P10 - PROJECT

Jesper Fyllgraf
Ruben Henrik Mensink

Early Open Source Development: Reports on Bootstrapping an Open Source Project with Distributed Users

Supervisor:
Peter Axel Nielsen

Department of Computer Science
Aalborg University
Denmark

19/06-2020

This page was intentionally left blank

AALBORG UNIVERSITET
STUDENTERRAPPORT

Department of Computer Science
Aalborg University
Selma Lagerlöfs Vej 300
9220 Aalborg East, DK
Telephone: +45 9940 9940
Telefax: +45 9940 9798
http://www.cs.aau.dk

Title:

Early Open Source Development: Reports on Bootstrapping an Open Source Project with Distributed Users

Theme:

Master Project

Project Period:

03/02-2020 - 19/06-2020

Project Group:

SD108f20

Authors:

Jesper Fyllgraf
Ruben Henrik Mensink

Supervisor:

Peter Axel Nielsen

Total Pages: 70 + frontpage
Appendix: 4
Completion Date: 19/06-2020

Abstract:

The subject of this master's thesis is concerned with three related areas of interest; open source development, software engineering processes and tools, and the domain of qualitative research and how these researchers use tools for qualitative coding. The purpose is to explore each area and their relation, through developing an open source real-time collaborative coding tool. We report on findings made throughout three phases of distributed development, where each phase is concluded with a demo. We conclude that implementing a real-time collaboration tool as a web application can be done with a distributed architectural pattern, where the server side uses the socket.io component to ensure consist data between clients. For early stage low-overhead, rapid feedback, we use UserReport, a tool compatible with all web applications in order to get feedback from an anonymous group of users. The primary results are summarized in the implications: *Early technology familiarization requires considerable time investment and can be difficult to estimate* and *A distributed team needs heavier methodology weight than a collocated team, and adjusting it requires intentional efforts.*

The contents of the report are free to use, yet any official publication referencing this requires an agreement with the authors of the report.

Preface

During the development of our application we have received useful feedback from a group of anonymous users. Without it we would not have been able to develop a greater understanding of the domain of qualitative coding, nor would we have been able to evaluate our overall methodology and tools used in the development. Your feedback is much appreciated. We would also like to thank our supervisor Peter Axel Nielsen for constructive criticism throughout the whole semester and for support of the overall development process.

Table of Contents

Chapter 1

Introduction

With software such as LibreOffice, the Linux operating system, VLC Media Player and much more, the open source software development model has proven to be very capable of producing high quality software. A great amount of research has been made on the subject of open source development, analyzing and describing its characteristics. It is a phenomenon which differs a great deal from closed industrial software development processes; the projects have no allocated resources, the developers are volunteers motivated by non-material values, there are no restricting time-frames on delivery and everyone can access and contribute to the source code. From a systems development perspective, these characteristics make an interesting domain to investigate, but is it possible to understand the processes of such a seemingly chaotic model?

In this thesis, we investigate the earliest stages of an open source project. Specifically, we are reporting our experiences on bootstrapping an open source project by developing a piece of software from scratch, using agile development as well as suggestions from literature on open source development. To clarify, the source code was made public, and we were primarily developing distributed, but there were never additional contributors apart from ourselves. Thus we are not reporting on an actual open source project, but on how to possibly bootstrap one.

Open source is not the only area of interest in this thesis. Since we are in the general domain of systems development, we are also interested in the development process. As this is the bootstrapping of an open source project, and not an open source project initially, we need to structure a process for ourselves. Additionally, since we are only two contributors, and because we previously did a project on agile development, an agile methodology makes sense for this thesis. A third area of interest is the domain and its users. The domain we chose for our application concerns qualitative data coding [1]. While several applications for this kind of data analysis already exist, research has has described an incentive for an application with better collaborative features [2][3]. For that reason, and the fact that it is relevant for systems development research, this domain is interesting for our thesis. As close collaboration with users is a priority in

agile software development, we are also interested in the users of the domain and on getting their feedback.

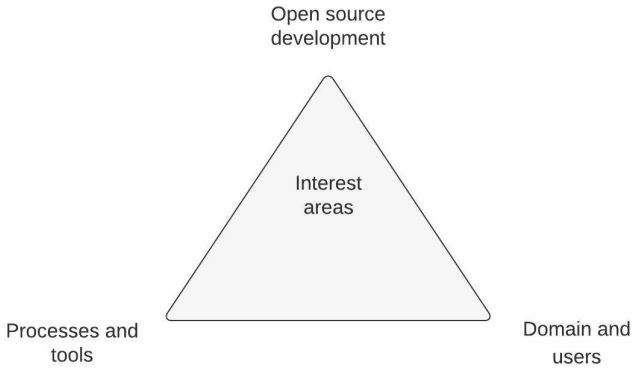

Figure 1.1: Illustration of the three areas of interest contained in this report

To summarize, this thesis has a threefold nature, which is illustrated in figure 1.1. As the figure shows, each of the areas are connected to each other, representing their relation: We are developing an application for a specific *domain* with related *users*. During this development, we are using agile *processes and tools* appropriate to our situation, which we modify and report on incrementally. These processes and tools are rooted in literature on both agile development and open source development. At the end of the development process, we will discuss the processes and tools specifically in relation to *open source development*; are they applicable in an open source project with more users and open source code? What do we need to adjust to be able to extend the project to be open source?

The remainder of this chapter will introduce each of the areas of interest, and give an overview of their characteristics relevant to our thesis.

1.1 Open source

The purpose of this section is to introduce *open source*, from where the term originates, the culture that surrounds it, as well as to introduce the difference of philosophy behind *open source software* (OSS) and *free/libre open source software*, which is abbreviated both as (FOSS) and (FLOSS).

The technical meaning of the term is well known, which is that code can be in either human-readable form (the source code) or in compiled form (machine code). *Open* refers to the fact that the source is available to the public, as opposed to *closed* source. According to Haff [4], sharing the source code was a necessary practice in the 50's, 60's and 70's, since software was viewed as a way to gain access to the *real* product

(the hardware). Since software was written for specific hardware architectures, one would often need to change the software for new purchases, adding features or fixing bugs. To do this, one would need the source code, and it was due to such problems that lead to the sharing of source code between users to become a culture. The values that developed in this culture, is the idea that if someone had resolved a problem, that it was one's duty to share it with others. It was this sharing of the modified source code that lead to communities of people sharing solutions to problems that they had in common. This is the beginning of the *hacker culture*, the origins of which is described in the essay, that later became a book, *The Cathedral and the Bazaar* [5]. According to Raymond [5] hackerdom is a culture and a community, which is centered around technology and that stays connected through the internet (in the early days the ARPANET). They refer to themselves as the *real programmers* and *hackers* and not in the way the word is used now to describe IT-criminals nor are they *crackers*. Hackers are enthusiasts, hobbyists, artists, people who like to tinker with technology. They are problem solvers and thrive on solving difficult technical problems. With these kinds of people, and their intrinsic motivations for creating beautiful software, lie the roots of open source software; where software development is considered more of an art or craft than engineering. The way of thinking that permeates open source communities, seems to be fundamentally different than that of software engineering. In section 1.2 we continue on these differences.

1.1.1 Open source licensing

The reason that software licensing is an important topic, is because without a licence, no one is allowed to use the software that is written, since copyright law is automatically applied upon creation of the software. There are two classes of licences, namely *copyleft* and *permissive*. What distinguishes these licences is that a copyleft licence infers mutual obligation. If we have some code that has been licenced under a copyleft licence, and we use some of that code in another project, and then distribute the new machine code, then we also have to make available the source code and a way for the end user to obtain it. Permissive licenses are a contrast in this sense, since software with these licenses can be modified and distributed (and sold for monetary gain), without having to make the source code available. An example of an open source project is *Socket.io* [6], which is licensed with a permissive license, specifically an MIT licence. If we decide to use the socket.io component in our software, and we want to distribute and/or sell our software, because it is licensed with a permissive MIT licence, then we do not have to make the source code available.

In order to choose between a myriad of licences, the *OSS-watch* has developed a tool to pick an appropriate open source licence [7].

The notion of free software originates from Richard Stallman, who developed the *GNU* (Gnu's not UNIX) manifesto (1984), the Free Software Definition (1985) and the

GNU public license (GPL, 1986) [4]. As software was becoming more commercialized his goal was to make sure that one was able to distribute and change the software, as one sees fit. The words *free*, and *libre* (the french word for free), are used to describe such software. Thereby the meaning of the word free, is as in *freedom of speech* and not free as in *free beer*. If we want to say something about the price of software, we use the word *gratis* as in free of charge. For a piece of software to be considered free/libre it must adhere to the *four essential freedoms* [8]:

- The freedom to run any program as you wish, for any purpose.
- The freedom to study how the program works, and change it so it does your computing as you wish. Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help others.
- The freedom to distribute copies of your modified versions to others. By doing this you can give the whole community a chance to benefit from your changes. Access to the source code is a precondition for this.

There are many nuances to these four freedom definitions, but for our purposes these are the foundation for considering whether a software license can be considered *free* or not.

There are various categories of software in terms of legality. We refer to figure 1.2 with an overview of libre software and proprietary software.

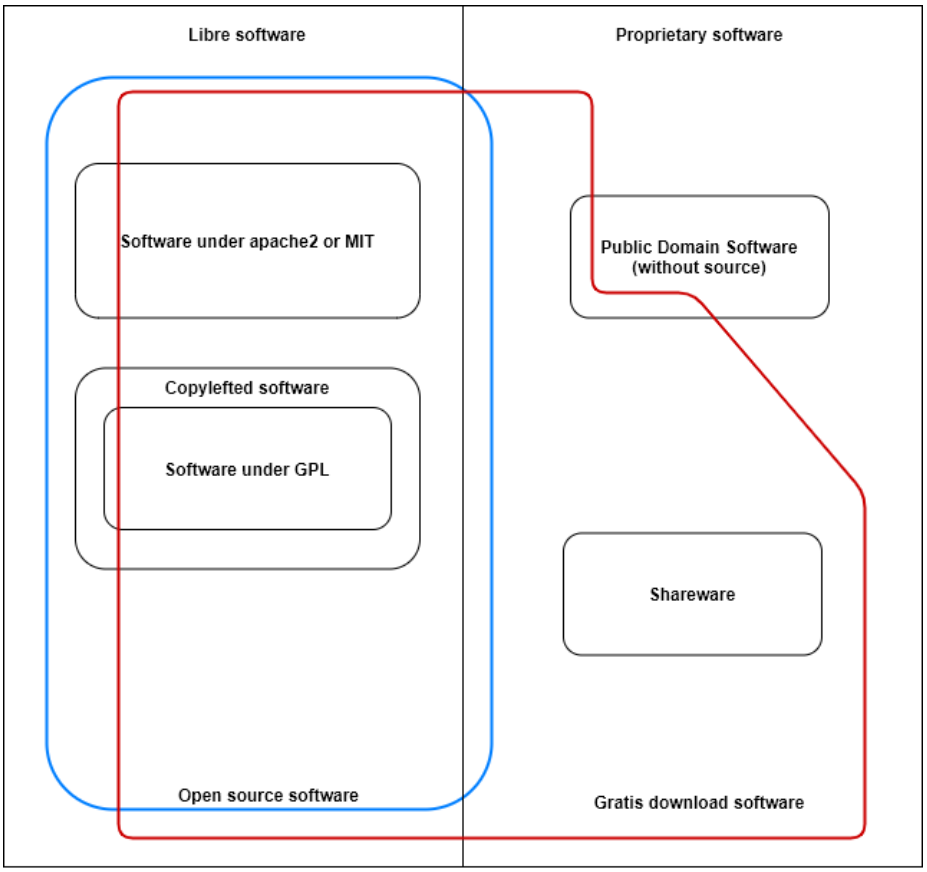

Figure 1.2: An overview of libre and proprietary software. The figure is built around inclusions; to illustrate that for example, all software under a GPL licence belongs to the greater class of copyleft licenced software. According to the GNU homepage [9], there are some cases, depending on the terminology used, of libre software that is not open source, however the examples are few. The blue container for open source software, shows first of all that all software under copyleft licences or permissive licences are considered open source, but also that there is open source software included in proprietary software. The red container shows that much open source software is *gratis*, however, it also indicates that it does not have to be. It also shows that some public domain software is available for gratis download; the authors presumably mean that such software can either be acquired through a trial period or illegally. Figure taken from [9] with minor modifications and exclusions.

1.2 Process and tools

The topic of software engineering spans a wide range of topics [10] and stands in a stark contrast with the Bazaar approach mentioned in 1.1. Especially in the early days of proprietary software development, where IEEE standards were rigorously enforced and the overall approach to software development was determined by plans and documents. A paradigm shift occurred in 2001 where the release of the agile manifesto [11] changed

the old way of thinking. A change of culture and values towards a more flexible style of development, where individuals, working software collaborating with customers and accounting for change are prioritized.

Software engineering is thought of as a set of activities. According to Sommerville [12] these activities are *software specification*, *software development*, *software validation* and *software evolution*. In a plan driven approach, the waterfall model takes these activities and views them as phases of development; where a single phase is executed at a time. *Incremental* development is an appended approach, where parts of the system are delivered to an end user, but still in a structured manner. The agile approach also includes these same activities, yet development occurs in *iterations*, and with frequent releases to the customer; meaning that all activities are touched upon even in small iterations.

Another aspect of software engineering are the tools that are used. In the book *Software Engineering Body of Knowledge* (SWEBOK) [10] Bourque and Fairley discuss, rather meticulously, all aspects of software engineering; specifying all possible activities and sub-activities that software engineering can undertake, also mentioning tools for almost each activity.

For software requirements, we have tools for both *modeling* requirements, but also for managing the *changing* requirements, where tools can provide support for tracing these changes. For designing software components and architectures we have methods and also tools such as *visual paradigm* [13]. Different software engineering tools span as wide as the activities themselves; tools for writing the code, such as compilers and text editors, to tools for managing the methodology.

While these overarching paradigms decide the approach; e.g either driven by plans or by responding to change, these are not detailed enough to be considered a *process*. In his book *agile software development* [14] Cockburn uses the word *methodology*, to describe "how software developers work together" or their "working agreement". A methodology then describes a number of elements, that may or may not be written down, that specifies how a team works together. In Cockburn's view a methodology captures thirteen such elements, which ranges from what tools the developers use in the project, how long increments last and to what different roles there should be in a development team. A methodology is therefore important in software engineering, since it supports the development, such that people can work together efficiently. How many members a team consists of and how many teams are attached to a project varies greatly, as well as the spatial conditions of the teams.

Cockburn distinguishes between a set of different spatial conditions, where *virtual* is the general term used to refer to teams not being collocated. He describes three different spatial conditions: *multi-site*, *offshore*, and *distributed* development. Multi-site means multiple independent teams working on different subsystems, offshore refers to having *designers*, *programmers* and *testers* in different locations, typically in another country,

and distributed development being that members of the same team reside in different locations.

1.3 Domain and users

The domain we have chosen to work with throughout this project is that of qualitative research; specifically a process used by experts in this domain. This is the process of *coding* [3], which we cover in section 2.4.

A part of working with software development is working with the users of that software. The way that is achieved is partly determined by the processes and tools used. The purpose is then to achieve insight into the complex knowledge domain that these users work in and to gain an understanding their needs in a tool in order for them to be more efficient at doing what they want to do; to do qualitative research. In a plan-driven view of software engineering, such understanding is gained through having the users specify requirements to a system that would be beneficial to them. In an agile approach we focus on continuous interaction with the users, where rapid feedback is sought after.

In our case of open source development, we interact with our users through a tool; where the choice of such a tool can limit how much understanding we gain of the domain. In this project we only had a few users; if we had many more this would have an effect on our process and which tools we would use, since we would have to accommodate many users by adjusting our process and use different tools. Therefore we have a relation between process and tools, and domain and users, where they both affect each other.

Since this is bootstrapping of an open source project, there are more reasons to consider a user as a valuable resource. According to Raymond [5] if the application solves some problem then there will be enthusiasm for it while confirming the need for the application itself. Secondly, it is possible to have users become co-developers, which in the long term would be one of the goals of an open source project. There are some challenges here that any open source project faces; the challenge of interacting with users in a distributed setting and fostering potential future contributors.

Chapter 2

Related work

While the three areas of interest introduced in chapter 1 form a large area of research to consider, we narrow the scope by presenting relevant work related to our project. This research will be used to give an overview of these domains, as well as to argue and discuss the present project. We present some of the relevant themes in each of the areas, what the challenges are, and what the literature suggests in order to accommodate them.

The first section will describe agile methodology in general, and specific related literature we used in a previous project. The second section will present work related to open source software development and challenges we should consider. The third section presents research on getting feedback from geographically distributed users and the last section describes related work on the domain of qualitative coding.

2.1 Agile methodology

Using an agile methodology has almost become a de facto in software development. Naturally, an intangible amount of literature, including teaching material and research, has been made on the subject. In general, the paradigm of agile development includes a focus on prioritizing people and communication over bureaucratic processes and tools. It has a focus on customer collaboration, getting iterative feedback, and advocates embracing change, i.e. accepting the fact that you cannot predict and plan a development process at the beginning, but you should do it iteratively.

One of the people who has made noticeable contributions to the field of agile methodologies is Alistair Cockburn. In a previous semester on our masters, we made a qualitative case study, where we interviewed employees of a software consultancy company, and made observations on how practitioners balance agile methodologies with more plan-driven approaches [15]. The primary literature used for the project, was his book [14]. Being one of the authors of the Agile Manifesto [11] it is clear that Cockburn is an advocate for agile software development. However, he suggests that in a practical

world, some projects allow for greater agility than others; specifically, one should balance methodology *weight* according to a certain situation. As Cockburn puts it: "The question for using agile methodologies is not to ask, 'Can an agile methodology be used in this situation' but 'How can we remain agile in this situation?'"[14](p. 149). In other words, some projects can be more agile than others, and this should be reflected in the choice of methodology. To understand Cockburn's points, it can be necessary to understand a few of his notions [14](p. 107):

Methodology Size The number of control elements in the methodology. Each deliverable, standard, activity, quality measure, and technique description is an element of control. [...]

Ceremony The amount of precision and the tightness of tolerance in the methodology. Greater ceremony corresponds to tighter control. [...]

Methodology Weight The product of size and ceremony, the number of control elements multiplied by the ceremony involved in each. This is a conceptual product [...]

In short, methodology weight is how large and how well-defined (ceremonial) your methodology is. Agility and methodology weight are not synonymous notions in this context, but there is an important relation between them. The relation is important, because Cockburn's overall point is that when using agile methodologies, your methodology should be as *light* as possible, for the given situation. However, not all projects will have the same degree of agility and depends on a different methodology. A large team with offshore developers will require a heavier methodology weight than a small collocated team. Similarly, a critical project, like developing precise medical applications, will require a more formalized process and higher ceremony than a webshop for a clothing line.

