELT, L.; SCHMEISKY, H.; ZIERIS, F. Quality experience: A grounded theory of successful agile projects without dedicated testers. In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2016. p. 1017–1027. ISSN 1558-1225.

RAHMAN, F.; DEVANBU, P. Ownership, experience and defects: A fine-grained study of authorship. In: *Proceedings of the 33rd International Conference on Software Engineering*.

New York, NY, USA: ACM, 2011. (ICSE '11), p. 491–500. ISBN 978-1-4503-0445-0. Disponível em: <<http://doi.acm.org/10.1145/1985793.1985860>>.

RAM, A. What makes a code change easier to review? an empirical investigation on code change reviewability. In: *26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. [S.l.: s.n.], 2018.

RAMLO, S. Centroid and theoretical rotation: Justification for their use in q methodology research. *Mid-Western Educational Researcher*, v. 28, n. 1, 2016.

RAY, B. et al. A large-scale study of programming languages and code quality in github. *Commun. ACM*, ACM, New York, NY, USA, v. 60, n. 10, p. 91–100, set. 2017. ISSN 0001-0782. Disponível em: <<http://doi.acm.org/10.1145/3126905>>.

SAE-LIM, N.; HAYASHI, S.; SAEKI, M. How do developers select and prioritize code smells? a preliminary study. In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.: s.n.], 2017. p. 484–488.

ŚLIWERSKI, J.; ZIMMERMANN, T.; ZELLER, A. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 30, n. 4, p. 1–5, maio 2005. ISSN 0163-5948. Disponível em: <<http://doi.acm.org/10.1145/1082983.1083147>>.

SMITH, E. K.; BIRD, C.; ZIMMERMANN, T. Beliefs, practices, and personalities of software engineers: A survey in a large software company. In: *Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering*. New York, NY, USA: ACM, 2016. (CHASE '16), p. 15–18. ISBN 978-1-4503-4155-4. Disponível em: <<http://doi.acm.org/10.1145/2897586.2897596>>.

SOUSA, L. et al. Identifying design problems in the source code: A grounded theory. In: *Proceedings of the 40th International Conference on Software Engineering*. New York, NY, USA: ACM, 2018. (ICSE '18), p. 921–931. ISBN 978-1-4503-5638-1. Disponível em: <<http://doi.acm.org/10.1145/3180155.3180239>>.

SOUZA, J. *Complementar Material*. 2019. Disponível em: <<https://sites.google.com/icufal.br/esem2019/>>.

SPADINI, D. et al. When testing meets code review: Why and how developers review tests. In: *Proceedings of the 40th International Conference on Software Engineering*. New York, NY, USA: ACM, 2018. (ICSE '18), p. 677–687. ISBN 978-1-4503-5638-1. Disponível em: <<http://doi.acm.org/10.1145/3180155.3180192>>.

STEPHENSON, W. Correlating persons instead of tests. *Journal of Personality*, v. 4, n. 1, p. 17–24, 1935. Disponível em: <<https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-6494.1935.tb02022.x>>.

TAO, Y. et al. How do software engineers understand code changes?: An exploratory study in industry. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. New York, NY, USA: ACM, 2012. (FSE '12), p. 51:1–51:11. ISBN 978-1-4503-1614-9. Disponível em: <<http://doi.acm.org/10.1145/2393596.2393656>>.

Thongtanunam, P. et al. Investigating code review practices in defective files: An empirical study of the qt system. In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. [S.l.: s.n.], 2015. p. 168–179. ISSN 2160-1852.

TUFANO, M. et al. An empirical study on developer-related factors characterizing fix-inducing commits. *Journal of Software: Evolution and Process*, v. 29, n. 1, p. e1797, 2017. E1797 JSME-15-0185.R2. Disponível em: <<https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1797>>.

Verner and J.; ; Babar, M. A. Software quality and agile methods. In: *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004*. [S.l.: s.n.], 2004. p. 520–525 vol.1. ISSN 0730-3157.

WATTS, S.; STENNER, P. *Doing Q Methodological Research: Theory, Method and Interpretation*. [S.l.]: Sage Publications, 2012. ISBN 9781849204149.

WHITTAKER, J. A.; ARBON, J.; CAROLLO, J. *How Google Tests Software*. 1st. ed. [S.l.]: Addison-Wesley Professional, 2012. ISBN 0321803027, 9780321803023.

