Abstract

Cognitive biases appear during code review. They significantly impact the creation of feedback and how the feedback is interpreted by the author. These biases can lead to illogical reasoning and decision making. However, code reviews rely heavily on accurate and objective code evaluation. This article explores harmful cases due to cognitive bias during code review, as well as potential solutions to avoid such cases or mitigate their effects. Therefore, we developed several prototypes covering confirmation bias and decision fatigue. They were developed by conducting usability tests and validated with a user experience questionnaire accompanied by participants' feedback about the developed techniques. It was shown that some techniques are well accepted to be used by reviewers and help preventing behavior that is detrimental to code review. This work provides a solid first approach to treat cognitive bias in code review.

Keywords. Cognitive bias, Tool-assisted code review, User centered design

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This work is about the relationship between two quite different domains. On one hand it treats cognitive biases which are investigated in psychology. On the other hand it treats code reviews which are applied in software engineering. Software engineering activities do not only consist in applying technical knowledge. These activities are heavily influenced by cognitive and social aspects which are often neglected. In this research we focus on the trigger and the impact of cognitive biases during code review. The first goal is to find cases in which cognitive biases appear. The second goal is to design solutions to either avoid the bias or to mitigate its effects.

There exist many different kinds of cognitive biases. They can be classified by using a taxonomy. For this work we focus on the triggers of *confirmation bias* and on the effects of *decision fatigue*. Both cognitive biases appear during code review. But each one of them is tackled differently. The objective of this thesis is the creation of a concrete solution that can be tested with reviewers to see whether the solution is accepted or not by the users. Here, tests focus on the user experience in terms of the visual layout.

As described before, in a first phase we explore potentially harmful situations due to cognitive biases and design theoretical solutions to prevent or mitigate the bias. In a second phase we aim to improve the designed solutions by conducting usability tests with students. These tests serve as feedback to gather the requirements of the users for an acceptable solution. To achieve this, we iterate multiple times over the prototypes. During this process we develop a prototype that the users can actually work with so that feedback is based on a realistic context. In this second phase we also conduct an evaluation of the prototype's final result by testing the user experience with a questionnaire (UEQ). This questionnaire provides quantitative data which is used to support the verbal feedback from the tests.

We start the testing with a base prototype. While iterating over the prototypes, we introduce new techniques as we interpret the participants' feedback as well as their behavior. Techniques that do not yield expected behavior are removed or improved. After development, the final result is evaluated on the basis of the feedback and the questionnaire. Here we mostly take into account qualitative data, the feedback, in order to make conclusions about the user experience. A rather small sample size is used, nonetheless we use that data to confirm the user's tendencies. Those tendencies are taken from the means of their answers and a benchmark of the user experience test.

With the prototypes as final result and an evaluation of their usability, we propose a first piece of work on solving problematic relationships between cognitive bias and code review.

Chapter 2

State of the art

This chapter is designed as a necessary introduction to help the reader understand the topics that are going to be developed in the next chapter. Basic concepts of cognitive biases as well as how code reviews are conducted are explained, so that the reader can better comprehend the more complex concepts approached in the next chapter Design; including how different aspects of cognitive biases influence code reviews and what techniques can be applied to mitigate their effects. Here we first go over the basic knowledge required to understand what cognitive biases are. Then we explore where cognitive biases appear in software engineering. Outside the context of software engineering we explore how the cognitive biases get triggered and what effects they have. And to prepare a technique that is discussed later, we will see what constructive feedback is.

2.1 Cognitive bias in psychology

Cognitive biases and fallacies "In psychology cognitive biases are cases in which human cognition produces reliable representations, that are systematically distorted compared to objective reality". In order to judge situations in a fast and accurate way, human processing applies different types of biases. Multiple reasons can trigger the occurrence of cognitive biases [14]. Cognitive biases are not to be mistaken for logical fallacies even though they appear similar. Biases are patterns of thinking that affect how we interpret new information and processes. They are applied systematically and influence our behavior, opinions and the decisions we make. On the other hand logical fallacies occur during arguments. They are arguments that are based on invalid conclusions. Contrary to cognitive bias, we sometimes apply them on purpose or by accident.

Heuristics When time and resources for processing are limited, the brain uses shortcuts or rules of thumb to solve a problem or judge a situation. However, the so-called heuristics are prone to break in some cases [14]. "People rely on a limited number of heuristic principles, which reduce the complex task of assessing probabilities and predicting values to simpler judgmental operations" [31]. According to a study by Arkes, "The extra effort required to use a strategy is a cost that often outweighs the potential benefit of enhanced accuracy" [1].

A concrete example of using heuristics is the judgment about others, depending on their position in the social hierarchy. People with a higher position tend to apply stereotypical views on others more often, than those who occupy a more precarious position. The latter group of people has to invest more time and energy in social judgment [14].

Artifacts In other cases the mind uses artifacts in order to solve tasks that the mind was not designed for, such as tasks involving probabilities or abstract rules of logic. Firstly, the human brain is better at making predictions with information presented in frequencies, than when it has to compute probabilities. This behavior is due to the frequencies being observable in nature, for example the number of times an event has occurred in a given time period. Probabilities, on the other hand, are a more abstract concept. Therefore, using frequencies allows the brain to better estimate the likelihood of events. Secondly, when a problem's content is of abstract or logic nature, then humans solve it by applying problem-solving mechanisms developed for recurring problems. This means that humans are not particularly good at solving such problems, rather they kind of cheat their way to the answer through an approach that is adapted to a specific set of problems [14].

Error management Further, according to Haselton et al. (2015), unlike optimal mechanisms which do not make any errors, cognitive mechanisms can produce false positives and false negatives. For example, taking an action that would have been better not to take; or failing to take an action that would have been better to take. The cost of making a certain decision error is not always the same. So, depending on the inconvenience an erroneous decision brings, a greater error can be avoided. For example, when "fleeing from an area that contains no predator results in a small inconvenience cost, but it is much less costly than the failure to flee from a predator that really is close by". The error management theory shows that "a bias towards making the less costly error will evolve. This is because it is better to make more errors overall as long as they are relatively cheap" [14].

Thinking, fast and slow In section 2.4 we will discuss antecedents for certain cognitive biases. However, it is quite challenging to identify their causes. The causes are based on heuristics. These principles, as defined earlier by Tversky and Kahneman (1974), can be analyzed further [31]. So, Kahneman also started differentiating the way of thinking into either fast or slow. Respectively, the difference between the two is that conscious, effortful and more rational thoughts (called system two thinking) are less prone to cognitive bias than unconscious, effortless, intuitive thoughts (called system one thinking). However, as explained before, cognitive biases can be triggered by other factors than "fast thinking" [16].

Debiasing Next to the knowledge about cognitive biases in general, how they are triggered and what effects they have, we can find approaches in the literature, that aim to eliminate certain biases. This is called debiasing. It is shown that neither applying more effort, nor being more experienced in a field helps mitigate cognitive biases. On the other hand, being trained on how cognitive biases function and applying specific techniques can make a substantial difference. This has been proven not only for experts in the field but also to have effects on the judgment by non-experts. Also, Fischoff demonstrates that aids like training are most efficiently developed when the process of the aimed cognitive bias is well understood [12].

2.2 Cognitive biases in software engineering

This section contains findings from existing literature about cognitive and social elements interfering with software engineering. We first explain the importance of the, sometimes neglected, cognitive aspects in software engineering. This gives a brief understanding about the vast space of possible impacts on software development activities. Next we see what role a reviewer's personality plays in software engineering and how the kinds of cognitive biases can be classified.

Social aspects In the study "Measuring cognitive activity in software engineering", Robillard et al. (1998) want to understand the processes behind software development by observing professionals during their activity. The goal is to derive good practices from an empirical analysis of their subjects. For this purpose analysis strategies derived from cognitive science are adapted to the context of software engineering. In general, software development processes are measured in terms of code properties or the time spent for a certain activity. However, cognitive aspects are not much investigated, even though software development processes are rich in information about cognitive activity [24]. Software development is sometimes done individually, sometimes in a team. As for peer reviews, the activity relies on the author and at least one reviewer, but it sometimes it relies on multiple reviewers. Such activities require a lot of interaction between multiple participants, which in turn require strategies to ensure an efficient and unbiased procedure. Empirical studies like in Robillard et al. demonstrate a clear interest in the analysis of cognitive and social aspects during the different software development activities.

Reviewer's personality In the study (Barroso et al., 2017) a personality's influence on tasks in software engineering is investigated. The reason is that especially during team oriented tasks, personality shapes the group and makes for a huge difference when talking about how efficiently a software product is produced. Thus the quality of the product depends on the interaction between members of the team combined with their professional capabilities. To understand these factors, relationships between the participant's personality and their professional activity can be evaluated [2]. These factors are tied closely to the human's cognitive activity, which in turn influences interpretation and how a developer reacts to feedback for example. Therefore, cognitive aspects play an important role when observing software development processes. The fact that the cognitive field of code reviews is currently less investigated, offers an opportunity to gather more knowledge on how to improve this crucial part of software development; especially in regards to cognitive issues.

A taxonomy of cognitive biases Even though cognitive research is sometimes not taken into account in information technology, some research investigates the topic. Different types of cognitive biases are analyzed. So, Fleischmann et al. went to create a taxonomy about cognitive biases in software engineering. Fleischmann et al. conducted the study on what research already exists about cognitive biases in information systems (IS). Results are that some biases appear more often than others in research. Based on the findings he/she could analyze the different categories of biases that exist. Those categories allow to classify biases focused on pattern recognition, memory, decision-making, stability, social aspects and interest. The most investigated cognitive biases are framing and anchoring bias [13]. The defined categories according Fleischman et al. (2014) are summarized in Table 2.1; n indicating the number of investigations of biases of a certain category in research.

Category	**Biases**
Perception biases (n=40)	framing, negativity bias, halo effect, ...
Pattern recognition biases (n=11)	confirmation bias, availability bias, reasoning by analogy, ...
Memory biases (n=1)	reference point dependency
Decision biases (n=24)	irrational escalation, cognitive dissonance, input bias, ...
Action-oriented biases (n=9)	overconfidence bias, optimism bias
Stability biases (n=24)	anchoring, sunk cost bias, loss aversion, ...
Social biases (n=9)	herding, stereotype, value bias, ...
Interest biases (n=2)	after-purchase rationalization, self-justification

Table 2.1: Categorization of biases.

2.3 Software inspection and code reviews

In this section we explain with what process we are working. Cognitive bias can happen anywhere. In this work however we will focus on modern code review. First we explain how companies used to search for defects by applying the software inspection process. Then we address the more modern and also less formal way of analyzing code today. This section also serves as introduction to inform the reader about terms including code review, author, reviewer and feedback.

2.3.1 Software inspection process

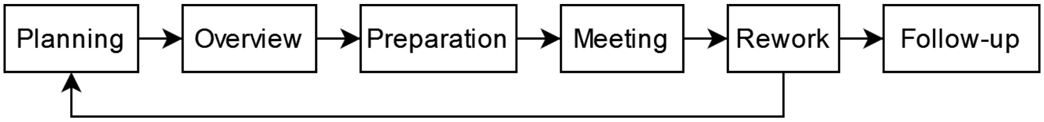


Figure 2.1: Software inspection process as defined by Fagan.

The management of every process requires planning, measurement and control. So does the process of software development. In order to improve software quality and increase the productivity of developers, IBM created the software inspection process, which got later refined by Fagan in 1974. This is a process which allows the developers to follow a clear structure and achieve a higher defect detection efficiency [10]. So, thanks to a completely defined approach, that gets applied routinely, errors in the result can be prevented. This is achieved with several phases that the review process goes through. First, we go through three stages, from planning and individual preparation of the documents to collective examination searching for defects. Then we have the last two stages, which are applying the changes and having a follow-up to verify that the recommendations were implemented correctly. However, even though the benefits are remarkable, the whole process is very intensive in its application. Every step is formally defined, takes a lot of time to walk through and requires multiple people to inspect a huge piece of software if not the complete project.

2.3.2 Modern code review

The disadvantages of formal software inspection shall be resolved with modern code review. Such reviews are more light-weight; they avoid using strict protocols, meetings, guidelines or using checklists like in the formal software inspection process. Modern code review got introduced by Google. The main goal was however, not to detect defects more easily, but rather to make it easier to understand new code, by checking its readability. Like this, others can comprehend the author's intentions and follow this thought-process. It is important to make sure to respect the company's norms such as formatting and the technologies that are used. Code reviews also serve as gatekeepers to prevent developers from committing arbitrary code without verification. This aspect improves security [25].

During a modern code review, often a single reviewer is sufficient. This is possible, as in comparison to the software inspection process, an informal code review is done more often and therefore applied on much smaller changes. This way, less time is needed to go over a change and defects can be detected earlier. Nevertheless, multiple reviewers can be taken into code review.

Today, modern code review can generally be summarized with the following steps:

Creation First the author creates, modifies or removes documents.

Review request After modification, the author inspects the changes he/she made and as soon as he/she is ready, notifies the reviewer about them. This happens either manually, per mail for example, or by being assisted by tools like GitHub, Gerrit or Microsoft TFS.

Reviewer inspection Then the reviewer analyses the changes, mostly using tools to see the difference between current and previous versions. While doing this, the reviewer makes comments to the individual changes if necessary. As soon as the reviewer is done, he/she notifies the author about the feedback.

Rework The author then either applies the suggestions from feedback or comments back by engaging in a discussion about the comment with the reviewer.

Approval When all comments have been addressed by the author, he/she awaits approval from the reviewer in order to finally commit.

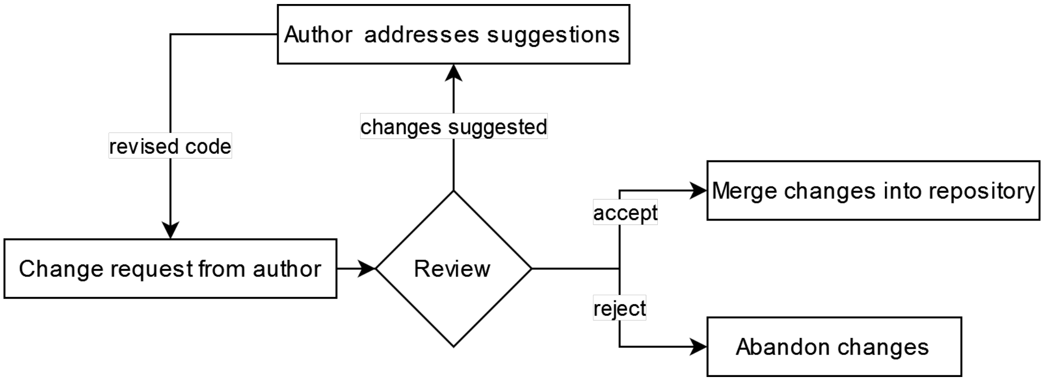

Author addresses suggestions
revised code
changes suggested
Merge changes into repository
accept
Change request from author
Review
reject
Abandon changes

Figure 2.2: Modern code review process.