Recognizing the fact that the open source paradigm is very different from industrial projects, Cockburn does touch upon the subject in relation to this theory. He argues that the primary motivators for open source developers are intrinsic values, such as pride of contributing and accomplishment. One key aspect that differs in this unique culture is that all communication should be visible to everyone. In an industrial project, Cockburn suggests that communication breaks down when teams evolve into *upper* and *lower class* societies, creating an "us-them" separation. Similarly, in an open-source project, all communication should be transparent to everyone, promoting a culture supporting that there are only "us". Cockburn does not specifically mention methodology weight in terms of open-source development, but does so in terms of distributed (virtual) development. In short, he suggests that a heavier methodology weight can be necessary to e.g. create essential communication between project participants. One should aim for agile

qualities (or "sweet spots"), such as collocation, proximity to users, experienced developers and so on, however, less agile mechanisms must be used in order to be sufficient when these qualities cannot be reached - for instance when developing distributed.

2.2 Open source software development

The popularity of open source software (OSS) development has definitely increased since the model first emerged, and the open source paradigm is now a serious contender to commercial software development [16]. Software companies are even using resources to contribute to open source projects causing new challenges to emerge. This section presents work related to general methodology considerations of open source projects, as well as specific concerns including quality and usability assurance, affiliated participation and non developer user-feedback.

2.2.1 Open source participation

As the popularity of open source projects have grown, so has the participation of the industry in the OSS communities. Companies not only use OSS code in their products, they are also increasingly inclined to open their own source code. In a qualitative study, Alami and Wasowski [17] examine how companies participate in OSS communities, what participation barriers exist and make suggestions on how to address them.

In terms of *how* companies participate, they observe three types of participation models: passive, active and latent. The passive model is observed when some companies deliberately decide to only benefit in an inbound-only manner, meaning they only use the OSS without contributing to it. Not surprisingly, this is argued to be a concern, adding no value to the community. In contrast, the active model is observed when companies acknowledge that it is possible or even beneficial to actively contribute both pecuniary and non pecuniary to open source projects. The latent model is a compromise between the former two, where companies are delaying their inbound participation to e.g. secure their own economic gains first.

Six barriers of participation are identified with associated suggestions on how to address them. While a main focus of the paper is affiliated participation in OSS projects, some of the barriers are not exclusive to institutions, but also individuals and communities. The first barrier deals with senior management objections. Their qualitative interviews suggest that sometimes the company managers do not recognize or understand the values of the open source concept, and therefore object to an active participation. Similarly, the second barrier deals with concerns of companies' images. Some companies do not want to participate, because they do not want to be associated with potentially bad quality software. The third barrier is about protecting companies intellectual property; i.e. some institutions are afraid of revealing parts of their solutions to com-

petitors. The forth barrier concerns undefined processes and policies. In other words, companies do not have clearly defined internal processes on how to actually participate in such projects and therefore are reluctant to do so. The last two barriers are argued to be relevant in the scope of communities and individuals as well as companies. The fifth barrier concerns the potential high cost of participation. Some OSS projects may reject contributions because they are either irrelevant or because they do not meet the quality requirements. This means time and effort may be wasted, and can be a barrier to participation. The sixth barrier is the challenge of trying to contribute to an unfamiliar system. The *system* in this context is the social order, rituals, norms and practices of the community. Learning such a system can be a heavy investment and thus be a barrier of participation.

2.2.2 Open source methodology

Raymond [18] recounts his experience of running an open source project by the name *Fetchmail*. Firstly, he describes two high-level approaches to open source development, the *Cathedral* approach and the *Bazaar* approach. The idea behind the Cathedral approach is that a small group of experts design, implement and test a system in iterations, and that when they are satisfied with the result, then they release the source code to the public. In the Cathedral approach of development, all communication is reserved to this group and contributions to the project is also limited to these select few. On the other hand the Bazaar approach is focused on transparency, where code and discussions are in view of the public and that anyone can contribute at any time. Raymond, inspired by the Bazaar model of development used by Linus Torvalds on the Linux project, decided to use this approach in order to empirically test software engineering theories. The primary contribution of Raymond's empirical testing is a list of 19 lessons, where the most significant is the following: *"Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone."*. The less formal version of this lesson is *"Given enough eyeballs, all bugs are shallow."*. Raymond was testing the Bazaar approach and this lesson indicates, that when you have a large user base and a plenty of developers *bugs* are easily found and fixed. It is also this lesson that indicates the core difference between these approaches; the Bazaar approach views bugs as problems that are easily fixed when exposed to flood of users, whereas the Cathedral approach views programming and bug finding as *"tricky, insidious and deep phenomena"*.

By proposing an agile methodology for distributed development, Angioni et al. [19] outline general considerations about software development in an open source context. In short, they describe different scenarios and literature where agile development methods have been added to a distributed context, in order to improve the processes. Ad-

ditionally, they compare agile methodologies to open source values, and propose an initial methodology for distributed development (MADD). While the process itself is not elaborated thoroughly, they evaluate this MADD by surveying 41 developers from 27 different OSS projects, for a refined proposal.

In their comparison between agile methodologies and open source values, Angioni et al. [19] find a series of apparent similarities, which they use to argue for an incentive to their MADD proposal. Both agile development and open source projects value the importance of being able to adapt to change. Frequent releases and continuous feedback from involved actors are preferred. Another apparent similarity is the emphasis on valuing individuals, their skills and self-organizing teams. The agile manifesto mentions a clear priority of *individuals and interactions over processes and tools* [11], and open source projects are built and owned by motivated volunteers. Furthermore, much like agile development values, open source communities value communication and promote mutual respect in projects with possibly hundreds of developers around the world. Frequent and incremental debugging to ensure code quality early instead of a late, high-cost debugging is also a resemblance between the two. The main difference, however, is that of distributed developers. While agile methodologies such as Extreme Programming and methods such as pair programming promote close proximity of developers, this is an obvious challenge in an open source context, where distributed development is a given.

In relation to this comparison, and the feedback from the survey, the MADD proposal consists of four areas: Communication, planning and design, coding and testing, and feedback. In terms of communication, their recommendations include establishing interpersonal relationships between the members, making a common vision and the progress transparent, using tools such as dictionaries to define the users (with mails, roles, and more), instant messages, and wikis for fast knowledge sharing. For planning and design, their recommendations are to break down and describe the features into use cases or user stories, use periodic releases with different versions of the software, use different levels of detailed planning e.g. quarterly, monthly and weekly, and to evaluate the process between phases. In relation to coding and testing, their recommendations when having a diverse set of developers are to promote coding standards, testing and continuous integration to continuously verify quality of the system. Lastly, the recommendations for feedback include getting frequent feedback from the development team as well as the customer, to avoid high cost of changes. Additionally, they recommend using collaborative management of code quality, by methods such as pair programming, unit testing and automated tests.

2.2.3 Quality and user-centered design

The success and potential of OSS is apparent. However, as OSS development becomes more popular, and software companies contribute to OSS and use it in their commercial products, new challenges emerge. Hedberg et al. [20] discuss challenges about quality and usability assurance, specifically when OSS target a larger user audience, including non computer professionals. Specifically, they present current open source practices in terms of quality and usability assurance, and describe recommended changes. It is important to make a distinction between the notions *usability* and *user-centered design* (UCD), since they are not the same. Hedberg et al. [20] have a focus on usability throughout their paper, but describe a strong relation between the two notions. Where usability is concerned with how well specific goals can be achieved by specific users of a product, UCD is a more vague concept, focusing on a user-driven approach to development. However, they are using the terms in combination or intertwined, because they encapsulate challenges concerning the inclusion of users in open source development - or specifically, distributed development. Thus the conclusions of their paper still apply to the context of this thesis, even though we are not specifically performing usability testing.

Hedberg et al. [20] explains that producing a high quality OSS is motivated by the fact that the developers are using the software themselves. There are usually no strict processes to follow, but instead, the development follows a more ad-hoc manner. There will typically only be a lead developer or a small core team to be in charge of the overall architecture. This introduces challenges in the new situation, where the end-users no longer only consist of co-developers, but also non computer professionals. This will in turn result in higher quality expectations. However, while proven quality assurance models, such as CMMI and ISO 15504 exist, these are not applicable in an open source context. OSS development is challenged by developers being distributed, dynamic hierarchies, motivation factors and even culture. The recommendations presented in the paper to overcome these challenges include stricter methods and processes as well as advancing quality assurance to development phases before the product is delivered to users. Specifically, they recommend producing plans and documents to support communication, using inspections and reviews, pay more attention in test coverage, using test-driven development and testing performed by developers and not users.

Hedberg et al. [20] describe user-centered design (UCD) as a vague concept with a lot of definitions and research attached to it. Nevertheless, the area of UCD is popular, and all its methodologies include a focus on the understanding of users and their work context as well as a clear incentive to get early and often feedback from users. However, usability assurance is also challenged by introducing new non computer professionals as users. The developers do not have the necessary knowledge and skills to do UCD in OSS development and there are often no resources to do it. The recommendations to overcome these challenges are much aligned with general UCD suggestions: under-

stand and specify the users and their context of use, actively involve the user and gather early and iterative feedback.

Another paper dealing with code quality in OSS communities is that of Alami et al. [21]. Specifically, they examine code reviews in this context, why they work, and find that they may be an important factor in the high quality software produced in these communities. They find that code reviews can be an emotional practice, but seem to be important for some open source communities. By interviewing participants of an open source community, and participating in meetings, they formulate 20 proposals for how OSS code reviews can be improved.

In general, they find that the underlying reason behind why code reviews work in these communities is hacker ethics. In other words, the contributors of such a community are motivated by virtues such as passion, caring, creativity and joy. Furthermore, they found that non-material extrinsic motivators, such as gaining reputation and learning, are key factors as well. They summarize six observations and suggest actions/interventions for each of them. The first observation is about having a work environment that embraces rejection. Thus it is important to welcome critique and wanting to improve. Along these lines, the second observation is about the iterative improvement cycle that code reviews enable; programmers should appreciate the process of iterative feedback. Similarly, the third observation deals with the importance of embracing passion by developers. Open source communities are driven by volunteers, and their passion should be cultivated. The fourth observation states that the ethic of care is a positive contributor for code reviews, and that should be supported. While acknowledging that there is a lack of research in the area, the fifth observation is about the incentive to nurture engineers' intrinsic motivators. Likewise, the sixth observation states that the extrinsic motives are important drivers for the OSS contributors, and that they make code reviews effective.

2.3 Distributed user feedback

User testing is almost an obligatory practice for software engineering. However, while the advantages of such practices are apparent, it is argued that they require considerable resources. In a qualitative study, Bruun et al. [22] compare traditional usability testing - i.e. testing in a lab setting - to remote asynchronous testing, and discuss their trade-offs. Practices of usability testing, such as planning for tests, establishing test settings, conducting the tests and analyzing the results, require substantial resources and can result in software teams neglecting the practice altogether. These considerations have introduced remote methods, where the evaluators are separated in both time and space from the users.

As a benchmark for the remote methods, Bruun et al. [22] use conventional laboratory-

based think-aloud testing. These results are compared to the results of three different remote methods for user testing, in terms of critical, serious and cosmetic usability problems. All the methods are used on an existing mailing system by 40 test users; 10 for each method. The first method is called user-reported critical incident, where the users are supposed to report the potential problems themselves. In their study, this was done immediately using a web form. The second method is a forum based condition, where the users are asked to post and discuss their findings with each other. The last method is a diary-based longitudinal approach, where users are asked to report problems in a certain time frame with no restrictions on the format.

The overall findings made by these comparisons were in terms of how many problems identified, and how much time was spent on the entire usability method. In short, a significantly higher amount of problems were identified in the traditional laboratory setting (approximately double) compared to all of the remote approaches. However, all of the three remote approaches required less than a third of the time to complete, making remote usability testing an appealing practice for many software projects.

An important thing to note in relation to the present report, is that they discuss the potential challenge of doing usability testing early in the development process. Since the users are on their own in these remote approaches, they have no possibility of support, which can influence how such a method can be structured.

A more recent paper also discussing user feedback in the context of OSS development is that of Llerena et al. [23]. They argue that one of the main challenges in OSS applications is poor usability. Similar to the related work presented above, they suggest that OSS developers have tended to develop software for themselves, however, as the popularity of the open source paradigm has grown, the communities are unsure of who the users actually are. Additionally, a lack of resources and the fact that the developers are volunteers are obstacles preventing thorough usability testing.

In a multiple case study, Llerena et al. [23] adapt four user feedback techniques to fit the OSS development process. Since these techniques often involve direct contact with users, some OSS characteristics present apparent challenges to overcome; development is feature driven, developers are geographically distributed worldwide, there are a shortage of resources, and it has a culture that is alien to interaction designers. Since the existing user feedback in OSS development mainly focus on usability testing, they purposely include two adapted techniques focusing on requirements analysis. The four techniques adapted for the case study are personas, user profiles, post-test information, and direct observation. The former two are mainly techniques for requirements gathering, and the latter two for usability evaluation. In total, the techniques were applied to four different OSS communities; one for personas, one for user profiles, and two in combination with direct observation and post-test information. The techniques were analyzed to identify adverse conditions in relation to OSS development resulting

in suggested adaptions to overcome them.

The main conclusions and contributions of this study, are the identified obstacles for each of the techniques, as well as their suggested associated adaptions, made by observing and evaluating the techniques in the case studies. Since the developers are distributed across different time zones, they found it difficult to arrange meetings and have consistent communication. To overcome this, developers have to communicate via e-mail, and users have to participate via online wikis. Additionally, usability experts are required to analyse and interpret results from some of the techniques. The suggested adaption here, is to replace the usability expert by a team of junior experts supervised by a senior expert. Another major concern was that of user participation. Even identifying the users can be a challenge, but making them engage in voluntary usability testing presented an even greater obstacle. To accommodate this, users can participate online via online surveys and social networks. Also, if there is a shortage of actual voluntary users, a suggested adaption is to include biased user sampling including family and friends of the contributors. Besides the obstacles and adaptions, Llerena et al. [23] present guidelines for each of the user feedback techniques, i.e. how they can be adapted to a distributed open source context. For instance, while direct observation presents clear challenges, this can be overcome by the aforementioned biased user samples as well as using online tools for information sharing, to simulate the observations.

2.4 User domain: qualitative coding

Within the field of qualitative research, one of the methods to process qualitative data is through *coding*. Through the coding process the researcher gains greater understanding and insight into the data, as well as organizing data according to *codes*. The purpose of the process is to extract themes, that are based on the codes and categories, in order to prove an existing theory or generate a new theory, according to Saldana [3].

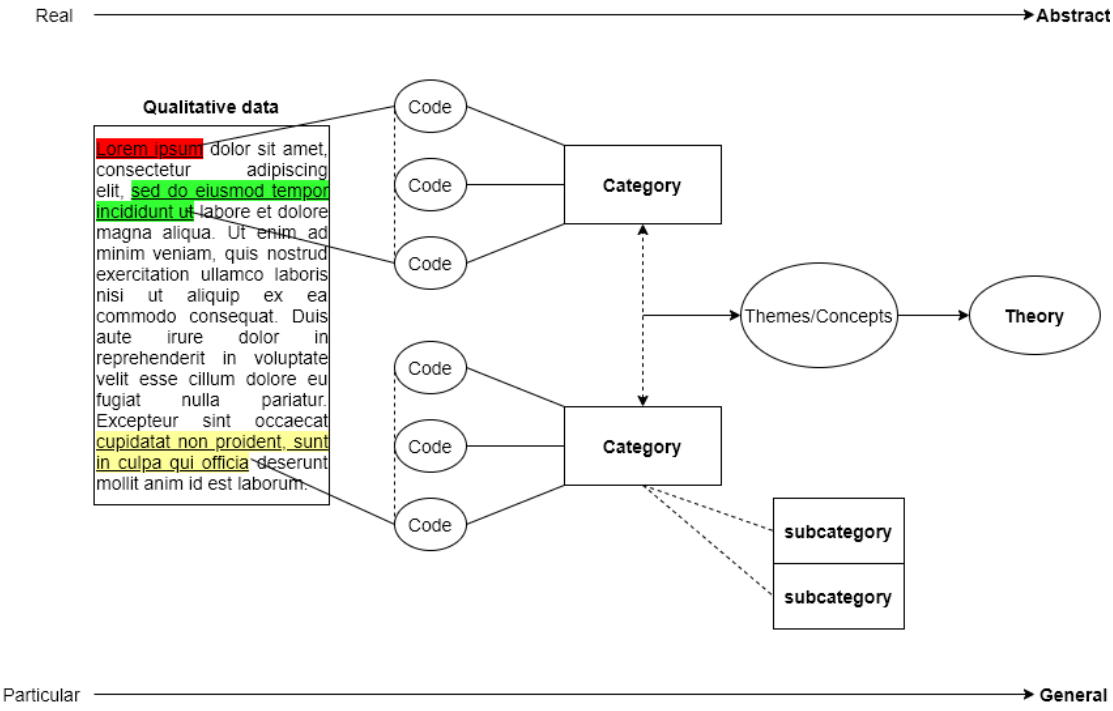

Figure 2.1: A model that describes going from qualitative data to codes and then to theory for qualitative inquiry. Figure adapted from [3].

The coding process takes as input raw data and at least a single researcher and through the process of coding, the researcher aims to organize qualitative data and find patterns within this data. The *format* of the data can be text, audio, video, and the *source* of the data can be interview transcripts, participant observation field notes, journals, documents, literature, artifacts, photographs, video, websites, e-mail correspondence etc.

The way data is organized is through the use of *codes*. The purpose of the code is to act as a container, that captures the essence of a piece of data. After a researcher has a created a code, subsequent pieces of data can be grouped as to belong to a code. This is the way that data is organized. The part of the data that is selected is referred to as a *quote* of the data, and the quote always belongs a code or perhaps to several codes. Analyzing patterns in the data is done by linking codes to other codes. At a greater abstraction level we have *categories*. A category contains a set of codes, where the category itself can be expressed as the essence of the set of codes it contains, thereby creating a hierarchy in the following way [3]: In the coding process new categories, subcategories and codes are discovered and therefore a great amount of editing of these codes and categories takes place. The specific data that is *summarized* by a code, are quotes and the following example shows how quotes are linked to codes.

Category: Teacher skills

Subcategory 1: instructional skills

- Code: Pedagogical
- Code: Socio-Emotional
- Code: Style/Personal expression
- Code: Technical

Subcategory 2: Management skills

- Code: Behaviorist Techniques
- Code: Group Management
- Code: Socio-Emotional
- Code: Style (overlaps with instructional style)
- Code: Unwritten Curriculum

Figure 2.2: Example of codes in a hierarchy. Example taken from [3].

