Design of Flexible Code Block Comparisons to Improve Code Review of Refactored Code

Heander, Lo; Söderberg, Emma; Rydenfält, Christofer

Design of Flexible Code Block Comparisons to Improve Code Review of Refactored Code

Lo Heander
lo.heander@cs.lth.se
Lund University
Sweden

Emma Söderberg
emma.soderberg@cs.lth.se
Lund University
Sweden

Christofer Rydenfält
christofer.rydenfalt@design.lth.se
Lund University
Sweden

ABSTRACT

Code review occupies a significant amount of developers' work time and is an established practice in modern software development. Despite misaligments between users' goals and the code review tools and processes pointed out by recent research, the code review tooling has largely stayed the same since the early 90s. Improving these tools, even slightly, has the potential for a large impact spread out over time and the large developer community.

In this paper, we use the Double Diamond design process to work together with a team of industry practitioners to find, refine, prototype, and evaluate ways to make it easier to compare refactored code blocks and find previously hard-to-see changes in them. The results show that a flexible comparison modal integrated into Gerrit could reduce the mental load of code review on refactored code. Potentially, it could also have effects on how code is written by no longer discouraging refactoring due to it complicating the review. The user interface created in this collaborative manner was also intuitive enough for all of the participants to be able to use it without any hints or instructions.

CCS CONCEPTS

• **Human-centered computing** → **HCI design and evaluation methods**; **Empirical studies in HCI**; **Interaction design process and methods**; • **Software and its engineering** → **Software verification and validation**.

KEYWORDS

code review, human-computer interaction, interaction design, software engineering

ACM Reference Format:
Lo Heander, Emma Söderberg, and Christofer Rydenfält. 2024. Design of Flexible Code Block Comparisons to Improve Code Review of Refactored Code. In *Companion Proceedings of the 8th International Conference on the Art, Science, and Engineering of Programming (‹Programming›Companion '24), March 11–15, 2024, Lund, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3660829.3660842

1 INTRODUCTION

Software developers today spend between 10-20%[9, 21] of their working time doing code reviews. With the total number of software developers expected to reach 28 million people by 2024 [1], this could mean between 22-44 million hours spent doing code reviews every day! When usage is on this scale, even small improvements in code review tools and processes can have a significant effect.

Yet, the tools used have not changed in their approach since they were first introduced with ICICLE in 1990 [10]. Research has shown that there are misalignments [23] between the tools used and the desired goals, such as code quality and knowledge sharing [4].

There are not very many studies published that explore how changes or new features in the code review tools can affect the review experience or quality. Bagirov et. al.[5] investigates if the ordering of the files in the review could be improved. Baum 2019 [6] studies how code review tools could be improved using cognitive support techniques to reduce the cognitive load of the task and improve code review quality. Baum et al.[7] study the (mis)alignment between the code review task and requirements and the tools in use today. They believe that there is room for improvement and that a new generation of more specialized tools could lead to "increased review efficiency and effectiveness".

In this paper, we explore ways to improve the code review developer experience by applying a double diamond design process (Section 2) to the code review tooling. Our research questions are:

- **RQ$_1$** What developer experiences during code review can cause frustration?
- **RQ$_2$** How can code review tools be modified to improve the developer experience?
- **RQ$_3$** How can developers be involved in the design process to better discover, understand and design tooling improvements?

To answer these questions, we study the code review experience of software developers, with an established code review process in Gerrit, working at a company developing embedded systems. Through a focus group session with the developers, we identify several problems that could be addressed by improved tooling. We select one of these problems and organize a co-design workshop with the participants, focusing on coming up with a range of possible solutions. One solution, a flexible code comparison modal to compare moved and refactored code blocks, is chosen. After developing this solution into a high-fidelity prototype, we bring it back to the software developers for evaluation and feedback.

The feedback from the participants (Section 3), during both the workshops and the evaluation, suggests that this feature could simplify code reviews of moved and refactored code. It could potentially also have effects on how code is written, by no longer discouraging change and refactoring in moved code to avoid complicating the review.

‹Programming›Companion '24, March 11–15, 2024, Lund, Sweden

ACM ISBN 979-8-4007-0634-9/24/03
https://doi.org/10.1145/3660829.3660842

2 METHOD

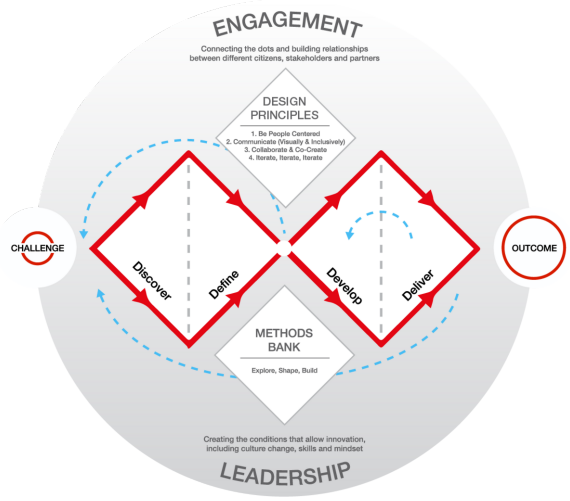

Figure 1: The Double Diamond design process.
From https://www.designcouncil.org.uk/

To work towards answering **RQ$_3$**, we wanted to choose a method that involved the developers using code review tools in the design process. The purpose of this is both to increase the chance of designing something that would be genuinely useful to the developers and also to invite their experience and expertise into the project.

For these reasons, we have used the Double Diamond design process [13] (Figure 1) to structure the work. The double diamond consists of two consecutive steps both consisting of one exploratory, i.e., divergent, and one converging phase. These phases live inside a context of Engagement and Leadership which makes the importance of involving the community explicit. These two steps form two "diamonds". The steps in the project and how they relate to the phases in the process can be seen in Table 1.

In the first diamond, the process starts with the challenge. The design project's purpose and starting point. From here, through the "Discover" phase, the purpose is to deepen the understanding of the challenge. This is a divergent phase, expanding outwards to discover more and more about the challenge, its characteristics, the people affected, the limitations, and use cases to work with. In this process, it is important to refrain from self-censoring or jumping to conclusions based on your prejudices about the problem.

