Task-focused Editor for Qualitative Data Analysis

BACHELOR THESIS

Leonie Färber

Submitted on 23 October 2023

Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Julia Mucha, M.Sc.
Prof. Dr. Dirk Riehle, M.B.A.

FAU
Friedrich-Alexander-Universität
Faculty of Engineering

Declaration of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

Erlangen, 23 October 2023

License

Erlangen, 23 October 2023

Abstract

Qualitative Data Analysis (QDA) requires a researcher to perform a wide range of tasks, such as editing text documents, defining codes, coding documents, refining the code system, etc. Software supporting qualitative research can assist in many of these processes. However, providing all functionality in one place can lead to distraction, potentially complicating a focused workflow. To address this issue, it is beneficial to divide functionalities based on tasks and present them in separate views. Furthermore, these views allow for a better use of available space, ensuring only relevant functionality is displayed, thereby allowing the user to work more efficiently.

In this thesis, we apply the concept of task-focused views to QDAcity, a cloud-based web application supporting QDA. Currently, the Coding Editor provides a multitude of functionalities without a clear overarching structure. To facilitate task-focused work, we restructure the Coding Editor, enabling users to select tasks from a central tab header. Additionally, this thesis extends the existing functionality by developing two new task views. The first concentrates on refining the code system, offering information on selected codes, and allowing users to restructure associated codings. The second view is customizable, permitting users to define new tasks and arrange editor elements efficiently.

Contents

List of Figures

List of Tables

Acronyms

API	Application Programming Interface
CAQDAS	Computer Assisted Qualitative Data Analysis Software
DTO	Data Transfer Object
GT	Grounded Theory
GUI	Graphical User Interface
QDA	Qualitative Data Analysis
UI	User Interface
UML	Unified Modeling Language

1 Introduction

Qualitative Data Analysis (QDA) is a valid research method aiming at the development of theories (Stol et al., 2016). QDA is especially popular in the field of social science and arts but can also be applied to research in other areas such as software engineering (Dey, 1993; Stol et al., 2016).
CAQDAS can aid many of the tasks performed in QDA. Several software tools support storing and structuring text and image data, coding documents, analyzing and displaying code information, and numerous other features (Creswell & Creswell, 2018). Using CAQDAS has various benefits for the user such as allowing for more time-efficiency that requires less effort, inspiring a more precise analysis of data as well as performing graphic visualization of findings. However, these benefits have to outweigh the disadvantages of having to learn how any of these software tools work (Merriam & Tisdell, 2015).
There are several approaches to qualitative research that imply different practices for the analysis process (Graue, 2015). While all of these approaches differ in some way or another, there are common components or tasks to be found between many of them (Miles & Huberman, 1994). When developing software for assisting QDA, it is important to consider how to provide the functionality for performing these tasks in a user-friendly manner. As discussed before, using a QDA tool should benefit a user in their workflow rather than requiring more effort. This implies structuring the functionality in a way that is efficient for a QDA workflow. Some tools such as MaxQDA will provide all functionality in one place. This approach may be overwhelming for new users which will in turn require more support in the form of tutorials and other explanatory content.[1]
To avoid overwhelm, the available features can also be provided in the form of task-focused views. Here, the functionality for each task will be displayed in a different view. Therefore, a user will only see features that may support the currently performed task. This is also a more space-efficient approach, allowing task-related interfaces to be larger.

[1]https://www.g2.com/products/maxqda/reviews/maxqda-review-4552929

1.1 Goal of this Thesis

This thesis aims at developing a task-focused editor for QDAcity, a cloud-based web application supporting QDA. The coding editor is a central part of the tool and provides most of the features necessary for performing QDA. It already contains a number of functionalities that are mostly provided in one place. The goal of this thesis is to restructure the editor allowing for task-focused work, and efficient task-switching. Additionally, we ensure that the editor is capable of supporting most of the standard tasks performed in QDA. The developed task-focused interface should allow for more efficient and focused work.

2 Related Work

This chapter briefly outlines common QDA practices and emphasizes potential tasks that can be derived from them. Then we go into detail on other CAQDAS tools before discussing the current functionality and interfaces of QDAcity.

2.1 Qualitative Data Analysis

Qualitative Data Analysis (QDA) is part of the process of qualitative research. Researchers in this field can collect data using a variety of strategies, which in turn shape the general course of the research process (Creswell & Creswell, 2018). As a result, different approaches and coding strategies can be formulated for QDA.

2.1.1 Grounded Theory

One approach to deriving theories from data in qualitative research is Grounded Theory (GT). GT is an inductive strategy, generating theory from the gathered data (Stol et al., 2016). GT is a broad framework but also consists of the performance of QDA. We will now take a closer look at its implications on QDA. While there is some level of agreement on key elements of the QDA process, such as continuous comparison in data and developed concepts, there are different approaches on how to develop a grounded theory. One of these approaches is formulated by Corbin and Strauss which we will analyze as an example for QDA procedures. We will omit other aspects of this approach that go beyond the performance of QDA since they go beyond the scope of this thesis.

Grounded Theory after Corbin and Strauss

This approach applies three iterative activities to the coding process, "open coding", "axial coding" and "selective coding". These three activities are often not entirely separate, rather a researcher can alternate between the three or even combine them. However, one procedure might be prevalent depending on which phase a researcher is in. Usually, the research process begins with open coding

while the end phase of research often contains more analysis supported by the selective coding approach (Flick, 2009).
Open coding aims at identifying categories in the form of codes and attaching them to the data which is called coding. Throughout open coding, the data is often segmented and then analyzed (Corbin & Strauss, 1990). Consequently, it encourages a better understanding of the gathered data. The researcher is going through segments, comparing the data, and formulating new concepts. Open coding is usually the initial step in coding the data (Flick, 2009).
In **axial coding**, the aim is to refine categories that have already been identified. This refinement can be in the form of developing subcategories, formulating relationships between categories, and verifying categories in the data. In this coding strategy, the researcher has to switch between identifying concepts and relations on the one hand and validating and testing these concepts on the other hand (Flick, 2009). Verifying concepts with new data is crucial because a theory needs multiple indications to be considered valid (Corbin & Strauss, 1990).
Selective coding focuses on central categories and their relationships. The researcher can further elaborate on these categories by looking at more evidence. The aim is to identify one core category and formulate the developed theory (Flick, 2009). Hence, the analyst must integrate all categories around this central category (Corbin & Strauss, 1990). This procedure is usually the final step performed in QDA. When theoretical saturation is reached, which implies that further analysis will not provide any new insights, the analysis of data ends (Flick, 2009).

Other Approaches to Grounded Theory

Other approaches differ more or less from GT. In addition to GT, Flick (2009) names the approach of theoretical coding, developed by Glaser. That approach differs from Corbin and Strauss by proclaiming that axial coding does not encourage the categories to emerge from the data. Instead, theoretical coding suggests formulating groups of basic codes which are called coding families. These are then the foundation for further code development (Flick, 2009).
Charmaz developed another approach to coding in GT, as mentioned by Flick (2009). This strategy begins with line-by-line coding, which is utilized to exclude personal bias from the coding process and focus more closely on the content. Then focused coding is performed. That approach delves deeper into the collected data, with a particular focus on certain developed codes.
When analyzing these three approaches one can see that there are common elements between them. All of them see open coding as a vital element of QDA and finish their analysis when theoretical saturation is reached. Another similarity is the emphasis on constant comparison between the data segments as well as the categories (Flick, 2009).

2.1.2 Common Procedures in QDA

As mentioned before, GT is not the only approach to qualitative research. When utilizing different approaches the coding strategies will be different. Nevertheless, there are some common components to many of the derived strategies for QDA (Mayer, 2015; Miles & Huberman, 1994). The QDA components include data collection, data display, data reduction, and conclusion drawing/verification, between which a researcher may alternate. The researcher iterates between components until theoretical saturation is reached and the data collection is finished. Data reduction describes the simplification and abstraction of data. In data reduction, an analyst can select, summarize, or paraphrase data. Data display focuses on organizing the data so that a researcher may draw conclusions more easily. Strategies for drawing and verifying conclusions are asking questions and noticing patterns in the data (Miles & Huberman, 1994).
Miles and Huberman (1994) derive some specific practices from these components. One of the practices is the revision of codes, where a researcher reevaluates the structure and content of codes. Deleting, adding and redefining codes may be part of this process. Another step is the definition of new codes. When a researcher defines new codes, it's important to concentrate on providing clear definitions and names, ensuring that these codes can be used for subsequent data analysis. Corbin and Strauss (1990) and Miles and Huberman (1994) also mentions the step of memoing. Analysts record their ideas and thoughts in memos, which can be written at various points during the coding process. Memos might encompass additional relationships or supplementary conceptual insights about a code, which can be integrated and utilized at a later point. (Miles & Huberman, 1994).
One can identify further tasks outside of the data collection and coding process. An area that might imply more possible tasks for QDA is the evaluation of the reliability of theories. Concepts such as intercoder reliability, which requires the collaboration of multiple researchers, can help in developing more criteria for tasks (Campbell et al., 2013). We limit the scope of this thesis to tasks directly derived from QDA procedures.