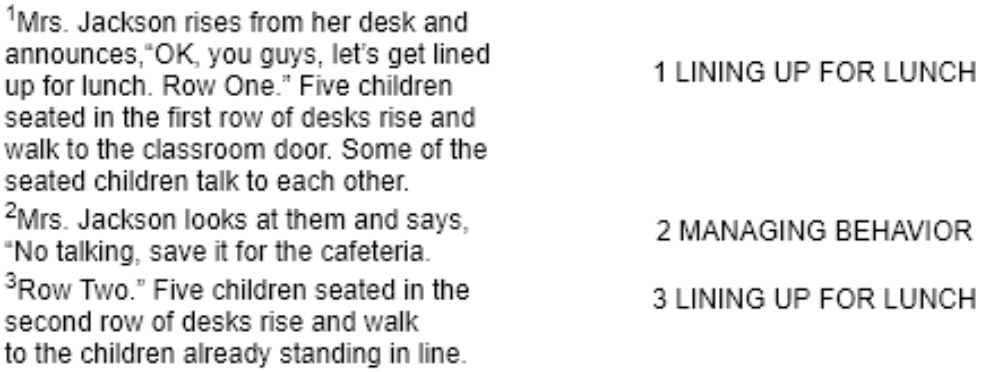

Figure 2.3: An example of how codes are linked to quotes. We have two codes; 1LINING UP FOR LUNCH and 2MANAGING BEHAVIOR and three quotes, where the numeric value of the code indicates the quote (in this case text). Example taken from [3].

Saldana [3] recommends a *pre-coding* activity, which includes *circling, highlighting, bolding* etc. important quotes or passages, through the use of software such that these quotes can be accessed at another time.

The coding process is a cyclic process, which in software engineering terms would refer to as coding in multiple iterations, and during this process the researcher gains greater insight into the data. Because of the iterative nature of this process, there is

often a need to revisit codes; either adding new ones or editing the existing ones, and adding new links between exiting codes. Although this process is considered to be individualistic, there is an advantage in having multiple researchers code the same data. Since this kind of analysis is prone to one's own perception of the data, it can be useful to compare one's analysis to the analysis of another researcher. However, according to Nielsen [2], this process can become more fruitful through researchers collaborating, not only for reducing the reliability of the analysis, but also to *"inform, influence and justify through dialogue"* in order to achieve a consistent analysis.

2.4.1 Applications supporting the coding process

In this section we document the part of our analysis that pertains to the technical part of the problem analysis. Since the purpose of the project is that of bootstrapping an open source project, as described in section 3.1, and to work with some level of user feedback in a distributed setting, there remain the following questions:

- What kind of system or application should we develop?
- Which features should the application have or focus on?
- Which technologies should we use to implement the system or application?

In section 1.3 we started by limiting the scope of the project to be exclusively about qualitative data analysis, where we in section 2.4 discussed this process at a higher abstraction level. The question that we ask is: *what feature of functionality do current qualitative data analysis tools not support?*. To answer this question we start of with the whitepaper [2]. The purpose of the whitepaper is to explore how qualitative data analysis tools support the process of coding, as well as evaluating the potential of *collaborative* coding and how tools support this collaboration.

We start off with table 2.4, which gives an overview of tools that were available in 2012.

Application	Source	Cost
NVivo Windows client	www.qsrinternation-al.com	1 license USD 670; 10 licenses USD 5360
Atlas.ti Windows client	www.atlasti.com	1 license USD 670; 10 licenses USD 4800 or USD 1400/year
HyperResearch Windows & Mac client	www.researchware.com	1 license USD 495; 10 licenses USD 4050
CAT Web service	cat.ucsur.pitt.edu/	Free
Knowdoo Web service	knowdoo.net	Free
Dedoose Web service	www.dedoose.com/	1 license USD 132/year; 10 licenses USD 1122/year

Figure 2.4: A list of qualitative data analysis tools. Table taken from [2].

We are presented with six applications, two of whom are free, which are the ones of primary interest since we will be developing an open source application. At the time of writing this report (February-June 2020) we cannot find any reference to *Knowdoo*; neither through the provided link *Knowdoo.net* nor through search-engines. The open source project *CAT* is still available, however documentation has not been updated since 2009 and the source code, although available for download, has no public repository nor is there a mailing list or anything that indicates a community around the software [24]. We searched for other open source projects. We found the project *RQDA* [25], which is a small python program written by a single developer, which focuses on fundamental features, such as creating codes and marking quotes. A primary feature of RQDA is exporting codes and quotes to R, which has a lot of packages for handling data presentation. Similarly, from an online discussion on GitHub, from the 2018 unconference hosted by rOpenSci an *open science hackathon* for developers; there was a discussion on open source qualitative coding tools [26]. The request was for a *coding* tool, not necessarily an *analysis* tool, where the tool should be be simple to use where the user used the analogy of a *paring knife*, in order to describe the characteristic of the required product. The user gives an example of codes and quotes, the same as we gave in section 2.4, where pieces of text can be coded by setting a '1' indicating that the coding tool not necessarily has to be based on a graphical user interface. Since this conference was hosted specifically by a community focused on using R, this coding should also be able to export codes and quotes. Another open source project *QualCoder* developed by Colin Curtain a lecturer from Australia [27], which includes many features, most notably being able to handle many data sources, such as text from many file formats, as well as audio, video and image data. What we gather from these sources is that there is potential for an open source community. However, we also have proprietary applications namely NVivo, atlas.ti, HyperResearch, and Dedoose. Nielsen [2] provides a

comparison of these applications, as figure 2.5 shows.

	NVivo	Atlas.ti	HyperResearch	Dedoose
Analysis function	Predefined & Query	Predefined & Query	Query	Predefined
Linking codes	Hierarchy	Complex network	Simple network	Hierarchy
Authority	Simple	Simple	None	Elaborate
Interleaved coding	File sharing	File sharing	File sharing	Server-based
Simultaneous coding	None	None	None	Synchronization
Learning curve	Steep	Steep	Gentle	Medium

Figure 2.5: Differences between proprietary applications. Figure taken from [2].

While these applications have a range of features, the one feature that is missing, is that of *real-time* collaboration. Real-time collaboration is a well known feature of applications such as Google docs and Overleaf, which allows users to concurrently edit a text document in real time. However, since this whitepaper was written in 2012, some of these applications may support this feature. Currently, ATLAS and Dedoose support real-time collaboration, NVivo supports collaboration by synchronization and HyperResearch supports none [28][29][30][31]. Since there are no open source alternatives to the proprietary applications in terms of real-time collaboration, we will use that feature as our starting point.

Chapter 3

Problem analysis

In chapter 1 we limited the project to three areas of interest; (1) open source development, (2) software development processes and tools, (3) domain and users, and in chapter 2 we presented theory related to each of them. In this chapter we examine the described theory and present research questions that address the three aspects.

3.1 Bootstrapping an open source process

As we have presented, the open source development methodology is a unique paradigm which does not conform to traditional industrial methodologies. The hacker culture, as introduced in chapter 1, entails different motivators for participants of OSS communities, and this has to be reflected in the work processes. In his book [14], Cockburn outlines these differences, in a tangible way. Using an analogy of a cooperative game, specifically rock climbing, he describes software development. He uses game terminology to describe the characteristics of agile software development, such as being cooperative, goal-seeking, skill-sensitive, using tools, require training, consisting of teams, and having individuals with talent. For instance, both rock climbing and agile software development relies on cooperation, communication and aim to reach a goal; reaching the top or producing a piece of software. Though open source development is still goal-seeking, the goals are different. Instead of trying to reach the next release, or finishing an application, OSS developers are not usually restricted in time-frames, but may have a goal to have fun or make something good. Cockburn mentions children carpet wrestling or musicians playing music as a better analogy. The point is that these characteristics of the hacker culture should not be neglected. If open source communities are dependent on having skilled, motivated contributors, we have to be careful not to change the circumstances in a way that drives them away - for example by enforcing a lot of heavy bureaucratic rules and processes.

Although we have to be careful not to ruin the culture that makes up the open source communities, we have presented literature in chapter 2 that introduces a series of con-

siderations regarding OSS methodologies. The remainder of this section will summarise these:

- A number of barriers can prevent people from participating in OS projects. This includes individuals, communities and industries
- Agile development and open source systems share similar values. This should be considered when designing a methodology for OS development
- As OS development becomes more popular, quality expectations will increase. OS characteristics such as distributed development challenges these requirements
- Getting the right user feedback is a valuable practice in OS development, however, existing methods are difficult to adopt in this context

Alami and Wasowski [17] identified six barriers of participation, that could lead to so called passive participation, where participants only use the code, but do not contribute - or even prevent participation altogether. Although these considerations and their related resolutions are relevant, they are mostly related to the side of the contributor. For instance, the barrier regarding objections from senior management are suggested to be resolved by improving communication between developers and management. However, the barriers regarding a high cost of participation and being unfamiliar with the system can be accommodated by the OS community. Alami and Wasowski suggest that the communities should consider whether some of the participation costs should be reduced and if the communities should be more helpful to integrate newcomers. For instance, experienced OSS contributors could mentor inexperienced people trying to enter the project.

Similar to some of the points made by Cockburn in his cooperative game analogy, Angioni et al. [19] suggest that open source development is very similar to agile values. Thus when designing methodologies for this context, these values are important to reflect; transparent communication, iterative development, continuous integration testing etc. (as described in section 2.2.2).

As described by Hedberg et al. [20], the expanded use of OSS result in higher expectations to quality assurance. Due to the unique characteristics of OSS development, this may not be a trivial problem. Contributors can be distributed across the world, hierarchies can be dynamic or unclear, and the motivations of volunteering developers may be misaligned with thorough testing and code reviews. Hedberg et al. [20] recommend using stricter quality assurance methods to resolve this. A somewhat different approach is presented by Alami et al. [21]. They suggest embracing the motivating factors of the hacker ethics (or culture), and create an environment promoting these values. For instance, as these volunteering contributors are motivated by intrinsic values, such as writing good code and improving themselves, they should welcome critique and thorough code reviews.

Getting the right user feedback is generally a valuable practice for software development, including open source projects. However, characteristics such as having distributed developers all over the world create problems to overcome. These are elaborated in section 3.2, on user feedback.

As the applied literature have argued, the above problems are relevant in the context of open source development, but what does this mean in terms of bootstrapping such a project? We have not been able to find any research on how to initialize an open source project, or how the process should look at its early stages. Combining the problems and suggestions from this literature, practically forms a catalogue of development processes, but how can they be used at the start of a project? What should be prioritized and what is less relevant, when a project is small and at its early stages? Our research question is the following:

- **RQ1:** *How can we bootstrap an open source project?*

3.2 Distributed feedback practices

One of the interesting aspects of open source is how to handle feedback of users, considering that open source development is challenging due to spatial and temporal differences; there is a similar challenge in interacting with users. In section 2.3 we covered literature that shows that remote feedback is certainly possible. The problem is that these methods are geared towards usability evaluation and not for *utility* evaluation. These usability evaluation methods from Bruun et al. [22] also do not consider the resource constrained context of open source communities and volunteer users, as described by Llerena et al. [23].

Nielsen [32] in his book *Usability Engineering* gives an overview of what utility and usability entail, which figure 3.1 illustrates.

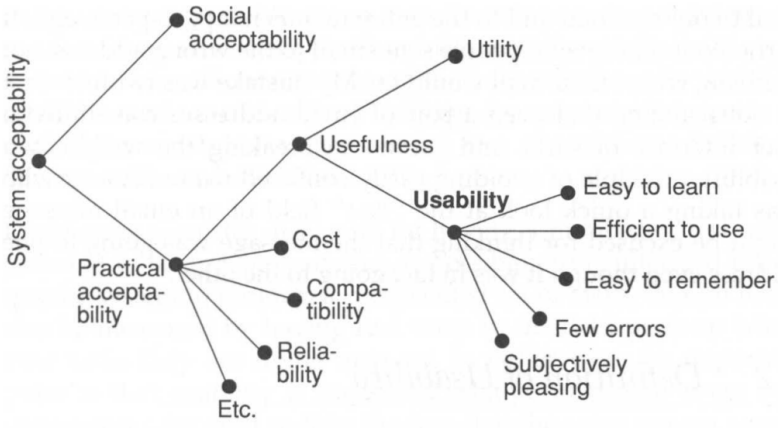

Figure 3.1: We see that although utility and usability are closely related, both being a subcategory of *usefulness*, they do differ. Figure taken from [32].

The core difference being that utility refers to the functionality of the system, e.g. does the system have the necessary features, such that the user can perform whatever task he or she might want to accomplish. Tarkkanen et al. [33] conclude, after reviewing 173 usability problems from three separate usability tests, that early usability testing with a *think-aloud* protocol and open task structure measures both utility and usability equally well. They report that the number of utility problems ranges from 51% to 74%. Our problem is then reduced to adapting one or more of the usability evaluation techniques to that of utility testing.

We formulate research questions based on the triangle described in the beginning of chapter 1, which were expunged in the previous sections: 3.1, 3.2, and subsection 2.4.1. The research questions are as follows:

- **RQ2:**
  - **RQ2.1:** *What kind of functionality, architecture, and software components can be used to build a real-time collaborative qualitative coding tool?*
  - **RQ2.2:** *How can we apply user feedback methods in the context of an open source project, with voluntary distributed users?*
  - **RQ2.3:** *What kind of tools and practices can be used to support a minimal open source process in the context of distributed development?*

Chapter 4

The Setup

Our methodology is reminiscent of the Cathedral approach, which is one of the open source approaches we described in section 2.2.2. The characteristics of our methodology are as follows: we are a small distributed team that designs, implements and releases software that is evaluated against a distributed user base, while all our communication is kept internally, yet the source code is made available at any time. If we compare these characteristics to the Cathedral approach, then it becomes apparent that there are a lot of similarities in terms of the overall approach. At the same time we can also say that we are not developing with a Bazaar approach, since we are not transparent with project decisions, nor do we have any procedures for contributing or recruiting new developers.

In section 4.1 we will present a timeline and the phases included in our development project to give an overview of the whole first. In section 4.2 we will elaborate on this, and describe our methodology in more detail, providing arguments for the choices we made regarding practices as well as tools.

4.1 Development phases

As figure 4.1 shows, our development process consisted of three general phases and two demos. At each demo, the application was made available on a website, to get feedback from relevant users.

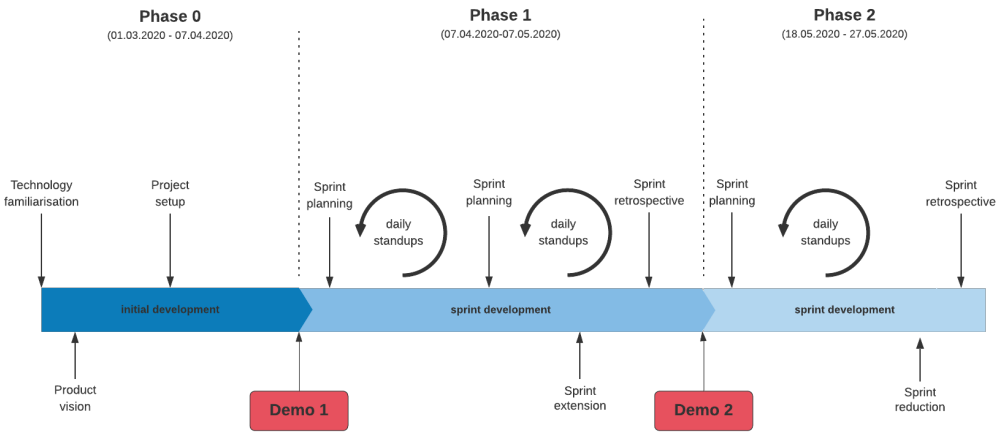

Figure 4.1: Overview of the development phases (timeline)

Phase 0

Phase 0 was mostly about setting up the project, and developing a minimal application viable for the first demo. We acknowledged the fact that we had to invest some time into learning the technology stack before we could start developing anything. As we became more familiar with the technologies, we made a few sketches of an initial application, based on what we already knew about the domain. Then we set up project formalities including an initial kanban (Trello), configuration management (GitHub), continuous integration (Heroku and Netlify), and a tool for user feedback (UserReport). The rest of phase 0 was spent on developing the application to be ready for the first demo.

For the first demo, we made the application available to the users for one day, and collected their feedback through the tool UserReport, which was attached to the site itself.

Phase 1

In this phase we started using sprints. We had two sprints in this phase, which were followed by a demo. Only one sprint was planned initially, but as we did not have much new functionality to show for a demo, we planned another sprint for this phase. Thus the second sprint is a small extension of the first, to complete the leftover tasks.

Since we started using time-boxed sprints, we would also do planning, which consisted of translating all the feedback from the first demo into work items, and creating a sprint backlog to be done throughout the sprint. Additionally, we held daily stand-up meetings online using Discord, where we discussed the features we were working on. At the end of phase 1 we held a sprint retrospective, however, only for the last sprint, discussing how the sprint had gone and what should be changed for the next sprint.

For the second demo, after having written an e-mail instructing our users, the application was also made available to the users for a day. For this demo we also used the

tool UserReport just as with demo 1 to collect feedback.

Phase 2

Phase two started with wrapping up from phase 1, and then focusing on other tasks than development. We did sprint planning and we did daily stand-ups in the same way as in previous phases, where another sprint was planned that should have lasted from 18.05.2020 to 01.06.2020, however, due to time constraints we ended the sprint early on 27.05.2020, and since we did not have much extra functionality, we ended up with not releasing a demo to our users. The purpose of the sprint was changed midways, in order to experiment with significantly increasing the level of ceremony with regards to pair programming. The reason that we ended up with not much extra functionality was because did a lot of refactoring, which was not planned initially, but was necessary for further implementation. Screenshots of the application from each demo, as well as the feedback received, can be seen in appendix A and B.

4.2 Development methodology

In this section we will describe the choices of our development methodology as it was for the first phase. Changes to the methodology are described in chapter 5.

We know from the principles of methodology design, which we described in section 2.1, that a team with a few members need very little methodology. Our project is of very low criticality and as such we need not have high ceremony for the practices that we establish. In the first two weeks of the initial phase, we were collocated, and for the rest of the development we were distributed, where our users were distributed during the entirety of the development.

Activities, Milestones, and Process

Cockburn describes *activities* as to mean what people in the team spend their time on during the day, *milestones* being measures of progress and *process* as how activities are strung together over time. We described milestones with very low ceremony, i.e. we talked about what we wanted to achieve, not writing down any formal milestones, in the following phase in terms of functionality and the milestone would then consist of a set of tasks to be completed, which would reside on our Trello board.

We have no explicit document that says how much time we spend on different types of activities. In phase 1 we spend much time learning about technologies, tools and operations, and we spent some time on designing and programming.

We conducted sprints in the following way:

- Sprint planning on the Friday before the start of the sprint, which would be the following Monday.

- Talk about the overall goal of the sprint (milestone).
- Possible distribution of tasks of this sprint
- The length of a sprint is decided at the sprint planning.

Daily stand-ups had the following structure:

- Meetup on voice-chat at 10:00 AM each weekday
- Status of work (current task)
- Discuss encountered problems
- Possible distribution of new tasks (switching tasks)
- Evaluation and possible revision of the current sprint

The daily stand-ups ensure that we keep the level of communication and knowledge sharing up. Increments, in the form of sprints, allows us to set milestones.

4.2.1 Tools

For web development, we use WebStorm and the open-source text editor Atom. We use Git as our version control system, and GitHub as the hosting site. Considering that this is how most open source projects host their code base, and that Git is the only system we have experience with, these are good choices for our team.