The second half of the first diamond, the "Define" phase, uses the understanding of the challenge obtained in the Discover phase, narrowing it down to a clear problem definition. The definition needs to be a real and urgent problem for the intended users and manageable to tackle within the scope of the project.

The first phase of the second diamond, the expansive "Develop" phase, is an exploration of the solution space for the previously defined problem. In practice, this means developing as many different solutions to the problem as possible. Rough prototyping can take place in this phase to describe the ideas more visually.

In the second half of the second diamond, the "Deliver" phase, the purpose is to reduce the scope down to a single final design. Here the team can iterate through a variety of methods. Creating prototypes, evaluating prototypes, and finally refining the prototype to arrive at a final design choice.

Table 1: Design process steps and participants.

Discover phase		
Sep 2023	Literature review	1st author
Sep 2023	Developer meeting	5 developers
Oct 2023	Focus group	5 developers
Define phase		
Nov 2023	Definition workshop	3 faculty members
Nov 2023	Problem statement	1st author
Develop phase		
Nov 2023	Co-design workshop	5 developers
Dec 2023	Conceptual design	3 faculty members
Deliver phase		
Dec 2023	Prototype development	1st author
Dec 2023	Prototype persona verification	3 faculty members
Jan 2024	Prototype evaluations	7 developers
Jan 2024	Prototype refinement	1st author

2.1 Discover phase

In the Discover phase, we performed a literature review, a developer meeting, and a focus group (Table 1). The literature review was intended to orient the study about problems in code review as found by current research [17, 19, 23, 24] and give us an idea of what and where to explore for **RQ$_1$**. After this, we held a first developer meeting to introduce the project to five professional developers participating in the study. During the meeting we got to know their backgrounds, the teams' code review processes, and collected informed consent for participating in the study.

The participating developers work at a medium-sized embedded software development company. Their prior experience ranges from over 20 years for one system architect, to 2-3 years for some of the software developers. They work in 4 different teams, which all have established code review practices. All new code in the teams undergoes code review by usually two other developers, who both need to approve the change before it is merged.

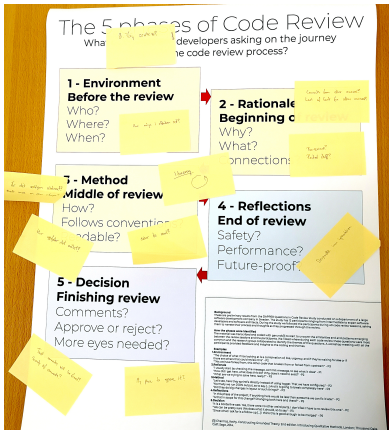

Figure 2: Post-it notes from the focus group.

In the next step, we arranged a focus group [18] with the same five developers. The output from the literature review and the developer meeting were used as input to create a design brief and an interview protocol for the focus group (Appendix A). The topics for the interview questions were aiming to explore $\mathbf{RQ_1}$ and $\mathbf{RQ_2}$. We first explored the practices the developers used when reviewing code (i.e., "Do you read the files in a code review one time or multiple times?"), and then experiences reading code in an IDE (i.e., "How is the experience of reading and understanding new code in other tools, environments or situations?"), and finally comparing the two (i.e., "How is the experience different compared to code review in Gerrit?")

The problems and challenges discussed in the focus group were gathered as post-it notes on a big sheet of paper (Figure 2) and sorted into categories depending on if the challenge usually occurred before, in the beginning, middle, or end of looking through a new code review. The session was held in Swedish and recorded for later transcription and translation.

To balance the participation in the meeting, the facilitation was done in a way where every participant got the same time to talk about their perspective and then with a discussion where everyone could contribute before moving on to the next person's perspective. See results in Section 3.1.

2.2 Define phase

The Define phase consisted of a workshop in the research group where the sorted and categorized challenges from the focus group were described based on potential impact and feasibility to prototype within the scope and time limits of the project. Based on the description, the group prioritized the challenges to conclude which problem areas to move on with. After this, the first author put together a problem statement as input to the next phase. The results from this process are described in Section 3.2.

2.3 Develop phase

In the Develop phase, we conducted a co-design workshop [22] where five industry practitioners (four who also participated in the focus group, plus one more developer from the same company but a different development team) co-created different solution designs. The reason for choosing a co-design workshop as the method was to deeply involve the developers in designing solutions to improve their own working tools, in accordance with $\mathbf{RQ_3}$.

The first author facilitated the co-design workshop and prepared hand-drawn low-fidelity mockups of the Gerrit user interface (Appendix B) and different kinds of widgets, buttons, and overlays on overhead film. New interfaces and ideas could be created by cutting and moving parts of the interface around. The output of the meeting was documented with photos of the mockups and notes on the ideas and principles behind them.

The ideas from the co-design workshop were elaborated through conceptual design [16] to make them more rich and substantial. Conceptual design is the definition of the metaphors, use cases, concepts, and actions that can be involved in the design. This also included the creation of two personas [11] based on the concerns and challenges emphasized by the participants in the focus group and co-design workshop.

2.4 Deliver phase

The final Deliver phase distilled the solution ideas and conceptual designs from the Develop phase and reduced this to one high-fidelity prototype that can be evaluated towards the problem statement developed in the Define phase. The phase contained 4 steps (Table 1). First, a high-fidelity prototype was created using Figma[1], a tool for creating interactive prototypes of computer interfaces.

The prototype was verified by using the personas and their questions. After some updates, the practitioners from the focus group, co-design workshop, and one additional external developer from a different company were invited to individually test and evaluate the prototype. During the test, the participants did a code review in a private Zoom meeting without detailed instructions or guidance (see Appendix C. All of them reviewed the same code that contained examples of blocks moved both within a file and between different files. Their running commentary and their shared screen was recorded to evaluate the interface's usability and how well it supported the challenge selected in the Define phase. The meetings were held in Swedish and the recordings were transcribed, referenced, and translated into English where needed.

Finally, the design of the prototype was adjusted based on the results and suggestions from the evaluations.

3 RESULTS

The result section follows the structure of the Double Diamond process and the sequence of method steps described in Table 1.

3.1 Discover phase

In the developer meeting and literature review (Section 2.1) several developers and articles mentioned file order as an important factor in code reviews. For this reason, the design brief and questions for the focus group, started out exploring this problem area.