2.1.3 Identification of tasks in QDA

From the previously presented processes in QDA, we can obtain a number of tasks. While some tasks may be specific to an approach to qualitative research, others may generally help in conducting QDA. In this thesis, we will focus on the latter group of tasks, specifically tasks that software can actively support.
A task that we can deduce from the process of data collection is the storing and editing of documents. This task is crucial for any QDA software since it is the basis of working with data. Furthermore, when looking at the steps of data display and reduction, the coding process formulates multiple required tasks.

As discussed above, open coding is a widely accepted step in performing QDA. Therefore, CAQDAS should support the task of coding text segments while allowing the user to quickly create new codes.
The concept of axial coding might vary in different QDA approaches, but comparing and refining codes is generally a vital step for QDA. This step indicates the task of codebook refinement. Codebook refinement includes the quick restructuring of the code system, redefining codes, and deriving new theories from the existing codes and their codings.
Moreover, the area of conclusion drawing and verification has an implication on QDA tasks. This step overlaps in many aspects with the selective coding strategy of Corbin and Strauss. From this step, we will develop the task of analysis. For analysis, an overview of the developed concepts is important. Information on codes and the relationships between them is crucial for the researcher.
The activity of writing memos is necessary for all coding steps. The researcher should always have the option of quickly noting down ideas and thoughts with minimal disruptions in the workflow.
When evaluating different QDA approaches more closely, more tasks can be specified. But for the purpose of this thesis, we will focus on the discussed basic tasks that can support a variety of approaches.

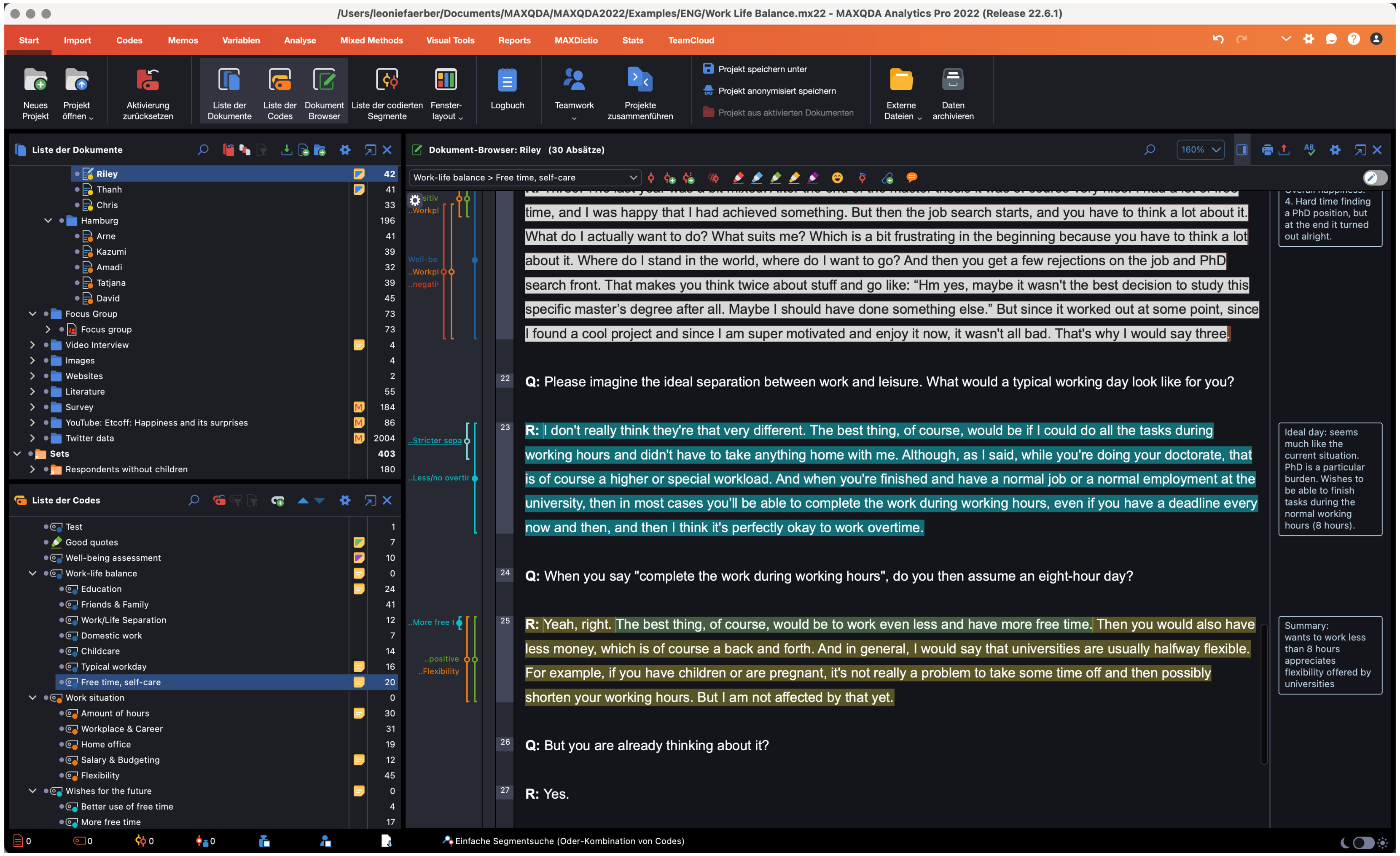
2.2 CAQDAS comparison

Several tools can be used to perform QDA. Many of these tools supply a number of generic features such as storing and organizing documents, analysis tools, and search functionality (Creswell & Creswell, 2018). Therefore, working with a large amount of data is more feasible and efficient since software can automate many monotonous tasks. Using CAQDAS tools is not only time efficient but also enables a more detailed analysis of the data as well as better visualization of theories (Merriam & Tisdell, 2015). In the following sections, we will analyze two different CAQDAS tools, MaxQDA and WebQDA. This analysis will include a focus on whether the tools support the prior formulated tasks as well as how the functionality is provided. Therefore, the main focus of this analysis will be the coding editor interface of both tools. We will omit any other functionality.

2.2.1 MaxQDA

MaxQDA[1] is a software tool for qualitative and mixed-methods research that is commonly used. The tool has a coding editor interface providing different elements that the user can enable and disable. Figure 2.1 gives an overview of this interface. Additionally, there are separate windows for some features. There

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


/Users/leoniefaerber/Documents/MAXQDA/MAXQDA2022/Examples/ENG/Work Life Balance.mx22 - MAXQDA Analytics Pro 2022 (Release 22.6.1)
Start
Import
Codes
Memos
Variablen
Analyse
Mixed Methods
Visual Tools
Reports
MAXDictio
Stats
TeamCloud
Neues Projekt
Projekt öffnen
Aktivierung zurücksetzen
Liste der Dokumente
Liste der Codes
Dokument Browser
Liste der codierten Segmente
Fenster-layout
Logbuch
Teamwork
Projekte zusammenführen
Projekt speichern unter
Projekt anonymisiert speichern
Projekt aus aktivierten Dokumenten
Externe Dateien
Daten archivieren
Liste der Dokumente
Liste der Codes
Dokument-Browser: Riley (30 Absätze)
160%
Work-life balance > Free time, self-care
Einfache Segmentsuche (Oder-Kombination von Codes)

Riley	42
Thanh	41
Chris	33
Hamburg	196
Arne	41
Kazumi	39
Amadi	32
Tatjana	39
David	45
Focus Group	73
Focus group	73
Video Interview	4
Images	4
Websites	2
Literature	55
Survey	184
YouTube: Etcoff: Happiness and its surprises	86
Twitter data	2004
Sets	403
Respondents without children	180

Test	1
Good quotes	7
Well-being assessment	10
Work-life balance	0
Education	24
Friends & Family	41
Work/Life Separation	12
Domestic work	7
Childcare	14
Typical workday	16
Free time, self-care	20
Work situation	0
Amount of hours	30
Workplace & Career	31
Home office	19
Salary & Budgeting	12
Flexibility	45
Wishes for the future	0
Better use of free time	4
More free time	17

time, and I was happy that I had achieved something. But then the job search starts, and you have to think a lot about it. What do I actually want to do? What suits me? Which is a bit frustrating in the beginning because you have to think a lot about it. Where do I stand in the world, where do I want to go? And then you get a few rejections on the job and PhD search front. That makes you think twice about stuff and go like: "Hm yes, maybe it wasn't the best decision to study this specific master's degree after all. Maybe I should have done something else." But since it worked out at some point, since I found a cool project and since I am super motivated and enjoy it now, it wasn't all bad. That's why I would say three.