2.4 Cognitive biases, their triggers and effects

In the following paragraphs we describe some cognitive biases happening in software engineering. In the chapter Design we do not investigate all of them, but it is important to know which kind of cognitive biases exist and which ones are most present in software engineering and identified in current literature. Therefore, we focus on certain biases that allow us to get familiar with the different aspects impacted by cognitive biases. Afterwards, we focus on a smaller selection in order to provide solutions that deal with specifically chosen aspects.

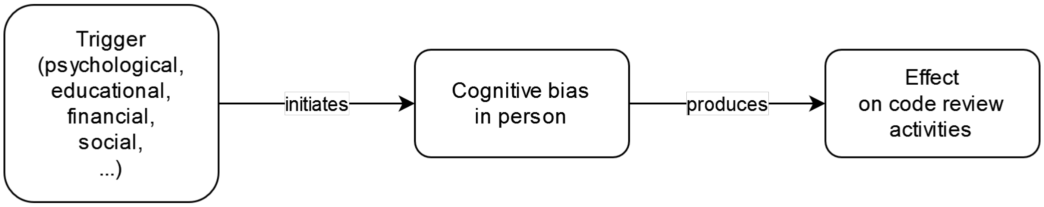


Figure 2.3: Relationship between triggers, cognitive biases and their effects

When we get to know a cognitive bias, we are interested in the triggers initiating it and what impact the bias has on the investigated activity. In this work a trigger is considered a specific condition in the environment, that can be of social, educational, financial, or psychological nature. Such elements can create an environment for certain cognitive biases to happen. One cognitive bias can potentially be triggered by multiple elements, not only one. When such a cognitive bias is triggered, it affects the person's activity. In our case, code reviews can be impacted severely by cognitive biases. One cognitive bias in turn can produce multiple effects during the different code review activities.

2.4.1 Overconfidence bias

As stated by Panko et al. (2014) [21] humans have in general a tendency to be over confident about tasks such as driving a car, business activities or fighting. This phenomenon, labeled as overconfidence bias, relates to how we assess the accuracy of our stored knowledge and perceptual models. It was hypothesized and thereafter proven, that overconfidence can lead to a significant reduction of error rates by getting feedback to one's behavior. Moores et al. (2009) explain this with an increase in self-efficacy when accomplishing a task and being rewarded through a feedback-loop. The feedback affects the future behavior. People have a tendency to seek confirmation rather than non-confirmation evidence for their actions. So, they overly positive self-evaluate. With greater self-efficacy people set higher goals for themselves, which in turn allows them to better cope with obstacles and thus deliver a greater performance [20]. However, other studies suggest that a too high estimation of self-efficacy is tied to bad performance. A too high estimation of self-efficacy creates a lack of motivation with decreased effort. Therefore, self-efficacy produces positive effects, as long as its estimation does not exceed the actual performance level of the person in question. In the latter case, performance actually decreases [32].

2.4.2 Confirmation bias

One of the most researched cognitive biases in psychology is the confirmation bias. Currently there exists a lot of knowledge about confirmation bias in psychology, however there is not much concerning software engineering. When talking about confirmation bias, one refers to the collection, interpretation, analysis and research for information in a way that confirms one's prior beliefs instead of searching for information disproving them. In practice, once the mind adopts an opinion, it does everything to support it, no matter how many counter-arguments to that very opinion can be found. All other factors are simply neglected in order to keep the point of view. Such behavior leads to wrong decisions that defy the sense of logical reasoning [15]. The impact of confirmation bias is severe, as stated in multiple studies. For example, confirmation bias leads to fixation on a certain tool, even when there are other tools suggested through evidence as being better supported [23]. In other cases software professionals show a biased behavior when searching for information in documents. Confirmation bias makes people only search as far as to confirm their hypotheses. Further research is neglected, which in turn creates incomplete knowledge about the element in question and misleading results [8]. This however, is a crucial factor during code review, which will be discussed later. According to McHugh et al. (2003) [9], confirmation bias is present during individual work, but it also concerns group work. As group work constitutes an important element of code reviews, it is necessary to investigate implications of confirmation bias during code review. As mentioned before, cognitive aspects play a significant role in software engineering activities. Through these means, cognitive biases are part of code reviews.

A well known example is the positive test bias happening among test developers. Based on confirmation bias, this behavior leads to tests that only confirm the code, instead of disproving it [30]. Tests are more effective with data, which is designed to disconfirm the hypothesis [19]. In general, and not only during tests, one's goal should be to fail the code in order to reduce defect density [5].

The factors triggering confirmation bias in software engineering tasks are already under investigation, because not only technical factors can be taken into account when analyzing those activities. Calikli et al. (2010) [6] searched for issues due to the company culture, education and professional experience. For this, multiple groups were constituted and compared by applying tests, such as the rule discovery task, developed by Wason et al. (1960) [34]. The findings are that neither education nor professional experience are factors that significantly contribute to confirmation bias. On the other hand, the subjects from a professional background, used to a tighter schedule, were more prone to confirmation bias than students. Students do not have the same perception of pressure for releases because they are not confronted with such a context yet. A similar discovery was made about the influence of time pressure on developers. However, in this empirical study no significant impact could be measured for time pressure as factor; nonetheless confirmation bias is strongly present in software engineering in general [26]. Even when no significant results were found for a specific factor, studies suggest a deeper investigation about confirmation bias in software engineering.

2.4.3 Decision fatigue

During a day thousands of decisions are made. This requires a significant amount of internal resources to process information constantly and make a decision. Similar to muscle depletion due to repeated use and loss of energy, the repeated act of making decisions results in the depletion of internal resources, which is called ego depletion by Baumeister (1998) [3]. Ego depletion in itself is rather a higher construct which can manifest as decision fatigue, with the consequences of attentional deficit and impulsive decisions. Another effect appearing with decision fatigue is postponing a decision with the intention to look at it later [7]. Also, people subjected to decision fatigue tend to have an impaired ability in making trade-offs; they prefer acting in a passive role and make irrational judgments. Unfortunately such changes in behavior are hard to recognize [22].

Stewart et al. (2012) conducted a study on how decision fatigue impacts the acceptance rate of manuscripts submitted to the Annals of Neurology journal. Those manuscripts are peer-reviewed when submitted and as a result accepted or rejected. With this study the effect of decision fatigue should be demonstrated. Having assigned more review tasks was supposed to provoke decision fatigue. It was demonstrated that the more reviews an editor got assigned, the more submissions he/she refused. Even though the difference with those getting assigned less reviews is small, the study still shows a significantly more severe evaluation by the editor. Also, the review process consists of multiple checkpoints that a review has to pass. This makes sure that even with a potentially biased editor, a submission is evaluated in a fair manner [29]. Therefore, using multiple instances to review a document seems to be suggested to fight the effects of decision fatigue. This is a hypothesis that can be used for code reviews too, as code reviews use the same principles.

Code review is essentially a task of deciding whether a change is accepted or not. Therefore it is crucial to avoid biased decisions. Knowing when this effect appears helps avoid it and so provide better results in code reviews. As mentioned before, decision fatigue appears when internal resources are depleted. The experience of decision fatigue depends on the time of day as shown by Sievertsen et al. (2016) during a study [28]. As the day progresses, moral decisions become harder for people active in healthcare and judgment. Specific reasons are sleep-deprivation, happening for example to nurses asked to work for long hours, more than they are capable of. This thus provokes illogical decisions due to fatigue. But also being tired after lunch can be a reason triggering such phenomena [22].

2.5 Constructive feedback

Concerning influencing people, feedback is one of the most important interpersonal skills that you can develop, because it is an integral part of communication. But making constructive feedback is not as obvious. In this section we discuss what makes for a great feedback and what does not.

Why giving feedback? Giving feedback is the opportunity to provide the recipient with information about behavior and performance. First, in order to maintain a positive attitude towards themselves and their work. Second to encourage the recipients to move towards agreed goals [4].

When to give feedback? It is recommended to let the recipient know as soon as possible, what about the work is good and what needs to be changed. Otherwise, when waiting too long, you run the risk of confronting the recipient with too much negativity at once. On top of that, when too much time has passed, the consequence can be that it is already too late to apply the feedback. Also, a factor to consider is if the recipient is ready to receive feedback or not. Does he/she have the time right now and can he/she handle it. The same way, the feedback creator must ask himself/herself if he/she is capable of making constructive feedback [4].

Constructive vs non-constructive feedback On the one hand, constructive feedback is characterized by either positive expressions, reinforcing good performance or negative expressions, improving poor performance.

On the other hand, problems can arise with non-constructive feedback. Such feedback lacks positive expressions and goes without any recognition or affirmation. Negative feedback can become destructive criticism [4].

In order to provide constructive feedback, the following recommendations help according to Waggoner Denton (2018) [33] and Bee et al. (1998) [4].

- The person giving feedback should identify a specific problem.
- The identified problem should be accompanied by an explanation regarding why it is a problem.
- A concrete solution to solve the problem should be provided. This can be done with examples. They make for an easier comprehension.
- It should be specific, clearly expressing the objective.
- The identified problem should be achievable by yourself, the reviewer.
- The feedback should be expressed in a positive way. For example, say *"Provide more details please"* instead of criticizing by saying *"There are too few details"*.
- The feedback should be accompanied by the expected experience when achieving the objectives agreed upon. Such an image of *what you will hear and see when achieving the objectives* motivates people to actually achieve the objectives. Sportspeople, for example, imagine what it would be like to win.
- Getting the feedback sooner than later is beneficial to the author. Getting it sooner, allows for smaller corrections and thus results in less negative input at once.
- Feedback that helped the reviewer in a similar position can be transmitted the same way to the person whose artifact is being reviewed. It is recommended to think about how you got impacted by an earlier feedback, and then react with the gained knowledge in the position as a reviewer.
- It is important to also recognize when a job is well done to ignite a sense of achievement, which in turn unlocks the potential for growth and development.

Chapter 3

Design

In the first part of this work, we discussed the existing knowledge concerning cognitive bias in software engineering and modern code-reviews as well as how certain cognitive biases happen and what effects they have in general. This knowledge serves as a basis to start investigating both topics further; there will be a specific focus on what the effects on code review are. The contribution of this research lies in the development of solutions used to avoid cognitive biases or to circumvent their effects. The effects on code review need to be described and understood in detail. This is necessary to find techniques that deal with the cause beneath the bias or its impact.

One needs to differentiate between mitigating a cognitive bias by circumventing the trigger and mitigating the effects that a bias provokes. Both ways of handling are useful. However, one can be more appropriate than the other depending on the context where the bias takes place.

In the first section the triggers and effects of biases are adapted to code review activities. Potential cases where cognitive biases occur are described in detail. This concerns cases that decrease the quality of code reviews. In the third section we describe potential solutions for confirmation bias and decision fatigue; there are solutions that are supposed to improve the code review process and increase the result's quality. In the last section the methodology is described. This includes how the addresses cognitive biases were selected, as well as how the impact they have on code review was defined. Also, the way of searching for solutions is described. Part of the methodology is validation of the techniques with users, which is based on a user-centered approach. After that, we explain why prototypes serve perfectly as a solution to validate their design.

3.1 Impact on code review

Problems In the previous chapter State of the art, we already discussed the triggers initiating certain biases and the effects they create. The description of the biases was done from a purely psychological perspective. Now, we approach an explanation with a focus on the code review side of those biases. The goal is to find out how cognitive biases affect code review. However, there is not much literature available about this particular relationship. Therefore the objective is to fill that research gap by providing potential cases in which effects of cognitive bias can be found. First such cases are developed, based on the functioning of biases as found in literature. These cases are adapted to scenarios arising during modern code review. Then, solutions are designed with prototypes, which subsequently are tested with users to verify whether the users consider the solutions as an improvement for code review or not.

In this first section we look at the triggers and effects of confirmation bias and decision fatigue. Here we choose cases where a trigger is the most intuitive to create scenarios. This means that one can develop scenarios easily when using these triggers. Some triggers are hard to understand and difficult to reproduce with a scenario. Therefore we limit ourselves to a subset of all available triggers. This is also due to the limited time available for this work. One could have dug even further into the science of triggers and effects of biases; nonetheless, those that are explored here are a solid base that can be extended with future research. The same approach was applied to the research of cognitive bias effects.

3.1.1 Confirmation bias

Like in the literature about decision fatigue, psychology investigated confirmation bias a lot, providing a solid basis to research its relations to software engineering, especially to modern code review. Here too, many triggers as well as effects can be discovered. Yet we will focus only on some of them.

Triggers Factors initiating confirmation bias during code review.

- The author receives non-constructive feedback, hurting his/her self-esteem.

- Time-pressure on the reviewer under certain circumstances.

Effects The impact visible after the mentioned factors have triggered confirmation bias during code review.

- The author refuses recommendations from the feedback in order to protect his/her self-esteem.
- The reviewer tries to validate the existing code instead of analyzing it objectively.

Now we describe in detail what consequences some cases provoke and how they are in relation with factors that can trigger it during some activities.

Refusing the code review feedback

Sometimes we tend to hold on to already existing beliefs, in order to protect our self-esteem. Changing our beliefs sometimes means changing our values, which is not as easy as keeping them. This can hurt our self-esteem because it suggests that we might lack intelligence for example. As a result we often look for information that confirms our beliefs instead of rejecting them [17].

Following this mechanism in code reviews the same can happen. When the author develops code, he/she puts much effort and time into it. This work makes him/her a confident believer of the correctness of his/her code. However, when working on complex tasks such as developing software, mistakes happen even to the most experienced developers. Fortunately code reviews allow us to view results from a different perspective than the author. Thus, errors and misconceptions can be detected by reviewers.

When the reviewer detects such mistakes, he/she comments about it by asking the author to justify a decision or to adapt according to the reviewer's suggestion. Feedback like this can be interpreted negatively by the author. It is possible that he/she takes the advice like criticism, instead of using it's constructive value. Further effects are a hurt self-esteem, because the author might think he/she is perceived as lacking intelligence for the task he/she worked on. As a result, the author of the code will search for counterarguments to defend the solution he/she built. Even though the suggestions given by the reviewer are valid, in order to protect his/her self-esteem, the author does not accept applying changes to the current code.

The assumption made here is that the way a reviewer builds the feedback has an influence on the author's perception and therefore his/her acceptance of the feedback.

The reviewer tries to confirm the code

When writing tests for software, the goal should always be to fail the code. However, software developers and testers are more likely to create positive tests rather than negative ones due to the phenomenon called confirmation bias [5]. Similarly, when a reviewer gets to see code changes, he/she is already exposed to code in front of him/her, influencing his/her perception during review. This could lead to searching for arguments that confirm the way the author programmed. Such behavior is even more likely to appear under time-pressure. As described for decision fatigue, the same trigger applies for confirmation bias [26].