> *Design brief.* When doing a code review, the developer often has to read the files multiple times because they are presented in an order where early files are not understood until after reading files further down in the review. Design questions:
>
> (1) In what ways could reading the code in a review be improved so that fewer passes or even a single pass through the files in the review would be enough to understand the changes?
> (2) Are there ways to improve the ordering or let the author convey more of the narrative when sending the code for review?

However, during the meeting our participants raised six different problem areas that they felt were more frequent and impacting their experience more. The areas are described below:

Diff problems. There are many cases when the diff algorithms break down and require tedious manual comparison word-for-word. For example, if a function is moved within a file from the bottom to the top, or maybe refactored into two separate functions, it will all show up as just deleted lines and completely different added lines.

[1] https://www.figma.com/

This makes it hard to see if the code was only moved or moved *and* modified. Also, if a file is renamed and then changed, or a function is moved into a different file, it is impossible to use the built-in diff tools to compare the code.

Suggestions from the focus group included manually selecting files, lines or blocks, to compare sections that the algorithms themselves don't match for diffing.

> *Participant 3.* "I was working with this today and had to sit with two separate windows and go through it, just like, Control-F in this file and see if I find it. Is it added? Is it completely new? Or is it just moved from further down in the file?"
>
> *Participant 1.* "Do we want a more semantic diff? Where you can kind of say that this has been extracted from over there or it has been moved between here and there?"

Finding similar but unchanged code. It might be the case that a code base contains several similar snippets of code and that a code change should affect all of them. In the code review tools, this is difficult to verify. There might be a forgotten snippet left in an unchanged file that is never even shown in the review. It would be helpful to have a tool to find similar code that maybe also should have been changed or looked at.

> *Participant 1.* "It is one of the things that are easiest to miss during a review, regardless of reviewing a document or code. It is like, you only look, think, and look at what *is in the diff*. Not what *should have* been there."

Lack of navigation. Lack of navigation in code review tools causes problems, such as making it difficult to go from a variable's or function's use to its definition, or finding all uses of a variable or function. In IDEs such as VS Code, this kind of navigation is easy and commonly just one or two clicks away, but the same convenience is missing from code review tools.

> *Participant 4.* "I mean, say that you could just press it and «yeah, you have 5 references here» and then you see that, yeah, but the reference down there is not changed in this commit. Why? Then it would be very fast to get to that insight."

Ordering of files. When changes are big and spread out over many files, the alphabetic ordering of files in the code review tool is essentially random, in regards to how the code should be read and understood in the best way. Suggestions for how to address this problem from our participants include placing generated code last or placing the tests last. The uploader could also draw a path through the change with commentary for each file, to clarify the story told by the code under review.

> *Participant 4.* "You could make it easy for yourself and just, like, let the person uploading the review decide or give a suggestion for an order. Then people can choose to go back to their own order, but you can say, kind of, that *I* suggest you look at it in this way. Then you can do it in call stack order if you like that.

Overlays and annotations. The continuous integration (CI) pipeline used by our participants already includes support from running a wide variety of testing and code analysis tools, but these results are disconnected from the code. In the best case, they are shown as a pass/fail stoplight in the code review tool with a link to the full logs. It would be useful to show these results as inline overlays on top of the code. To see, for example, test coverage, linter warnings, execution traces, loops with frequent execution, failing tests, etc. The overlays need to be easy to select and toggle on and off so that the user interface stays easy to use. For adoption, it needs to be easy to integrate the results from the CI without modifying the linters, tests, etc.

> *Participant 5.* "It would have been nice to have a code coverage overlay because then you would have been able to see that (if more tests were needed) in a completely different way."

Unchanged files. Finally, it was discussed that unchanged files are not shown at all in code reviews today. It would help to have a way both to find and navigate to unchanged files and also write review comments in them. There may be places in unchanged files that have not been changed, but that should have been changed, or that affect some parts of the changed code.

> *Participant 5.* "But if you don't want to be marked as the uploader, you have to do it (commenting on an unchanged file) through URL-hacking. (...) NN does it fairly often and I do it sometimes when I realize that there is a change in a nearby file that should have been there."

3.2 Define phase

When considering the problem areas discussed in the focus group, we made a first selection of problems that can be addressed by collaborative design (**RQ$_3$**). This selection removed *Lack of Navigation*, *Ordering of files* and *Finding similar but unchanged code*, since these problem areas would have put more of the focus on deeper code analysis instead.

The three remaining problem areas, *Diff problems*, *Overlays and annotations* and *Unchanged files* are all very interesting areas to explore under the scope of interaction design. In *diff problems*, you want an interface that is flexible in choosing the blocks to compare. It should also be intuitive to use to quickly make comparisons, and at the same time not get in the way of the classical code review interface.

For *overlays and annotations*, completely new concepts of layers would need to be designed and introduced in a way that fits well into the existing code review interface. It needs space for rich information and at the same time needs to be easy to navigate and turn on and off.

To create a design solution for *unchanged files*, the existing interface for presenting and commenting in the files could largely be reused. The challenge is rather in the navigation and to make it clear that the unchanged file is outside the changed code.

In the end, *overlays and annotations* was estimated to be too large for the scope of this project and that *diff problems* was more important and had a greater impact on the quality and ease of reviewing than *unchanged files*. Because of this *diff problems* was chosen as the problem area to explore during the next phase.

3.3 Develop phase - co-design workshop

During the co-design workshop (Section 2.3), our industry practitioners were given hand-drawn cut-and-paste prototyping kits. With these, solution suggestions for the problem of comparing code, that the diff algorithms do not detect as moved, could be created and discussed. Two guiding principles, which should always be present, and three separate solution ideas were formulated:

***Principle 1**: Show context.* One important principle is to always show context for both blocks. The context lines should be syntax-highlighted and displayed in a muted way but still show where the two blocks were found originally.

***Principle 2**: Review comments while comparing blocks.* Since it will often be during these more detailed comparisons that ideas or comments about the code will be found, it is an important principle to be able to write comments right there and then. Maybe it should also be possible to view and read previous comments from other reviewers or the author, even if these were not written in the new comparison views.

***Principle 3**: Support comparison within and across files.* The *diff problems* discovered and defined in the previous phases can occur both within the same file and across different files depending on the types of changes and refactorings done. Support for both these cases is needed to get the most benefit from the tool.