4. Hard time finding a PhD position, but at the end it turned out alright.

22 **Q:** Please imagine the ideal separation between work and leisure. What would a typical working day look like for you?

23 **R:** I don't really think they're that very different. The best thing, of course, would be if I could do all the tasks during working hours and didn't have to take anything home with me. Although, as I said, while you're doing your doctorate, that is of course a higher or special workload. And when you're finished and have a normal job or a normal employment at the university, then in most cases you'll be able to complete the work during working hours, even if you have a deadline every now and then, and then I think it's perfectly okay to work overtime.

Ideal day: seems much like the current situation. PhD is a particular burden. Wishes to be able to finish tasks during the normal working hours (8 hours).

24 **Q:** When you say "complete the work during working hours", do you then assume an eight-hour day?

25 **R:** Yeah, right. The best thing, of course, would be to work even less and have more free time. Then you would also have less money, which is of course a back and forth. And in general, I would say that universities are usually halfway flexible. For example, if you have children or are pregnant, it's not really a problem to take some time off and then possibly shorten your working hours. But I am not affected by that yet.

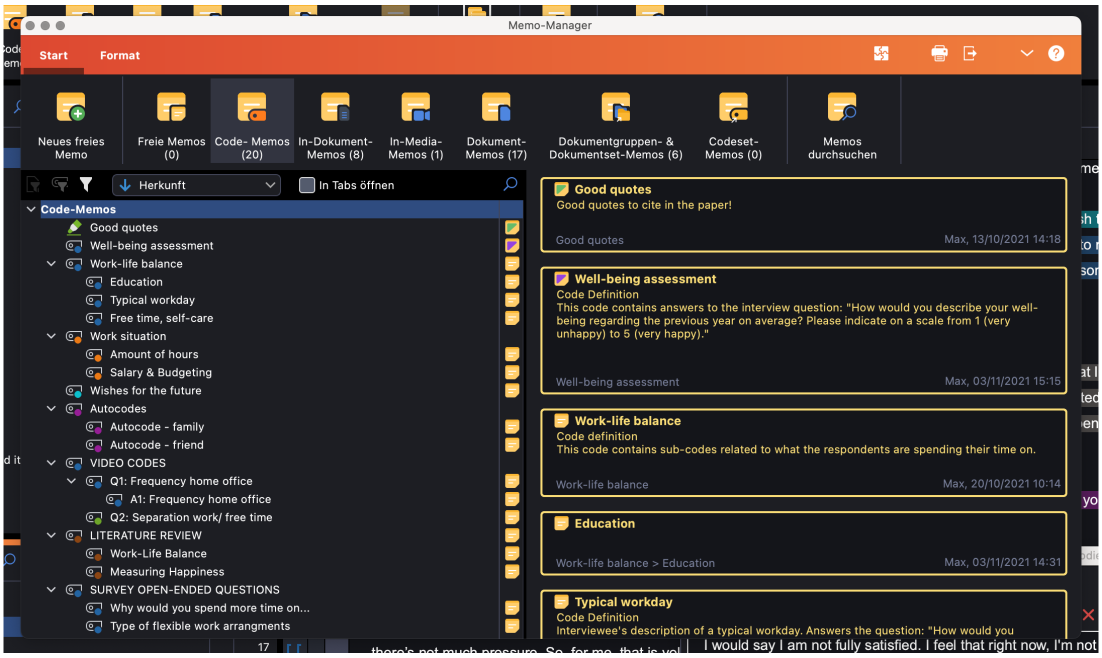
Summary: wants to work less than 8 hours appreciates flexibility offered by universities

26 **Q:** But you are already thinking about it?

27 **R:** Yes.

Figure 2.1: Coding editor in MaxQDA

is one main menu in the top column of the page that modifies the content and layout of the rest of the editor. For layout configuration, MaxQDA provides four different layouts. The layouts allow the user to modify the size of the elements. The main elements are a code system list, a documents list, a document browser with code brackets, text, and comments, as well as a coding overview.
Since all elements can be activated and deactivated individually, a user can adapt the coding editor to the workflow. However, this absolute configurability is not compatible with providing any task-focused views. A user has to find all necessary elements in the provided functionality depending on the task. This might be overwhelming and hard to learn for new users. In addition, MaxQDA provides all functionality in one place which is not space-efficient.
When analyzing the previously defined tasks, all of them can be performed in MaxQDA. A user may carry out the task of storing and editing documents using the documents list and the documents browser. The task of coding while defining new codes can easily be executed using the aforementioned elements in addition to the code system list. The code system offers the option of adding, defining, renaming, and deleting codes which makes quick code modification very efficient. MaxQDA also supports the task of codebook refinement. However, there is no view of all code information in one place. The user can modify the code properties through the context menu of the code system. The same context menu offers the option to open a separate window with the code memo. This window also displays other properties such as a code summary. Figure 2.2 shows the content of the code memo window. However, the window has to be opened for each code



Code-Memos
- Good quotes
- Well-being assessment
- Work-life balance
  - Education
  - Typical workday
  - Free time, self-care
- Work situation
  - Amount of hours
  - Salary & Budgeting
- Wishes for the future
- Autocodes
  - Autocode - family
  - Autocode - friend
- VIDEO CODES
  - Q1: Frequency home office
    - A1: Frequency home office
  - Q2: Separation work/ free time
- LITERATURE REVIEW
  - Work-Life Balance
  - Measuring Happiness
- SURVEY OPEN-ENDED QUESTIONS
  - Why would you spend more time on...
  - Type of flexible work arrangments

Good quotes
Good quotes to cite in the paper!
Good quotes — Max, 13/10/2021 14:18

Well-being assessment
Code Definition
This code contains answers to the interview question: "How would you describe your well-being regarding the previous year on average? Please indicate on a scale from 1 (very unhappy) to 5 (very happy)."
Well-being assessment — Max, 03/11/2021 15:15

Work-life balance
Code definition
This code contains sub-codes related to what the respondents are spending their time on.
Work-life balance — Max, 20/10/2021 10:14

Education
Work-life balance > Education — Max, 03/11/2021 14:31

Typical workday
Code Definition
Interviewee's description of a typical workday. Answers the question: "How would you

I would say I am not fully satisfied. I feel that right now, I'm not

Figure 2.2: Memo window in MaxQDa

individually and does not allow for switching between codes. While there are other options such as displaying all code memos, there is no way of viewing and modifying all fields related to codes while being able to switch from one code to another. Hence, the task of codebook refinement will require multiple clicks and windows. The task of analysis is well supported. MaxQDA offers a variety of visualization tools to analyze the code system. The tool also offers several memo options. A user may write memos in an individual window and those memos can refer to most of the MaxQDA elements such as codes, documents, and projects. Consequently, MaxQDA allows the performance of all basic QDA tasks but could benefit from better supporting codebook refinement. Additionally, providing all functionality in one space might overwhelm new users and generally distract from focused work. On the other hand, an experienced QDA user might prefer the flexibility and modification options that MaxQDA allows.