WOHLIN, C. et al. *Experimentation in Software Engineering*. [S.l.]: Springer Publishing Company, Incorporated, 2012. ISBN 3642290434, 9783642290435.

YANG, Y. A brief introduction to q methodology. *Int. J. Adult Vocat. Educ. Technol.*, IGI Global, Hershey, PA, USA, v. 7, n. 2, p. 42–53, abr. 2016. ISSN 1947-8607. Disponível em: <<http://dx.doi.org/10.4018/IJAVET.2016040104>>.

YIN, Z. et al. How do fixes become bugs? In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. New York, NY, USA: ACM, 2011. (ESEC/FSE '11), p. 26–36. ISBN 978-1-4503-0443-6. Disponível em: <<http://doi.acm.org/10.1145/2025113.2025121>>.

ZABALA, A. qmethod: A package to explore human perspectives using q methodology. *The R Journal*, v. 6, n. 2, p. 163–173, 2014. Disponível em: <<https://journal.r-project.org/archive/2014-2/zabala.pdf>>.

ZABALA, A.; PASCUAL, U.; XIA, Y. Bootstrapping q methodology to improve the understanding of human perspectives. In: *PloS one*. [S.l.: s.n.], 2016. p. 1–19.

ZHANG, H.; GONG, L.; VERSTEEG, S. Predicting bug-fixing time: An empirical study of commercial software projects. In: *2013 35th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2013. p. 1042–1051. ISSN 0270-5257.

ZIMMERMANN, T.; NAGAPPAN, N.; ZELLER, A. Predicting bugs from history. In: _____. *Software Evolution*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 69–88. ISBN 978-3-540-76440-3. Disponível em: <https://doi.org/10.1007/978-3-540-76440-3_4>.

APPENDIX A – Supplementary Material

1 Complete List of Assumptions

Table 10 shown the entire list of assumptions collected from our study.

#	Assumptions
1	The comprehension of rationale behind changes is essential to avoid the bugs introduction.
2	If a change meets all non-functional requirements of an application, it is less prone to introduce bugs
3	Changes involving a large number of files lead to the introduction of bugs.
4	Changes involving a large number of features (fields, methods or classes) lead to the introduction of bugs
5	The presence of documentation associated with source code is essential for developers to avoid bug-introducing changes on it.
6	Familiarity of developers with source code avoids bug-introducing changes.
7	It is essential to know all co-changes in order to avoid introducing bugs.
8	When analyzing multiple changes, reviewers can better identify bug-introducing changes if they are strongly related because it avoids context switches.
9	By grouping related change parts together, we avoid context switches and reduce the cognitive load for the reviewers. As a consequence, a reviewer can better identify bug-introducing changes if they are grouped.
10	The introduction of bugs is inversely proportional to the complexity of the change. (bikeshedding).
11	The higher the number of test cases associated to source code, the lower the number of bug-introducing changes.
12	The lower the test coverage related to a change, the higher the number of bugs.
13	Parts of a change that lead test cases to fail are the main responsible for introducing bugs.
14	The presence of a testing team reduces considerably the bugs introduction.
15	The use of integration tests during a change aim test individual modules combined avoid bugs introduction.
16	Mutation Tests improves the quality of a test suite, avoiding the introduction of bugs.
17	Continuous Integration requires developers to integrate code into a shared repository several times a day. Each check-in is then verified by an automated build, allowing teams to detect problems early. Therefore, it can prevent a bug introduction.
18	Acceptance tests verify if the change meets the user requirements, and consequently avoids the bugs introduction.
19	It is always appropriate the presence of tests cases in a change to prevent bugs.
20	Changes that break the architectural design tend to introduce bugs.
21	High impact changes lead the introduction of bugs
22	The stability of a system can be measured by its sensibility to changes. A negative impact on the system stability leads to the introduction of bugs.
23	Architectural changes are the main responsible for the bugs introduction.
24	The introduction of bugs can be reduced by avoding code smells
25	Avoiding code smells reduces the introduction of bugs.
26	Introduction of bugs are decreased by avoiding recurring changes
27	Changes affecting old versions of a program influence directly the introduction of bugs.
28	The triage process aims at prioritizing specific changes based on its severity, frequency, and risk. Therefore, a triage process performed properly avoids the introduction of bugs.