***Principle 4**: Integrated in code review environment.* There are existing software that can do comparisons of any code blocks or texts that you choose such as Meld[2], KDiff3, git diff, etc. The issue with them is that it requires the reviewer to either check out the code locally or copy-paste the code blocks they want to compare into another window. This could switch them out of the code review task [23] and would require extra steps and time. To make reviewing faster and easier the tool needs to be integrated into the code review environment.

***Idea 1**: Scroll-lock one side.* One solution idea that came up was to be able to scroll-lock one side of the diff view and then scroll only the other side, to be able to align code blocks that you want to compare so that they are next to each other. In this way, you would not have to select any lines for a block and would save the extra work and navigation of opening a new Gerrit tab and trying to place that next to the code you want to compare.

The idea is to keep the coloring and the diff the same, and just change which lines are shown next to each other. Probably some kind of snapping at line alignments would be nice. A further improvement could be a feature to mark a segment and then tell the view to scroll through matching segments on the other half.

***Idea 2**: Switch comparison base.* Another idea is for the case of comparing blocks, or whole changes, across different changesets, for example after a revert and re-submit with changes. Here, the proposed solution is to let the user switch the base commit-id to compare against.

Code that has changed independently, by other commits, should be hidden. Only changes in the diff are highlighted so it becomes easy to compare what the difference is between the old faulty code and the new suggested changeset.

Being able to input a git commit-id manually could be a first step, with a possible extension of automatically finding suitable comparison bases that have very similar diff towards the main branch.

***Idea 3**: Workspace for blocks.* To have a workspace where interesting blocks can be placed as they are seen in the code, such as a sidebar or a drawer, is another interesting workshop idea. These blocks could then be brought up and listed in a modal dialog to get an overview of all the interesting blocks. From here the user could compare them to each other, or search for other similar blocks to compare them to.

3.4 Develop phase - conceptual design

From the sketches and discussions during the co-design workshop, we developed a conceptual design meant to capture and enrich the metaphors, concepts, mappings, objects, and personas that can be involved in the design.

Figure 3: Developed analog film on a lightboard. From 'Museum of Obsolete Media', used under CC BY-SA 4.0.

Metaphor: Lightboard. In analog photography, a lightboard is a flat luminous surface where the developed film can be previewed and frames compared to each other before choosing which to enlarge onto photo paper (Figure 3). It can also be used for drawing, to copy or compare art drawn on paper. In this project, the lightboard is a metaphor for a work surface where pieces that you need to illuminate or inspect can be kept and compared.

[2]Meld is a desktop visual diff tool available for many operating systems https://meldmerge.org/

Concept: Changeset. The changeset is the central concept in Gerrit and is what you are approving or rejecting in a code review. It is submitted by an author and contains a commit message written by the author to describe the contained changes.

Concept: File. A file is part of the changeset and has a name, a path, an old and one or more new versions. The diff view in Gerrit can show the comparison between any pair of selected versions for the same file.

Concept: Line. A line in a file within the changeset. Line numbers and contents might not correspond between versions of the same file, so a line only makes sense as a concept when referring to a specific version.

Concept: Blocks. A block consists of one or more consecutive lines of code in a file. Since it is built up of lines, it also must refer to a specific version of the file.

Concept: Diff view. A view to show differences between two text-based contents. One content is designated as the older one, and the other as the newer one. Differences can be shown interleaved or in a side-by-side view.

Concept: Matches. Two blocks that either have a similarity score above a certain threshold or that the user has manually selected.

Action: Select block. Select a block by selecting lines on either the old or the new side of the file diff view.

Action: Compare blocks. Open up a comparison of a match.

Action: Comment on match. While looking at a diff view of a match, write review comments.

Action: Read comments in context. When seeing a review comment in a file, open up the match in a diff view to see the same context as the comment was written in.

Persona: Willow. Willow is new to the team and inexperienced with the particular code base. They have some experience in general software development and code review from education and previous projects, but not seasoned enough to feel super secure in a new code base and environment. Willow is part of a small team with five colleagues, two of whom are on a similar level and three who are more experienced, especially in this particular codebase. Their process is mutual peer reviews where at least one, preferably two, developers should look at and approve new changesets. When reviewing code and finding whole functions or blocks that have been removed, Willow often asks the following questions and would like the design to help them answer them:

- Where did this code go?
- Was this block deleted or moved?

Persona: Raven. Raven has experience with many different software projects in a range of teams, tools, and programming languages. In the project she is working on, she knows most of the codebase by heart and is aware of most of the interactions and intricacies in how it interweaves with its environment. Raven is part of a small team with four colleagues, two on a similar level and two less experienced. Their process in the team is peer code review where one other developer should look at and approve all changesets. When finding files, functions, or blocks that have been moved, she often asks:

- Is this moved block identical to the original?
- Why was it moved?
- How does moving the code affect the surrounding code and projects?

3.5 Deliver phase - first high-fidelity prototype

To converge the ideas and concepts from the Develop phase we focused on **Idea 3**. **Idea 1** was discarded since it would be hard to support comparison across files intuitively with just scrolling. Also, it would require the reader to compare manually line for line without diff coloring. **Idea 2** was also discarded since it would need a functional version of **Idea 3** to start with so that wherever the blocks come from (same patchset version, other versions, or other changesets) they could be compared easily and intuitively.

The prototype[3] (Section 2.4) simulates both source code that has been moved and then modified within a single file, and code that has been broken out into a new file and modified in the process. The prototype is designed as if being fully integrated into Gerrit [**Principle 4**].

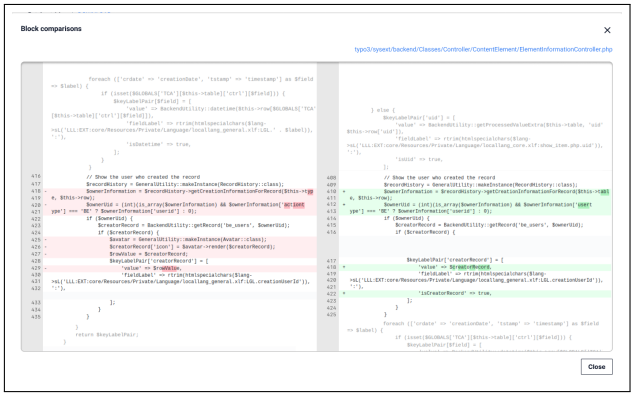

Figure 4: Modal dialog for detailed comparison of blocks.