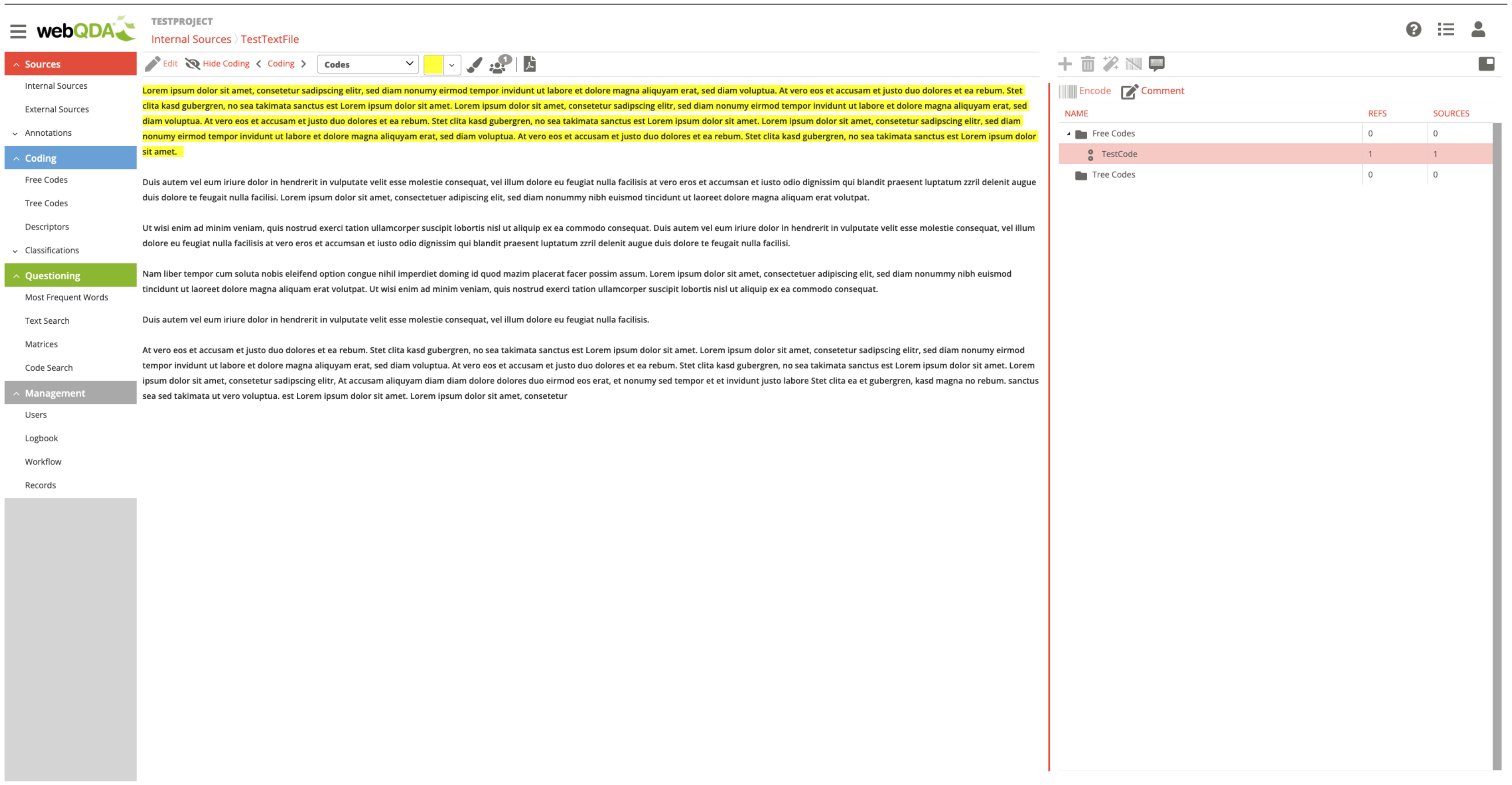
2.2.2 WebQDA

WebQDA[2] is a web-based tool that has a coding editor interface allowing little modification. This interface is displayed in Figure 2.3. The left sidebar is a menu divided into the sections "sources", "code", "questioning", and "management". These sections function as task dividers. The sources view permits the modification and coding of documents as well as the quick initiation of codes in a smaller code system view in the right column. Therefore, the view allows for data modification and coding while defining new codes. The sources view also provides the option of adding notes and comments that can be used to write down thoughts and ideas. The task of codebook refinement is supported in the code section of the editor. This section provides a bigger view of codes that enables the user to restructure and redefine codes. The questioning section permits the analysis of the gathered data and concepts. It provides analysis and search functionality. The last section for management contains other functionality.
The task division enables the tool to display less functionality in each view, which is not only more space-efficient but also allows for a less overwhelming interface. However, an experienced user might miss the option of modifying the layout. Most elements are fixed and not modifiable in size. The user may at most collapse a small number of elements.

2.3 QDAcity

QDAcity is a cloud-based web tool supporting QDA. The focus is on the storing and structuring of data, while best assisting the process of analysis as well as allowing for collaboration.[3] This thesis focuses on the functionality provided in

[2]https://www.webqda.net
[3]https://qdacity.com



Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis.

At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, At accusam aliquyam diam diam dolore dolores duo eirmod eos erat, et nonumy sed tempor et et invidunt justo labore Stet clita ea et gubergren, kasd magna no rebum. sanctus sea sed takimata ut vero voluptua. est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur

NAME	REFS	SOURCES
Free Codes	0	0
TestCode	1	1
Tree Codes	0	0

Figure 2.3: Coding editor in WebQDA

the coding editor which a user can access from the project overview. There are various other functionalities that are beyond the scope of this thesis.

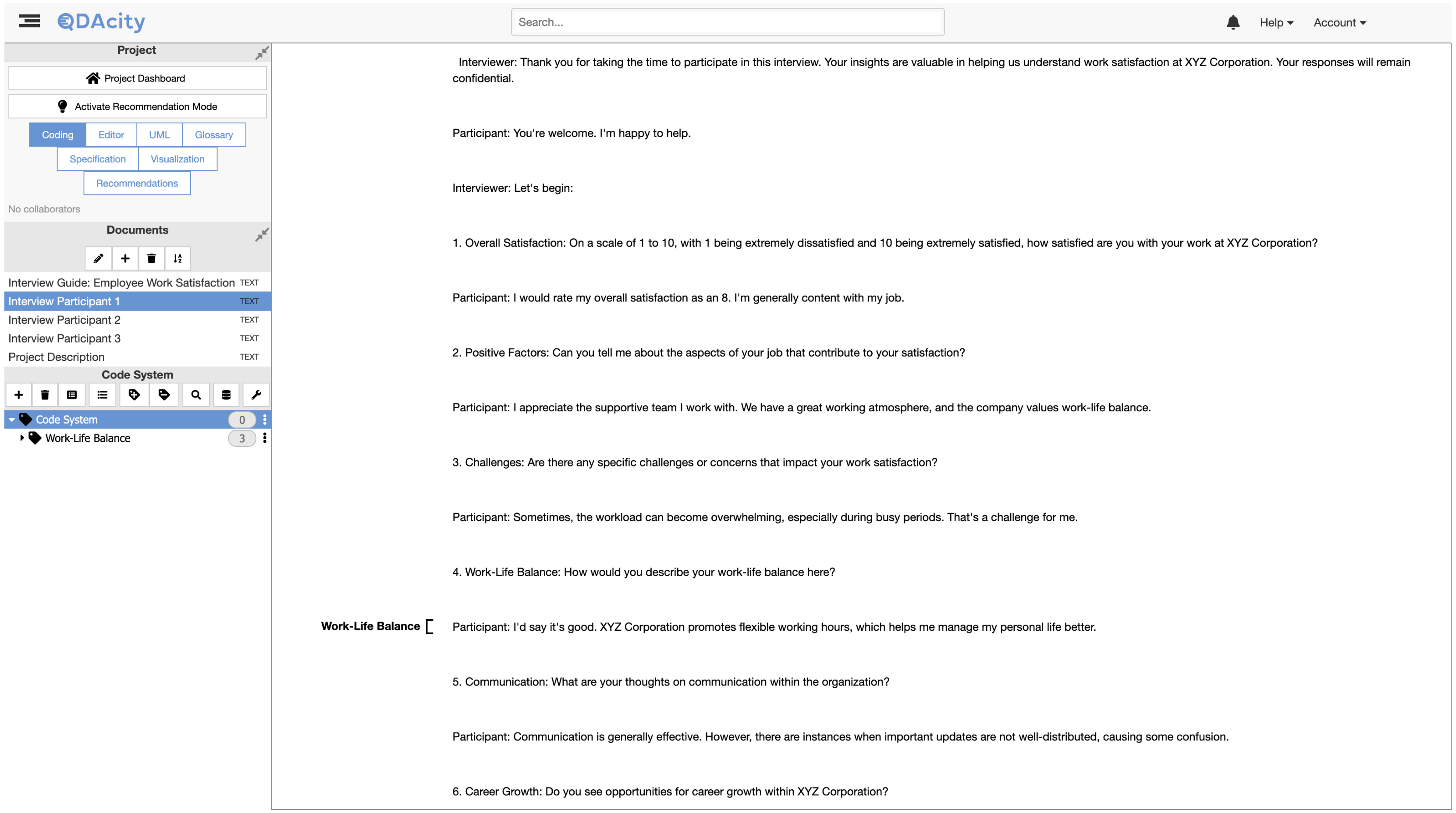
Coding Editor

The coding editor is the main feature in supporting a QDA workflow. Figure 2.4 shows an exemplary view of this coding editor. The interface is split into three sections, a sidebar on the left, a main editor interface on the right, and a footer element on the bottom of the page. Figure 2.5 shows all sidebar elements.
A user may select the content of the sidebar and the editor interface through buttons in the project section. This section is located at the top of the sidebar and is visible in all editor views. In addition, the project section contains a button for navigating back to the project overview as well as a display of collaborators active in the editor. The buttons that allow for navigating between editor views change depending on the selected document.
A user may select a document in the document view located beneath the project section in the sidebar. There are three different types of documents, PDF, text, and audio file. As mentioned before, the type of the selected document determines the editor view buttons in the project section. The available editor views for each document type are displayed in Table 2.1. The document view that allows the

Table 2.1: Compatible views for document types

Document type	Compatible views
PDF	coding, UML, glossary, specification, visualization, recommendations
text	coding, editor, UML, glossary, specification, visualization, recommendations
audio	transcription editor, UML, glossary, specification, visualization, recommendations

user to change the selected document, is not displayed in all editor views.
The last section of the sidebar is the code system. The code system is the central feature of the coding editor since it is also at the core of QDA. Therefore, all editor views contain this section. The code system allows for adding, deleting, and accessing codes as well as further functionality such as giving the option of opening code-related interfaces.
The content of the editor interface on the right depends on which editor view is selected in the project section. The content of the editor views will be discussed further later on.