For example, at the end of a day of work or shortly before a release date, a higher pressure on developers and reviewers arises. A tendency to merely confirm code would replace an objective analysis of the code.

The hypothesis states that under time-pressure a reviewer develops the tendency to search for fast review approval, instead of searching for correct implementation.

3.1.2 Decision fatigue

Many triggers that can initiate decision fatigue were found in psychology literature. So, in some cases inexperience can trigger this bias. In others, the cause can be the time of day during which the person is dependent on cognitive capability and yet experiences a loss in decision accuracy. Those causes reflected on code review can provoke multiple side effects.

Triggers Factors initiating decision fatigue during code review.

- Inexperience in the field of the currently investigated code.
- Doing code reviews at times of day known for decreased internal resources.
- Doing code reviews while being in a state of less energy.
- Getting assigned a too important number of code reviews.
- Spending too much time on details to achieve perfection.

Effects The visible impact after the mentioned factors have triggered decision fatigue during code review.

- Skipping code changes of the review because of specific properties (too small, too big or too complicated).
- Missing motivation to do code reviews and postponing them for later.
- Making impulsive comments instead of constructive suggestions for the author.

Now we describe in detail what consequences some cases provoke. As there are many possible relationships between the causes and their effects, we are not going to explain all of them.

Skipping code changes of the review For this case the hypothesis states that the reviewer is subjected to decision fatigue because he/she might do reviews at a certain time of day. Some times are less recommended for doing tasks requiring high attention and thus requiring much of the internal resources [28]. Such times are generally when the person is in a state of fatigue; often provoked by previous activities, such as having eaten just before or having completed an intensive task accompanied with making many decisions. Studies were made on how repeated decision making impacts the state of people [3]. These studies reveal that a person might become overwhelmed and mentally exhausted from making too many decisions. In the case of code reviews, a developer too has to make many decisions during a day when it comes to making software and analyzing code. Most of the work a developer does, is reading and analyzing code he/she does not know.

Now, if a reviewer is subjected to decision fatigue for the reasons explained above, he/she might handle a review differently than if he/she was in a non-impacted, clear state of mind. The hypothesis made here is that a reviewer then has the tendency to skip reviewing changes with specific characteristics, such as very small changes, significantly big changes, changes of a less known topic or changes that are complicated to understand. However, if the reviewer skips certain parts of the code, he/she unavoidably misses information. And we are referring here to missed information that is crucial to understand the whole of the code and to comprehend the though-process of the author of the code.

Understanding the code is key to making constructive comments for the author. Thus, based on this hypothesis, a review under the impact of decision fatigue might provoke misleading results in the feedback for the author, simply because not all elements are taken into account during the code review.

Postponing code reviews

As said before, code review is an energy intensive task. The reviewer needs motivation to tackle new code or potentially new topics. Humans have a tendency to do tasks where they are rewarded early. So, specifically small tasks correspond to this desire. This behavior is called hyperbolic discounting [18]. However, in code review it can happen that a reviewer gets assigned a great amount of reviews to do. The sheer number of reviews in front of him/her, can discourage the person from even starting. This behavior can be observed when the person is subjected to decision fatigue. It leads people to postpone tasks for later which is also called procrastinating.

In this case, the reason that decision fatigue gets triggered, might be tied to certain conditions of the company. For instance, just before the release of a big project, many review requests can pop up and overwhelm the assigned reviewer. Another case occurs at the end of the a working day. It can for example happen when a developer decides to do all the reviews at once at a certain time of the day or even of the week. He/She risks accumulating a great number of reviews waiting for him/her to be completed. It is similar to the previously described effect. The time of day has a great influence on the results of the work.

So the hypothesis arises that postponing code reviews is the effect happening when decision fatigue is triggered due to circumstances that are unfavorable for starting tasks intensive in cognitive resource.

Impulsive comments

Impulsive comments from the reviewer can be due to decision fatigue, not allowing the reviewer to evaluate the code objectively. As a result, comments are expressed in a familiar way. Such comments can be destructive even though feedback should always be constructive for the author. This behavior might be initiated when doing code review at certain times of the day, like close to the end of a day of work or when being tired just after lunch.

3.2 Mitigating the impact on code review

Solutions As explained in the first chapter, cognitive biases are surrounded by their triggers and effects. In this section, we are interested in how we can treat the problem of biases during code review. Now it is important to understand that a solution can either deal with the trigger initiating the bias or it can try to mitigate the provoked effects. Both ways of solving the problems can be useful. But depending on the problem one might correspond to a better way of doing so than the other one.

In this section several cases are described with a proposed solution. A case is characterized by the following elements. A case starts with a trigger that initiates a certain cognitive bias as explored in the State of the art chapter. This bias provokes potentially multiple effects on the activities of code review. For such cases, we provide a solution; a technique to solve the case's problem.

Now we discover several proposed solutions for decision fatigue and confirmation bias. A summary of the described cases with the associated solution is in Table 3.1.

3.2.1 Scheduled reviews

As described in the sections before, it happens that a reviewer gets assigned a great amount of reviews. If the reviewer perceives that the number of tasks waiting for him/her is too high, he/she can loose motivation to actually start working on them. This phenomenon is called decision fatigue. That cognitive bias being triggered, the reviewer has a tendency to postpone tackling his/her list of assigned reviews.

In order to prevent the trigger so that the cognitive bias does not even happen, the amount of reviews per day should be decreased or at least a maximum of reviews per day should not be exceeded. When a manager or author is about to assign a developer to a review, he/she can not know how overworked the developer in question is. For this information to be available an intermediate step is recommended, showing the scheduled reviews of the developer. An option is to provide an intermediate such as a calendar combined with the possibility for the developer to indicate a maximum amount of reviews per day. This way the developer makes sure that he/she will not be assigned too much work unknowingly by the manager or author, leading to less or no decision fatigue as long as the limits are respected.

This step however comes with possible limitations too. The reviewer has to know himself/herself very well, in order to indicate a relevant maximum number. Also, not every company has so many reviewers, to choose another one in the case of overwork.

3.2.2 Warning when the needed time is exceeded

Like in the previous case, it happens that a reviewer gets to deal with too many review tasks. Decision fatigue leads the reviewer to skip analyzing seemingly complicated or big changes because such reviews require a significant mental investment. In the state of mental resource depletion, also called ego-depletion [3], the quality of reviews declines. But reviews are in fact a tool used to ensure qualitative code changes. Therefore it is not recommended to work on decision intensive tasks when being in such a state.

In order to prevent triggering this bias in the first place, one must pay attention to the current work efficiency of an employee. Judging oneself is difficult, when even detecting such phenomena from the outside is not an easy task. But the employee can be reminded at certain times that he/she might not be efficient enough at the moment. The solution is to measure the time a developer requires for undertaking a review. Learning the usual duration needed, helps get an idea of when a reviewer takes too much time and thus might be in a state of decision fatigue. Once the usually required time is significantly exceeded, the reviewer is warned. He/She can then decide how he/she reacts to the warning by either taking a break or halting the review altogether.

This solution comes with the side effect of potentially postponing reviews. As explained in the previous solution, this is considered as an effect due to decision fatigue. However, a combination of both solutions brings an optimal way of handling the trigger.

3.2.3 Find an expert for unfamiliar topics

Every developer knows some fields better than other fields. A developer, who is comfortable with every topic is rare or does not even exist. A limiting factor is therefore the experience a developer can use to make a qualitative code review. For example, knowing edge cases of specific domains requires experience to analyze code changes with a trained eye. Not having this experience forces the reviewer to either not verify code changes in a precise way, or to gain the experience on the spot by searching and learning about the topic. The latter however leads to a greater investment than usual, which in turn provokes decision fatigue and skipping code changes that the reviewer is not familiar with.

It is therefore recommended to assign a developer who is in fact familiar with the topic of the code in question. The goal is to find an expert or at least the most fitting developer to do the code review. This way additional research by the developer is unnecessary and thus eliminates decision fatigue. The solution is to present the developers available for review with a list of familiar topics next to them. This allows the author or manager who chooses the reviewer to find the best fitting one.

One limitation is to always find an expert, similar to the first solution to decision fatigue, because not every company has that many developers specialized in a certain field. Also, a reviewer gains experience when he/she is not familiar with a certain topic and has to do research. This knowledge is lost when choosing someone else, who is familiar with the topic. However, a crucial stage like code review is used to ensure quality and not especially to provide developers with additional experience.

3.2.4 The author guides the reviewer

Multiple factors can trigger decision fatigue. In this case we take into account the time of day at which code reviews are done. This factor occurs in many varieties such as doing code review in the morning as a first task, after lunch or just before the end of the working day as the last task. Depending on the context and the amount of work that has been put in, the state of the developer is different. For example just before the end of the work day, most employees have used up much of their internal resources. This, as described before, leads to decision fatigue and thus potentially skipping big or complex code changes. No matter what triggers decision fatigue, the impact is the same. Knowing that most of a developer's work is reading and understanding code, we search for a solution that supports the reviewer when he/she experiences decision fatigue. Here, the author could help out by providing an explanation about the code change. With such help the reviewer does not have to identify the reason for the change before actually evaluating it. The idea is to provide an interface where the author can add a description about specific parts in the code change, but only to the main changes so that the reviewer understands them. The goal is not to provide a description to every changed line of code, but rather to the more complicated parts. The way the author provides a description resembles the way a reviewer writes feedback. The reviewer then can choose a guided review to see annotations above some code changes. By going from one explanation to the next, the author guides the reviewer through the main parts so that he/she gets an idea of how the code was designed. After following the guide, the reviewer can start giving feedback to the individual code changes.

This solution is a significant additional step for the author between writing the code and committing it. Writing explanations seems very useful to the reviewer, but requires the author to make an additional investment.

3.2.5 Help make a complete comment

Not only can a developer be inexperienced in a certain field, but he/she can also be inexperienced in code review altogether. A student who has recently graduated or a developer who has never done code review can be unfamiliar with the concepts or simply miss the practice of the theoretical concepts. Again, inexperience forces a developer to do research about the task he/she wants to accomplish. This requires additional investment, leading to decision fatigue. The effect that is observed here, is a decline of the quality of comments. In the state of decision fatigue comments become illogical, without proper justification. Comments are made impulsively without questioning much their relevance.

In order to prevent such effects, the reviewer can be guided when making a comment. For developers who are inexperienced in code review, a form provides sufficient guidance, allowing the developer to answer questions like, “What have you identified?”, “Why is this a problem” and “Provide multiple alternative solutions to the problem!”. This way the important aspects of a comment are answered, however, if a reviewer is left alone, with a single field to make a comment, the provided comment might miss information necessary for the author to understand the issue and implement the correction adequately.

On the other hand, most reviewers are not new to the task of reviewing. They are used to their way of commenting. Such a form guided review is only useful to novices. Alternatively, such advice can also be taught through training courses.

3.2.6 Advice to make constructive feedback

The result of code review is the feedback that the author receives. Such feedback is crucial to become aware of issues one was unable to see. As a developer one spends much time solving one and the same problem. This allows errors to slip in because our view can be distorted from constant input of the same idea. Therefore techniques such as code review exist, to allow discovering the code from a different perspective. As much as a different perspective can make issues visible, it can also resemble questioning the author's way of thought, his/her decisions and even his/her skills. A developer's self-esteem is tied to the perception of his/her capabilities and relevancy of his/her decisions, how the author is perceived by himself/herself as well as how he/she is perceived by others. As the author develops code, he/she is accompanied by confidence about the code's validity.

However, when feedback criticizes the author's decision, his/her self-esteem can be undermined, which in turn leads to a bad perception of the feedback. Discussing about mistakes made in the code should be done in an objective manner. Having hurt the author's self-esteem hinders such a discussion. It occurs that the feedback is not accepted, that provided suggestions are refused. On the contrary, in such a situation the author might be prone to confirmation bias, searching for arguments to refuse feedback comments and searching for arguments proving the validity of the existing code. This problem shows the importance of developing a good feedback, providing grounds for an objective discussion between author and reviewer. Creating good feedback is not a innocuous task. It requires paying attention to details, detecting relevant issues and formulating comments in a constructive way, which has to be accompanied by an objective code analysis. It is expected that authors are less prone to confirmation bias when given constructive feedback.

A concrete solution can be provided by instructing the reviewer on how to write constructive feedback. Every time the reviewer is about to create a comment a reminder of best practices goes a long way. The challenge is to incite the reviewer many times to pay attention to the advice. Learning how to provide constructive comments is a skill that takes time, but it is worth it because it will yield great results in the long term.

3.2.7 Asking others for feedback about the review

In the previous case we discussed how to circumvent confirmation bias due to non-constructive feedback. A first solution is providing the reviewer with advice to create constructive feedback. Sometimes we miss a point of view, which leads to illogical reasoning and incomplete feedback. In order to prevent the cognitive bias from happening in the first place, it is recommended to question the discovered issues. This can be done by discussing the points of view with other developers. They can evaluate the feedback of the reviewer from an objective stand point. This procedure resembles a smaller review of the review itself. This way the feedback must first pass a checkpoint before getting to the author.

However, also this solution comes with a drawback. Adding a verification step slows down the review process and requires additional work force.

3.2.8 Encourage brainstorming and complete feedback

In software engineering tasks become intensive quite fast. Especially when nearing a deadline, time becomes scarce and developers tend to reason less accurately. Then the reviewer will be prone to confirmation bias. Instead of analyzing the code objectively, time-pressure pushes him/her to accept code as proposed by the author. He/She solely confirms the code changes, whereas he/she should evaluate them objectively. This provokes feedback that is less thought trough. A great problem with non-constructive feedback is the lack of alternatives that help the author understand how to solve the discovered issue. It is favorable for the author when the reviewer provides not only one alternative, but multiple ones.

So, the goal is to encourage the reviewer to brainstorm on how to solve the problem. Similar to the previously described form that guides the reviewer, now we offer multiple fields destined to contain different alternative solutions described by the reviewer. On the one hand, a placeholder for alternatives reminds the reviewer to think about providing alternative solutions, on the other hand he/she is incited to give more than one alternative. This way it can be made sure that the reviewer responds at least to this aspect of constructive feedback. As a reminder, many more aspects have to be respected as described in the State of the art chapter.

To go further, review tools could provide a more detailed form, destined to guide the reviewer especially in cases where time-pressure or decision fatigue occurs. Having a form at hand helps the reviewer not to fail to respond to important aspects of feedback.

3.3 Methodology

Now that the problems have been identified and that the first solutions are described, implementation of the latter shall prove their usability. In the following chapter we are going to test some of the solutions. The chosen methodology for the development of the solutions and testing, is the user centered design process. But first we are going to focus on what that process looks like. Then we are going to explain why prototyping is the way to present solutions to the future user and in the next chapter we are going to disclose the results of the testing as well as the evolution of the techniques.