Feature I: Comparison modal. The comparison modal (Figure 4) shows a detailed diff view between two blocks of code with intra-line markings to highlight the changed parts. The context around the selected blocks is shown without background colors and with lower contrast to make it clear that it is not part of the current comparison [**Principle 1**].

Clicking on the row numbers opens up a comment text field, so the reviewer can write directly in the context that made them notice an issue or questions [**Principle 2**].

Feature II: Comparison within a single file. If code has been moved within a single file, that is detected and a hint is shown on the line above the moved code (Figure 5). The user can click the link to open the comparison modal with the two blocks loaded.

Feature III: Lightboard toolbar. If the user wants to compare blocks across files [**Principle 3**], an added or removed code block can be

[3] Available at https://figma.com/proto/KZIrsBH8DZ22ZI0B0YD2BC/GBC?hide-ui=1

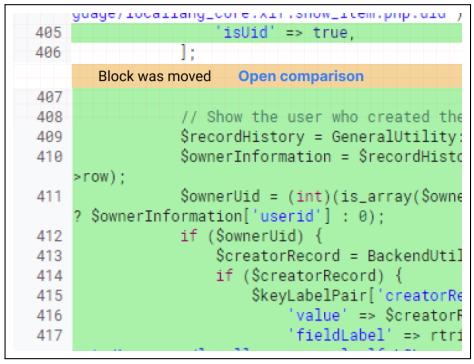

Figure 5: Moved code detected within the same file.

Figure 6: Floating toolbar with blocks marked as interesting for comparison.

added to the list of interesting blocks to compare (Figure 6). This toolbar can be minimized to only take up less space and attention while reviewing and navigating the files, and then expanded to show the list of selected blocks.

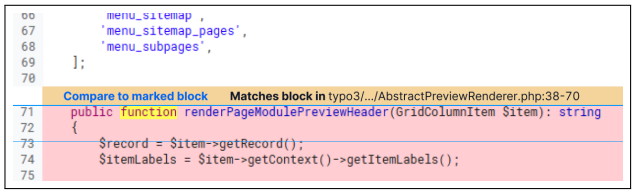

Figure 7: Moved code that matches marked block on lightboard.

Feature IV: Comparison to lightboard block. When navigating through the files under review, the current file is checked for blocks that are similar to any block that is on the lightboard. These blocks show a hint on the line above the code block (Figure 7). Clicking there will open the comparison modal dialog between the block and its closest match on the lightboard.

3.6 Deliver phase - prototype persona verification

The first sanity check of the prototype was done by the research group by checking if the questions and use cases described by our two personas, Willow and Raven, could be answered and performed using the flow in the prototype.

Willow: Where did this code go? This question is answered by the move-detection and the headings that come up over a block [Feature II, Feature IV] and allows Willow to compare it to similar blocks in the same file, or files that have been added to their lightboard.

Willow: Was this block deleted or moved? This question is also answered by the move-detection, where moved blocks will have headings over them showing Willow where the block was moved to or from [Feature II, Feature IV]. However, if the block is moved between different files, and the source or destination blocks are not on the lightboard, the heading will not show and it will look the same as if the block was deleted or newly created. Integrating a clone detection [2] tool could make it possible to scan the whole changeset for similar blocks to detect those cases.

Raven: Is this moved block identical to the original? This question is answered by the comparison modal [Feature I]. Here, Raven can see detailed differences between the block before and after it was moved, with intra-line diff markers to highlight changes.

Raven: How does moving the code affect the surrounding code and projects? This question is a bit more complex. The comparison modal [Feature I] should give Raven a detailed view of any changes in functionality, which will help judge the effects on surrounding code. It also shows the context before and after the blocks, so that Raven can look for potential side effects there as well.

Raven: Why was it moved? **Not supported** - The design of the prototype does not give any extra help for this question. Knowing the details of the changes might in the best case give you a hint, but without a clear rationale being stated by the changeset author it is hard to infer it just from the code.

3.7 Deliver phase - prototype evaluation

The prototype evaluation (Section 2.4 and Table 1) showed that all of the 7 participants could complete the code review task, and were able to use the comparison modal dialog to clarify questions that they had about the moved and refactored code blocks without any additional instructions except for the user interface.

One thing that stood out, when analyzing the recorded evaluations, was how positive the sentiment was regarding the usefulness. For example, in the co-design workshop some of the participants commented on how the solutions they used today, e.g., opening two browser windows next to each other and comparing the code line-by-line manually, worked pretty well, but when using the prototype they expressed surprise at how much easier it was to read and validate the changes by using the new comparison modal instead.

> *Participant 3.* "So I am, if anything, yes, positively surprised that it is, yeah, that it feels like it works and is maybe also not a lot of different things that needs to be developed to still kind of make a pretty big difference to the better."
>
> *Participant 1.* "Well, when code has been moved around and when it is so easy to use so you get comfortable with it [...] it is a really big difference. And then, like I said, it makes you able to stay in Gerrit the whole time. You don't need to,

as said, cut and paste into a Meld-window and figure out what happened that way."

One participant also commented that this improvement could potentially change how they write code themselves. Today, they avoid moving and refactoring code in the same changeset since it was so difficult to review. They therefore try their best first to move the code, commit that, send that for review, and only after the move is approved go ahead and also do the refactoring.

Participant 1. "This thing can *really* make the difference between how you today tend to only move code but not touch it. *Then* you make the actual changes. [...] One reason today is that it is hopeless to review if you do, if you don't separate that into two steps."

For the case when code had been refactored into a new file, 3 of the 7 participants needed several passes back and forth between the two files before they understood how the feature worked and how the new file could be marked for comparison and then used to verify the changes by bringing up the comparison modal in the old file.

Participant 6. "But then there was a feature here in the second file, file 2. It said «Mark for comparison». At first, I did not understand what to do with it. Then I understood that I could go to file number 3 and then click on «Compare to marked block»"

Comments on the functionality and interface included that it was confusing sometimes which of the selected blocks were shown on the left or the right side of the modal diff dialog. Also, 4 out of the 7 participants commented that it should be possible to mark either the new or the old blocks for comparison and add them to the lightboard in any order. When comparing in a single file, several of the participants would have wanted a visual marker linking the two blocks that were detected for comparison.