Interviewer: Thank you for taking the time to participate in this interview. Your insights are valuable in helping us understand work satisfaction at XYZ Corporation. Your responses will remain confidential.

Participant: You're welcome. I'm happy to help.

Interviewer: Let's begin:

1. Overall Satisfaction: On a scale of 1 to 10, with 1 being extremely dissatisfied and 10 being extremely satisfied, how satisfied are you with your work at XYZ Corporation?

Participant: I would rate my overall satisfaction as an 8. I'm generally content with my job.

2. Positive Factors: Can you tell me about the aspects of your job that contribute to your satisfaction?

Participant: I appreciate the supportive team I work with. We have a great working atmosphere, and the company values work-life balance.

3. Challenges: Are there any specific challenges or concerns that impact your work satisfaction?

Participant: Sometimes, the workload can become overwhelming, especially during busy periods. That's a challenge for me.

4. Work-Life Balance: How would you describe your work-life balance here?

Participant: I'd say it's good. XYZ Corporation promotes flexible working hours, which helps me manage my personal life better.

5. Communication: What are your thoughts on communication within the organization?

Participant: Communication is generally effective. However, there are instances when important updates are not well-distributed, causing some confusion.

6. Career Growth: Do you see opportunities for career growth within XYZ Corporation?

Figure 2.4: Coding editor in QDAcity

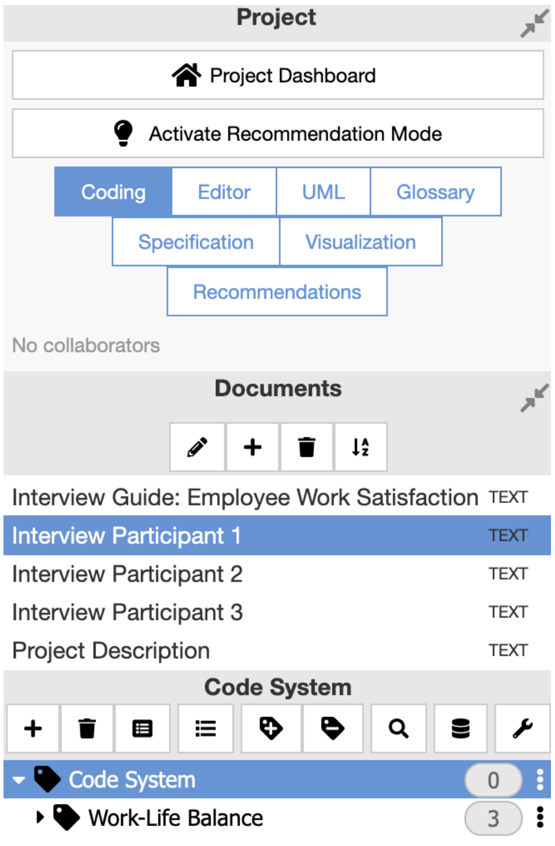
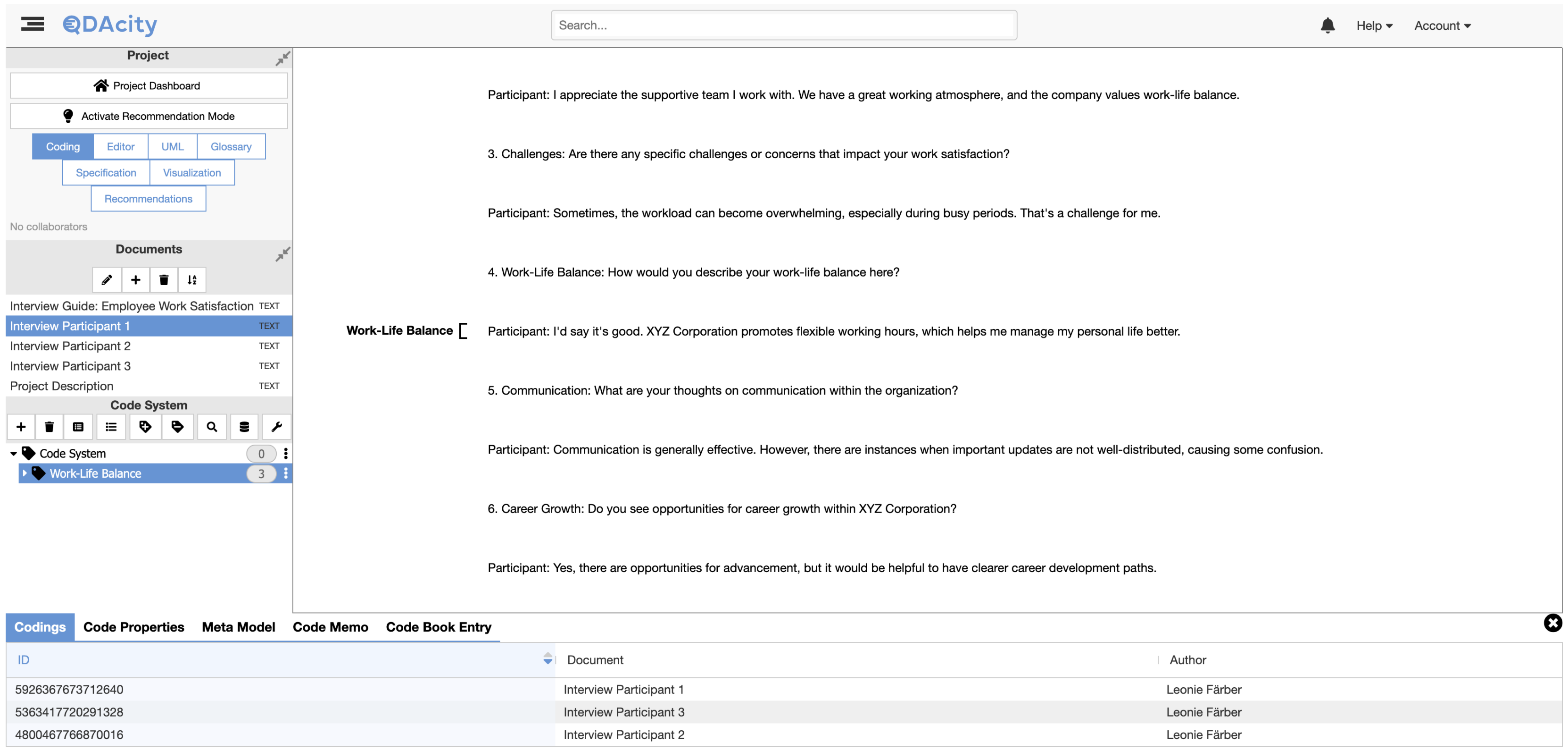


Figure 2.5: Sidebar in QDAcity

The last section of the coding editor is the code view footer which is displayed over the full width of the page and has a fixed height. Figure 2.6 displays the code view footer in the coding editor. The code view footer allows the user to access further information on the selected code including the code properties, a meta-model, the code memo, and the codebook entry. A user may enable and disable this footer element in all editor views allowing for quick access when necessary. As mentioned above there are different editor views that we may already interpret as different task views. These currently include:

1. Coding
2. Editor
3. Transcription Editor
4. UML
5. Glossary
6. Specification



Documents	
Interview Guide: Employee Work Satisfaction	TEXT
Interview Participant 1	TEXT
Interview Participant 2	TEXT
Interview Participant 3	TEXT
Project Description	TEXT

Participant: I appreciate the supportive team I work with. We have a great working atmosphere, and the company values work-life balance.