Branching, continuous integration and continuous deployment

We decided to *branch* based on features. When starting on a new feature the programmer branches from the *master* branch, work on that feature while committing new changes to that branch everyday. When the feature is finished the programmer would do some manual testing and then integrate it on the master branch, and delete the branch that was used to develop the feature. This means that our integration cycle is not a constant value, such as every day or every three days, but rather it is dependent on when a feature is finished. Integrating changes will, however, not take any longer than the duration of a sprint.

Since we needed a way for users to be able to test the application and since it is a web application, the most straightforward way is to host the application in the cloud. We tried several different cloud services, however, we ended up using Heroku [34] for hosting the back-end and Netlify [35] for hosting the front-end. These services also come with built-in support for continuous integration (CI) and continuous deployment (CD). The pipeline is illustrated in figure 4.2.

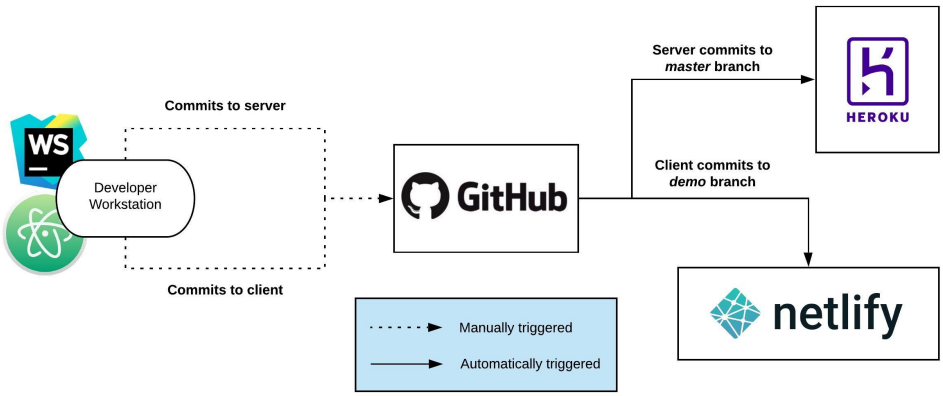

Figure 4.2: Deployment pipeline with Heroku and Netlify cloud services.

Bug tracking

For *bug tracking* we initially referred to Serrano and Ciordia [36], where a number of bug tracking tools are compared. While the paper is from 2005, the most notable systems mentioned are *Bugzilla* and *ITracker*, where Bugzilla is a system that is still currently used. There are many more such systems, and the authors of the paper provide a list of 70 other bug tracking tools in addition to the two mentioned [37]. The most important things for this project, considering we are two developers with a handful of users, are ease of use, quick setup, and low maintenance, and since we have not used *any* bug tracking tool before, familiarity plays no role. It would make sense to use the industry standard Bugzilla, because of its wide adoption, however, this would require us to install the system and grant access to our users to this system. Another argument for using Bugzilla is that when our open source policies and methodology is established, it would be easier to transition from *closed* development to *open* development. At first glance, the bug tracking tool MantisBT seemed a good fit for this project. It is open source, but there is also a paid cloud hosted version available, therefore we spent some time researching the possibility of hosting it ourselves, which we did, but after installing the system we experienced that it would take some overhead to manage the system. If we also want to have users be able to add and track bugs, then it would either require users to create an account for the system or we would have to create accounts for them. In either case more effort is required. Bug tracking was therefore limited to being a checklist on a single card on our kanban. This may seem as limited functionality, since bug tracking tools have a range of specifiable attributes of a bug, such as which browser the bug was encountered with. For the beginning of this small OSS project, we prefer low overhead by writing bugs in a checklist on our kanban.

Kanban

We have previous experience with the kanban style work visualization tool [38]. We use the kanban for maintaining an overview of the project as well as visualizing what work items are in which state. The specific tool that implements the kanban is the web application Trello [39]. We have the following states, that a work item can be in: *(1) Backlog*, *(2) Sprint backlog*, *(3) Implementing/doing*, *(4) Needs Review*, *(5) Done*. Figure 4.3 is a screenshot of our kanban with the mentioned columns.

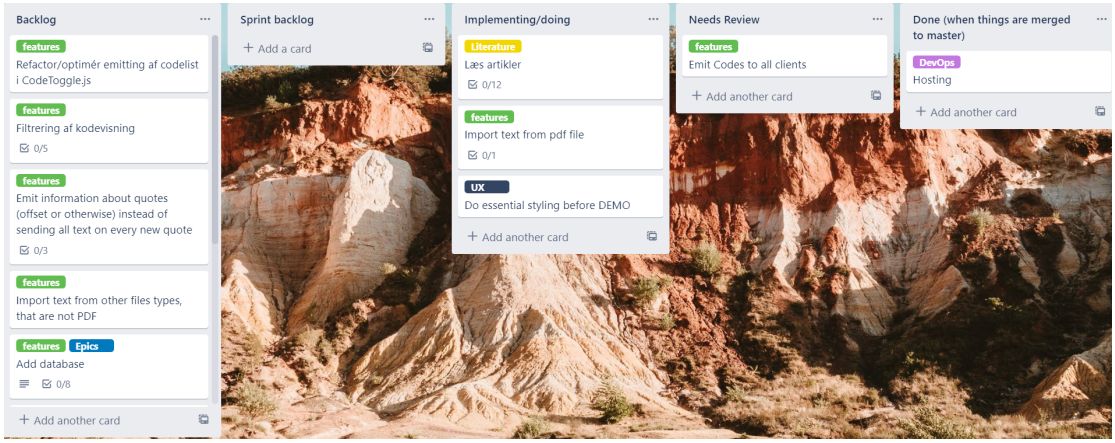

Figure 4.3: Kanban columns. These are the columns we had in phase 0.

Each work item contains a description of a feature or something that has to be done, such as setting up continuous integration or a demo, and is then labelled with one or more labels. Work items are labelled with one of the following labels:

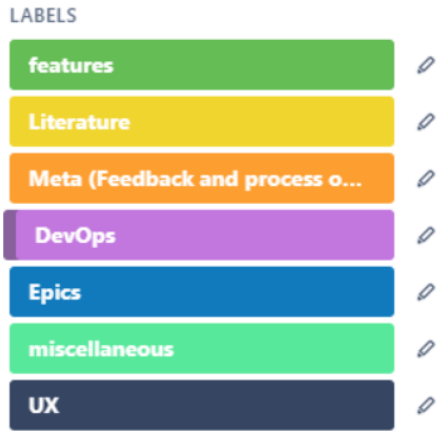

Figure 4.4: We have included some non-development labels since this project includes both the development of a web application, but also consists of other non-development tasks.

We have no entry or exit criteria for changing the state of a task, where this is done mostly ad hoc during daily stand up or during sprint planning. The difference between

the *needs review* and *done* is that a work item is considered done when it is integrated into the master branch. During development we added labels as needed as well as adding extra columns that happened during development, and the reasoning for these changes we describe in chapter 5.

Feedback tool: UserReport

In order to get frequent feedback from our users we used the tool *UserReport*. UserReport is an external service, which provides websites (and web applications) a way to directly leave feedback on the web application without having to switch to another 'tab' or other program to leave feedback. An example of the UserReport interface can be seen in appendix A. To use UserReport we add a *script tag* to the *index.html* file of our web application. The script tag is a single line of HTML, that is copied from the UserReport service. The way we would interact with our users would be per e-mail, instructing them how to engage wit the application as well as how to leave feedback. The e-mais can be seen in appendix D.

Users can submit ideas for features and submit bugs. The screen that is displayed to the users can be seen in figure 4.5

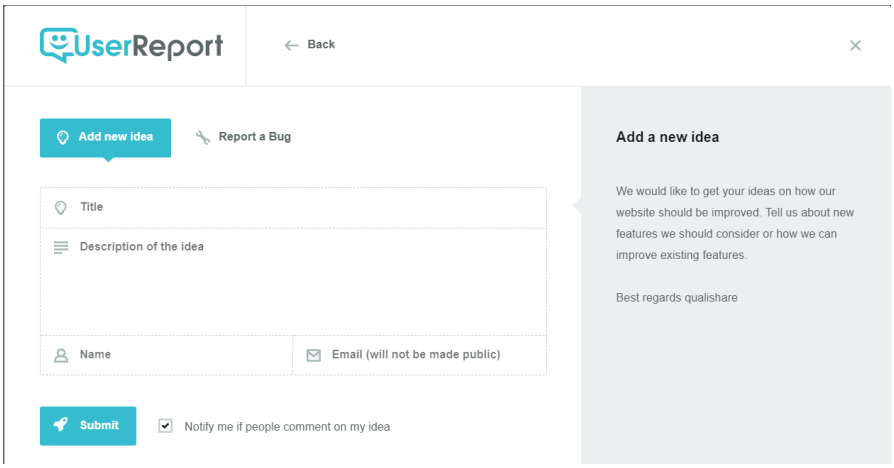

Figure 4.5: Screenshot of the UserReport interface where users can submit ideas and bugs.

This is the interface that is provided by UserReport. All feature requests are visible to all users, where a voting mechanism is provided, allowing a user to vote on any feature requested by another user. A user can also submit a bug that will be anonymous to other users. The tool is limited with regards to customization; we cannot add nor subtract any functionality. The advantage of this tool is that setup takes very little effort and that it can be applied to any web application, regardless of underlying architecture, since the feedback option will be available on any web application page. Disadvantages are that bug descriptions are limited to text and no options can be specified, such as what browser was used or browser version. The user could specify such details. Feature

requests are also limited to text descriptions. With these tools in place, most notably a CI/CD pipeline and UserReport, we can frequently deploy new features and acquire rapid feedback of users.

4.3 Application design

In this section we describe a set of features and components that are necessary to do basic collaborative coding. These features are based on the whitepaper from Nielsen [2] and from the description in section 2.4. This is the design created at the beginning of phase 0 from figure 4.1. The functionality that is essential to the coding process:

1. That all users can create codes, and that all users independently of who created the code, can dispose of it, and that these changes are always reflected among all clients in real time.

2. That all users can highlight text in a document, and that all users independently of who created the highlight, can remove the highlighting and that the highlighting is reflected among all clients in real time.

3. That data is persistent, such that users can return to the document at a later stage and resume the coding activity.

Figure 4.6 illustrates the intended data model, where we only implemented a minor part of it.

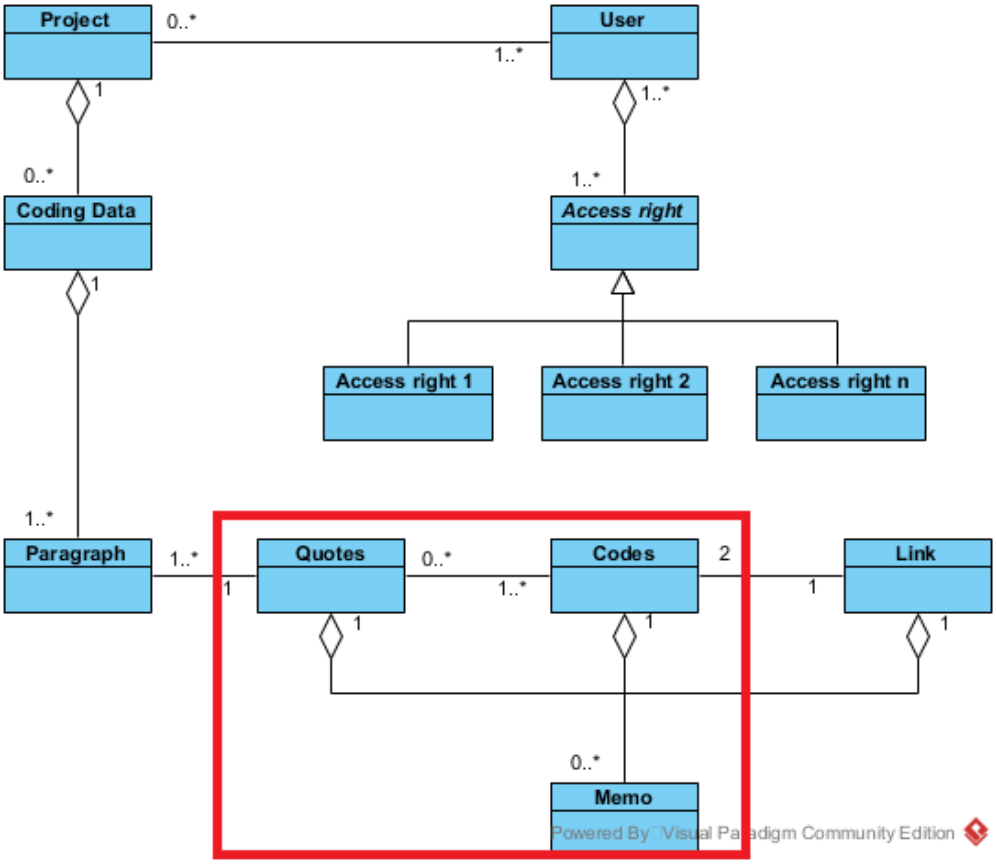

Figure 4.6: The red square marks the part of the data model that was implemented.

We ended up only implementing the red part of the model, since we only consider *codes*, *quotes*, and *memo* to pertain to essential functionality.

We started off the user interface design with the following three sketches 4.7, 4.8, and 4.9.

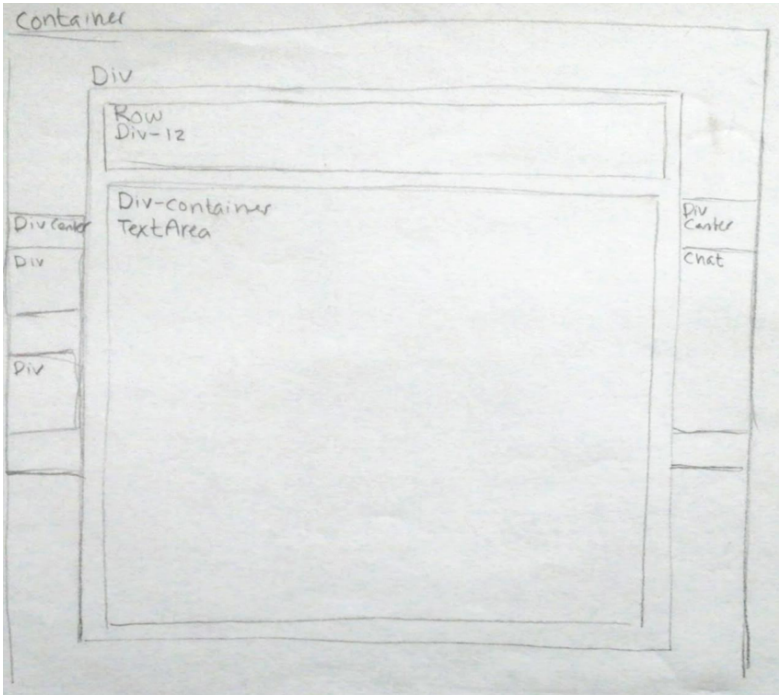

Figure 4.7: Partitioning of the UI. The idea was to divide the user interface into multiple sections, and how to achieve that with HTML.

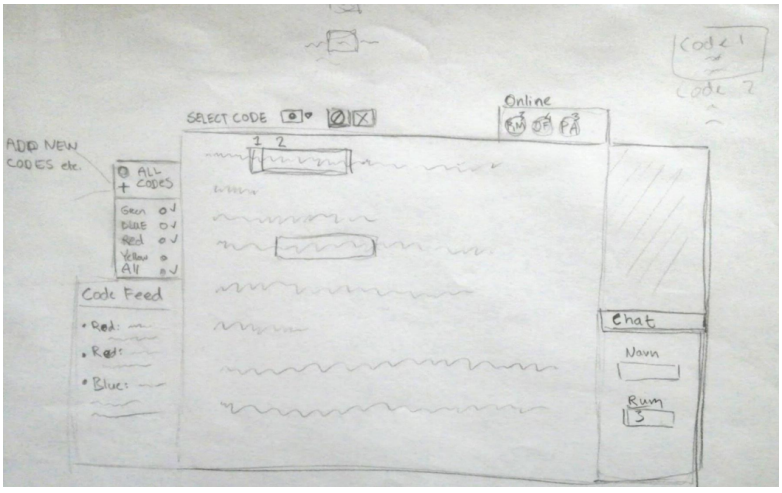

Figure 4.8: Specific buttons and layout possibilities. Here we added ideas in terms of where specific functionality could be located in the user interface.

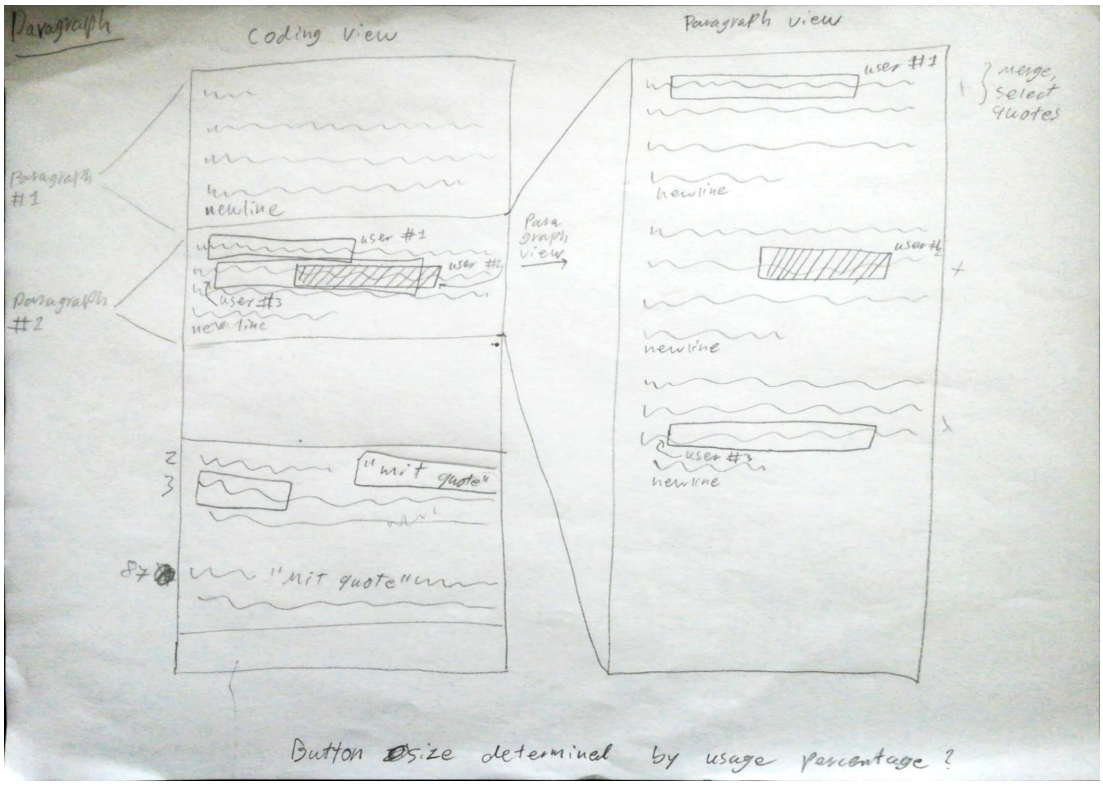

Figure 4.9: An *extracted* view of quotes that are doubly linked to a code or overridden by other quotes. Our considerations with this view, was to be able to open a separate view, in order to view quotes that were overlapping. We speculated that this would occur often, because of the collaborative functionality in the application.