3.3.1 User-centered design process

The user centered design process guides development with an iterative approach. That means requirements for the user are analyzed, a solution is designed, presented and finally corrected according to the needs of the user. It involves the users throughout the design process. The goal is to understand the user's tasks and how to solve his/her problems. Following this process we go through five phases:

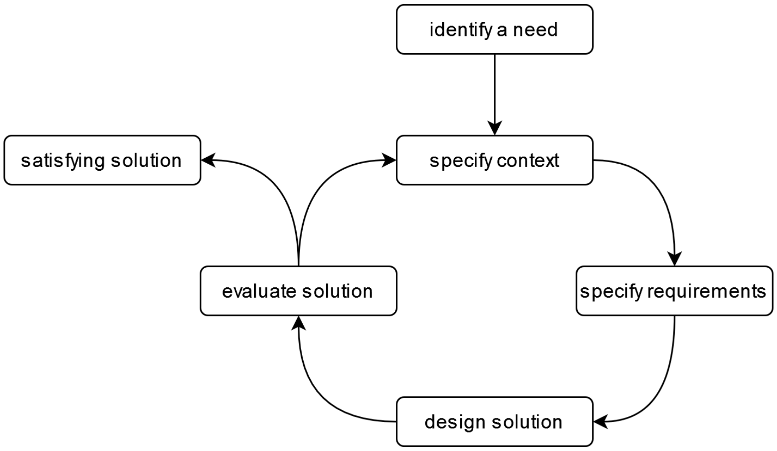


Figure 3.1: Phases of the user centered process.

1. Specifying the context. Define the user's profile, who is going to use the product and under what conditions that will be.

2. Specifying the user's requirements. What are his/her goals and how he/she is trying to achieve them.

3. Creating the solution. In this case solutions will be presented as prototypes, so that the user can express his/her opinion about it.

4. Evaluating the solution. Testing the product shows if the prototype is working. In this case we will conduct usability tests with the users.

5. Accept the solution. If the evaluation yields satisfying results, the solution can be accepted. Otherwise the design process restarts with specifying the context.

3.3.2 Prototyping

Objective is to test the designed solutions with the user. therefore we need a way to show the user a concrete, usable medium. The chosen medium for the prototypes is a simple HTML document. The reason is clear; because the solutions are destined to be implemented in already existing tools, such as Gerrit. Gerrit is a tool to facilitate the code review process along with GitHub. Gerrit is already developed as a web application. This makes the choice of HTML as technology relevant. Nonetheless there exist tools that generate dynamic prototypes, and that could have been considered too. However based on HTML, the generation of prototypes is independent of mentioned tools. Additionally, HTML provides a great flexibility in terms of design using CSS.

3.3.3 Usability test

In the last phase of the user centered design process, the solution needs to be tested with the users. Here we use an implementation of the user centered design process, called Usability testing. Usually, multiple participants are asked to complete a task while observers watch, listen and take notes about the user's behavior as well as about what the user says about the prototype during navigation. Such tests help identify problems, identify if the user is satisfied with the solution and how long it takes him/her to complete a task.

This is useful as finding problems early reduces complications compared to discovering them only later. With an iterative approach the path of the solution can be corrected step by step, instead of having to invest into large adaptations afterwards like with a Waterfall approach.

3.3.4 UEQ

For the final step of this research, we want to analyze whether the new techniques serve as an improvement, do not change anything or even have a negative impact. Previously we explained usability testing to develop solutions that correspond to the user's needs and respond to their requirements for specific tasks. Now we use user experience testing (UEQ) to verify the solution's relevancy, especially in terms of the user interface. Testing like this allows making data-backed decisions on what feature enables an optimized experience. UEQ testing can also be used to understand the user's experience, their pain points and satisfaction with the features.

3.3.5 Development process

In Figure 3.2 we get an overview of the applied development process. What starts with the identification of triggers and effects for cognitive biases during code review, continues with the design of theoretical solutions to the problems. These solutions require testing to improve according to the needs of the future users. Once the development is finished, the resulting tool must be verified in terms of design. This last step uses a different group than the one used for design. In the last step the collected feedback is evaluated in order to make conclusions about the developed prototypes.

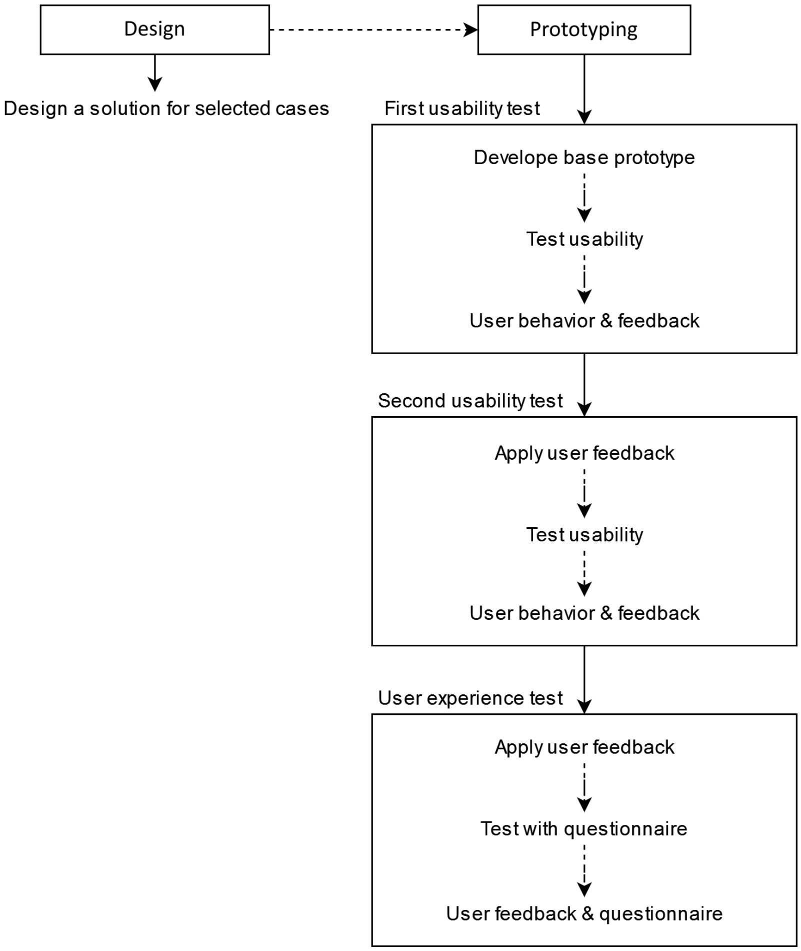


Figure 3.2: Design and prototyping of the techniques.

Problem	**Solution**
The reviewer is over-solicited, which triggers decision fatigue.	*Scheduled reviews.* Prevent the bias by limiting the amount of reviews with a maximum number and a calendar to schedule.
The reviewer is over-solicited, which triggers decision fatigue.	*Observe needed time.* Prevent the bias by reminding the reviewer to halt when too much time is needed for review.
Missing experience about a specific topic in the code triggers decision fatigue.	*Find an expert.* Prevent the bias by assigning the best fitting reviewer according his experience in the topic.
When decision fatigue is triggered from working at a bad time of day, the reviewer is prone to skipping big or complicated changes.	*Guide with comments.* Mitigate the impact by guiding the reviewer through the files with comments made by the author.
Being inexperienced in making code reviews leads the reviewer to impulsive/illogical comments due decision fatigue.	*Help commenting.* Mitigate the impact by providing a form with key words to help the reviewer to include all essential elements.
Feedback of bad quality makes the author defend his/her self-esteem due to confirmation bias, thus refusing suggestions and searching for arguments to leave the code unchanged.	*Constructive feedback.* Prevent the bias by providing the reviewer with advice about how to make constructive feedback. *Feedback of review.* Prevent the bias by suggesting to the reviewer to ask another developer for feedback about his/her review.
Under time-pressure reviewers are prone to decision fatigue and make incomplete comments.	*Encourage brainstorming.* Mitigate the impact by providing a form with empty solution fields to encourage the reviewer to think about multiple solutions.

Table 3.1: Summary of described cases with the associated solution.

Chapter 4

Prototyping

In the first chapter, State of the art, we explore the theoretical space of cognitive biases, their triggers and effects. This knowledge leads to the selection of certain possible cases applicable in code review, that are discussed in the third chapter, Design, with the goal to design potential solutions in order to circumvent the triggers or mitigate the effects. In this chapter we select some of the discussed solutions with the goal to test those techniques with participants, in order to explore their efficiency and whether they actually work. The techniques that are tested here, are developed as HTML prototype and then tested; first doing two iterations of usability testing, and second applying the user experience questionnaire. While evaluating the prototypes with a usability test, we use two groups of participants serving a different purpose. We employ a total of eight participants. This is supposed to help us find on average more than 80% of the problems during an evaluation according to Faulkner (2003) [11]. The first group is active during the development of the prototypes. A first version is presented to them in order to execute a requested task and give feedback. After the first iteration, according to the notes and feedback from the participants, modifications are made to better respond to the user's needs as well as to make the solution more efficient. We proceed the same way with a second iteration. The second group is active during the last phase, which is destined to evaluate the usefulness of the solutions. The purpose of a second group is to avoid misleading success due to biased feedback from the first group, as the participants help develop the tools. The second group is not part of the usability test. With the second group we use a User Experience Questionnaire as described in the Methodology section.

As a reminder, we describe the three selected cases.

Case 1 In the first case we focus on the trigger of confirmation bias. Confirmation bias can be triggered when a person feels the need to protect her/his self esteem. This happens during code review as well. A non-constructive feedback can lead the author to ignore suggestions on reception of the feedback. Further, he/she tries to defend his/her code with arguments that are potentially illogical, even if the proposed comments from the reviewer are relevant.

The goal is to avoid the cognitive bias to be triggered in the first place. The solution is to develop a technique to advise the reviewer during review in order to produce constructive feedback.

Case 2 In this case we do not look at the triggers; instead we focus on the effects of the cognitive bias. For several reasons decision fatigue might appear during code review, leading to incomplete comments and to situations where the reviewer takes a passive role.

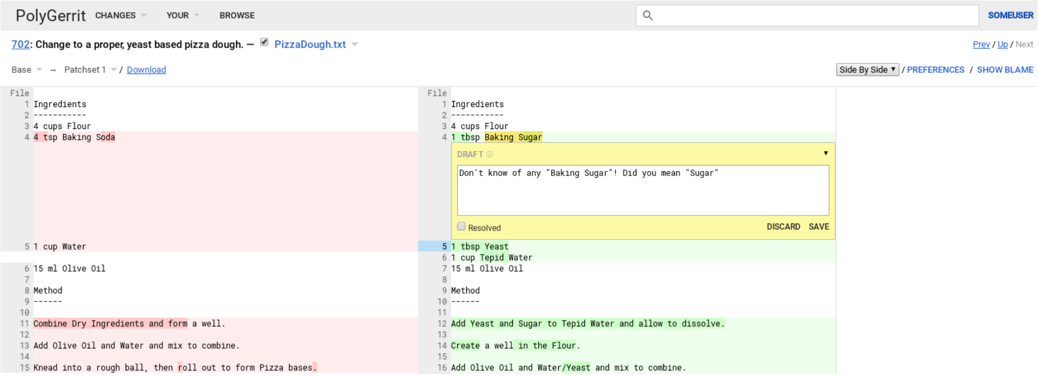
The goal is to mitigate the effects of decision fatigue. The solution is to develop a technique to help the reviewer make a complete comment even though he/she is in a state of mental fatigue. This could be achieved by providing form fields that the reviewer only has to fill out, making sure that the comment does not miss any important information.

Case 3 In this case we also focus on the effect of decision fatigue. One of them is the temptation of skipping code changes that appear big or complicated. This in turn leads to misunderstanding and decreased quality of the code review.

The goal is to prevent the reviewer from skipping important changes and thus help understanding why specific changes were made. The solution is to develop a technique that guides the reviewer with comments through the most important changes before starting the actual review. This way the reviewer gets informed about the reason for modifications, skipping is prevented and understanding the code requires less analysis from the reviewer.

4.1 Base prototype

Developing a technique requires time and goes through multiple steps until a presentable prototype is available. To begin, brainstorming over elements for a potential technique is done. The ideas are inspired by the solutions discussed in the previous chapter (summary in Table3.1). Later the ideas that we came up with and we find useful, get mixed together and adapted. This process already involves creating sketches of the final solution. At this stage, a first prototype can be developed by using HTML to see what the solution would look like if really implemented. However, these prototypes do not reflect a fully working tool by any means. But this is not the goal of such a prototype. Here, solely the idea of the solution is represented so that the user can imagine how the technique would function during code reviews.


PolyGerrit
CHANGES
YOUR
BROWSE
SOMEUSER
702: Change to a proper, yeast based pizza dough. — ☑ PizzaDough.txt
Prev / Up / Next
Base → Patchset 1 / Download
Side By Side ▾ / PREFERENCES / SHOW BLAME
File
File
DRAFT
☐ Resolved
DISCARD
SAVE

```
Ingredients
-----------
4 cups Flour
4 tsp Baking Soda
1 cup Water
15 ml Olive Oil

Method
------

Combine Dry Ingredients and form a well.

Add Olive Oil and Water and mix to combine.

Knead into a rough ball, then roll out to form Pizza bases.
```

```
Ingredients
-----------
4 cups Flour
1 tbsp Baking Sugar
1 tbsp Yeast
1 cup Tepid Water
15 ml Olive Oil

Method
------

Add Yeast and Sugar to Tepid Water and allow to dissolve.

Create a well in the Flour.

Add Olive Oil and Water/Yeast and mix to combine.
```

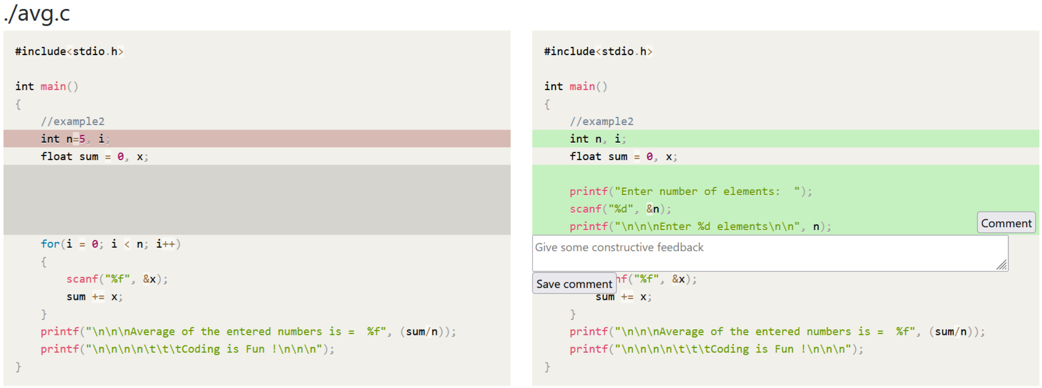
Don't know of any "Baking Sugar"! Did you mean "Sugar"

Figure 4.1: Gerrit quick review line to make a comment.