3. Challenges: Are there any specific challenges or concerns that impact your work satisfaction?

Participant: Sometimes, the workload can become overwhelming, especially during busy periods. That's a challenge for me.

4. Work-Life Balance: How would you describe your work-life balance here?

Work-Life Balance [Participant: I'd say it's good. XYZ Corporation promotes flexible working hours, which helps me manage my personal life better.

5. Communication: What are your thoughts on communication within the organization?

Participant: Communication is generally effective. However, there are instances when important updates are not well-distributed, causing some confusion.

6. Career Growth: Do you see opportunities for career growth within XYZ Corporation?

Participant: Yes, there are opportunities for advancement, but it would be helpful to have clearer career development paths.

ID	Document	Author
5926367673712640	Interview Participant 1	Leonie Färber
5363417720291328	Interview Participant 3	Leonie Färber
4800467766870016	Interview Participant 2	Leonie Färber

Figure 2.6: Coding editor with code view footer in QDAcity

7. Visualization
8. Recommendations

Coding The coding view includes the document section in the sidebar. When selecting a document in the document view, the editor interface on the right will display its content. Moreover, the editor view includes a column showing the coding brackets that a user may modify in size. These brackets visualize any codings that have already been applied. This view allows the user to add and remove codings for the selected document.

Editor A user may edit text documents in the editor view. The view displays the same elements as the coding view in addition to a toolbar for editing text at the top of the editor interface. In the editor view, a user may edit the content of the selected document but cannot modify the applied codings.

Transcription Editor Transcribing audio files can be performed in the transcription editor view. This view displays the audio track as well as the transcribed text. Additionally, it contains the document view in the sidebar. In the transcription editor view, a user may listen to an audio file and correct or modify the automated description. When reaching an adequate result the user may export the file to a text document to start the coding process.

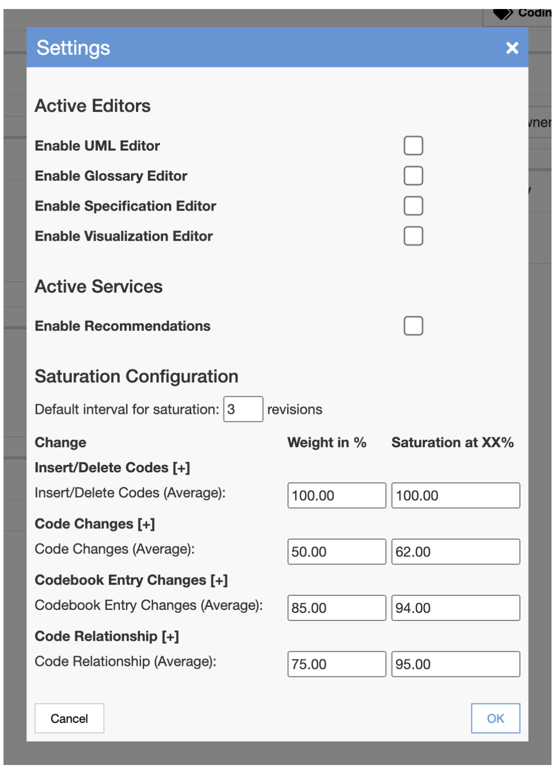
UML, Glossary, Specification, Visualization These views provide the user with further functionality that is not part of the focus of this thesis. Some of these views are still being developed.

Recommendations The Recommendations view allows users to review recommendations made for codes and other elements. This view is part of the recommendation service. The recommendation service can be enabled and disabled and the recommendations view will only be visible if the recommendation service is enabled. The feature of writing and reviewing recommendations is outside of the scope of this thesis.

A user may enable or disable most of the above editor views to adjust the editor interface to the workflow. The modification of visible editor views can be done in the project settings which are located in the project overview and can be seen in Figure 2.7. Excluded from this are the coding, editor, and transcription editor views. Furthermore, a user may enable the recommendation service in the project settings which determines whether the recommendations view is enabled.

2.4 Discussion

QDAcity contains different views that a user can switch between in the coding editor. We may already interpret these views as task views. In the following dis-


Settings

Active Editors

Enable UML Editor ☐

Enable Glossary Editor ☐

Enable Specification Editor ☐

Enable Visualization Editor ☐

Active Services

Enable Recommendations ☐

Saturation Configuration

Default interval for saturation: 3 revisions

Change	Weight in %	Saturation at XX%
Insert/Delete Codes [+]		
Insert/Delete Codes (Average):	100.00	100.00
Code Changes [+]		
Code Changes (Average):	50.00	62.00
Codebook Entry Changes [+]		
Codebook Entry Changes (Average):	85.00	94.00
Code Relationship [+]		
Code Relationship (Average):	75.00	95.00

Figure 2.7: Project settings modal in QDAcity

cussion, we will evaluate, how well they support the aforementioned basic tasks for QDA.
The task of storing and organizing data is well supported in QDAcity. The document list gives quick access to all documents as well as an option of sorting them alphabetically. The editor and the transcription editor view allow for modifying text and transcribing audio files. These two views allow the user to quickly add, store, and sort a number of data sources. A possible improvement for the clarity of the views is the removal of the add and remove coding buttons from the code system. These buttons currently have no functionality in any view apart from the coding view. Therefore, they should only be displayed in this view and otherwise be hidden when adding and removing codings is not allowed. Another task that is well supported is the task of coding and adding new codes. In the coding view, the code system allows for adding and removing codes as well as codings. For more information on codes, the codebook entry, and the code memo, a user may enable the code view footer. The footer element is displayed in Figure 2.6. This footer allows for even better quick access to codes. However, the footer leaves room for improvement. The element currently spans over the whole width of the page, causing the view of the code system to be smaller. Therefore, fewer codes are visible, requiring more scrolling from the user when switching the selected code. We can avoid this by decreasing the width of the footer to fit in the editor

interface so that the code system view can remain the same size as before.
The task of codebook refinement requires improvements in the QDAcity interfaces. Similarly to MaxQDA, QDAcity allows access to code information in different areas. There is a separate overlay for displaying codings for a selected code. When this coding overview is enabled, the selected code cannot be changed. A user may not access any other elements until the overlay is closed. Moreover, the code view footer allows for quick access to the code properties, code memo, and codebook entry but it does not enable a user to see all elements at once. The footer is fixed in size which implies that when refining the codebook a large part of the page containing the editor interface and possibly the documents view is not being used. Therefore, a new task view focusing on the task of codebook refinement will allow for a better QDA workflow.
The task of analysis is supported in QDAcity. There is a code analysis view that the user can access through the code system. However, the functionality of this view needs to be revised. Furthermore, the coding editor provides the option of constructing UML models and meta-models to better visualize code relationships.
QDAcity well supports memoing. The code view footer enables the user to write memos at any point in the workflow.
A feature that improves both codebook refinement and analysis is the code status bar next to each code in the code system. Additionally to the number of codings for each code, the status bar displays whether a memo, a codebook entry, or meta-model information has been added to the code.
QDAcity allows for little modification, similar to WebQDA. An experienced user who wants to perform more complex tasks might know best what interfaces are necessary as well as what size these interfaces should have. While QDAcity supports a number of tasks through the different editor views a user could benefit from the option of creating custom task views. A custom view would not only maximize the tasks that users can perform in QDAcity but also exploit the benefits of more customizable tools such as MaxQDA.
Another aspect that we should regard when evaluating task-focused work is how efficient task navigation can be performed. The current design where the displayed navigation buttons depend on the selected document is not beneficial and confusing in the workflow. Moreover, the task navigation elements should be a central focus of the coding editor to show a user that the view selection will determine the displayed editor content.

3 Requirements

This chapter presents the requirements for this thesis structured thematically. There are three main thematic areas. First, we focus on requirements that are valid across all editor views and are thus formulated for the entire coding editor. Then requirements that are limited to the task of codebook refinement and the customization of views. For all requirements, we give a short description that we could convert into further more fine-grained sub-requirements. However, this thesis focuses on satisfying the main requirements given in this section. We adopted a template-based approach to improve the clarity of the requirements. (Sophisten, 2016).

3.1 Coding Editor

Req 1.0 - All task views in the coding editor should not include any functionality or information that is unrelated to their task.

All existing and newly created task views should be space efficient, thus only displaying functionality required to perform its task. Any elements providing no functionality should be removed. In the existing task views, this mostly concerns the add and remove coding buttons in the code system. These buttons should only be visible when they are functional.

Req 2.0 - The coding editor shall provide the user with the ability to easily switch between task views.