We based these sketches on the previous description on necessary functionality, the literature on qualitative data analysis and our own experience with tools that support the process of coding.

4.3.1 Technology stack and components

We decided to use the MERN web stack (MongoDB, Express, React, Node), for the development of the application. This stack is fully open source, therefore when updates are made to these technologies our open source project will evolve with the open source community as a whole. In our case this has the drawback, that we have no experience with any of the technologies in use, nor do we have any experience with web development in general. However, considering that our users are distributed it will be much easier for us to have our users visit a web application from their browser, than to have them install an application on their own system. Continuously deploying a web application also takes less time in general, and therefore this is the preferred choice.

MongoDB is an open source database management system that is based on NoSQL. It contains JSON objects, which are commonly used in web applications. On the other hand we have SQL databases; these are stricter in terms of having to define a schema,

whereas NoSQL JSON objects can contain any attributes without having to redefine their attributes. This is important for this project, because we are exploring the problem domain and we therefore prefer ease of modification. Express is a web application framework for Node, that can be used for creating web services and web application programming interfaces (API), we use it to write a REST API on the back-end. A REST API may not be necessary for this project, however since the MERN stack advocates this approach we will use it as well. React is a library that contains functionality for re-rendering the HTML DOM (the UI), when there is a change in the data model. This makes it useful to create an interactive web application that reacts to user input, for example when a user adds a code, then React handles the re-render automatically to display the change.

We use the open source component Socket.IO to implement real time collaboration between clients [40]. Clients will have a local copy of all the data, and each time a client adds a quote or code, this needs to be reflected at the other clients. Socket.io can be used to broadcast such a change to all clients.

Chapter 5

The Evolution

In the previous chapter, we described the structure of our development process, and how our associated methodology was initially designed. In this chapter, we describe the observations and findings made throughout our development phases, and how the methodology changed. As this thesis is based around the three areas of interest introduced in chapter 1, we will present findings for each of them. This will be done incrementally with respect to each of the phases illustrated in figure 4.1. In other words, we will go through each of the phases following phase 0, and the demos, and present what we learned with respect to each of the three areas; (1) open source development, (2) processes and tools, and (3) domain and users. The purpose of this format is to give a clear overview of our observations and findings, however, we do note that a clear separation between the areas can not be done, since there are natural relations between them.

5.1 Phase 0: Project startup

As this phase was primarily about setting up the project, we do not have notable findings from the stage. At this stage of the project, we barely had any defined development processes, such as daily stand-ups or code sprints, and we were still collocated for a portion of this phase. The focus was to setup the necessary tools, and design the initial processes, as described in section 4.2.

As described by Angioni et al. [19] in section 2.1, open source development development values have a lot in common with agile methodologies. Also, open source systems are comprised of volunteering contributors with different intrinsic motivations, creating a unique hacker culture. Using Cockburns terminology, these facts solicit a light methodology weight; few elements with low ceremony. This worked well at this stage, because it did not create unnecessary overheard, and because elements of the three areas of interest were still uncertain.

5.2 Demo 1

The first demo marks the beginning of where we started to develop as our methodology from section 4.2 prescribes. Naturally, every software development project needs a start-up phase like ours, where code repositories and task management is set up, thus not much can be said about phase 0. On the contrary, we did learn something about the three areas of interest, during the first demo; primarily about the domain and our approach to getting feedback from the users.

Domain and Users

In terms of domain knowledge, we received an array of useful feedback concerning new features, the purpose of the application, and on the feedback process itself. We have summarised the feedback as follows:

- When selecting a code, the relevant quotes should be highlighted in the text. This helps interpret how collaborators are working, and gives an overview of related codes.
- Shortcuts to apply codes would be appropriate, particularly when coding a large amount of data.
- It should be possible to apply two different codes on the same area.
- It should be possible to see metadata about the document that is being coded in.
- It should be visible how many times each code has been applied.
- It should be possible to see who of the collaborators have added which code.
- It should be visible who has access to the document, and who is currently online.
- It should be possible to add new codes while highlighting the text.
- It is unclear what the purpose of the feedback is. What questions should be answered? Crowd sourced analysis of qualitative data may be a more accurate description of the process.
- Consider what collaboration entails.

Each of these items are the comments received through the UserReport tool. We have slightly changed the format so they are understandable, and are phrased in a similar way. As expected, the users had several suggestions on potential new features where several of them are also present in similar applications, as described in section 2.4. The most apparent theme here, is information about other users and general collaboration.

The users want to be able to see who has access to the application, who is currently working on the document and information about the applied codes.

Some interesting things can be noted about the feedback process itself as well. It was only the point about being able to see who added which code that was submitted twice. While all of the feedback was available to every user as soon as it was posted, this can be an indication that the users were aware of the submitted posts and therefore did not make duplicate feedback. On the other hand, none of the suggestions had any up- or down-votes, which could have been useful when trying to decide what tasks to implement. This can be an indication that the users were not aware that the voting feature was available at all. Furthermore, one of the feedback posts suggested that the purpose of the feedback process was unclear, and that he or she did not know what kind of feedback we were looking for. A potential adaption to the process could be to elaborate further on the purpose of the demo, in the e-mails to the users beforehand. Lastly, the post regarding missing document metadata presented a different challenge. Although the suggestion could be valuable to the application, we were missing an elaboration of what kind of metadata should be included. Since the application was only available in a certain time-frame, and the user-posts were anonymous, we could not expand the feature further. This suggests a potential shortcoming in this way of getting user feedback.

The UI of the application at the time of this demo can be seen in figure 5.1, the corresponding component hierarchy in figure 5.2 and an architectural overview in figure 5.3.

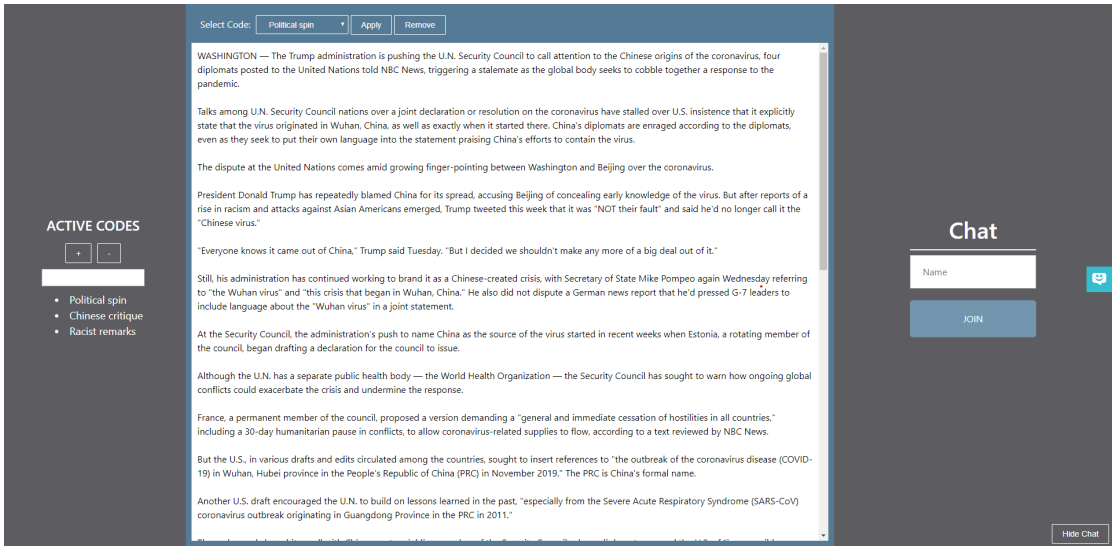

Figure 5.1: Screenshot of the application at the first demo. Left hand side we have an overview of *codes* (*Component: Code Toggle*), where codes can be added and removed. In the middle we have placeholder text, where text can be highlighted as quotes (*Component: Toolbar*). The right hand side we have the Chat (*Components: Join* and *Chat*).

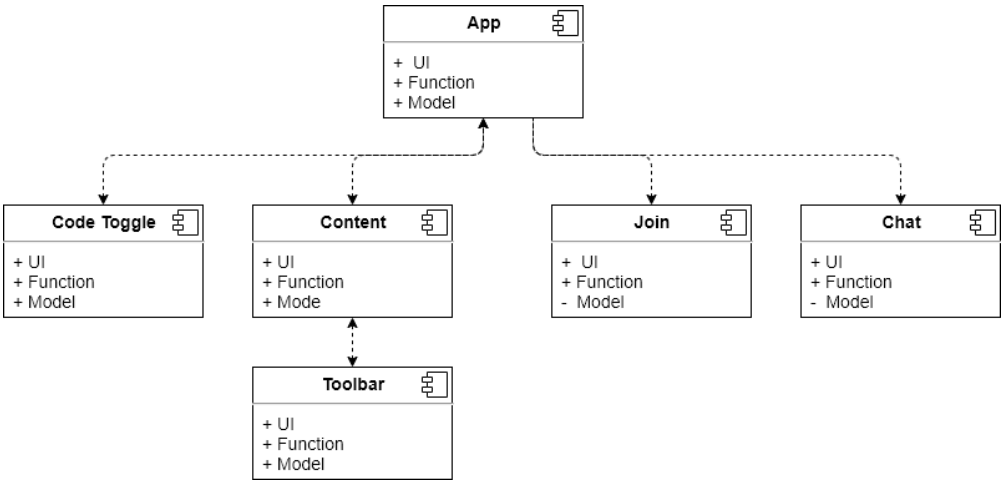

Figure 5.2: React component hierarchy. Here we have the components that are mentioned in figure 5.1, in a hierarchy. *App* is at the top of the hierarchy, since with regards to the UI, the children of the *App* component are nested within it. However, the components *Code Toggle*, *Content*, and *Toolbar* have dependencies to the *App* component, since these components call functions, and get parts of the model passed to them, from the *App* component.

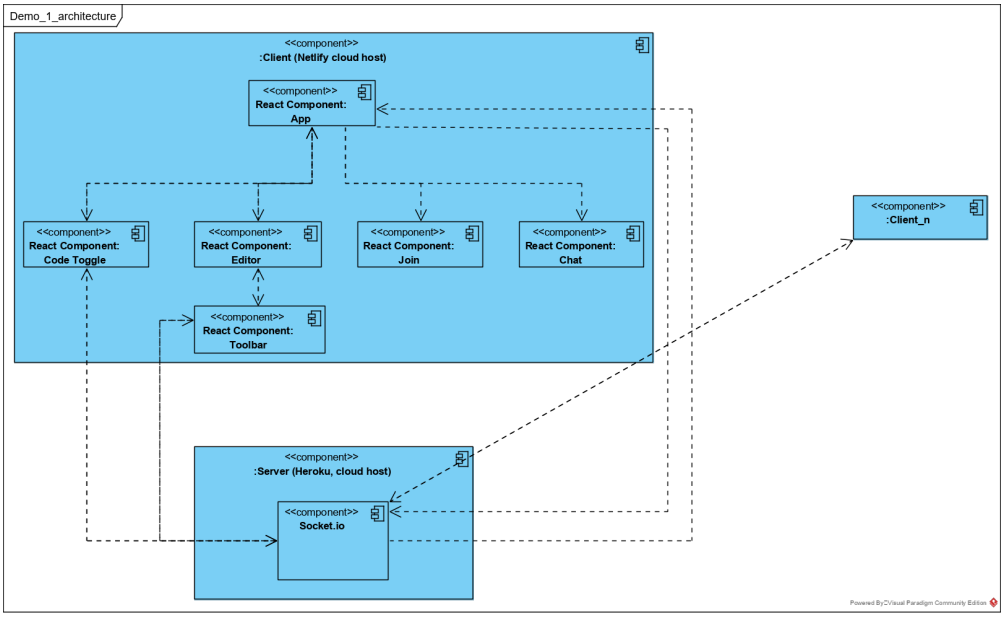

Figure 5.3: Architecture of our application at the time of Demo 1. Clients are connected only through socket.io events. In this version we have no database.

The functionality supported by this version of the application, is that of highlighting text, which is displayed in real time to all other clients. When clients log in however, no highlighting will be displayed, since we do not have persistent data. In this early version we have used socket.io extensively in multiple components, since this took the least time. Using socket.io in this way was the cause of some bugs, which would be solved

at a later stage, by having the *App* component be responsible for passing instances of the socket.io component to other child components.

Open source development

As presented in section 2.3, Bruun et al. [22] tested and found potential for three different types of remote feedback methods, where one of them was a forum based condition. In short, the idea in their test was for users to discuss their use of the tested application on a forum with only a few instructions. Considering the issues we presented above about not using the voting system and a confusion about the goal of the feedback, a forum based approach could be a potential solution. On such a forum, we would have been able to post simple instructions and comments on the process itself, and the feedback would be more transparent to all the users. Additionally, considering the issue about needing elaboration, we would be able to communicate with the users and discuss potential changes to the application.

Processes and tools

In general, our approach to getting user feedback was successful. Using a lightweight tool such as UserReport proved to be very useful for several reasons. One of the biggest challenges of getting distributed feedback in an open source context is, as described in section 2.3, to engage users in participation. There should be virtually no barriers when giving feedback, such as installing external software or creating unnecessary user profiles. UserReport proved to be a subtle addition on top of our application, in the sense that it was not intrusive and posting feedback could be achieved with only few clicks. Additionally, from our perspective, the online platform provided by UserReport made it easy to see all the posts, which we could then easily translate in to development tasks for the backlog - if sufficient information was posted. One issue that we did encounter with the tool, was the lack of customization. We were not able to add any instructional text, or edit the features. We would like to have been able to make a few comments on the feedback process, in particular when and how to use the *idea* or *bug* tabs to post feedback.

We changed our Kanban by adding another column and another label in order to incorporate the feedback, such that we have a column with *Developer backlog* and a column with *User backlog*, which is shown in figure 5.4. We added the label *Demo feedback* to indicate that a work item was the direct result of feedback. With the extra column we are continually reminded of what the users would like, and we assure that no mix-up happens, which is important since user requirements should be prioritized higher than what we want to implement. However, we still had non-functional requirements that users may not have directly asked for, but that would be necessary for the system to work well, such as being able to upload files.

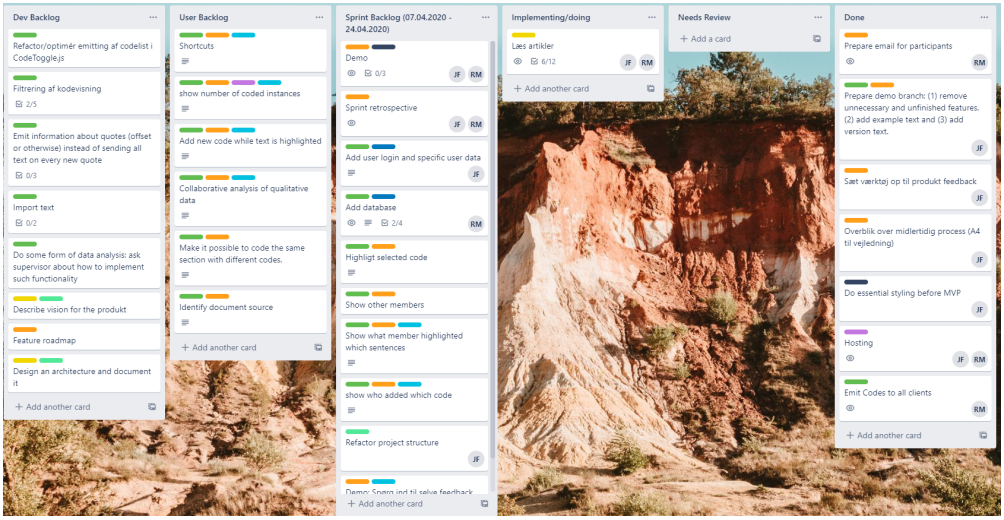

Figure 5.4: Adding another column to our Kanban. This Kanban is from the beginning of phase 1.

5.3 Phase 1

For this phase we planned the first development sprint, and the use of our light methodology. As described in chapter 4, this was also the phase where we had to extend the second sprint of this phase, in order to get ready for the next demo. We learned a great deal from the experience.

Processes and Tools

In general, the few processes and tools we were using during the first sprint of phase 1 were useful. The daily stand-ups were done through Discord, which a good way to get an overview of task progression, and what tasks were currently being worked on. However, some days we would skip a daily stand-up, for example if we were still working on the same thing as the day before.

We tracked bugs in an ad hoc manner, where the bug label on Trello and a single work item with a checklist were sufficient. Generally, we did not spend much time on fixing specific bugs throughout this phase, since some of the bugs would be solved through a change of architecture. Therefore spending time on specific bugs was not a priority. We had experienced some issues that had to be adapted in the next sprint. These were especially related to the fact that we had to add an additional short sprint, in order to have enough new functionality to show for the following demo. Using the sprint retrospective method was particularly useful when trying to clarify these issues.

During phase 1, we found that we had to break down the backlog tasks more than we did at the first sprint. In practice, we did this by adding a checklist for each of the tasks on Trello, representing a series of sub-tasks. By breaking down the tasks further, it was both easier to understand the problem and to quantify the size of the task. Addition-

ally, discussing the tasks together helped discover additional challenges and possible solutions. Another useful adaption to our process was the use of pair programming for tasks that were particularly difficult. One may fear that being two developers working on the same task is less effective, yet in our case, we ended up being more productive. We had technical challenges regarding some of the collaboration aspects of the application, which partially caused us to miss the sprint deadline as well as other unforeseen problems with other technologies. Utilizing pair programming helped us understand the problems and to implement a working solution faster. Lastly, we encountered problems regarding version control, when we deviated from our established approach. As described in section 4.2.1, our application was automatically deployed on Netlify, when we made commits to a demo branch. When we were setting up the demo for user feedback though, we discovered a few bugs, which we fixed immediately on the demo branch. This was a bad practice, since we then had to either merge the demo branch into the master afterwards, or fix the bugs one more time. The point is that we had to be more disciplined about following the version control processes, to avoid overhead later. During this phase of development we did some minor refactoring that turned out unsuccessful. Because of our lack of experience with React, we added unit tests where the purpose was to ensure correct re-rendering behavior of React components based on the updating of data. This lead to some unplanned tasks, that resulted in adding a label to our Kanban, such that we could mark work items with *unplanned*. The idea was that we during a sprint retrospective we could evaluate the amount of unplanned work.