#	Assumptions
29	Deviating from the architectural design of the system in a change may introduce bugs, which may be an unacceptable risk for some developers.
30	My confidence in accepting a change is impacted by compiling and running the code to prevent the introduction of bugs.
31	The use of continuous integration tools over the lifecycle of a software avoids the introduction of bugs.
32	Changes affecting end user experience lead to the introduction of bugs.
33	Bugs can be avoided by reviewing changes properly.
34	The higher the value of change to a customer (or client), the less bugs are introduced by them
35	The higher the value of change to a customer (or client), the higher the awareness by the developer, and less bugs are introduced by them.
36	The developer's experience is directly related to the introduction of bugs
37	The higher the developer's experience, the less he introduces bugs.
38	The familiarity of developers with the programming language adopted in a project is essential to avoid bugs.
39	The more consistent the reviewing process, the less bugs are introduced; the higher the effort of reviewers, the less bugs are introduced.
40	Changes recommended by the community introduce less bugs.
41	The older the source code, the higher the proneness of developers to introduce bugs.
42	The use of pair programming behind a change produce less introduction of bugs.
43	The more time of a developer in a project, higher is the familiarity with the code, and less changes will introduce bugs.
44	Source code frequently changed is a hotspot of bugs.
45	Understanding the software history is essential to developers avoid the introduction of bugs
46	When predicting changes, they can be fairly reviewed, avoiding the introduction of bugs.
47	Changes fairly reviewed avoid the introduction of bugs.
48	Performing changes equivalent to previous ones decrease the bug introduction.
49	Code reuse avoids the introduction of bugs
50	A proper exception handling avoids bugs.
51	Developers should use static analysis tools to avoid bugs.
52	A code break occurs when a change make previously working code stop working. Therefore, code breaks are hotspots of bugs.
53	Changes easily repaired can be performed without concerning with bugs introduction.
54	Changes to deal with compiler's error messages do not introduce bugs.
55	Knowing the source code history related to a change is essential to avoid bugs.
56	Knowing the professionals involved in a change is essential to identify if it introduces bugs.
57	If a source code already had bugs previously, changes on it are more propeness to introduce bugs.
58	The higher the traceability of changes, the less the propeness of a developer introduce bugs.
59	Refactoring changes are less prone to introduce bugs.
60	Adaptive changes aims at migrating systems to a new version of a third party API. Therefore, adaptive changes usually introduce bugs.
61	Migrating systems to a new version of a third party API usually introduce bugs.
62	Code changes impacting unrelated code components are more likely to introduce bugs.
63	A set of changes for completing two or more tasks lead to the introduction of bugs.
64	The features referenced by changes are the main responsible for introducing bugs
65	The introduction of bugs are reduced by avoiding code clones.
66	The changes that do not satisfy the requirements are bugs hotspots.

#	Assumptions
67	When people make different changes in the same file, it leads to a bug introduction.
68	It is always appropriate the presence of tests cases in a change to prevent bugs.
69	Deviating from the architectural design of the system in a change may introduce bugs, which may be an unacceptable risk for some developers.
70	To avoid the introduction of bugs, it is always appropriate to conduct the test cases when performing a change.
71	There is a risk of introduction of bugs involved in accepting changes that we do not yet understand.
72	My confidence in accepting a change is impacted by compiling and running the code to prevent the introduction of bugs.
73	I tend to do a lighter/faster review when the change is performed by a senior teammate due the less introduction of bugs.
74	I put more confidence to avoid the introduction of bugs running the functional tests related to these changes.
75	The more time of a developer in a project, higher is the familiarity with the code, and less changes will introduce bugs.
76	The analysis of history about the changes performed can be timely and valid to avoid the introduction of bugs.
77	A change with code smells cannot be approved to avoid the introduction of bugs.
78	The lack of third-package updates is prone to introducing bugs.
79	The use of continuous integration tools over the lifecycle of a software avoids the introduction of bugs.
80	The introduction of bugs is inversely proportional to the complexity of the change. (bikeshedding).
81	The technical debt of developers behind a change may introduce bugs.
82	The presence of commit-level comments during changes improves the comprehension of code, and avoid the introduction of bugs.
83	The use of pair programming behind a change produce less introduction of bugs.
84	The use of integration tests during a change to test individual modules combined avoid bugs introduction.
85	The higher is the quality of a test suite in a change, the lower is the number of bugs introduced.
86	Mutation Tests improves the quality of a test suite. Therefore, mutation tests avoid the bug introduction.
87	Continuous Integration requires developers to integrate code into a shared repository several times a day. Each check-in is then verified by an automated build, allowing teams to detect problems early. Therefore, it can prevent a bug introduction.
88	Acceptance tests check if the change respects the user needs. Therefore it prevents rework and consequently avoids the bug introduction.
89	The change did not introduce bug if the application is running properly.
90	Bikeshedding spends a disproportionate amount of time and effort in an unimportant detail of a system. Therefore, it leads to the introduction of bugs in complex parts of the system.