To make the user's workflow more efficient, the selection of task views shall not depend on any other elements in the coding editor. There shall be a navigation element that is superordinate to the rest of the coding editor elements.

Req 2.1 - The coding editor shall provide the user with the option of quickly adding and removing displayed task views.

The displayed task views in the coding editor shall be modifiable to adjust to the user's workflow. To make customization more time-efficient and user-friendly, this functionality shall be provided directly in the coding editor rather than the project settings.

Req 2.2 - The display of the task views should be space-efficient for smaller window sizes.

To make the navigation element space efficient, the navigation element should not display the names of the editor view for smaller window sizes. Instead, an icon should be associated with each editor view that will be present for all screen sizes.

Req 2.3 - The display of the task views should provide the user with the ability to change the order of the tasks.

The user should be able to rearrange the task views via drag-and-drop. Therefore, task switching becomes more efficient and the user may group tasks depending on the workflow.

3.2 Codebook Refinement

Req 3.0 - The coding editor shall provide the user with the ability to select a view that supports the task of codebook refinement.

The user shall have access to an editor view that provides functionality for the task of codebook refinement. The following requirements establish the required content of this editor view.

Req 3.1 - The codebook refinement view shall provide the user with the ability to see and modify the code properties, code memo, and codebook entry of a selected code.

The user shall have the option to see and modify all information assigned to a code. The code information enables the user to get a better overview of the code system as well as refine definitions and theories.

Req3.2 - The codebook refinement view shall provide the user with the ability to see all codings of a selected code as well as quickly switch to a specific coding in the coding view.

The user shall be able to see all codings in the code editor view since knowing what codings exist for a code can help in refining definitions and theories. Switching to a specific coding in the coding view can give further context on a coding as well as allow the user to quickly modify the coded data section. This context on codings can assist the user in further understanding the data and developing more concise theories.

Req 3.3 - The codebook refinement view should provide the user with the ability to move or copy codings from one code to another as well as delete codings efficiently.

The task of codebook refinement includes the restructuring of the code system. To ensure that codings can quickly be adapted to a new code system structure, the user needs to be capable of quickly modifying which code a coding belongs to.

Req 3.4 - The codebook refinement view should allow the user to write comments for codings.

Comments on codings enable users to quickly note down thoughts about a specific data segment. These notes may include summaries, further ideas, theories, etc. A user can then use these comments to refine the codebook further.

3.3 Custom View

Req 4.0 - The coding editor shall provide the user with the ability to add custom editor views to the selectable task views.

The user shall have an editor interface to customize the editor content to their liking. The following requirements establish the content and functionalities of this editor view.

Req 4.1 – The custom editor views shall provide the user with the ability to name each view.

To ensure that the user can differentiate between multiple custom editor views, these views must be nameable. Naming views also allows the user to clearly assign a task to a view so that other collaborators in the project may know its purpose instantly.

Req 4.2 - The custom editor views shall provide the user with the ability to change the visibility of each view in the project.

The user must have the option of changing the visibility of a custom editor view. There must at least be a distinction between editor views that are private, thus can only be seen by the user themselves or public, thus being visible to all collaborators on the project.

Req 4.3 - When selecting a custom view, the user should be able to choose which elements out of the available coding editor elements the view should display.

The user should be able to choose the content of the custom editor view from the most common coding editor elements. Regrouping editor elements in different views maximizes their usability.

Req 4.4 - The custom editor should be highly modifiable regarding the size and layout of its components.

The user should be able to size all elements according to their needs. Moreover, the user should be allowed to choose the position of an element in the layout.

4 Architecture

This chapter describes the developed architecture based on the requirements formulated in the previous chapter.

4.1 Task Navigation

The task navigation element shall be efficient and as independent from the rest of the editor as possible. However, there are constraints on which editor views a document type can support. Table 2.1 shows the compatible editor views for each document type. For an overview of the existing editor views refer to Chapter 2.3. We need to address these limitations in a way that is user-friendly. The initial step toward an independent navigation element involves merging the editor and the transcription editor view. Both views are concerned with the editing of a document and thus have the same overarching task. The merging of the two editor views is possible since there is at most one of them available for any document type. Therefore, no distinction is necessary.
We also have to address the issue of the dependency of the coding and the editing view on the selected document type. Merging the coding and the editing view is impractical due to the fact that for text documents both views have to be available. However, the disappearing and reappearing of navigation elements, depending on the selected document, may cause confusion. Therefore, all available navigation elements shall be visible at all times. Consequently, we must answer the question of what to display, when an incompatible editor is selected. It's possible to display a replacement message for unavailable views, but it's advisable to avoid doing so in order to reduce empty, non-functional interfaces. Therefore, we will implement an automatic tab-switching logic that reduces the usage of replacement messages. Figure 4.1 shows the final tab-switching logic for all editor views.
For the editor view, we developed the following solution. The editor navigation element will appear disabled if a PDF document is selected. Additionally, when a PDF document is selected while in the editor view, the selected view will automatically switch to coding. This switching logic allows a constant display of

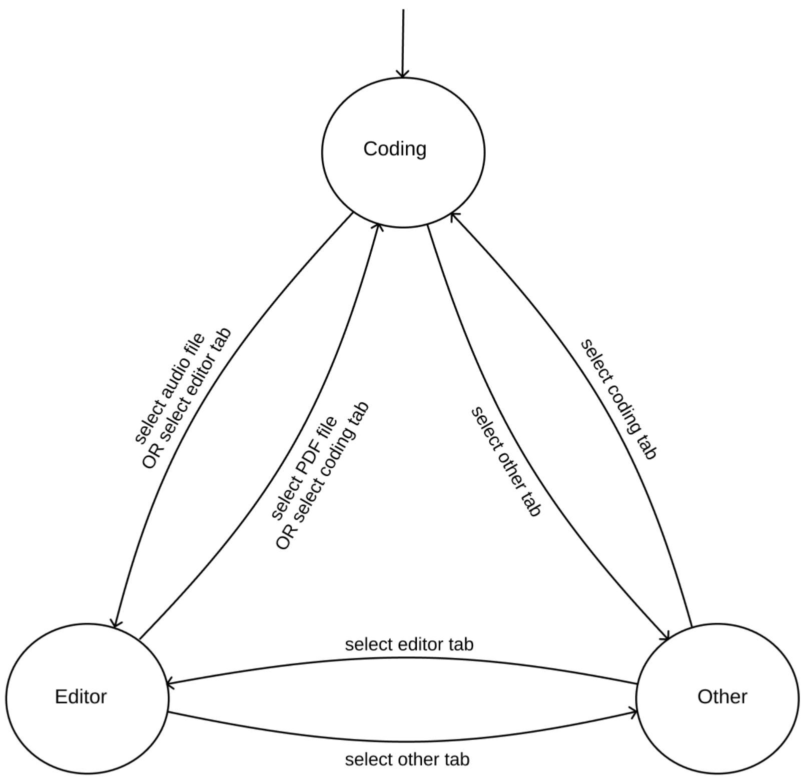


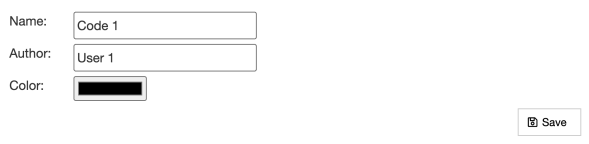
Figure 4.1: State diagram of the tab-switching logic between the coding, editor, and other (including UML, glossary, specification, visualization, and recommendations) tabs

the editor navigation element, while not requiring a replacement message. We considered the same approach for the coding view. However, there are conflicts with this solution. When disabling the coding navigation element for audio files, a user may still switch to any other view. Some views, such as the UML, the specification, and the recommendations view, do not contain a document list. In these views the user cannot change the document, thus the coding view cannot be directly enabled. Therefore, a user must now switch back to a view containing the documents list before being able to select a text or PDF document. Only then can the user switch back to the coding view. This behavior is inefficient since the coding view is at the core of performing QDA. A user may often want to access this view. Consequently, the coding navigation element should always be enabled. To minimize the use of a replacement message when the coding editor view is unavailable, the view is switched automatically to editor when selecting an audio file. However, users may still return to the coding view, where they will be notified of the coding view being unavailable.
Another conceptual change to the navigation logic is the enabling and disabling

of views while remaining in the coding editor. Thereby, it is possible to disable a currently selected task view. When this occurs, the view has to automatically switch to a default value. We select the coding view for this purpose since it is a central view in the coding editor.