Open source development

Although open source projects do not necessarily have the same requirements in terms of deadlines and sprint planning, the above findings can still be relevant to an open source context. Using daily stand-ups in our project not only gave an overview of the sprint progression, but also of the tasks being implemented, and thus the state of the application. This was an advantageous quality for us when we were prioritizing what to implement next, and what had to wait. In general, maintaining good communication proved an important quality in our project. Before breaking down the tasks thoroughly, our time estimation was way off, and we had an insufficient understanding of the tasks. Going through the user feedback suggestions together, breaking them down into backlog tasks and discussing potential steps on how to implement them, we got a much better understanding of the problems and how long they were going to take. This could potentially be an important practice in an open source project with additional developers. If contributors of such a project choose to include user feedback similar to us, creating a backlog based on the results, the incentive to break down tasks carefully could be even greater. Specifically, this practice may be a valuable way to ensure commonly perceived user profiles and product visions between contributors. Regarding version control, the presented issue may have been caused by our inexperience, however, ensuring a similar

discipline may be important in an open source project as well. If an open source project has DevOps practices, or some sort of live version like our demo, handling bugs and persistent changes should be done carefully. Depending on the version control practices, having multiple practitioners with different versions of the system, this issue could be important to consider.

5.4 Demo 2

For the second demo, we also received a series of useful feedback posts through UserReport. These included suggestions on new features, bug reports, and general domain knowledge. Although we did not plan on implementing all of the suggestions, or present a third demo, the feedback was still useful to the three areas of interest.

Domain and users

For this demo, we gave a few instructions to the users to ensure that they noticed the new features. This resulted in approximately the same amount of feedback posts on the UserReport form, which is summed up below. Each of the items translates to a feedback post, similar to how we presented the results from the first demo:

- It should be possible to see where quotes are in the source text, from the the quote window; either by adding line numbers to the code window or having the option of clicking a quote and it then getting highlighted in the source text.
- It should be possible to delete a code from the code window.
- It should be possible to see how many times a code has been applied to the source text.
- Bug: logged out of the system when clicking the '-' button.
- It should be possible to write a memo after marking a piece of text with the cursor.
- I rarely discuss codes synchronously, therefore the chat is not of much value.
- Colours should be more.

Most of these feedback posts were once again very useful in terms of domain knowledge, and gave a good idea on how to proceed with the development process. This time however, some of the posts were more specific in terms of the coding process itself. For instance, one user suggested that the chat gave little value to the application, based on his or her experience with similar applications. Similarly, a user suggested that it should be possible to see where quotes are in the source text, when showing the code window. This gives useful insight as to how this user works when coding, and what

features are important. This difference in feedback fidelity may be because the application was more developed, or it could be the result of giving the users brief instructions. Additionally, unlike the first demo, there were no feedback expressing a confusion on the purpose of the feedback, which may also be due to some clarification caused by these instructions. Another thing to note, is that there were no duplicate posts for this demo at all. As discussed for the previous demo, this could suggest that the users are aware of existing feedback posts. On the other hand, the voting system was hardly used for this demo either. Only a couple of posts were up-voted once. Lastly, considering the post titled "Colours should be more", we could still benefit from having a way to communicate with the users, since we do not know how to interpret this feedback.

The following figures document the version that was released as the second demo: figures 5.5, 5.6, and 5.7 show the user interface, while figure 5.8 shows the new hierarchy of components and 5.9 shows the architecture of the application.

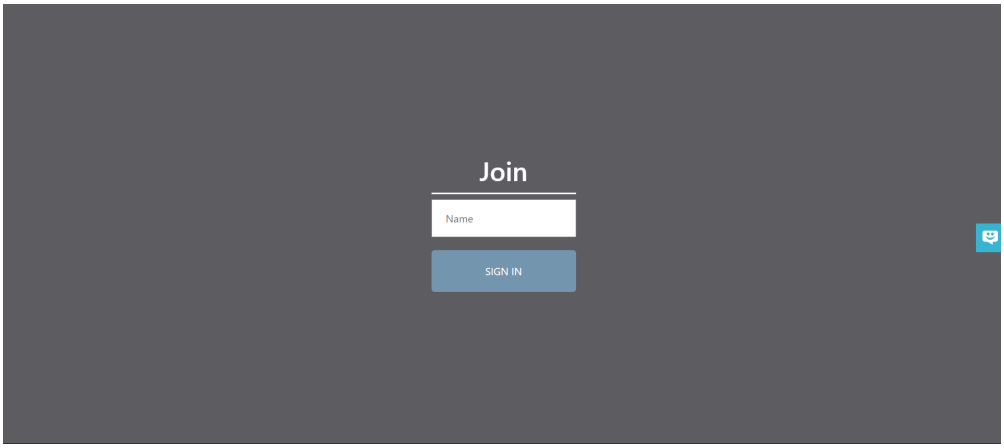

Figure 5.5: Screenshot of the application at the second demo (1/3). We re-purposed the *login* to the chat component to be the login screen to the application.

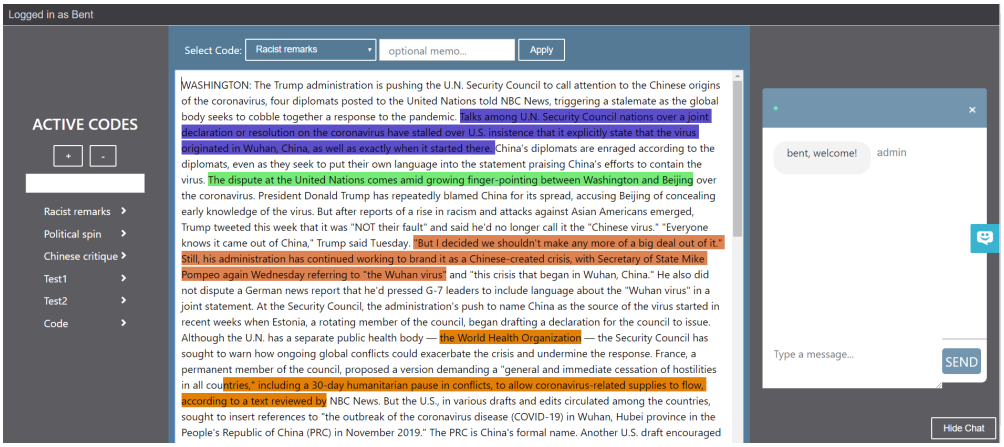

Figure 5.6: Screenshot of the application at the second demo (2/3). We now have persistent data, so the user is shown highlights when first logging in. We added the name of the user at the top of the screen, as well as being able to add a *memo* to a quote.

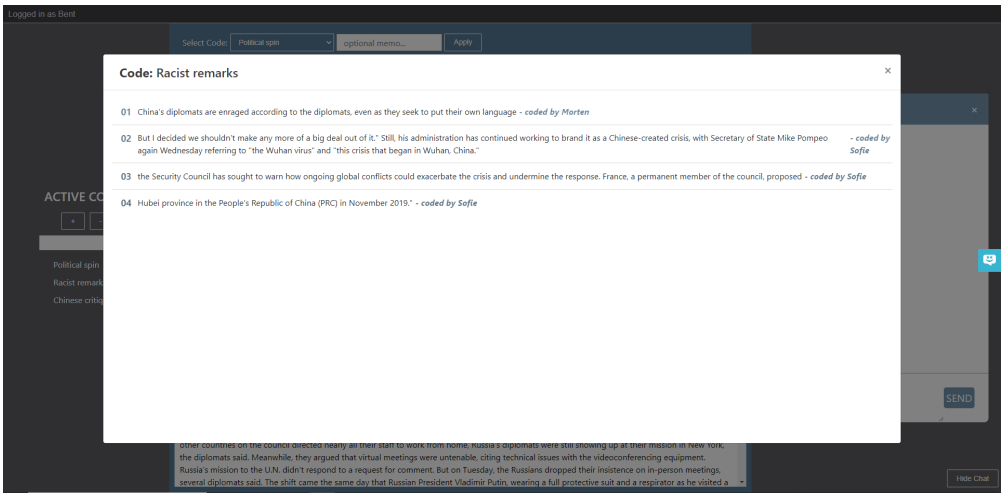

Figure 5.7: Screenshot of the application at the second demo (3/3). It is now possible to get an overview by clicking a code (figure 5.6) in the left hand side, after which this screen is presented to the user and all aggregated quotes of a code are shown.

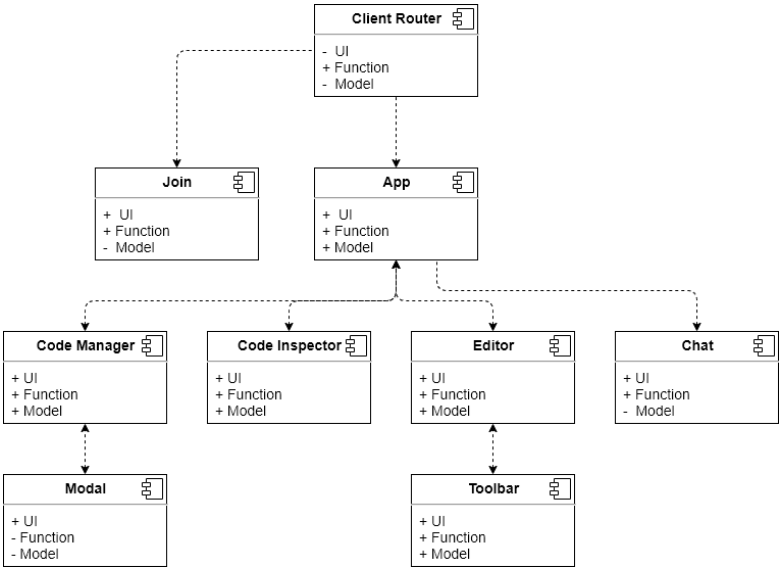

Figure 5.8: The component has changed, since we have moved the login screen to be the first view, this is reflected in the hierarchy through the *Client Router* component, which takes the user from the login screen to the main screen.

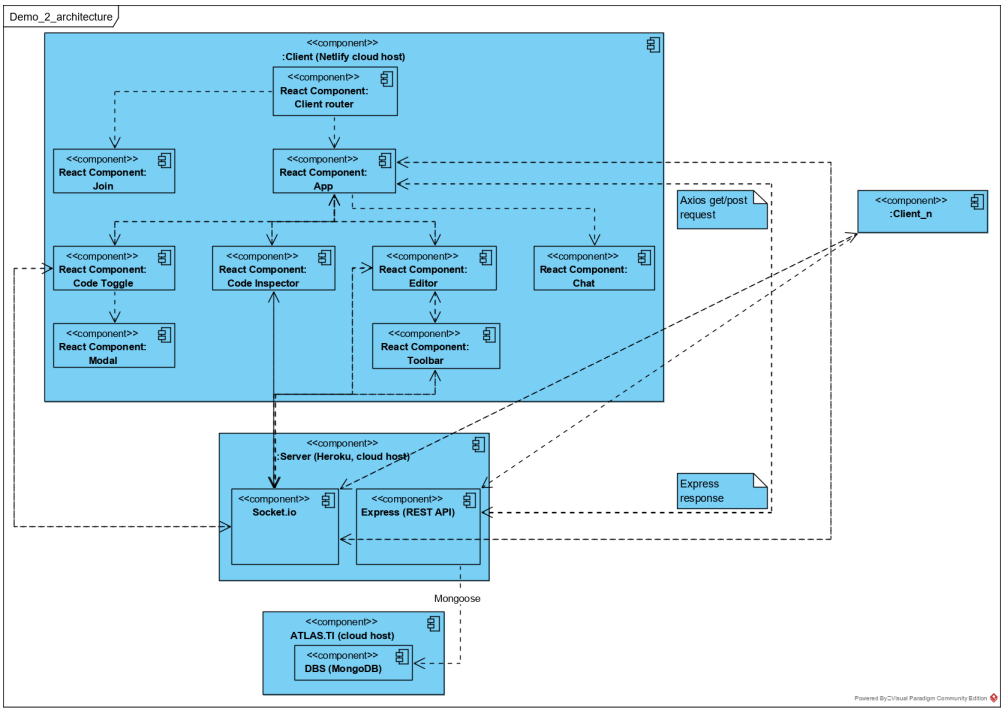

Figure 5.9: Architecture with a lot of dependencies between the socket.io component and various React components. We added a cloud hosted NoSQL database for persistent data, where we used Axios and Express for communication between client and server, and the Mongoose framework for object-document mapping (ODM), that also acts as the interface between the server and the database.

In terms of functionality we focused partly on adding persistent data e.g not something the users had requested, but something that is necessary for other features. The code view from figure 5.6 is an example of a feature that was requested. In phase 1, section 5.3, we mentioned that we had some unforeseen problems with some of the technologies. It was partly due to adding Axios/Express, which on a higher level is quite straightforward as figure 5.9 shows, however specific implementation details with these frameworks caused long delays, that ultimately lead to the second sprint and also to the extension of that sprint. In order to enable such communication, Express has to be configured with certain CORS policies enabled, such that any client from any origin can connect. Such details are solved by experience and we found that such details make it difficult to predict and estimate without thorough understanding of the technologies.

Open source development

The findings from the first demo in terms of open source systems are also applicable for the second demo. Using some sort of forum based approach, where we would be able to communicate with the users, could help clarify feedback that needs elaboration. Similarly, we could also make the voting system more apparent, which would help us prioritize what future development tasks will give most value to the application.

Process and Tools

Adjusting the feedback process by adding a few instructions to the users seems useful. As described above, some of the feedback was more detailed and gave us even better domain insight. Additionally, as we received no feedback regarding process ambiguity, the process may have been easier to understand. On the other hand, as argued at the findings from the first demo, there were a few shortcomings in terms of using the voting system and to get elaborations of the feedback. This could potentially be accommodated by the open source adaptions described above.

5.5 Phase 2

Process and tools

We decided to do high-ceremony pair programming at the end of the last sprint. We did in total three sessions spanning three days, where each session lasted four hours. During the session we would switch roles every 30 minutes, which was done by setting a timer, after which the programmer role would not be allowed to keep on writing code, but had to commit the code. The reason for this was that we had focused much on different aspects of the code, separating responsibilities in front-end and back-end. Another reason was because we used JavaScript, and because this language is dynamically typed, it sometimes takes longer to figure out what kind of objects are transmitted between the client and server and the server and database. With pair programming we are quicker to make these look ups, and since we both know different parts of the code-base, we experienced quicker implementation of functionality, since we typically need information about the whole stack, to implement a single feature. We did not do any formal tracking of velocity, but through our experience with these sessions we estimate that we achieved a greater velocity than before. Although refactoring was not an indented part of these sessions, we ended up re-writing how different React components would receive data; instead of many React components having their own reference to a socket, we moved data into a React component of a higher level in the hierarchy. Specifically, the *App* component, shown in figure 5.10.

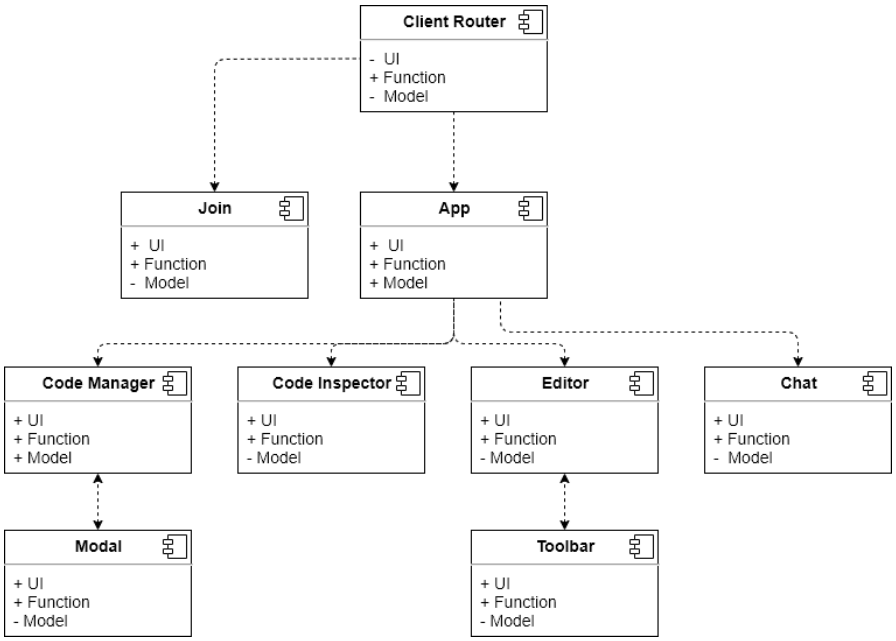

Figure 5.10: React components of the application. For each React component we included whether the components contain UI, functions, and Model.

Testing a new feature is also easier with pair programming; since the burden of having to think of all possible cases or scenarios that can go wrong can be put on the non-programmer, which leaves the programmer to executing the tasks and staying focused.

After the second demo, we started creating work items on our kanban that specifically addressed how to perform a demo; this included a basic checklist procedure to follow. The checklist included documenting the UI, as well as asking users about the feedback tool, to run a script that checks if the application is still running. It also includes general clean-up of the repository such as deleting old branches, and ensuring that changes to the demo were added to the master branch, thereby ensuring consistency of versions.

Just as in phase 1, we also skipped daily stand-up once in a while. However, we did more pair programming in general in phase 2, however, that pair programming through out phase 2 was generally of low ceremony, except for the three pair programming sessions at the end of sprint 3.

Open source development

The pair programming session resulted in significantly better code quality and a more coherent design. Previously, the React components *code manager*, *code inspector*, *Editor*, *Toolbar* and *Chat* from figure 5.10 would all instantiate socket.io components, resulting in duplicate messages for each client. Another problem was that the model was kept in separate components. The socket.io events, that were used to update the

model and keep data consistent between clients, would be used throughout the application instead of in a specific component. We refactored the application with regards to instantiating multiple sockets into passing along a single socket through a *context*, which solved the duplicate messages bug. The components *Code Inspector*, *Editor* and *Toolbar* would have part of the model as their state. This was removed during pair programming and the replacement was to pass functions to these components from the *App* component. We can see this reflected in the architecture by the removal of dependencies between these components and the *socket.io* component, by comparing figure 5.9 with figure 5.11. The last thing to refactor would be to move socket.io events into the *App* component as well, such that we get the ideal architecture outlined in figure 5.11, where all communication is restricted to a single React component.