Table 10 – List of Assumptions

2 Q-Sort Matrix

Table 11 present a two-dimensional matrix obtained from the 41 participants and assumptions. The value of each cell in this matrix is the level of agreement (or disagreement), e.g. +3 or -3, attributed by participant to the assumption in the Q-Sort.

Participant's ID	Assumptions										
	A1	A2	A3	A4	A5	A6	A7	A8	...	A40	A41
#1	0	-1	0	2	-1	0	1	0	...	1	0
#2	0	-2	0	2	0	2	2	-2	...	-2	-1
#3	0	0	0	0	0	3	2	0	...	-1	-1
#4	1	-1	-1	1	2	1	1	2	...	-2	0
#5	3	0	1	3	0	1	2	0	...	-1	1
#6	-1	-1	0	1	2	-1	1	-1	...	-1	0
#7	0	1	-1	0	-1	0	-2	0	...	1	0
#8	2	-1	1	3	2	0	2	1	...	0	1
...
#40	1	1	-1	0	1	-3	2	2	...	-1	1
#41	1	-2	-2	2	-1	1	1	1	...	0	0

Table 11 – Part of the Initial Data Matrix

3 Correlation Matrix

Table 12 shown a correlation matrix between participants (q-sorts) and assumptions. We use the *Pearson's* correlation coefficient where a perfect positive correlation is registered as 1, and a perfect negative correlation is -1.

Participant's ID	Assumptions										
	A1	A2	A3	A4	A5	A6	A7	A8	...	A40	A41
#1	1	0.33	0.26	0.32	0.43	0.39	-0.01	0.16	...	0.15	0.41
#2	0.33	1	0.29	0.32	0.28	0.36	0.34	0.15	...	0.32	0.20
#3	0.26	0.29	1	0.27	0.45	0.26	0.08	0.20	...	0.11	0.32
#4	0.32	0.32	0.27	1	0.39	0.35	0.07	0.25	...	0.11	0.37
#5	0.43	0.28	0.45	0.39	1	0.13	0.08	0.26	...	0.02	0.45
#6	0.39	0.36	0.26	0.35	0.13	1	0.32	0.21	...	0.30	0.52
#7	-0.01	0.34	0.08	0.07	0.08	0.32	1	-0.05	...	0.36	0.30
#8	0.16	0.15	0.20	0.25	0.26	0.21	-0.05	1	...	0.50	0.34
...
#40	0.41	0.20	0.32	0.37	0.45	0.52	0.30	0.34	...	1	-0.22
#41	-0.04	-0.17	-0.30	-0.29	-0.30	-0.27	-0.15	-0.26	...	-0.22	1

Table 12 – Correlations between Q-sorts

APPENDIX B – Characterization Questionnaire

Characterization Questionnaire

During a software development did you notice any changes that introduced bugs? If yes, how did you perceive the buggy code?

Please enter the country you currently work on.*

Please select your educational level.*

- PhD
- PhD Student
- Master
- Master Student
- Undergraduate
- Undergraduate Student
- Other

What is your current job? (You can select multiple options)?*

- Open Source Developer
- Academic Researcher
- Industrial Developer
- Industrial Researcher
- Other

Please rate your expertise with*

	Very poor	Poor	Fair	High	Very High
Software Development	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Code Review	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Bug Reporting	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Bug Fixing	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Tests	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Have you ever contributed for open source projects? If yes, could you provide the links of these projects?

Please enter your e-mail (it will not be disclosed), if you want to participate on the raffle of US\$ 100 voucher at Amazon.com or want be contacted in future researches.

Do you have any comments regarding this research?

Submit

Back