Participant 5. "But I, I think it would have helped to have an explaining text from both sides, absolutely."

Another more general comment from one of the participants was that this feature might mean that you see and read code in a file you have not fully visited in Gerrit yet, so when you finally get to that file it should be marked in some way that these particular lines have already been seen and potentially commented on.

Participant 4. "...so then if I do this review and compare and see that, well, this block looks good, then I am finished with this part. But I am also finished with the other file, I just compared to. So then it could like almost be defined as reviewed. And if I press «Next» here it is nice to know that in some way, yes like, you have already, you looked at this file just now. There is not a lot more to see."

In regards to code review comments written while inside the comparison modal, it would also be important to include a link so that the author could bring up and read the comment in the same context as it was written in so that it would be easier to understand what the comment means and how they saw it.

Participant 2. "But if you click on the comments in the change view do you come to this view (the comparison modal) then?"

One other idea from one of the participants was to be able to click a single link and button and bring up all relevant comparisons for a full file collected in one single modal to save the time of reading through and clicking each correspondence one by one.

Participant 7. "To select several from this page would have been nice so that you can see, just click, like, I want this *and* this and then look."

Overall, the participants thought that the user interface, with links above the blocks, was clear and easy to understand. However, it would take some time to learn which blocks are useful to mark for comparison and get into the habit of doing so while passing through a file. Especially, if they are not completely sure if a matching block to compare to will occur later in the review or not.

3.8 Deliver phase - updates after evaluation

After the evaluation meetings, the prototype was updated to incorporate some of the feedback, in particular:

Consistent placement of old and new versions. If a selected block is originally on the left side, it should be kept on the left side also in the comparison modal and vice-versa if it is originally on the right side. If the user chooses to compare two blocks that are both on the same side, we could place the first selected one to the right and allowing the user to flip the comparison with a button in the top part of the modal.

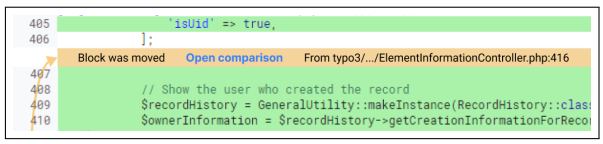

Figure 8: Moved code explicitly marked with arrow, filename, and line numbers.

Explicit location of matches. Every action block with a comparison link lists the file name and the line numbers it would open a comparison with to make it more explicit what you would be comparing to. If the match is in the same file, a line with arrows is drawn linking the two blocks to give a visual marker (Figure 8).

Mark blocks for comparison from both directions. It should not matter if you encounter the old or the new file first when dealing with blocks that have been moved between files. Therefore, the links to mark a block for comparison are made available for both removed and added blocks.

4 DISCUSSION

The goal of this study was to address the three research questions in Section 1 and through the choice of methodology, the execution and the analysis we have found new insights into all of them. Concerning $\mathbf{RQ_1}$, we found that several issues in the code review experience causes frustration for the group participating in the study. These issues are not directly related to the specific code, language, or process these developers use, so we believe that similar frustrations can be found among other developers elsewhere.

By designing and evaluating one possible modification to Gerrit, we also presented one answer to how review tools can be modified to improve the developer experience ($\mathbf{RQ_2}$). While this improvement can be seen as relatively small, we are convinced that even small improvements can have an impact since so much time is spent on code review and because the task is so cognitively demanding [8].

Finally, we explored one way to involve developers in improving their tools ($\mathbf{RQ_3}$), specifically code review tools. By centering developers' experiences and needs from the beginning, involving them both in the design and evaluation of solutions, we estimate that we have reached a prototype that is more in line with processes and flows in code review. Which helps give a larger improvement than its learning curve or distraction.

Overall, as a recommendation and reflection, we think that since the code review tools and processes today are so accepted and ingrained into modern software development practices, we are at a point where completely disrupting and re-designing these tools from the ground up would have a steep hill before reaching adoption and making a difference. One way around such adoption problems, is the way shown in this study; to make small improvements that have the potential to compound and over time, and after continued incremental development, make a big difference.

4.1 Reflection on the method

We estimate that the choice of method was a good fit for this type of study and we also discovered some ways in which the execution of it can be improved. During the co-design workshop, the participants were at first hesitant to directly change the prototype themselves. The first author, as the facilitator, had to draw, cut, and paste together the ideas in the room. Both to get things started, to keep the ideas going, and the solutions evolving.

After discussing this experience during a research group meeting, there were ideas shared about warming up the co-design team by first collaborating on designing something low-stakes and fun, like a celebratory garbage bin for redundant code. When the team has warmed up and gotten used to creating and discarding sketches together it could be easier move on to the real design task and see more confidence in the group to directly change and create prototypes. This would require 2-3 times as much time from the participants, which can be an issue in practice, but could have the potential of getting results that are more creative and more firmly anchored in the whole group.

4.2 Threats to validity

One threat to the validity of this study is that the participating software developers in the focus group and co-design workshop are a fairly small group of only 6 developers. They are also all working in the same company, albeit in different teams. While this focus group size falls within the ideal size of 4-8 participants described by Liamputtong [18], she also recommends conducting several focus groups with the same interview questions until reaching data saturation (i.e., no new ideas or data is being found). However, since the study is aimed at finding *some* (rather than all) possible improvements we believe that ideas and experiences from this group are still valid.

The study is also only valid for the code review tool Gerrit. However, since Gerrit has a large user base, this still means that the results could have broad applicability. Also, other tools in wide use today, such as GitHub and GitLab, are similar to Gerrit and there might be findings here that can be generalized to apply to them as well, but it is beyond the scope of this study to verify or evaluate that.

There is also a risk that the prototype validation has a positive confirmation bias since the developers evaluating the prototype have participated in the co-design workshop that led to its creation. In this way, they might both be more familiar with the concepts and also personally invested in the success of the prototype. To check for this we also did one extra validation with a software practitioner who has not been involved in the project at all and works for a different company than the other participants.

Finally, the prototype evaluation is also exposed to response bias [12]. We tried to counter-balance this by encouraging feedback.