The prototypes are based on the web tool Gerrit which can be seen in Figure 4.1. First we create a base prototype without applying techniques, as seen in Figure 4.2.

In the following paragraphs three techniques are described how they function. For each case one technique is developed respectively. Technique 1 for case 1, technique 2 for case 2 and technique 3 for case 3. In the First iteration Section 4.2.2, describing the feedback and changes, we also show what the prototypes look like.

Technique 1 – Advice The goal being to help the reviewer create constructive feedback, the first technique is based on advice displayed in the form of a list. This list can be opened by clicking the button *I need advice.*



./avg.c

```
#include<stdio.h>

int main()
{
    //example2
    int n=5, i;
    float sum = 0, x;




    for(i = 0; i < n; i++)
    {
        scanf("%f", &x);
        sum += x;
    }
    printf("\n\n\nAverage of the entered numbers is =  %f", (sum/n));
    printf("\n\n\n\n\t\t\tCoding is Fun !\n\n\n");
}
```

```
#include<stdio.h>

int main()
{
    //example2
    int n, i;
    float sum = 0, x;

    printf("Enter number of elements:  ");
    scanf("%d", &n);
    printf("\n\n\nEnter %d elements\n\n", n);
Give some constructive feedback
          nf("%f", &x);
        sum += x;
    }
    printf("\n\n\nAverage of the entered numbers is =  %f", (sum/n));
    printf("\n\n\n\n\t\t\tCoding is Fun !\n\n\n");
}
```

Figure 4.2: Base prototype without techniques.

Following this action, a popup shows a list containing the advice. The advice originates from literature investigating how to achieve constructive feedback [4, 33].

Technique 2 – Form Because under decision fatigue reviewers tend to make incomplete comments, the idea of a pre-structured form seems an appropriate choice. This way, the reviewer does not have to think about how to structure the comment. It should include the identification of a problem, a justification regarding why the discovery is considered a problem and also a suggestion to solve it. Here, three empty fields are available in order to incite the reviewer to brainstorm not only one, but multiple suggestions.

Technique 3 – Guide In order to avoid the user skipping changes or even entire files from being reviewed, a guide is offered just before starting the review. This guide consists of a certain amount of comments written by the author. These comments concern only the bigger and complicated changes. They contain an explanation of why a certain change was made. When launching the guide, the reviewer's attention is immediately drawn to the first comment, as decided by the author. The concerned piece of code is surrounded by a red border so that the reviewer focuses his/her attention on it. When the reviewer decides that he/she understands the change, he/she clicks on the button *Next* to go to the next comment. Once having been through every step, the reviewer starts the actual review. Now the reviewer has a better understanding of the code and the thought-process of the author when he/she modified the files.

4.2 Usability test

4.2.1 Plan

Before doing the actual usability test, a plan needs to be set up. It describes what is tested and how as well as what metrics are used and how many participants are required to gain representative results. Let us not forget that scenarios must be described in order to define user goals and the context where the tools are tested. So, the plan for a usability test is composed of the following elements. While doing multiple iterations, some of the plan's elements are adapted according to the results, such as the definition of sessions and the scenarios.

Scope This usability test covers the testing of multiple techniques used to address two cognitive biases during code review; confirmation bias and decision fatigue. The test is done by using a prototype of tools applying the designed techniques. The test analyses the navigation through the prototype, usage of the tools and reaction to the tools by the participants who act as authors or as reviewers depending on the technique.

Purpose We use questions and goals to describe what is supposed to be observed while the participant uses the prototypes. T1, T2 and T3 representing respectively technique 1 with advice to avoid confirmation bias, technique 2 with a more detailed comment form to mitigate decision fatigue and technique 3 with a change guide for the reviewer to mitigate decision fatigue.

T1 When is the advice button used; before or after writing the comment?

T1 What advice is applied after reading it?

T1 Is the advice understood correctly?

T2 What fields are filled out?

T2 Does the comment include all important aspects; problem identification, justification, solutions?

T2 How many solutions are provided?

T2 Do empty fields encourage providing a richer answer?

T3 How are changes investigated first; with or without the guide?

T3 Does the reviewer skip steps of the guide?

T3 How comfortably does the reviewer navigate through the files after using the guide?

Schedule and location All sessions are held online. In the first iteration three participants are observed while using the prototypes. The same participants are used for the second iteration. The sessions take place during the day, reflecting partly conditions of a working day.

Sessions A whole session takes one hour and 15 minutes. Before starting the test, the participant is briefed for 15 minutes. Because only few sessions are done, and each one is isolated from the others, there is enough time to evaluate and reset the sessions in between.

Equipment The participant gets access to the digital prototype over the internet. The prototype is shared by using a messaging service. The participant executes the prototype on his/her local machine, using a single monitor, a keyboard and a mouse to navigate the application. The prototype is executed in a web browser capable of interpreting JavaScript. During the test the participant is not recorded but his/her screen is shared so that notes about his/her behavior can be taken.

Participants For the usability tests that are executed here, we employed 3 participants in two iterations. All the participants are students at the [redacted] being in second year of the masters program of computer science. These students are not necessarily professionally experienced in code reviews. Nevertheless, they are knowledgeable about the theoretical concepts about code reviews and most of them have at least participated in a code review as an author.

Scenarios Even though evaluating multiple scenarios per case is recommended, due to limited time we focus on only one scenario per technique; Technique 1, 2 and 3 respectively as explained previously for the element Purpose.

S1 The code reviewer gives non-constructive feedback. The author receives the feedback and thinks that the comments do not fit.

He/She gets mad at the reviewer and does not accept the suggestions, not even the remarks. The author even starts defending his/her code that he/she worked on for so long. The author experiences confirmation bias, because he/she wants to protect his self esteem. This is due to non-constructive feedback.

S2 The code reviewer is exhausted, because of the mentally intensive work that was done previously. Now he/she feels overwhelmed by the huge number of reviews waiting for him/her. This leads to making incomplete, illogical comments. The code reviewer experiences decision fatigue, because he/she decided to do the reviews only at the end of the day.

S3 The code reviewer, inexperienced or working in a mentally exhausted state, starts skipping especially complicated or big code changes. This leads him/her to not understand what the code is about, thus making illogical comments and skipping the overall comment altogether. He/She experiences decision fatigue, due to inexperience or doing code reviews at the wrong time.

Metrics Before starting the test, questions about the participant's background are asked to verify the profile. Knowing the participant's previous experience is crucial to understand his/her behavior.

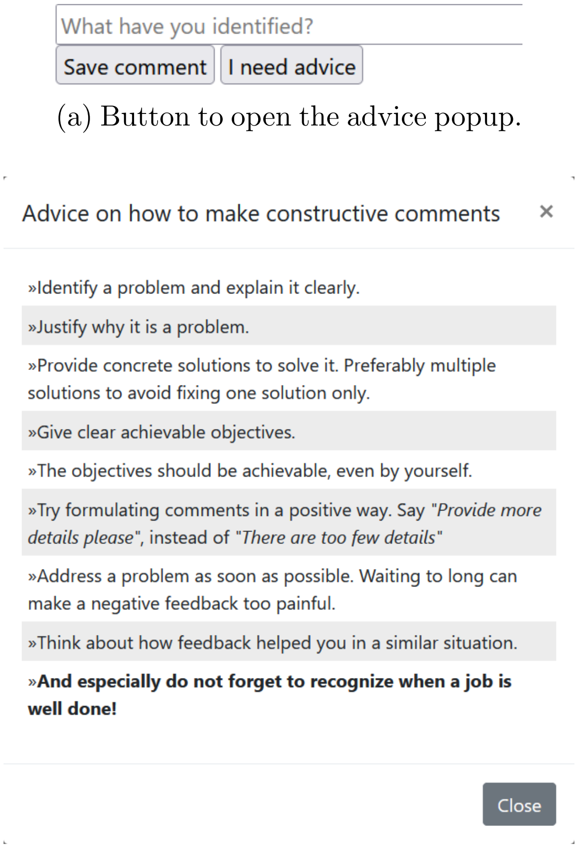
- Have you worked with code review tools such as GitHub before?
- During code review, what position are you experienced with; author and reviewer?

After doing the test, before closing the session, the participant is asked to give personal feedback about the tools that were tested. This feedback is used to understand the user's needs and provide a corresponding solution.

- How useful is the tested technique in order to mitigate explained effects or avoid their triggers?
- What modifications do you recommend to achieve the successful bias prevention?

4.2.2 First iteration

The first iteration starts with the creation of the base prototypes. After analyzing the user's needs, a first prototype for each technique is created. Now it gets tested and we take notes of the feedback from the test participants. The results of the tests are summarized in Table 4.1.



(a) Button to open the advice popup.

Advice on how to make constructive comments

»Identify a problem and explain it clearly.

»Justify why it is a problem.

»Provide concrete solutions to solve it. Preferably multiple solutions to avoid fixing one solution only.

»Give clear achievable objectives.

»The objectives should be achievable, even by yourself.

»Try formulating comments in a positive way. Say "*Provide more details please*", instead of "*There are too few details*"

»Address a problem as soon as possible. Waiting to long can make a negative feedback too painful.

»Think about how feedback helped you in a similar situation.

»**And especially do not forget to recognize when a job is well done!**

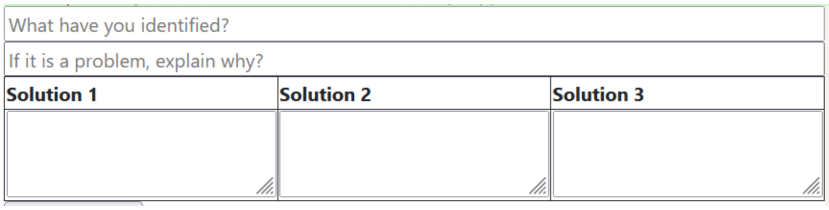
(b) Advice shown in a popup frame.

Figure 4.3: Advice as popup to help provide constructive feedback.

Technique 1 – Advice The first tested technique is the advice available in form of a popup, as displayed in Figure 4.3b. In the case where participants notice the advice button and actually use it, most of them read the advice and try to apply it while writing the comment afterwards. Some participants are convinced that they know the basic rules for constructive feedback already and do not need to be helped out. Using a button to open the popup requires an intermediate step. This leads some participants to not even open the advice.

The participants expressed feedback asking “to make the advice more accessible, instead of having to press a button first”. Also, seeing all the advice at once seems overwhelming, which hints to improve the technique by showing advice depending on the context. For example, making advice appear for positive commenting only when detecting that the entered comment is formulated negatively. Participants liked the short formulation of the advice. The advice is colored alternating, gray and white. However, participants did not understand the coloring even though the color is only meant to help distinguish one piece of advice from another.

For the next iteration, first the number of steps to see the advice is reduced. Second, the background color for the advice is replaced with a single color, green. Because the first three pieces of advice explain the structure of the comment and reviewers use this advice as a checklist, a new technique is added. This technique only shows an example of a constructive comment following the advice as explained previously. The changes and the new technique with feedback and observations can be seen in the second iteration Section 4.2.3.



What have you identified?

If it is a problem, explain why?

Solution 1	Solution 2	Solution 3

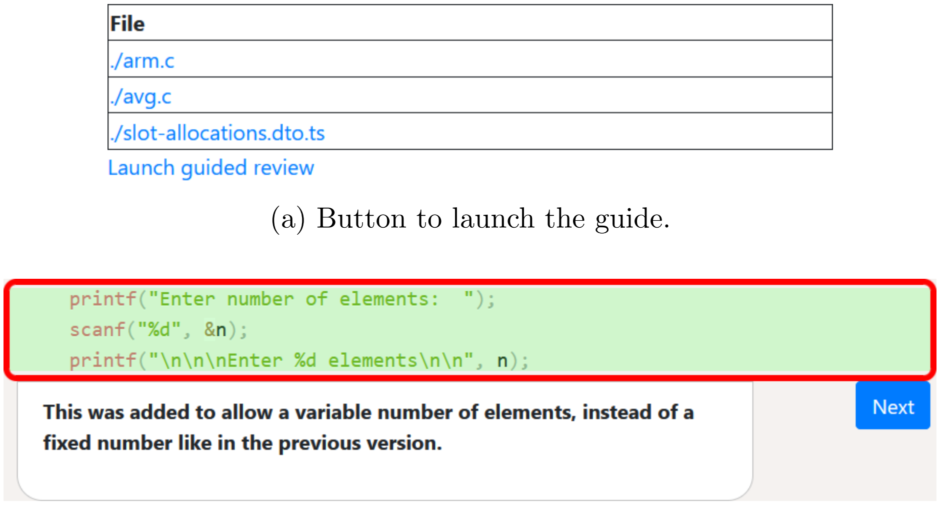
Figure 4.4: A form to help to structure comments.

Technique 2 – Form The second tested technique is a form made of multiple empty fields helping to structure the comment, as seen in Figure 4.4. A field asking to identify the problem, a field asking to justify the said problem and three fields asking for the description of a solution. All the participants filled out the fields, except for the solutions; mostly only one solution was given. This showed that reviewers are ready to follow a structure to create a complete comment, which was voiced to be “especially useful for beginners who do not have much code review experience”.

However, some participants made clear that “too many empty fields can make the reviewer feel overwhelmed, because it looks like now more effort than before has to be put into the comment”. Also, when brainstorming on multiple solutions to solve the found problem, one participant felt like he/she is “doing the author’s work”. Some misleading opinions came up too, such as the missing possibility of selecting multiple lines for comment. In the used prototype this functionality is not available, because it is not in the focus of the test. It is supposed that such a functionality does already exist like in tools such as Github and Gerrit. Here, the focus is put on the comment functionality combined with its psychological aspects.

In order to handle the reviewer’s feeling of being overwhelmed, some asked for techniques that auto-complete forms. They also asked for techniques that make the reviewer avoid unnecessary steps, such as opening a new tab for search or clicking a button to open a window to see specific information. To respond to these requests, a search tool is added in the next iteration. It allows reviewers to use a search bar integrated in the comment tool, in order to quickly find a solution or a code snipped from Google, Stack Overflow and other sites. Yet another suggestion was to facilitate the process of asking colleagues for help when the reviewer is unsure about his/her comment concerning a specific topic. Therefore the possibility to ask an expert in the field before saving the comment is added.