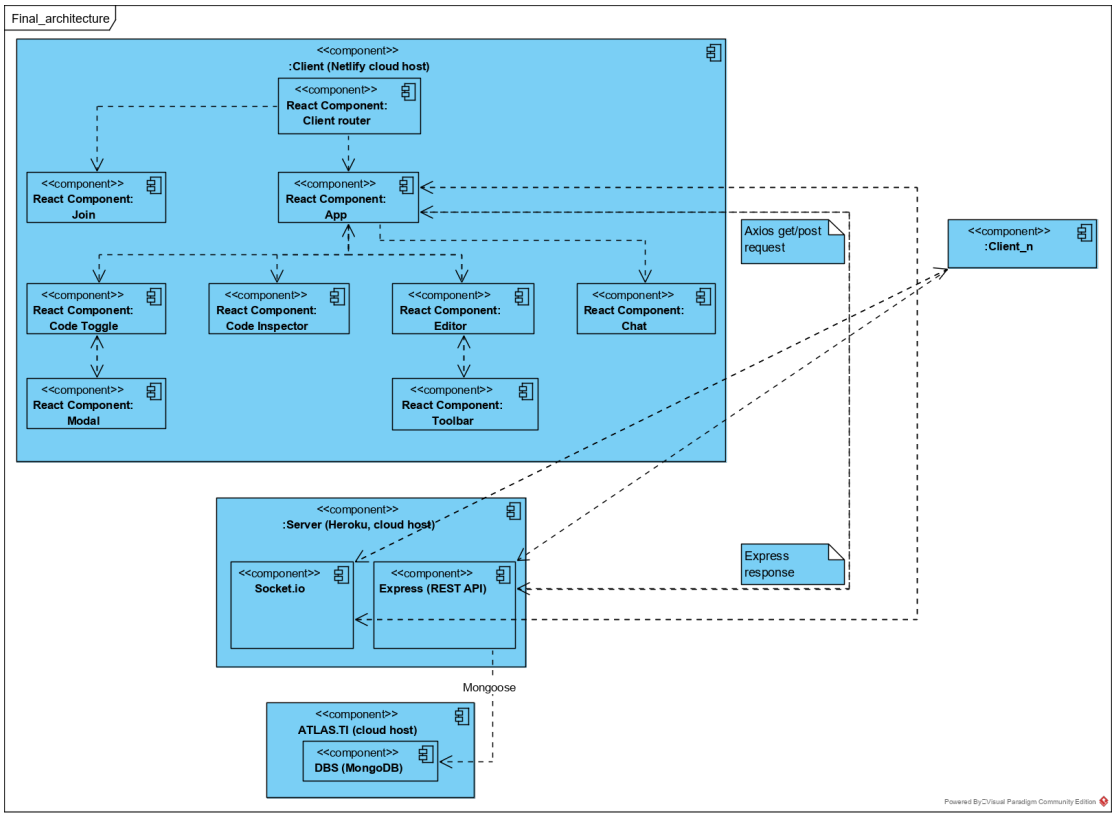

Figure 5.11: With an estimated days worth pair programming (refactoring) we will have all communication handled by the *App* component as portrayed in this figure. We follow the distributed architecture pattern, where [H server is kept relatively simple and its behavior is limited to forwarding data as well as acting as a REST API. The client implements the necessary functionality as well as each client having a copy of the model, where synchronization between components is handled through the socket.io component.

Domain and users

While we did implement three minor requests from the users, the main progress of the pair programming sessions was the refactoring. Figure 5.12 shows two of the requests and figure 5.13 shows the delete button for all the codes.

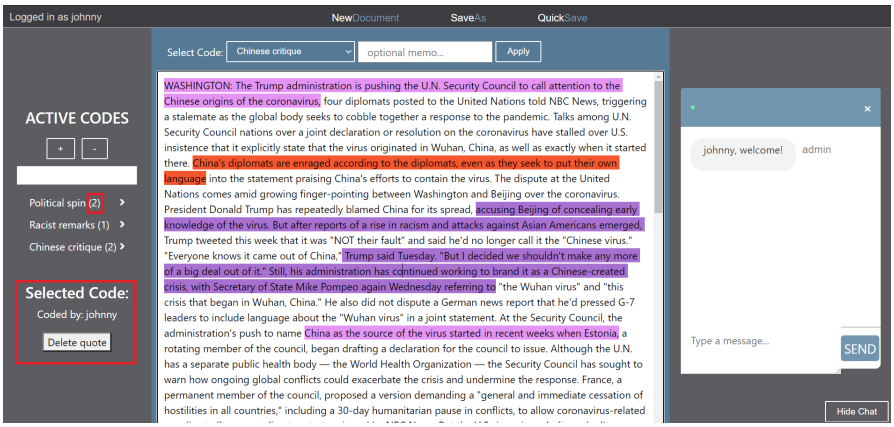

Figure 5.12: The areas marked with a red square indicate extra functionality. The upper red square indicates a number that corresponds to the number of times a code has been applied. The lower red square captures the part of the UI that shows which collaborator added the quote. These were both requested by the users.

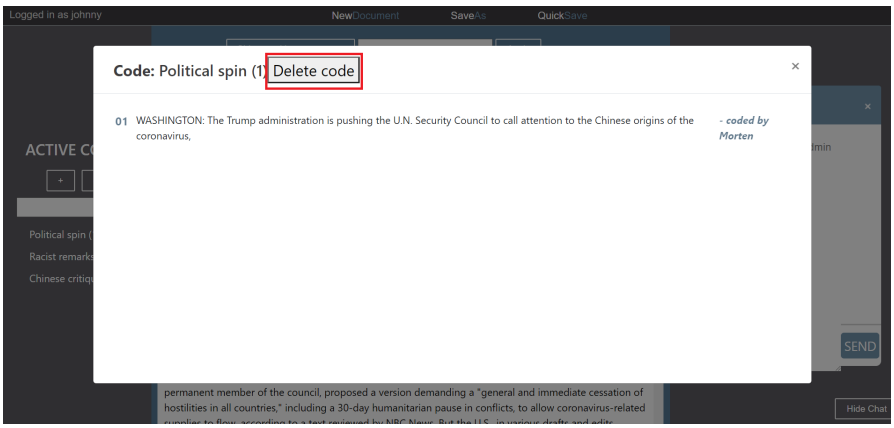

Figure 5.13: We added a delete button to this view, such that the code and aggregated quotes all can be deleted, as per user request.

Chapter 6

The project in retrospect

Throughout chapter 4 and 5 we have described our approach to development of the application and what we learned from the experience, respectively. In this chapter, we summarize our findings from chapter 5, as seen in table 6.1, which we will use throughout the chapter. We present implications from the development of an open source application and discuss the rationale for these implications as well as how they apply to literature. The discussion leads to the answering our research questions presented in chapter 3, where we end with a discussion on how to adjust our methodology to accommodate more contributors and users. Finally, we conclude the thesis, including potential future work.

Table 6.1: Summary of findings presented in chapter 5. The findings are divided into the three areas of interest, and related sub categories. The findings marked in bold are the ones we find the most interesting, and are discussing in this chapter.

Processes and tools	
Processes (and tools)	**1. Practices (e.g. daily stand-up, sprint planning) and tools (Discord, Trello) help with distributed communication and knowledge sharing.**
	2. Pair-programming can help simplify problems and can even be more efficient than working individually.
	3. Maintaining discipline regarding version control is particularly relevant when having "live" versions of the application.
	4. Pair programming is a good practice for ensuring a consistent design and high code quality, especially in a distributed setting
UserReport	**5. A light-weight tool with little overhead for both users and developers.**
	6. Seems to facilitate collaboration between the users.
	7. Voting system does not seem to work. Would be useful for developers.
Domain and Users	
Feedback	**8. Tools can be too light-weight for certain situations.**
	9. Users benefit from simple instructions about the feedback process.
	10. Having a user backlog and a developer backlog on the Kanban board helps sorting and prioritizing tasks.
Application	11. All unedited feedback about the application can be seen in appendix C.
	12. In general, the users want transparent information about collaborators; Who coded what? Which quotes are related to which codes? Who is currently online?
Open Source Development (future)	
General methodology	**13. Practices and tools facilitating distributed development may be essential for open source development.**
	14. We argue that collaboration is important when retrieving and breaking down tasks in a project with many developers.
	15. If an open source project uses DevOps or "live" versions of a system, version control discipline may be important to consider.
Forum-based feedback	16. Can enable instructions and communication between developers and users.
	17. Can enable elaboration of ambiguous feedback.
	18. Could create an incentive for user participation by showing a backlog and general progression.

The findings presented in chapter 5 are summarized in table 6.1. Although they are sorted in the three areas of interest, some of them are naturally interlinked. The findings written in boldface are the ones we find the most interesting in relation to our project, and they will be the basis of our discussion.

6.1 Discussion

In this project, we have developed an open source system at its early stages. We have designed an initial methodology, suited for a small distributed team, and adjusted it throughout the development process. We have inquired distributed user-feedback with users accustomed to the application domain. In chapter 5, we presented our findings observed throughout the process which we will discuss in this chapter. Some of these findings have certain implications, which we elaborate on and relate to existing literature below. Furthermore, we will discuss how to expand our project to additional contributors and users, thus answering the research question on how to bootstrap an open source project.

6.1.1 Implications

Distributed development requires effort to establish communication between developers.

Communication and collaboration is essential for any software development team. As we have described in a previous project [15], Cockburn dedicates an entire chapter of his book [14] to communication and team collaboration in agile teams. One of his implications is that teams should strive to be collocated and reduce physical barriers between them, to facilitate good communication. However, he acknowledges the fact that sometimes the circumstances do not allow for this behaviour, and has to be accommodated in different ways.

This is the same issue that we encountered when we went from being collocated developers at the initial phase 0, to being distributed for the remainder of the development process. Being collocated, it was much easier to keep a mutual understanding of the product vision as well as how the project progressed. When sitting next to each other, it does not take much effort to discuss issues or to be aware of what is being worked on. As described by finding 1, we found that the applied practices, and related tools, helped facilitate this necessary communication. Applying daily stand-ups, sprint planning, and sprint retrospective to our process, using Discord, was essential. As our experience shows in chapter 5, this was indeed helpful in terms of getting an overview of the project and its progression as well as to share knowledge and keep a common vision. However, we did skip daily stand-ups about every third day, and as such we deviated from the process.

Pair programming in remote development enables faster learning, better code quality and helps motivate development.

Another practice that we found particularly helpful to our development process was that of pair programming. Finding 2 encapsulates our thoughts on the values of using this method in our project, which we found to be particularly relevant when working distributed. As described in section 2.2.2, existing literature [19] recommends using pair programming in open source development, which made us apply it to our project. We applied pair programming in the second sprint of phase 1, while at the end of the third sprint we had a three days of high-ceremony pair programming sessions.

We found that using pair programming was more effective in terms of how much we could accomplish compared to working individually. Although one may think that distributing the tasks would prove more effective, this was not the case in our project. While we do not have any formal tracking of velocity, our intuition tells us that pair programming to be more effective in our case - particularly for complicated tasks (epics). Since we are both at a junior expertise level with respect to the technology stack we are using, this aligns with the findings of Dybå et al. [41]. In short, they present guidelines on when to use pair programming, and when solo programming is favorable, based on

the complexity of a task and the expertise levels of programmers. Their general recommendations are that junior-level programmers will benefit from pair programming, and experts should avoid it, unless the task is too complex. This is very much inline with our findings; breaking down tasks together and discussing possible solutions and issues helped us get a better overall knowledge of the architecture and flow of the application. Combining our knowledge of separate parts of the application and focusing on a single task at a time, not only helped on functionality, but also improved the code quality, as section 5.5 suggests through the improved architecture, which finding number 4 is based on.

A distributed team needs heavier methodology weight than a collocated team, and adjusting it requires intentional efforts.
Considering the implications above, our findings are inline with Cockburn's thoughts on methodology weight in relation to distributed development, and that this type of development requires a heavier methodology weight. As we had to add additional control elements to our process (daily-stand ups, sprint retrospective etc.) when we went from being collocated to distributed, our methodology size grew.

Additionally, as we experienced the negative effects of deviating from these processes, we had to be more disciplined (ceremonial) about it. In other words, we had to increase our methodology weight, to accommodate the project circumstances, of no longer sitting together, and having a good communication channel. One thing to note here, is that there also was a transitory change in our project. The point is that these practices also have been suitable to the project since we moved to a later an more defined stage, compared to when we were distributed. This is also one of Cockburns points, which we described in our previous project [15]; At the early stages of a problem, the methodology will naturally be lighter, since the software team needs a better understanding of the problem and the technologies in use. Similarly, when a project is at a later stage, the team can start applying methods suitable for the context.

In continuation of this, we found that this adjustment of methodology weight, is not a novelty, and takes conscious effort. As we have shown throughout chapter 5 our methodology *did* change, mostly during phase 2. Especially in the beginning we had a particular problem, which is expressed by Weinberg in [42] in the following way, of how problems are handled in an *oblivious culture*: *"Problems are suffered in silence."*. This was a characteristic of our early distributed development. In the beginning of phase 0, before the distributed setting this was not a problem, and it was during phase 2, where we were utilizing the practice of pair programming in a more structured manner that allowed better handling of problems.

This allowed us to go from an oblivious culture to the variable culture pattern. Cockburn mentions [14] the importance of *bothering to reflect* as well as specifying that *reflective improvement* is one of the three *core* properties of the *Crystal Clear* method-

ology, and without reflection one will not change their ways of working for the better. By looking back now, through this cultural pattern, we can see the necessity of reflection.

Teams getting user feedback in a distributed setting should balance low overhead and sufficiency.
As implicated by finding 5, the tool UserReport proved to be a light-weight tool with little overhead, and generally suitable for projects in an early stage of development. It is convenient and easy to use from a user perspective, and as our sprints show we never *lacked* quantity in feedback. In general the feedback can be characterized as *one-liners* (1-5 lines sometimes). However, this is to be expected with such a tool.

It was also easy to implement from a developers perspective, by adding a single script tag to the code. The platform provided by UserReport was also straightforward to use, and made it easy to find the feedback and translate it in to backlog tasks. On the other hand, as indicated by finding 8, we did encounter certain situations where the tool was too light-weight. We noticed this when one of the users made a feedback post about being confused about what the purpose of the evaluation was, and what we wanted to achieve with the application.

Additionally, we received a few feedback posts that were ambiguous or could preferably be elaborated on. In other words, while UserReport has low overhead, the lack of customization and limited functionality is a limitation in certain situations. It would be useful if we can customize the feedback tool, make small instructions and be able to communicate with the users during the feedback. At the same time, we would have to be careful not to make the process too complicated, where related work from section 2.3 suggests that this could negatively influence user participation.

Prolonged development would benefit from a more advanced tool. At a later stage when the application is closer to a finished product, our focus shifts to that of usability, which can be measured by different tools and methods. While the step towards usability would have to be taken at some point, one such intermediate step could be to use the project management features of GitHub. since we already use GitHub to host our source code, a natural progression would be to use those features as well.

Additionally, GitHub is easier to customize, and offers an array of additional features, such as general issue tracking (including bug reporting) and a public kanban. A disadvantage of GitHub is that we increase the level of effort for the users; by the requirement of having a GitHub account as well as switching between 'tabs' when providing feedback. We did briefly consider changing our feedback method to use GitHub issues instead, but since we were in the latter part of phase 2, we did not have time.

Early technology familiarization requires a considerable time investment, and can be difficult to estimate.

A pre-requisite of development and software engineering processes is a certain level of understanding of the technologies and programming languages that are being used. During development we found that a symptom of lack of understanding can be seen in task estimation; the more you know about the technological platform the better one is able to estimate a given task. The first sprint we had in phase 1 had work items left in the backlog, which was the reason for the second sprint. The second sprint was planned to last a week, in order to complete that same backlog, but was extended due to our estimations being off point. Then in the third sprint we did not progress very quickly, and therefore we changed the purpose of the sprint; to experiment with high ceremony practices. The reason that our estimates were incorrect was typically because of unforeseen problems with the technologies we were using, for example the problem with configuring Express, that we described in section 5.4.

Cockburn [43] recommends that each team has a designated lead designer/programmer on the team. This role should be covered by the person with the most experience as this person will typically have to mentor the other designer/programmers. The problem with configuring Express is not a difficult problem, however, the time an inexperienced programmer has to use compared to an experienced are tremendous. Therefore, having one experienced person on the team will make development much smoother, especially with osmotic communication, since such Express problems can be easily solved by a team member even with little experience.

In retrospect, the task of estimating work items in terms of how many man-hours (or days), did not support development. Therefore such methodology weight can be removed, since there is a high percent chance of getting the estimation wrong, which creates unnecessary tension and demotivates the team unnecessarily. This is an example of methodology weight that does not contribute to the overall development process, even though we do in general need more methodology, that it not the right *kind* of weight. Kniberg suggests in the book *SCRUM and XP from the Trenches* [44](p. 65) to skip task estimation entirely, since many teams through experience will be able to break down tasks into 1-2 day chunks at some point. We also experienced that breaking down tasks has advantages in our context, while time estimating did not.

An important issue that the agile manifesto raised was to change how progress was measured. Instead of measuring progress by the amount of documentation, in the form of requirements specification, design documents or otherwise, we measure progress by running code. According to the agile manifesto this is the primary measure of progress [11]. In line with this principle, we measured our progress by the amount of work items (features) that we had implemented. We did not measure this formally, but we needed a certain amount of features to show our users, and therefore features became the deciding factor of when to hold a demo.

Another practice we could have employed was to track *learning*. In the previous paragraphs we discussed how our estimates would not be accurate, and therefore we

had to adjust sprints and it would seems that we *lacked* progress. Cockburn mentions the effect of slower development, where one of the effects is on lower morale. Yet, we progressed in other manners, and visualizing this, perhaps secondary measure of progress, is important to boost morale.

Not all methodology weight is equal
We have already discussed, based on finding 2 and 4, that pair programming is a useful practice for junior developers in a distributed setting. As we discussed not all methodology weight is the right amount of weight. Pair programming in our case is, and with the added ceremony of the three day pair programming session was a significant methodological upgrade, which resulted in a much cleaner architecture with fewer dependencies.

However, we could have achieved this through other methodological elements. We could for example have declared a *standard* that says that all client side communication must be handled by the *App* component. Another approach could have been through automated tests; where a test suite will fail, if any other component besides the *App* component contains references to the socket.io component. A third option would be to hold code reviews. However, the advantage of pair programming is that we cooperate in order to get a consistent architecture and few unnecessary dependencies, rather than establishing standards or quality assurance measures that that are more burdensome. Therefore, even though we have many ways of increasing the methodology weight, the preferred choice is pair programming.

These practices share the same goal of assuring high quality code and few architectural dependencies, however the burden of these practices are different. Therefore the implication *Not all methodology weight is equal*, where in this context pair programming is the preferred practice.

6.1.2 Adjusting our methodology for additional contributors

Based on our experience with distributed development, that we summarized through implications, we describe how we would adjust our methodology in order to accommodate a large number of contributors and a large number of users. Several of our findings are about distributed development - which is a given in open source development. Therefore, these implications apply when designing an open source methodology.

An increase of contributors

The first implication emphasises the importance of communication and collaboration, and that the practices and tools we used helped facilitate this in a distributed setting. However, in an open source project with tens or even hundreds of contributors, our current practices (daily stand-up, sprint planning, pair programming) would not be ideal.

These practices require synchronization of developers, which is difficult in an open source context. Keeping a common vision, supporting communication, ensuring high code quality would have to be done through practices that do not require direct communication between developers, such as with daily stand-up, sprint planning, and pair programming.

We mentioned some of these alternative practices for quality assurance, where code review would be ideal, since this requires no synchronization of developers like pair programming. Pair programming can still be done between contributors, but coordination will be difficult and the things that are discussed during pair programming would not be transparent to the community at large. Code review on the other hand is something that a whole community can participate in, and therefore a more efficient practice for a larger community.

To accommodate additional developers it is also important to consider participation barriers, as we have elaborated in section 2.2.1. In particular, the barriers identified by Alami and Wasowski [17] regarding a high cost of participation and being unfamiliar with the system, can be managed by an OS community. Alami and Wasowski suggest reducing unnecessary participation formalities and find ways to help integrate newcomers in the community.