4.3 Directions for future work

Paths to broader adoption and impact. Exploring the prototype of more flexible diffing between blocks further and implementing it as a feature or a plugin to Gerrit (or GitLab or GitHub) would be very interesting and something we would like to address in the future. Further validation with several other teams of industry practitioners would be needed to refine the prototype into something generally useful. Implementing the features of the prototype would also require the integration of clone detection tools [2] to identify blocks to suggest for comparison.

To have any significant benefits compared to already available solutions and avoid context switches for the developers, we feel that the implementation would need to be available directly in the review tool. Therefore it is important to anchor the idea and the implementation in the Gerrit (or GitHub or GitLab) community. This could increase the support for a plugin implementation or possibilities for merging the feature into the tool itself. A first step here could be presenting the prototype and the thoughts behind it to the Gerrit community during the annual Gerrit User Summit.

Future design explorations. The other problem areas and solution suggestions uncovered during the workshops are also viable ideas for improving the experience of code review and reduce the cognitive load of the task while at the same time having the chance of

increasing the benefits of a team's code review process. The idea of overlays or layers of meta-information from continuous integration systems, tests and source code analyzers is particularly interesting to study and explore further.

Increase understanding of cognitive demands. While many papers agree [4, 6, 15, 20] that code review seems to be a cognitively demanding activity, there have not been many studies to measure the cognitive load during code review and compare that to other tasks that are known to be demanding. It would be interesting to do a study integrating EEG measurements [3] or fNIRS [14] with participants doing code reviews of different sizes and also, for example, general problems in math or programming. This could give valuable insight into code review's cognitive demands and guide future exploration into its design.

5 CONCLUSIONS

We used the Double Diamond design process to explore how the Gerrit code review tool could improved. By hosting a focus group we found several problem areas that are common experiences when doing code reviews. The problem of comparing moved and refactored blocks that the built-in diff algorithms don't pick up was chosen for exploring solutions.

A co-design workshop with industry practitioners was held and the prototype created collectively there was then refined to a high-fidelity interactive prototype that could be evaluated in one-on-one testing sessions.

The results show that making these kinds of comparisons has the potential of improving the code review experience both by reducing the mental workload and also give higher accuracy in the comments and the analysis of the code. The user interface of the prototype was also intuitive enough for all of the participants to be able to complete the assigned task without any hints or instructions.

ACKNOWLEDGMENTS

Special thanks to the study participants for sharing their insights, ideas, and creativity. Special thanks to Andreas Bexell and Peng Kuang for inspiration, discussions, and feedback. Thanks to Luke Church for valuable input regarding the method.

The authors would further like to thank the following funders who partly funded this work: the Swedish strategic research environment ELLIIT, the Swedish Foundation for Strategic Research (grant nbr. FFL18-0231), the Swedish Research Council (grant nbr. 2019-05658), and the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

REFERENCES

[1] 2023. Number of software developers worldwide in 2018 to 2024. https://www.statista.com/statistics/627312/worldwide-developer-population/

[2] Qurat Ul Ain, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam, and Bilal Maqbool. 2019. A Systematic Review on Code Clone Detection. *IEEE Access* 7 (2019), 86121–86144. https://doi.org/10.1109/ACCESS.2019.2918202 Conference Name: IEEE Access.

[3] Pavlo Antonenko, Fred Paas, Roland Grabner, and Tamara Van Gog. 2010. Using Electroencephalography to Measure Cognitive Load. *Educational Psychology Review* 22, 4 (Dec. 2010), 425–438. https://doi.org/10.1007/s10648-010-9130-y

[4] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*. 712–721. https://doi.org/10.1109/ICSE.2013.6606617

[5] Farid Bagirov, Pouria Derakhshanfar, Alexey Kalina, Elena Kartysheva, and Vladimir Kovalenko. 2023. Assessing the Impact of File Ordering Strategies on Code Review Process. (2023). https://doi.org/10.48550/ARXIV.2306.06956 Publisher: arXiv Version Number: 1.

[6] Tobias Baum. 2019. Cognitive-support code review tools : improved efficiency of change-based code review by guiding and assisting reviewers. (2019). https://doi.org/10.15488/9164 Publisher: Hannover : Institutionelles Repositorium der Universität Hannover.

[7] Tobias Baum and Kurt Schneider. 2016. On the Need for a New Generation of Code Review Tools. In *Product-Focused Software Process Improvement*, Pekka Abrahamsson, Andreas Jedlitschka, Anh Nguyen Duc, Michael Felderer, Sousuke Amasaki, and Tommi Mikkonen (Eds.). Vol. 10027. Springer International Publishing, 301–308. https://doi.org/10.1007/978-3-319-49094-6_19 Series Title: Lecture Notes in Computer Science.

[8] Tobias Baum, Kurt Schneider, and Alberto Bacchelli. 2019. Associating working memory capacity and code change ordering with code review performance. *Empirical Software Engineering* 24, 4 (Aug. 2019), 1762–1798. https://doi.org/10.1007/s10664-018-9676-8

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

[10] L. Brothers, V. Sembugamoorthy, and M. Muller. 1990. ICICLE: Groupware for Code Inspection. In *Proceedings of the 1990 ACM Conference on Computer-Supported Cooperative Work (CSCW '90)*. Association for Computing Machinery, New York, NY, USA, 169–181. https://doi.org/10.1145/99332.99353 event-place: Los Angeles, California, USA.

[11] Yen-ning Chang, Youn-kyung Lim, and Erik Stolterman. 2008. Personas: from theory to practices. In *Proceedings of the 5th Nordic conference on Human-computer interaction: building bridges*. 439–442.

[12] Nicola Dell, Vidya Vaidyanathan, Indrani Medhi, Edward Cutrell, and William Thies. 2012. " Yours is better!" participant response bias in HCI. In *Proceedings of the sigchi conference on human factors in computing systems*. 1321–1330.

[13] Design Council. 2023. Double Diamond framework for innovation. https://www.designcouncil.org.uk/our-resources/framework-for-innovation/. [Online; accessed 25-October-2023].