Technique 3 – Guide The third tested technique is the guide set up by the author. When the reviewer lands on the overview page for a changeset, the prototype shows under the changed file a link named *Launch guided review* to start the guide, as seen in Figure 4.5a. The reviewer voluntarily either opens a file directly to comment or launches the guide. This feature is used optionally. Once the guide is launched, the reviewer is presented a view focusing on a specific part of the code. The focused part was selected by the author when he/she created the guide. The reviewer sees a red border around the piece of code that the author wants to emphasize. To help the reviewer understand the piece of code in question, the author has put a comment to it, which is visible when hovering the mouse pointer over the code. When the reviewer feels he/she understand the the changed code and how the author thought, then he/she uses the *Next* button to show the following code change.



File
./arm.c
./avg.c
./slot-allocations.dto.ts

(a) Button to launch the guide.

```
    printf("Enter number of elements:  ");
    scanf("%d", &n);
    printf("\n\n\nEnter %d elements\n\n", n);
```

This was added to allow a variable number of elements, instead of a fixed number like in the previous version.

(b) Focus around code snipped with comment.

Figure 4.5: Guide with comments to help reviewers understand the changes.

This procedure is repeated for all the comments that the author put into the guide. The goal is not to explain every piece of changed code, but rather to give the reviewer a brief idea of the whole submitted change-set, to save time analyzing the code. All the participants intentionally clicked on the guide. While using the guide, the reviewers read every step carefully without skipping one of them. Some participants also read code outside the focused area, which is not intended by the feature.

One participant expressed a concern about “being potentially biased from the comments in the guide”. According to this, the reviewer could be led to accept code without objectively analyzing the code. Another concern was the signification of the *Next* button. Some participants could not anticipate what consequence clicking on that button has. Also, some misleading feedback appeared. Due to the goal of not biasing the participants opinion, some information about the psychological aspects were not explained, leading to misunderstand the use case of the tested technique. For example, some participants were unable to discern specification comments from comments explaining the reason for modification.

For the next iteration the launch button is enlarged and displayed in a more inciting way. The same way, the *Next* button is put in another location, making its use more intuitive.

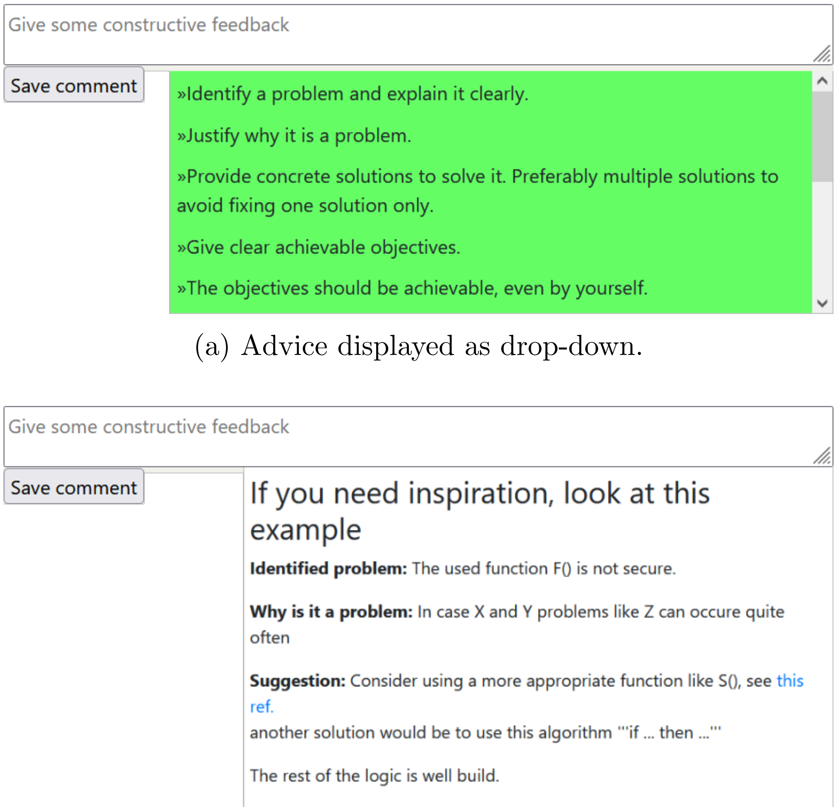
Technique	What works	What does not work
Advice	– Most users apply the advice after reading it. – The short formulated advice is appreciated.	– The popup button is not always noticed. – Not everybody wants to be advised. – Background color confuses participants.
Form	– All fields get filled out. – The form provides a coherent structure.	– Only one solution is given. – Some feel overwhelmed.
Guide	– Everyone uses the guide. – No code change is skipped.	– The guide could bias the reviewer's comment. – The Next button is not intuitive.

Table 4.1: Summary of first iteration.

4.2.3 Second iteration

In the first iteration we collected feedback from the base prototype. This allows us now to improve the prototype according to the observed behavior and the suggestions made by the participants. Some techniques get improved, others get extended. Also, two new techniques are added whereas one technique is removed. The results of the test are summarized in Table 4.2.

Technique 1 – Advice In the current iteration the advice popup was transformed into a drop-down list that is immediately visible when the comment tool is opened, as seen in Figure 4.6a. This way the reviewer does not need to go through extra steps, only to see the advice. To distinguish the advice from the actual input field, its background color is green and not alternating gray like before. Now, every participant notices the advice and reads it, compared to previously where not all reviewers opened the popup.



Give some constructive feedback

Save comment

»Identify a problem and explain it clearly.

»Justify why it is a problem.

»Provide concrete solutions to solve it. Preferably multiple solutions to avoid fixing one solution only.

»Give clear achievable objectives.

»The objectives should be achievable, even by yourself.

(a) Advice displayed as drop-down.

Give some constructive feedback

Save comment

If you need inspiration, look at this example

Identified problem: The used function F() is not secure.

Why is it a problem: In case X and Y problems like Z can occure quite often

Suggestion: Consider using a more appropriate function like S(), see this ref.
another solution would be to use this algorithm '''if ... then ...'''

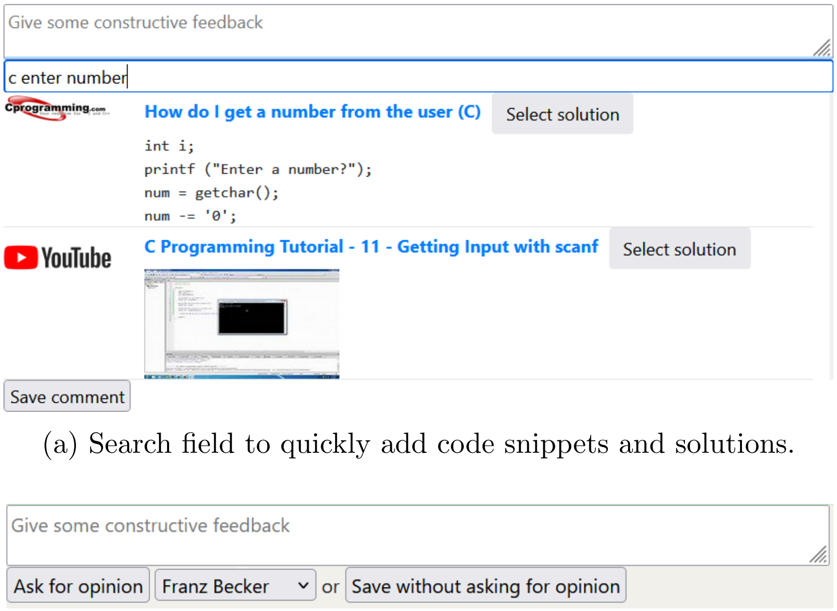
The rest of the logic is well build.

(b) Example displayed as drop-down.

Figure 4.6: Techniques to help to provide constructive feedback.

Most of the participants use the advice like a checklist, verifying one piece of advice after another, looking if they implemented it correctly. An impact on the overall comment that is given at the end of the review can be observed too. It seems like the reviewers apply the advice even outside the comment tool. Due to the usage of the advice as a checklist, one participant recommended "displaying the advice as a real checklist, so that each piece of advice can be checked one by one". The green background however does not show positive effects. One participant perceives the advice "as a checklist that is already completed". "Another, more neutral color would fit better". Also, though most elements can be used like in a checklist, some of them can not, because they are intended to incite the reviewer to analyze the code from another perspective.

Technique 2 – Example In the previous iteration a new technique emerged from the advice technique (T1), displayed in Figure 4.6b. Here the technique using an example shows participants are eager to use it. They follow exactly the structure presented in the example. Some type the keywords into the input field; others only type the response without typing the keywords before. The feedback confirms appreciation, stating "this technique is clear, easily applied and saves time to think about the comment structure". Participants reveal multiple times the advantage of such a structure, because like this "comments could be automatically organized more easily and foster following a clear convention". The only observed down-side of this technique is the necessity to manually type the keywords of the comment structure; *problem*, *justification* and *solution*. This issue is solved by adding a button that automatically types the structure into the comment field.



c enter number

How do I get a number from the user (C)

```
int i;
printf ("Enter a number?");
num = getchar();
num -= '0';
```

C Programming Tutorial - 11 - Getting Input with scanf

(a) Search field to quickly add code snippets and solutions.

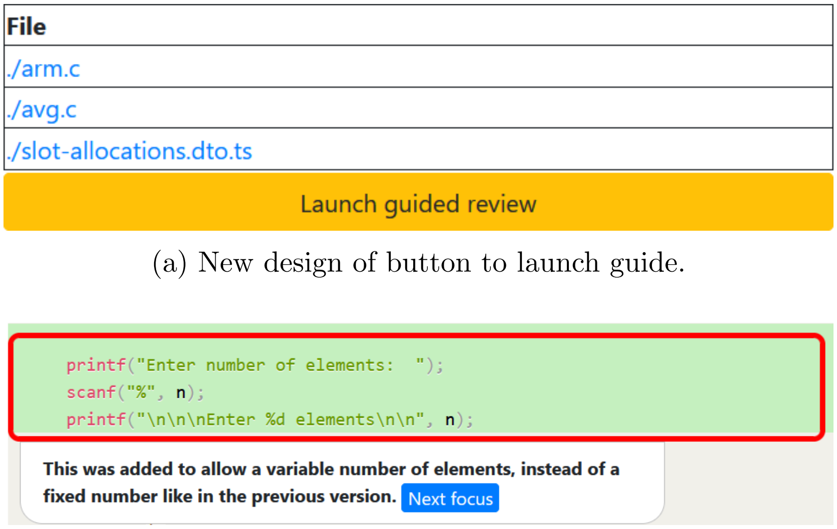
(b) Optional selection of an expert to review comment.

Figure 4.7: Techniques to make complete comments.

Technique 3 – Quick search As requested in the first iteration, tasks should take less effort, thus less intermediary clicks. The form that was used before is now replaced with a tool for a quick search, as seen in Figure 4.7a.

It was used in quite different ways. One participant fills out the comment field first, followed by using the quick search in order to find a code snipped that visualizes the comment given before. Another participant first searches a solution by using the quick search and only then writes the actual comment. The comment however, only refers to the selected solution from the search without giving further explanation. All participants say that "the technique is interesting to use".

Technique 4 – Expert feedback Like for the previous technique another help was requested. Providing expert feedback is a technique that is already advised in the Design chapter. It allows countering the effects of decision fatigue when the reviewer is prone to take a passive role. However, as observed during the tests, most participants do not use the expert feature because they either feel like they are experienced enough or because they do not want to needlessly annoy a colleague by asking for feedback. The prototype is displayed in Figure 4.7b



File
./arm.c
./avg.c
./slot-allocations.dto.ts

(a) New design of button to launch guide.

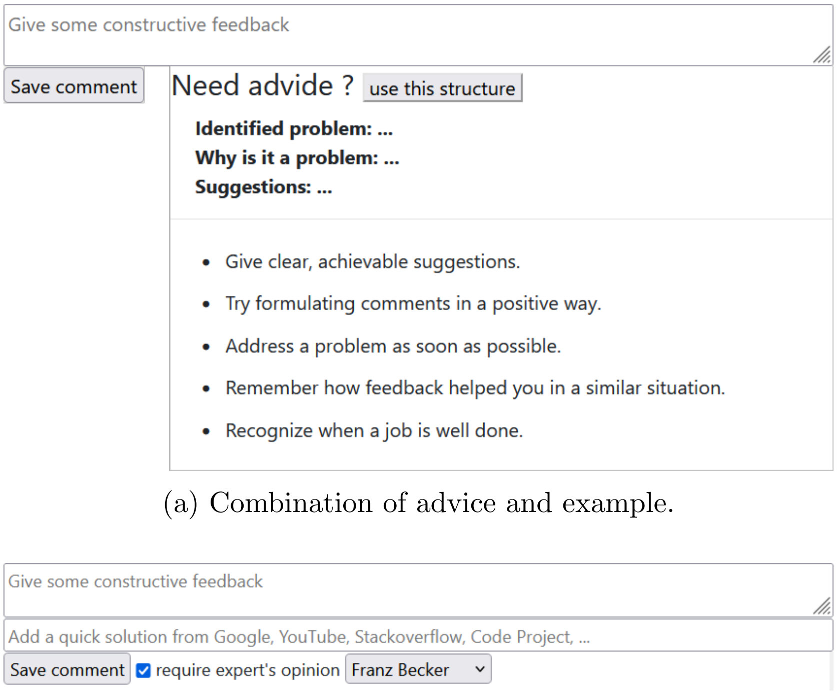
```
    printf("Enter number of elements:  ");
    scanf("%", n);
    printf("\n\n\nEnter %d elements\n\n", n);
```

This was added to allow a variable number of elements, instead of a fixed number like in the previous version.

(b) New location of Next button.

Figure 4.8: Improvements of a guide.

Technique 5 – Guide In this iteration the launch button for the guide is bigger and thus better visible, as seen in Figure 4.8a. The reviewers notice the offered guide right away and instantly use it. Also, the button used to get to the next step is placed in a more understandable location, right next to the comment, as in Figure 4.8b. Users do not ask anymore what the *Next* button is for. One participant explains that he/she "feels obligated to give a comment after following the guide, because using the *Next* button makes him/her believe he/she confirmed having seen the author's comment and therefore he/she feels like he/she cannot abstain from reviewing the concerned piece of code, like if he/she missed it by accident". All participants complain about not being able to give comments while using the guide. This however is a feature that would not support the intended effect. Being able to comment simultaneously hinders the reviewer from following the guide to understand the code as a whole and not only partially.


Give some constructive feedback
Save comment
Need advide ?
use this structure

Identified problem: ...
Why is it a problem: ...
Suggestions: ...

- Give clear, achievable suggestions.
- Try formulating comments in a positive way.
- Address a problem as soon as possible.
- Remember how feedback helped you in a similar situation.
- Recognize when a job is well done.

(a) Combination of advice and example.

Give some constructive feedback
Add a quick solution from Google, YouTube, Stackoverflow, Code Project, ...
Save comment

☑ require expert's opinion Franz Becker

(b) Combination of quick search and expert feedback.