This is essentially about knowledge sharing and our current practices have included daily stand-up and pair programming. These practices are not ideal to handle newcomers, since this would require a lot of effort by current contributors. Therefore, a well documented system with component hierarchies (figure 5.10) and general architecture outline (figure 5.11 would help newcomers in understanding the structure of the system. Such figures could be added to the source code project hosted on GitHub.

An increase of users

Based on related work, we analysed problems about getting user feedback from distributed users in section 3.1. Although getting the right user feedback is a valuable practice in OS development, the nature of OS projects presents several challenges. The contributors are volunteering, distributed developers with no allocated resources, and may not have the necessary knowledge in terms of user centered design.

Furthermore, conventional methods used to get user feedback require the physical presence of users. The users themselves may not even be identified, and can be difficult to engage in the development process. In fact, Llerena et al. [23] found that one of the biggest challenges when adapting feedback techniques to open source software, was to find and engage users. To find more users, they suggest promoting and communicating with the users through online wikis and social networks. In terms of engaging the users, they suggest finding some sort of incentive for users to participate, but do not specify how.

In continuation of this, Llerena et al. [23] also found that to keep users engaged,

the feedback process cannot be cumbersome and time consuming. To accommodate these challenges, we suggest considering our implication that teams should balance low overhead and sufficiency when getting distributed feedback. It has to be a quick and easy process to give feedback, while still providing sufficient contributions. Specific to open source systems, as we have summarised in findings 15-17, we suggest using a forum-based approach as described in [22], or extending our process to include a chatroom. That way, we would be able to ask users to elaborate on their feedback when necessary, answer questions, and give them instructional guidelines.

Finally, showing a backlog of the tasks drawn from the feedback, making the development process transparent, we may create an incentive for the users to engage in further participation. Being able to see the development progress and how the feedback suggestions are being implemented (or fixed), could motivate participation. We would do this with the built-in functionality on GitHub, essentially moving our current kanban to that of GitHub.

The culture

Based on the literature presented in section 1.1, we consider alternative ways of attracting potential contributors. We know from the literature that open source projects are driven by motivated individuals who voluntarily invest their own time. And that the underlying culture is driven by solving interesting problems, in an environment of absence of deadlines or other time constraints. Therefore, presenting clear and well formulated technical problems might attract contributors.

6.2 Conclusion

We conclude by answering our research questions from chapter 3.

RQ2.1: *What kind of functionality, architecture, and software components can be used to build a real-time collaborative qualitative coding tool?*
Based on literature, existing knowledge, and the feedback received from users, we have developed a web application with the MERN technology stack, where the application is implemented with the distributed architectural pattern. We showed through our findings that our implication *Pair programming in remote development enables faster learning, better code quality, and helps motivate development* that to ensure such an architecture and to ensure low dependencies among components implemented with socket.io, pair programming can be a suitable quality assurance practice for junior level developers. An overview of the functionality, as well as how it evolved throughout the process, is described specifically in chapter 5.

RQ2.2: *How can we apply user feedback methods in the context of an open source project, with voluntary distributed users?*
Through our findings we argued for the implication *Teams getting user feedback in a distributed setting should balance low overhead and sufficiency*. We found that a lightweight tool, integrated in the web application, provided appropriately low overhead, while still facilitating valuable feedback. The tool UserReport proved to be sufficiently easy to use, both from a user and a developer perspective, but did present a few shortcomings. Not being able to customize the input forms, and having no communication channel to the users, was a problem when analyzing the feedback. Therefore, we suggest adapting or combining UserReport with external communication channels - particularly in later stages of development.

RQ2.3: *What kind of tools and practices can be used to support a minimal open source process in the context of distributed development?*
The implications *Distributed development requires effort to establish communication between developers* and *A distributed team needs heavier methodology weight than a collocated team, and adjusting it requires intentional efforts* as well as *Teams getting user feedback in a distributed setting should balance low overhead and sufficiency* answer this research question. In particular, we found that when developing in a distributed fashion, certain practices enable a break of cultural pattern, e.g. going from an oblivious culture to a variable culture.

RQ1: *How can we bootstrap an open source project?*
We have bootstrapped an open source project, by using well known software engineering tools and practices, as is described in following implication *Early technology familiarization requires considerable time investment, and can be difficult to estimate*. While the mentioned implication covers early development, practices change as the project opens up for more contributors and users. This we have described in section 6.1.2.

Bibliography

[1] *Coding (social sciences) - wikipedia*. https://en.wikipedia.org/wiki/Coding_(social_sciences). Accessed: 2020-06-03.

[2] Peter Axel Nielsen. "Collaborative coding of qualitative data". In: *White paper—LA2020. Kristiansand S, Norway: University of Agder* (2012).

[3] Johnny Saldaña. "An introduction to codes and coding". In: *The coding manual for qualitative researchers* 3 (2009).

[4] Gordon Haff. *How open source ate software: Understand the open source movement and so much more*. Apress, 2018.

[5] Eric S. Raymond. *The cathedral and the bazaar - musings on Linux and open source by an accidental revoltionary (rev. ed.)* O'Reilly, 2001. ISBN: 978-0-596-00108-7.

[6] *Socket.io licence*. URL: https://github.com/socketio/socket.io/blob/master/LICENSE.

[7] Mark Johnson OSS watch. *Licence differentiator*. URL: http://oss-watch.ac.uk/apps/licdiff.

[8] Richard Stallman. *The Four Essential Freedoms*. URL: https://www.gnu.org/philosophy/free-sw.html.en.

[9] Chao-Kuei. *Categories of free and non-free software*. URL: https://www.gnu.org/philosophy/categories.html.

[10] Pierre Bourque, Richard E Fairley, et al. *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press, 2014.

[11] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. *Manifesto for Agile Software Development*. 2001. URL: http://www.agilemanifesto.org/.

[12] I Sommerville. *Software Engineering (Global Edition), 10e*. 2015.

[13] *Visual paradigm*. URL: https://www.visual-paradigm.com/.

[14] Alistair Cockburn. *Agile Software Development*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0-201-69969-9.

[15] Jesper Fyllgraf and Ruben H. Mensink. "Adjusting methodology weight: A case study examining how plan-driven qualities are added to agile methodologies". In: (2019).

[16] Adam Alami, Yvonne Dittrich, and Andrzej Wasowski. "Influencers of quality assurance in an open source community". In: *2018 IEEE/ACM 11th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE. 2018, pp. 61–68.

[17] Adam Alami and Andrzej Wąsowski. "Affiliated participation in open source communities". In: *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE. 2019, pp. 1–11.

[18] Eric Raymond. "The cathedral and the bazaar". In: *Knowledge, Technology & Policy* 12.3 (1999), pp. 23–49.

[19] Manuela Angioni, Raffaella Sanna, and Alessandro Soro. "Defining a distributed agile methodology for an open source scenario". In: *Proceedings of the 1st International Conference on Open Source Systems, July 11-15, 2005 Genova, Italy*. 2005.

[20] Henrik Hedberg, Netta Iivari, Mikko Rajanen, and Lasse Harjumaa. "Assuring quality and usability in open source software development". In: *First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07: ICSE Workshops 2007)*. IEEE. 2007, pp. 2–2.

[21] Adam Alami, Marisa Leavitt Cohn, and Andrzej Wąsowski. "Why does code review work for open source software communities?" In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 1073–1083.

[22] Anders Bruun, Peter Gull, Lene Hofmeister, and Jan Stage. "Let your users do the testing: a comparison of three remote asynchronous usability testing methods". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2009, pp. 1619–1628.

[23] Lucrecia Llerena, Nancy Rodriguez, John W Castro, and Silvia T Acuña. "Adapting usability techniques for application in open source software: A multiple case study". In: *Information and Software Technology* 107 (2019), pp. 48–64.

[24] *Coding analysis toolkit (CAT) documentation*. URL: http://cat-help.texifter.com/.

[25] *RQDA: How an open source alternative to ATLAS.ti, MAXQDA, and NVivo opens new possibilities for qualitative data analysis*. URL: https://philippejoly.net/2018/12/03/rqda/.

[26] *Open Source Qualitative Coding Tool #71*. URL: https://github.com/ropensci/unconf18/issues/71.

[27] *Qual coder*. URL: https://github.com/ccbogel/QualCoder.

[28] *Atlas feature comparison*. URL: https://atlasti.com/atlas-ti-product-feature-comparison/.

[29] *Dedoose features*. URL: https://www.dedoose.com/home/features.

[30] *NVivo collaboration*. URL: https://www.qsrinternational.com/nvivo-qualitative-data-analysis-software/about/nvivo/modules/collaboration.

[31] *HyperResearch*. URL: http://www.researchware.com/.

[32] Jakob Nielsen. *Usability engineering*. Morgan Kaufmann, 1994.

[33] Kimmo Tarkkanen, Ville Harkke, and Pekka Reijonen. "Are we testing utility? Analysis of usability problem types". In: *International Conference of Design, User Experience, and Usability*. Springer. 2015, pp. 269–280.

[34] *Heroku*. URL: https://www.heroku.com/.

[35] *Netlify*. URL: https://www.netlify.com/.

[36] Nicolas Serrano and Ismael Ciordia. "Bugzilla, itracker, and other bug trackers". In: *IEEE software* 22.2 (2005), pp. 11–13.

[37] *Web based bug tracking*. URL: http://www.aptest.com/bugtrack.html.

[38] David J Anderson. *Kanban: successful evolutionary change for your technology business*. Blue Hole Press, 2010.

[39] *Web service Trello*. URL: https://trello.com/.

[40] *Socket.io*. URL: https://socket.io/.

[41] Tore Dybå, Erik Arisholm, Dag IK Sjøberg, Jo E Hannay, and Forrest Shull. "Are two heads better than one? On the effectiveness of pair programming". In: *IEEE software* 24.6 (2007), pp. 12–15.

[42] Gerald M Weinberg. *Quality software management (vol. 3) congruent action*. Dorset House Publishing Co., Inc., 1994.

[43] Alistair Cockburn. *Crystal clear: A human-powered methodology for small teams: A human-powered methodology for small teams*. Pearson Education, 2004.

[44] Henrik Kniberg. *Scrum and XP from the Trenches*. Lulu. com, 2015.

Appendices

A First demo

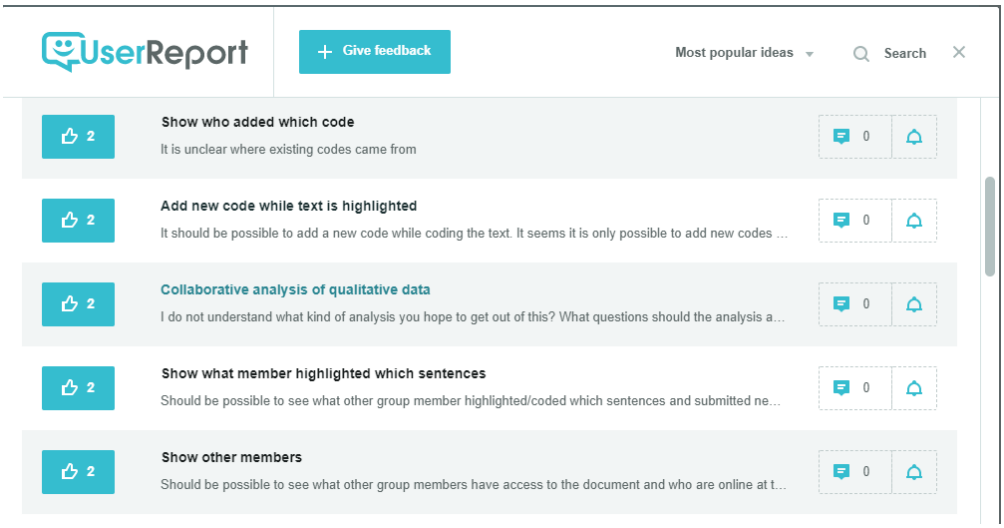

Figure 1: Screenshot of the feedback from the first demo, received through UserReport

B Second demo

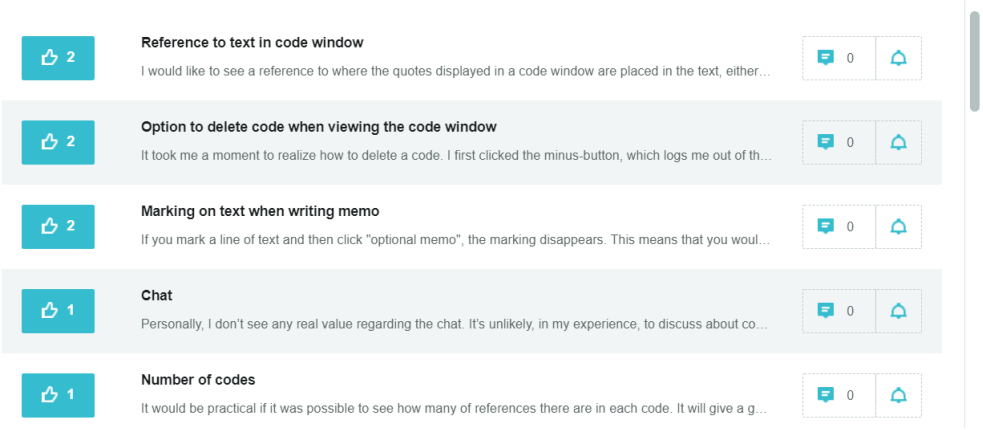

Figure 2: Screenshot of the feedback from the second demo, received through UserReport

C An overview of all received feedback without editing

C.1 First demo

- show number of coded instances: An overview of how many times each code is applied
- Show what member highlighted which sentences: Should be possible to see what other group member highlighted/coded which sentences and submitted new tags.
- Highlight selected code: When selecting a code the relevant codes should be highlighted in the source. When working together, this feature would show how my collaborators interpret codes and help get an overview over related codes.
- show who added which code: It is unclear where existing codes came from
- Shortcuts: When coding large amount of data, it can be time-consuming to click around. Shortcuts (right-click to code) would be appropriate, but it is a nice-to-have feature.
- Identify document source: It should be possible to see metadata about whatever document is being coded
- Show other members: Show other collaboraters*, other people who are online
- Add new code while text is highlighted: It should be possible to add a new code while coding the text. It seems it is only possible to add new codes prior to analysis
- Make it possible to code the same section with different codes: It should be possible to apply two different codes on the same area.
- Collaborative analysis of qualitative data: I do not understand what kind of analysis you hope to get out of this? What questions should the analysis answer? Moreover, I suggest you consider what collaboration entails. Crowd sourced analysis of qualitative data may be a more accurate label on what you hope to achieve.
-

C.2 Second demo

- Number of codes: It would be practical if it was possible to see how many of references there are in each code. It will give a good overview.

- Option to delete code when viewing the code window: It took me a moment to realize how to delete a code. I first clicked the minus-button, which logs me out of the system, before figuring out that you have to write the name of the code you want to delete and then click "-". I think it would be practical to have the opportunity to delete a code when you click it and view the code window. The existing solution seems a little odd to me.
- Marking on text when writing memo: If you mark a line of text and then click "optional memo", the marking disappears. This means that you would always have to write the memo before marking the text, which I could see be impractical in some cases. Would be nice if the marking persisted when writing memo.
- Reference to text in code window: I would like to see a reference to where the quotes displayed in a code window are placed in the text, either by adding line numbers or having the option to click on a code and then it get marked in the text. For the latter option, the code window should probably not cover all of the text view.
- Chat: Personally, I don't see any real value regarding the chat. It's unlikely, in my experience, to discuss about codes synchronously.
- Bug: Marking on text disappears when typing memo: If you mark a line of text and then click in "optional memo", the marking disappears. This means that you have to always write a memo before marking the text, which could probably be impractical in some cases.
- Bug: Logged out when clicking "-": I get logged out when I click the minus-button in the left panel.

D Instruction e-mails to our users

D.1 First email

"Thank you for agreeing to act as demo tester and your effort is much appreciated. We hope to develop our application driven by feedback from users, that is, we are pursuing a user-centric and agile approach.

We are iteratively developing an open source tool to support users (researchers) in the analysis of qualitative data. We hope that you have previous knowledge of what analysis of qualitative data is in an empirical research process. The application is at a very early stage of development, where we are currently working towards a minimum viable product (MVP). We are at this stage primarily interested in ideas for features and feedback on the user interface itself, but of course bug reports are always welcome.

The main focus is of the web application is to support collaborative analysis of qualitative data (i.e., coding) in real time. Therefore, during the demo, you may see codes being added, or text getting marked randomly. This happens because all participants work on the same set of qualitative data.

Getting a feel for the application, in its current state, should take no more than 5-10 minutes. The application will be open for demo testing from Monday at 9:00 to Tuesday at Noon.

The demo test can be conducted in the following way:

- Follow the link: https://fervent-albattani-caf461.netlify.com/ which directs to the web application itself.
- Move around on your own in the application – there is still very little functionality – and get a feel for it
- Try to select text and add a code to it, and check how it works. We rely here on your previous knowledge of what qualitative data analysis is.
- Try the chat panel to see if anyone else is here at the same time, and check how it works
- Provide feedback on the teal colored button with a smiley face, on the right side of the screen
- Select a name for yourself so that you are anonymous for the developers (your email will be hidden if you enter that)
- Enter new ideas and enter bug that
- Please also vote if you see comments that coincide with your own comments

Thank you for your participation and we appreciate the feedback very much!

Best regards, Jesper Fyllgraf and Ruben Henrik Mensink"

D.2 Second email

Thank you for agreeing to be demo testers once again, your feedback from the previous session is much appreciated and helped guide development in a good direction. While we have implemented some of the feature requests that were suggested, we have also implemented some additional infrastructure code, such as adding a database and a REST API. We have certainly not forgotten about the remaining suggestion of features.

The main difference between the two demo's is that now the state is saved, while maintaining the real-time collaboration aspect of the application. As such, when using the application during this demo, what all previous users and current users are working on can be seen by everyone. As for now we have no access rights between users, so everyone can delete anything.

The current state of the application is somewhat poor in performance, this is partly due to the free hosting services that are used, which most likely causes small delays when working with the application, as well as the first time the application is visited.

The demo test can be conducted in the following way:

- Follow the link: https://fervent-albattani-caf461.netlify.com/ which directs to the web application itself.
- Move around on your own in the application – there is a little more functionality now, and it should be much easier to get a feel for the real-time aspect of the application.
- Codes: we suggest to not delete the first three codes, but feel free to mark pieces of text with those codes.
  - Add a code of your own
  - Use your new code to select quotes
  - View your selected quotes by clicking on a code in the left hand panel
  - Delete your code (which will delete all the associated quotes).
- Quotes: select your own or one of the quotes already present in the document and view the associated memo of the quote and who added it
  - Create a quote and add a memo, and refrain from creating a quote, from text that is already marked, since this is an unhandled case and will cause a bug
  - Delete a quote
  - Consider using the chat if any other demo tester is present during your testing time.
- Provide feedback on the teal colored button with a smiley face, on the right side of the screen
- Select a name for yourself so that you are anonymous for the developers (your email will be hidden if you enter that)
- Enter new ideas and enter bug that
- Please also vote if you see comments that coincide with your own comments

Thank you for your participation and we appreciate the feedback very much!