[14] Frank A. Fishburn, Megan E. Norr, Andrei V. Medvedev, and Chandan J. Vaidya. 2014. Sensitivity of fNIRS to cognitive state and load. *Frontiers in Human Neuroscience* 8 (2014). https://doi.org/10.3389/fnhum.2014.00076

[15] Pavlína Wurzel Gonçalves, Enrico Fregnan, Tobias Baum, Kurt Schneider, and Alberto Bacchelli. 2022. Do explicit review strategies improve code review performance? Towards understanding the role of cognitive load. 27, 4 (2022), 99. https://doi.org/10.1007/s10664-022-10123-8

[16] Jeff Johnson and Austin Henderson. 2002. Conceptual models: begin by designing what to design. *interactions* 9, 1 (2002), 25–32.

[17] Oleksii Kononenko, Olga Baysal, and Michael W. Godfrey. 2016. Code Review Quality: How Developers See It. In *Proceedings of the 38th International Conference on Software Engineering* (New York, NY, USA) *(ICSE '16)*. Association for Computing Machinery, 1028–1038. https://doi.org/10.1145/2884781.2884840 event-place: Austin, Texas.

[18] Pranee Liamputtong. 2011. *Focus Group Methodology: Principle and Practice*. SAGE Publications Ltd,.

[19] Laura MacLeod, Michaela Greiler, Margaret-Anne Storey, Christian Bird, and Jacek Czerwonka. 2018. Code Reviewing in the Trenches: Challenges and Best Practices. 35, 4 (2018), 34–42. https://doi.org/10.1109/MS.2017.265100500

[20] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. 2018. Information Needs in Contemporary Code Review. *Proceedings of the ACM on Human-Computer Interaction* 2, CSCW (Nov. 2018), 1–27. https://doi.org/10.1145/3274404

[21] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern Code Review: A Case Study at Google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice* (New York, NY, USA) *(ICSE-SEIP '18)*. Association for Computing Machinery, 181–190. https://doi.org/10.1145/3183519.3183525 event-place: Gothenburg, Sweden.

[22] Marc Steen. 2013. Co-Design as a Process of Joint Inquiry and Imagination. *Design Issues* 29, 2 (April 2013), 16–28. https://doi.org/10.1162/DESI_a_00207

[23] Emma Söderberg, Luke Church, Jürgen Börstler, Diederick Niehorster, and Christofer Rydenfält. 2022. Understanding the Experience of Code Review: Misalignments, Attention, and Units of Analysis. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering 2022* (New York, NY, USA) *(EASE '22)*. Association for Computing Machinery, 170–179. https://doi.org/10.1145/3530019.3530037 event-place: Gothenburg, Sweden.

[24] Emma Söderberg, Luke Church, Jürgen Börstler, Diederick C. Niehorster, and Christofer Rydenfält. 2022. What's Bothering Developers in Code Review?. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice* (New York, NY, USA) *(ICSE-SEIP '22)*. Association for Computing Machinery, 341–342. https://doi.org/10.1145/3510457.3513083 event-place: Pittsburgh, Pennsylvania.

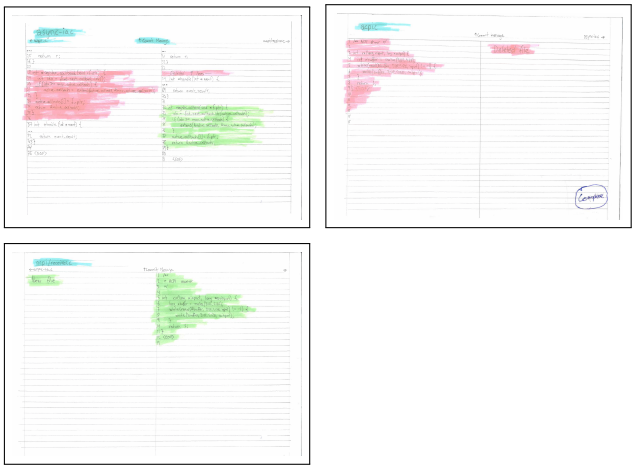

Figure 9: The three base views for the co-design workshop.

A FOCUS GROUP DESIGN BRIEF

Agenda for the focus group meeting (Section 2.1).

A.1 Goal

Collect information about challenges in understanding the changes in code reviews that span multiple files.

A.2 Warm-up questions

- Which is the best tool for code reviews? Gerrit, GitHub, or GitLab?
- Which is your favorite IDE? Vim, Emacs, VS Code, others?

A.3 Main questions

- Do you read the files in a code review one time or multiple times?
- Why?
- How do you choose which file you read first?
- How does it affect your review what kind of file that you read first? (API, test, etc.)
- In what way?
- What is your experience of reading and understanding new code in other tools, environments, or situations? (IDE, pair programming, explained by a colleague, on paper in a book, etc.)
- In what ways do they work?
- Where do you start reading in those cases?
- In what ways is that different compared to code review in Gerrit?
- Are there features and support from there that could have been applied also in code review tools?
- Is there anything else you would like to tell?

B CO-DESIGN WORKSHOP

The material for the co-design workshop consisted of three pre-drawn base views (Figure 9), blank paper, overhead film, scissors, tape, and pens in different colors. The base views displayed some of the use cases brought up during the focus group. The participants created simple prototypes and showcased ideas by overlaying new components on top of the base views. A small example can be seen in the top-right image where the first sketch of the "Compare" action overlay is drawn on overhead film and then taped onto the base view.

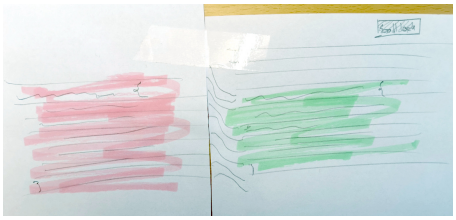

Figure 10: First sketch of comparison modal.

In Figure 10, we show an example of a rough sketch made during the workshop to show the level of fidelity that we think is feasible and appropriate for the setting. This is the first sketch of the comparison modal and shows the idea of lining up and comparing blocks that are in different places in the original Gerrit diff view.

C PROTOTYPE EVALUATION QUESTIONS

C.1 Task

- You are welcome to think aloud, but I will not respond or give hints until after the task.
- Perform the code review.
- Find changes in any moved code.

C.2 Interview questions

- What was your experience of trying this prototype?
- How often do you encounter the situation the prototype tries to aid in?
- How much would a fully functional version of the prototype help with the problem?
- What are the solution's greatest weaknesses?
- What are the solution's greatest strengths?

Received 2024-02-08; accepted 2024-02-26