Figure 4.9: Combined techniques.

Combining techniques One suggestion that was made multiple times, is the potential of combining some techniques. First, the advice (T1) can be combined with the example (T2), displayed in Figure 4.9a. This constitutes a single technique aiming for the same goal; creating constructive feedback. Second, combining the quick search feature for solutions (T3) with the expert feedback (T4) feature aims the same way at mitigating the effects of decision fatigue, displayed in Figure 4.9b. For the following step (UEQ) the combination of these techniques is applied and tested.

4.3 User experience questionnaire

4.3.1 Plan

For the final test of the techniques, five participants of a similar profile to the first test group were chosen. But compared to the first group most of them have already done code reviews in the position as an author or as a reviewer. Additionally all participants have a scientific background and are familiar with scientific research. This experience makes for adequate participants to evaluate the usability of the tested techniques.

A test session proceeds with every participant as follows. First questions about the participant's experience are asked. Depending on the knowledge we explain and demonstrate to him/her what code reviews are and how tools can be used to assist the code review process. Next, one technique at a time is explained and tested. The explanation however takes place without mentioning the investigated aspects concerning cognitive biases. This is necessary to not bias the participant's opinion. With this test the goal is only to gain feedback about the user interface and the usability; not about the psychological aspects.

The participant is briefed about his/her task, beginning with the position he/she occupies; for each technique the user plays the role of a reviewer. The task consists in using the code review tool in order to review the changes as shown in the prototype. While doing so, we can guide the participant, but only in the case where he/she does not know how to proceed. After completing the task, the participant is asked to fill out the questionnaire (UEQ [27]) based on the previous technique. This questionnaire is not specifically adapted to the tested tools, nor to the underlying investigated topic, cognitive biases. The measured scales are pre-defined for each user experience questionnaire and represent the 26 underlying items that participants evaluate:

Technique	**What works**	**What does not work**
Advice	– The advice is noticed and read immediately. – The advice impacts the overall comment.	– Green color signifies already complete. – Participants mistake inciting items for to-do items.
Example	– All participants use the technique. – Saves time to think about structure. – Comment analysis could be automated.	– Participants type the keywords manually.
Quick search	– All participants use the technique.	– Some only use the quick search, without making a comment.
Expert feedback		– Most participants do not use the technique. – Most users are concerned about annoying colleagues.
Guide	– Launch button is noticed faster. – Understanding the Next button is intuitive. – The Next button acts as obligation to comment.	– Unable to make comments inside guide.

Table 4.2: Summary of second iteration.

- Attractiveness: Overall impression of the product. Do users like or dislike the product?

- Perspicuity: Is it easy to get familiar with the product? Is it easy to learn how to use the product?
- Efficiency: Can users solve their tasks without unnecessary effort?
- Dependability: Does the user feel in control of the interaction?
- Stimulation: Is it exciting and motivating to use the product?
- Novelty: Is the product innovative and creative? Does the product catch the interest of users?

Additionally we ask every participant after the questionnaire for a personal feedback about the tested technique. “Do you think the technique improves the code review process?”. “What can be improved in your opinion about the presented tools?”. The discussion about the techniques is done after responding to the questionnaire to not bias the participant before giving feedback and catch his/her immediate impression towards the technique.

4.3.2 Results

The data collected with the questionnaires and the feedback from the participants are now used to determine whether or not the techniques fulfill the general user's expectations concerning user experience. Such expectations are generally formed by products that the users frequently use. In the first testing phase we analyzed the expectations of the users with usability testing. Now, we verify if the expectations are the same for the second group and if the expectations are fulfilled. First we summarize the personal feedback and the behavior we observed while participants used the techniques. Then we look at the questionnaire answers to determine if the tools enhance the user experience. The answers of the user experience questionnaires should align with the described feedback and represent a global opinion of all participants and the future users. In order to evaluate the questionnaires we used the UEQ Data Analysis Tool available on the website of the very questionnaire. This tool provides a brief quantitative analysis of the responses. The responses are on a scale from 1 to 7. For the analysis they get converted to scale from -3 (most negative) to +3 (most positive). According to this scale we calculate the mean and variance for each item and each scale. Here, the previously mentioned six scales are used to describe the user's experience. We focus on the benchmark results which classify the technique into five categories, one per scale. A scale can be evaluated as excellent, good, above average, below average or bad. However, we have to keep in mind that only a small sample of participants was used. This means that quantitative analysis might show less accurate results. Nevertheless, a quantitative analysis reveals at least tendencies of the user's experience. We focus on a qualitative analysis of user feedback and behavior in order to determine usability and what aspects to improve.

Advice with example With the combination of the two techniques, advice (T1) and example (T2), a new button is added to allow preparing the comment structure faster. This button copies the key phrases "Identified problem", "Why is it a problem" and "Suggestions" into the comment field. However, this button is not noticed and it is therefore not used by any of the participants. Most participants type the structure into the form manually like the participants from the first group in iteration 2. Others do not type the key words at all and only provide the response to the keywords. What can be noticed, is that even when the button is not used, the proposed structure is well respected by all the participants. Coming to the advice which is located beneath the example, only some of the participants read it; however only briefly. Only one participant reads it with the goal to apply the advice one by one to the comment.

While designing the techniques with the first group of participants, users showed willingness to be guided. An example is when users use the structure offered as an example to guide the construction of their comments. This expectation aligns with the feedback of the second group which expresses usefulness of a pre-defined comment structure, "because this makes for an easier analysis afterwards". Additionally, feedback expresses the advantage of structure to automate tasks, as well as standardizing the comment form. This way, the author knows what to expect when receiving the feedback; "leading to a less personal, more constructive discussion". Another expectation of the first group was to facilitate and speed up tasks. By providing an example, this expectation is met. The reviewer does not need to rethink the comment structure anymore. However, for some participants the user interface appeared confusing; especially the coloring was criticized, because "a white background color does not draw the user's attention to the advice". Also, the proposed structure is criticized due to "focusing mainly on negative aspects, although the advice recommends including positive aspects too". Following the feedback, we analyze the questionnaires, looking at the means and variances as well as at the benchmark comparison.

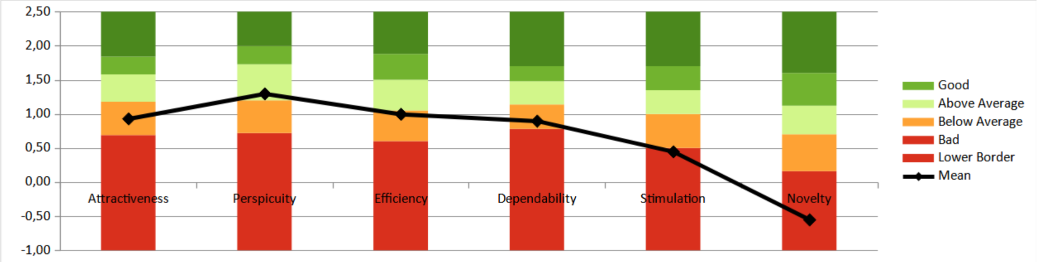


Figure 4.10: UEQ benchmark for advice with example.

The benchmark displayed in Figure 4.10 shows that the users perceive the techniques (T1 and T2) as understandable and easy to learn. However this aspect is to be considered with caution, because the answers reveal a high variance throughout the items of this scale. This indicates that some items of the scale perspicuity might be misinterpreted. To improve the quality of the results, data of more participants is required. For the scale stimulation, how exciting and motivating the technique is, we see that users perceive the technique as bad. Similarly, the combination of the techniques is not seen as innovative or creative but rather as conventional and usual. This last scale must however be used with caution, because similar to the scale perspicuity, the high variance between measured items indicates a potential misunderstanding. The results are displayed in Table 4.3.

Scale	Mean	Variance
Attractiveness	0.933	0.55
Perspicuity	1.300	2.67
Efficiency	1.000	0.53
Dependability	0.900	0.64
Stimulation	0.450	0.54
Novelty	-0.550	1.08

Table 4.3: Mean and variance for the technique advice with example.

Quick search with expert feedback For this test the Quick search (T3) and Expert feedback (T4) techniques were not modified, but only combined as requested by participants. This advice turns out not to enhance the interest in using the expert feedback feature. Most users immediately focus on the search field for a quick solution. However, compared to the first group from the usability tests, the second group does not enter a search query into the input field. Instead all participants, except one, open a new tab to manually search for a solution on mentioned sites like Stack Overflow and Google, only to copy and paste the link into the search field. The goal is to use the search field to quickly find a link or code snipped without leaving the code review tool. After intercepting by explaining the technique in more detail, all participants understand the feature and use it adequately. Contrary to the search field, the expert feedback is not used by any participant, even though some users of the first group used it.

Because acceleration of tasks is an expectation that users expressed quite often a quick search for solutions was added. Also the expert feedback is supposed to facilitate asking colleagues or experts to confirm in case of doubt. With a sped up process in mind we designed techniques that users understand quickly. Using elements such as buttons and text-fields in the right place is necessary to offer intuitive solutions. In the case of the search field (T3), at first users missed the point of the technique. Some participants suggested replacing the keyword *Add* with *Search* so that the emphasis is on the help to search and on not the obligation to add a link. As soon as the participants understand the use of the search field, the technique is well appreciated as "it simplifies commenting and saves time". This feedback aligns with the expectations of the first test group. Nonetheless, users are concerned about copyrights when using code snippets from an online source. Another concern is about the potentially low quality of code snippets proposed by the search. So, participants show a significant interest in a tool that facilitates the review process, but that does not make the result suffer in quality. Concerning the expert feedback (T4) some users state that they either do not feel the need to use the feature because they feel experienced enough, or because they do not want to annoy colleagues unnecessarily. The former could be due to over confidence, a cognitive bias which is not investigated further in this work. Overall the latter feature is not appreciated.

The benchmark displayed in Figure 4.11 shows that the users perceive the techniques as interesting, even exciting and motivating to use and to get familiar with. The same is observable for the tool's efficiency, which means

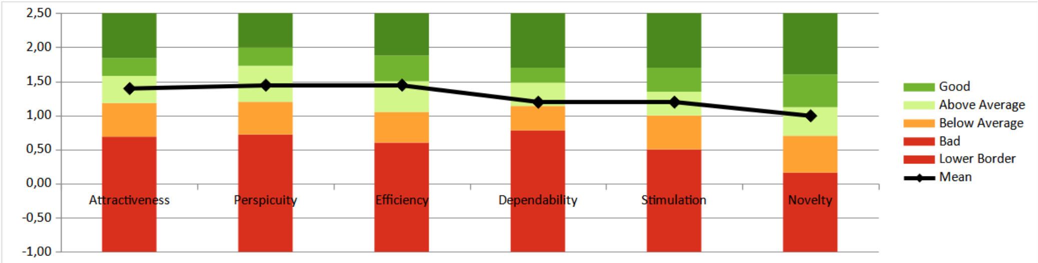


Figure 4.11: UEQ benchmark for quick search with expert feedback.

that users can apply the technique without a particular effort. This is in fact the goal of the technique, and we can see that the result is good. Furthermore, novelty being a less common characteristic to achieve, as noticeable in the benchmark, the technique still turns out to be perceived as innovative in the field of code reviews. Items of the scale dependability seem to be misinterpreted when referring to the variance of the responses, which leads to use these values only with caution. Overall the results are significantly positive in terms of user experience, which corresponds to the feedback. However, the responses are more targeted towards the quick search (T3) and less to the expert feedback (T4). The results are displayed in Table 4.4.

Scale	Mean	Variance
Attractiveness	1.400	0.19
Perspicuity	1.450	0.51
Efficiency	1.450	0.33
Dependability	1.200	1.42
Stimulation	1.200	0.29
Novelty	1.000	0.78

Table 4.4: Mean and variance for the technique quick search with expert feedback.

Guide After the second usability test iteration we optimized the guide by displaying the red border around the change only when the corresponding step is in focus. Before, the red border was shown constantly which could have mislead the user's focus. At the beginning the reviewer is shown a list of changed files and a button to start a guided review. All participants use this very button, explaining "that every proposed supporting tool is appreciated, as long as it is not overwhelming". Once the guide started, most participants immediately hover the mouse pointer over the focused area which makes the author's comment visible. One participant did not understand intuitively how to proceed. This could be due to a problem, causing the mouse pointer not to trigger the visibility of the author's comment. After getting explanations, the participant understands how to use the tool. All users carefully read the author's comment and know intuitively what the *Next* button is for. They use it until they arrive at the actual review interface, where the reviewer's comments fall out shorter, less detailed than compared to the previous techniques (T1 and T2, T3 and T4).

During the usability test users expressed the need for tools that guide the reviewer, as explained for the techniques advice (T1) and example (T3). Guidance supports the reviewer so that he/she is not overwhelmed by the quantity of changes to review. A tool that helps understand the code changes quickly is a requirement that was often expressed too. Coming to the feedback, all users say "the technique makes for easier understanding and pushes the reviewer to make useful comments, because now he/she goes through the most important aspects of the code in a concentrated way". Also, using the guide is seen as "logical, because this way the reviewer can follow the author's logic from file to file, which was not possible before where the review is done without respect to the relationship between the files". One criticism states "that the reviewer's opinion is influenced when he/she reads the author's comment". Another criticized limitation comes from the creation of a kind of a story for the guide; "this means that one review can only be about one specific topic or story". "Therefore, commits can not be made arbitrarily; commits must concern only one topic at time". This requires the reviewer to pay attention to what changes to include in the commit, whereas before a commit simply included all changes. Overall the feature is appreciated, but when using this technique, one must evaluate whether the improved review quality compensates the additional invested time and effort.

The benchmark displayed in Figure 4.12 shows that the users perceive the technique as very attractive and interesting to use. This is confirmed with the a clear indication of finding the tool stimulating and innovative. In terms of efficiency, it is evaluated positively as well and users mostly feel that the tool fulfills expectations by being supportive. Regarding perspicuity it is difficult to say whether participants find it easy or not to comprehend the technique and learn it, because a slightly high variance of responses is

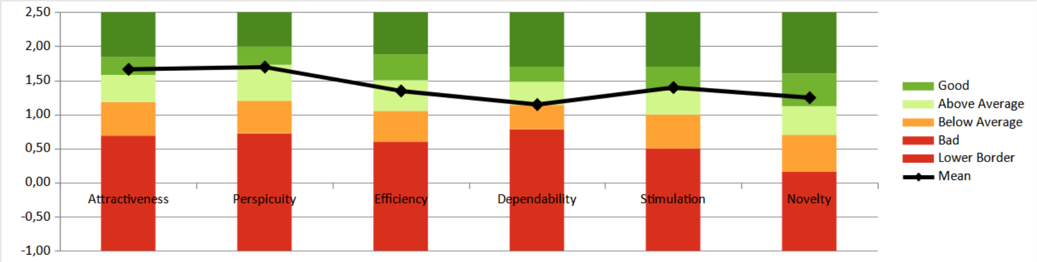


Figure 4.12: UEQ benchmark for a guide.

observable between the measured items; meaning that participants might have misinterpreted the items in this context. The results are displayed in Table 4.5.

Scale	Mean	Variance
Attractiveness	1.667	0.15
Perspicuity	1.700	0.83
Efficiency	1.350	0.71
Dependability	1.150	0.43
Stimulation	1.400	0.21
Novelty	1.250	0.63

Table 4.5: Mean and variance for the technique guide.

4.4 Discussion

In this last section we discuss the results previously observed about the techniques. We take a step back and look at the results from the perspective of the initial research question. The question is how to prevent or mitigate cognitive biases during code review. For each of the tested techniques we first interpret feedback and behavior. Then we conclude how a technique responds to the research question and what limitations we come to face.

4.4.1 Confirmation bias

Refusing feedback The hypothesis is, that during code review non-constructive feedback provokes confirmation bias from the author. Therefore a second hypothesis arises, stating that providing constructive feedback to the author prevents confirmation bias from happening through comments. Results from user experience tests show that reviewers are keen to be guided by applying a pre-defined structure (T2) for the comment. Results also show that most reviewers are not motivated to take written advice into account (T1). Overall advice and examples are used in a quick manner, without spending more than a few seconds to integrate them in the comments.

Concerning the technique example (T2) reviewers are ready to integrate it into the code review process. Also, most users agree on confirming the positive effects of this technique. Thus, based on feedback and the investigated knowledge about triggers of confirmation bias from the literature, appears to be helpful in prevent confirmation bias during code review.

With this research we do not test quantitatively to what extent the technique prevents confirmation bias. Nevertheless we developed a first prototype of a solution and tested the user experience. In order to analyze how effective the technique is, further investigation will be necessary using a greater sample size for the tests to get representative quantitative results.

4.4.2 Decision fatigue

Incomplete comments The hypothesis is that during code review decision fatigue leads to incomplete comments from the reviewer, due to fatigue or taking a passive role. Multiple hypotheses arise from here. First, a search tool (T3) to quickly find solutions from the internet helps reviewers while making comments, hence supporting completeness of feedback. Second, a tool to request feedback from an expert (T4) for one's comment helps reviewers when taking a passive role.

Results from user experience tests show that a search tool incites reviewers to add a code snippet to the comment. The dynamic interface is well appreciated and used by all participants, but only after the users understand exactly the use case of the technique. Thus, based on feedback and the investigated knowledge about the effects of decision fatigue from the literature, this technique appears to have significant potential to mitigate the impact of decision fatigue during code review.

Results from user experience tests show that requesting an expert for feedback is perceived as annoying for colleagues. None of the reviewers use this technique to reassure themselves about their comments. Others ignore this technique because they are confident enough about their comments.

Thus, based on feedback and the investigated knowledge about effects of decision fatigue from the literature, we can say that this technique in its state of a prototype can not be used to mitigate the effects of decision fatigue. It can potentially be improved in terms of design to incite reviewers to use it.

The design and layout of elements play a big role, which might explain the failure of the techniques advice (T1) and expert feedback (T4). The user's understanding and willingness to use a technique is strongly dependent on the design of the user interface. Therefore, if a technique might not seem useful at first, another design could reveal opposing results.

Skipping code changes The hypothesis is that during code review decision fatigue leads to skipping review of particularly short, big or complicated changes. This gives way to the hypothesis that a guide (T5), helping the reviewer to go through important changes one by one, prevents the reviewer from skipping them. Results from user experience tests show that reviewers always choose to use the guide instead of investigating the changes individually. While using the guide the users follow through until the end without skipping the presentation of a code change.

Feedback from participants confirms the appreciation of following a guide. Hereby the technique shows that it effectively prevents decision fatigue from making users skip code changes. Thus, based on the effects of decision fatigue described in literature and the results of the prevention technique developed here, we can say that this way certain effects of decision fatigue during code review can be prevented.

This research focused on the user's willingness to stay focused on reviewing code changes by not skipping them. However, concerns were expressed about the quality of comments after using the guide. Potentially the review is biased by the comments that the reviewer reads from the guide. This factor is however not investigated in this research, but could be in some future work where the focus is on the quality of the feedback after using the techniques we have described here.

Chapter 5

Conclusion

With this work we explored the relationship between cognitive bias and modern code reviews. The need to investigate this topic stems from the literature which clearly shows the impact of cognitive aspects on software engineering activities. Multiple studies investigated how social aspects and a developer's personality affect the software's development process. This previous knowledge gives us the opportunity to investigate this relationship more in detail. Here the first goal was to discover in what cases cognitive biases appear during code review and what impact they have. A second goal was to select some of the previously described cases in order to develop a solution to either prevent a cognitive bias or to mitigate its effects on code review.

The main objective was to create solutions with the end user's requirements in mind. This could be achieved by iterating multiple times over the solutions together with two groups of reviewers. A first group was used to develop the solutions by conducting usability tests. And a second group was needed to validate the final results without being biased in their opinion by conducting user experience tests (UEQ). The proposed result of this work are multiple prototypes which have been tested exhaustively. The tests were focused on qualitative data. By using the reviewer's feedback we were able to adapt the solutions depending on the needs of the participants. While testing the techniques, the limitation of working with prototypes sometimes led to confusion and misleading feedback. Some participants expected the prototype to work like a fully developed tool. Nevertheless, after reexplaining the goal of the test with more details, most participants understood the idea and the quality of the feedback improved.

Due to a limited number of participants and little time to test, we were obligated to use a rather small sample size to get quantitative data. Nonetheless the quantitative results could be used to explain the participant's tendencies and so help discover pain points as well as satisfactory elements. Knowing these tendencies was crucial to improve the prototypes during development. Another limitation could become the ethical question about some of the techniques, which on one hand support the reviewer but on the other hand take cognitive information of the user into account. Such information is regarded as personal and must be treated in way that takes ethical aspects into account.

Within this framework we designed potentially problematic cases due to cognitive bias for confirmation bias and decision fatigue. The final result consists of one prototype concerning confirmation bias triggered by non-constructive feedback, and of two prototypes concerning decision fatigue to avoid skipping code changes and incomplete feedback during code review. Different techniques were used in each prototype while some of the techniques showed positive and others negative results. Regarding the successful techniques, feedback clearly indicates acceptance of the techniques among the reviewers as well as the quantitative data confirming this very feedback.

Even though some techniques show bad results, the potential must not be underestimated, as oftentimes the quality of a technique depends on the visual layout. Therefore, it seems to make sense that further research focuses on the investigation of techniques from a design point of view so as to achieve better results. Some of the proposed solutions were prototyped only partially or not at all. Therefore those solutions could be part of a future research, including scheduled reviews to avoid decision fatigue or warning the reviewer when he/she requires too much time for the review. Many suggestion came from user feedback. Most participants suggested to automate and make the review task as easy as possible. For example, offering predictions for comments based on often used phrases. Another suggestion was to provide feedback depending on the context. For example, mention to stay objective only when the tool detects overly subjective formulations. Furthermore, the research could be extended to other cognitive biases as well as other triggers and effects of the currently investigated biases. Finally, the developed prototypes could be transformed into fully working tools that can be applied in real working environment.

Bibliography

[1] Hal R. Arkes. Costs and benefits of judgment errors: Implications for debiasing. *Psychological Bulletin*, 110(3):486–498, 1991. Place: US Publisher: American Psychological Association.

[2] Anderson S. Barroso, Jamille S. Madureira, Michel S. Soares, and Rogerio PC do Nascimento. Influence of human personality in software engineering-a systematic literature review. In *International Conference on Enterprise Information Systems*, volume 2, pages 53–62. SCITEPRESS, 2017.

[3] Roy F. Baumeister, Ellen Bratslavsky, Mark Muraven, and Dianne M. Tice. Ego depletion: Is the active self a limited resource? *Journal of Personality and Social Psychology*, 74(5):1252–1265, 1998. Place: US Publisher: American Psychological Association.

[4] Roland Bee and Frances Bee. *Constructive Feedback*. CIPD Publishing, 1998. Google-Books-ID: fFLX0vssr7kC.

[5] Gul Calikli and Ayse Bener. Empirical analyses of the factors affecting confirmation bias and the effects of confirmation bias on software developer/tester performance. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10, pages 1–11, New York, NY, USA, September 2010. Association for Computing Machinery.

[6] Gul Calikli, Ayse Bener, and Berna Arslan. An analysis of the effects of company culture, education and experience on confirmation bias levels of software developers and testers. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 2, pages 187–190, May 2010. ISSN: 1558-1225.

[7] Shai Danziger, Jonathan Levav, and Liora Avnaim-Pesso. Extraneous factors in judicial decisions. *Proceedings of the National Academy of Sciences*, 108(17):6889–6892, April 2011. Publisher: Proceedings of the National Academy of Sciences.

[8] Klaas Andries de Graaf, Peng Liang, Antony Tang, and Hans van Vliet. The impact of prior knowledge on searching in software documentation. In *Proceedings of the 2014 ACM symposium on Document engineering*, DocEng '14, pages 189–198, New York, NY, USA, September 2014. Association for Computing Machinery.

[9] Fadi P. Deek and James A. M. McHugh. Cognitive and Social Psychology in Collaboration. In Fadi P. Deek and James A. M. McHugh, editors, *Computer-Supported Collaboration: With Applications to Software Development*, pages 7–26. Springer US, Boston, MA, 2003.

[10] Michael E. Fagan. Advances in software inspections. *IEEE Transactions on Software Engineering*, SE-12(7):744–751, July 1986. Conference Name: IEEE Transactions on Software Engineering.

[11] Laura Faulkner. Beyond the five-user assumption: Benefits of increased sample sizes in usability testing. *Behavior Research Methods, Instruments, & Computers*, 35(3):379–383, August 2003.

[12] Baruch Fischoff. Debiasing. Technical report, DECISION RESEARCH EUGENE OR, 1981.

[13] Marvin Fleischmann, Miglena Amirpur, Alexander Benlian, and Thomas Hess. COGNITIVE BIASES IN INFORMATION SYSTEMS RESEARCH: A SCIENTOMETRIC ANALYSIS. *Tel Aviv*, page 23, 2014.

[14] Martie G. Haselton, Daniel Nettle, and Damian R. Murray. The Evolution of Cognitive Bias. In *The Handbook of Evolutionary Psychology*, pages 1–20. John Wiley & Sons, Ltd, 2015. Section: 41 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119125563.evpsych241.

[15] Magne Jørgensen and Efi Papatheocharous. Believing is Seeing: Confirmation Bias Studies in Software Engineering. In *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*, pages 92–95, August 2015. ISSN: 2376-9505.

[16] Daniel Kahneman. *Thinking, fast and slow.* Macmillan, 2011.

[17] Joshua Klayman. Varieties of Confirmation Bias. In Jerome Busemeyer, Reid Hastie, and Douglas L. Medin, editors, *Psychology of Learning and Motivation*, volume 32, pages 385–418. Academic Press, January 1995.

[18] David Laibson. Golden Eggs and Hyperbolic Discounting*. *The Quarterly Journal of Economics*, 112(2):443–478, May 1997.

[19] Laura Marie Leventhal, Barbee Eve Teasley, and Diane Schertler Rohlman. Analyses of factors related to positive test bias in software testing. *International Journal of Human-Computer Studies*, 41(5):717–749, November 1994.

[20] Trevor T. Moores and Jerry Cha-Jan Chang. Self-efficacy, overconfidence, and the negative effect on subsequent performance: A field study. *Information & Management*, 46(2):69–76, March 2009.

[21] Raymond R. Panko. Are we overconfident in our understanding of overconfidence? *Software Engineering Methods in Spreadsheets*, pages 48–55, 2014. Publisher: Citeseer.

[22] Grant A Pignatiello, Richard J Martin, and Ronald L Hickman. Decision fatigue: A conceptual analysis. *Journal of Health Psychology*, 25(1):123–135, January 2020. Publisher: SAGE Publications Ltd.

[23] Austen Rainer and Sarah Beecham. A follow-up empirical evaluation of evidence based software engineering by undergraduate students. *NA*, June 2008. Publisher: BCS Learning & Development.

[24] P.N. Robillard, P. d'Astous, F. Detienne, and W. Visser. Measuring cognitive activities in software engineering. In *Proceedings of the 20th International Conference on Software Engineering*, pages 292–300, April 1998. ISSN: 0270-5257.

[25] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: a case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, pages 181–190, New York, NY, USA, May 2018. Association for Computing Machinery.

[26] Iflaah Salman, Burak Turhan, and Sira Vegas. A controlled experiment on time pressure and confirmation bias in functional software testing. *Empirical Software Engineering*, 24(4):1727–1761, August 2019.

[27] Dr Martin Schrepp. User experience questionnaire handbook - ueq-online.org, December 2019. Publication Title: User Experience Questionnaire (UEQ).

[28] Hans Henrik Sievertsen, Francesca Gino, and Marco Piovesan. Cognitive fatigue influences students' performance on standardized tests. *Proceedings of the National Academy of Sciences*, 113(10):2621–2624, March 2016. Publisher: Proceedings of the National Academy of Sciences.

[29] Adam F. Stewart, Donna M. Ferriero, S. Andrew Josephson, Daniel H. Lowenstein, Robert O. Messing, Jorge R. Oksenberg, S. Claiborne Johnston, and Stephen L. Hauser. Fighting decision fatigue. *Annals of Neurology*, 71(1):A5–A15, 2012. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/ana.23531.

[30] Barbee E. Teasley, Laura Marie Leventhal, Clifford R. Mynatt, and Diane S. Rohlman. Why software testing is sometimes ineffective: Two applied studies of positive test strategy. *Journal of Applied Psychology*, 79(1):142–155, 1994. Place: US Publisher: American Psychological Association.

[31] Amos Tversky and Daniel Kahneman. Judgment under Uncertainty: Heuristics and Biases. *Science*, 185(4157):1124–1131, September 1974. Publisher: American Association for the Advancement of Science.

[32] Jeffrey B. Vancouver. The Depth of History and Explanation as Benefit and Bane for Psychological Control Theories. *Journal of Applied Psychology*, 90(1):38–52, 2005. Place: US Publisher: American Psychological Association.

[33] Ashley Waggoner Denton. Improving the Quality of Constructive Peer Feedback. *College Teaching*, 66(1):22–23, January 2018. Publisher: Routledge _eprint: https://doi.org/10.1080/87567555.2017.1349075.

[34] P. C. Wason. On the Failure to Eliminate Hypotheses in a Conceptual Task. *Quarterly Journal of Experimental Psychology*, 12(3):129–140, July 1960. Publisher: SAGE Publications.