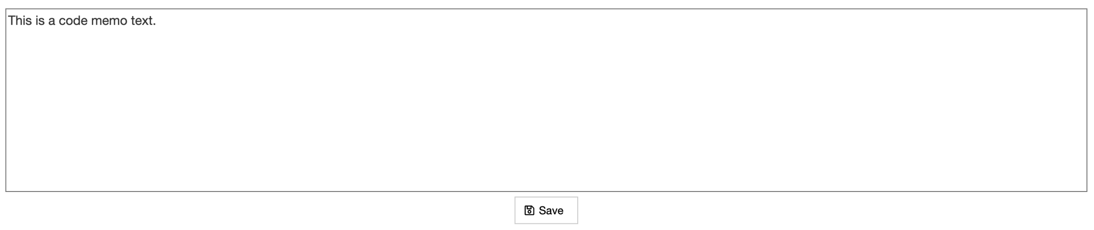
4.2 Code Editor View

The main task of the code editor view is a visualization of already existing data, that is better suited for the task of codebook refinement. Since we are displaying already existing data, sufficient backend structures for storing and retrieving this data are already present. Therefore, introducing new Application Programming Interface (API) methods or data models is not necessary. When constructing the code editor view in the frontend, we focus on reusing the existing QDAcity components where possible. Reusing components not only improves the maintainability of the code but also allows the usage of interfaces already known to the user.
The CodeProperties, CodeMemo, and CodebookEntry components that are part of the CodeView footer can be reused. Figures 4.2, 4.3, and 4.4 show the UI of these three components. However, these elements require some minor changes



Name: Code 1
Author: User 1
Color:

Figure 4.2: UI of the CodeProperties component



This is a code memo text.

Figure 4.3: UI of the CodeMemo component

to better adjust to the code view. On the one hand, the components have a fixed height because the footer element itself is consistent in height. The code editor view requires them to be flexible in size since the interfaces should be responsive to page size changes. On the other hand, all of the reused components have a save button. In the code editor view, having one central save button is more efficient

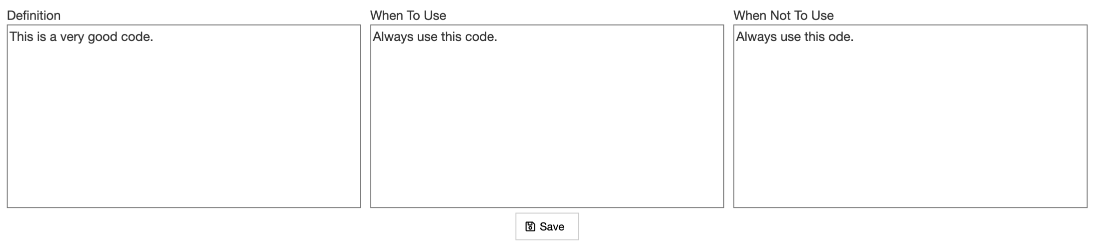
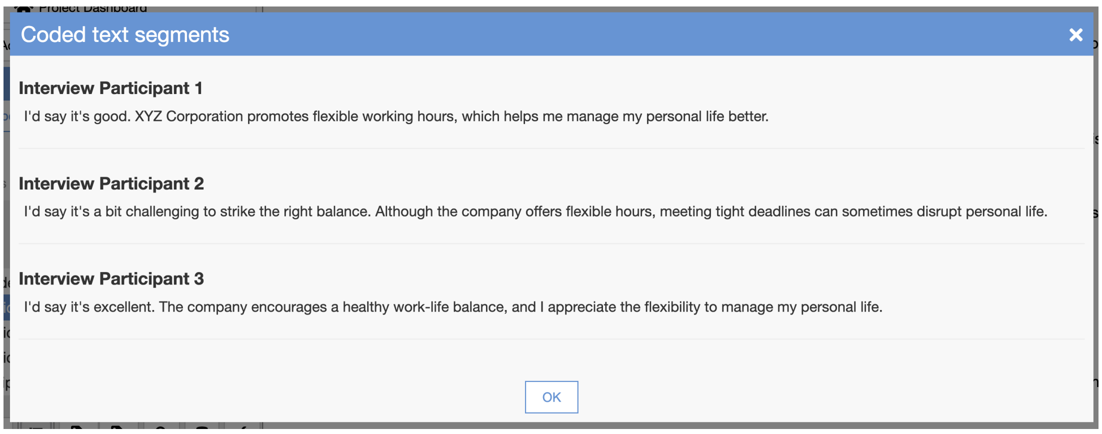


Figure 4.4: UI of the CodebookEntry component

than having to save changes in all fields individually. Therefore, the code editor view does not require the individual save buttons.
We could not reuse the CodingsOverview component, which can be seen in Figure 4.5, since the code editor view requires more than a simple display of the coding instances. Instead, the new coding overview should allow the selection



Interview Participant 1

I'd say it's good. XYZ Corporation promotes flexible working hours, which helps me manage my personal life better.

Interview Participant 2

I'd say it's a bit challenging to strike the right balance. Although the company offers flexible hours, meeting tight deadlines can sometimes disrupt personal life.

Interview Participant 3

I'd say it's excellent. The company encourages a healthy work-life balance, and I appreciate the flexibility to manage my personal life.

Figure 4.5: UI of the CodingsOverview component in the coded text segments modal

and moving of codes. The existing component structure does not allow for that. Therefore, a new component must be implemented.

4.3 Custom Editor View

This section explores the architectural development of the custom editor view. The necessary frontend and backend architecture is considered separately.

Frontend

To increase space efficiency and decrease distractions in the workflow, two different modes are required. The "edit" mode should allow the user to edit the layout

and content of the custom editor. When saving a layout the custom editor view will switch to the "work" mode. Here, no configuration elements are displayed, thus the layout can fill the entire screen.
To make the development of the custom editor view fit into the scope of this thesis, we must make some constraints. While a layout that is completely configurable in size and content may be most beneficial for the custom view, we can also achieve many of the same benefits through a simpler approach. The user will not be able to configure the layout structure but rather the edit mode will provide a number of layout structures to select from. These are configurable in size and content. The user may choose between eight different layout structures or "basic layouts". Figure 4.6 shows the developed layout structures. These basic layouts contain from one up to five fields. Each field can contain one coding editor element. The approach of supplying basic layouts is also advantageous for

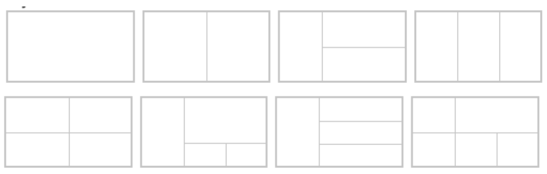


Figure 4.6: Basic layout structures for the custom editor view

users who might be overwhelmed when facing the task of configuring the layout. Therefore, we should extend rather than substitute the option of basic layouts when implementing further configuration options in the future. Consequently, the basic layouts are an excellent starting point for the custom view functionality since they remain useful in future work.
The elements provided in the custom editor need further consideration. A first constraint for all elements is, that they should allow for as much dynamic size modification as possible. Static sizes should be avoided since the elements are supposed to adjust to the layout. Furthermore, we must examine the dependencies between elements. Table 4.1 shows the existing element dependencies. An example of a dependent element would be the coding element. It cannot function without a code system and a document view. Both elements are necessary to allow for document selection and the coding of segments. When an element is added, that has dependencies missing from the layout, the view must display a message to inform the user. In the edit mode, this message should ask the user to add all required missing elements. In the work mode, the message should instruct the user to switch back to the edit mode to add the missing elements.
For the scope of this thesis, a user may only add each element once to each custom view to clearly dissolve the aforementioned dependencies. Developing a custom editor view that supports multiple uses of an element is possible and allows for comparisons. This could be explored in future work.

Table 4.1: Editor elements and their dependencies

Element	Dependencies
code system	none
documents	none
code memo	code system
code properties	code system
codebook entry	code system
coding overview	code system
specification	code system
UML	code system
editor	documents
coding editor	code system, documents
glossary	code system, documents

Backend

The backend functionality for the custom view is mostly concerned with storing and accessing the necessary data. The data model should be expandable for future features that allow for more customization of the layout. Figure 4.7 shows the developed data model. For each Project, a user may store multiple CustomEditor views. Each of these views contains a CustomEditorElement which is the root element of the layout. Each CustomEditorElement can be the parent element of multiple other CustomEditorElements. A CustomEditorElement that contains child elements will be a row or a column and its children make up the content of the element. Thus the CustomEditorElement components support the flexible building of layouts since they allow for free composition of rows and columns.
The status of a custom editor view will distinguish two values. These values are "private" and "public" and control the visibility of the custom view in the project. Table 4.2 describes the differences between the two status types.
The API methods for the custom view should only allow access to the full CustomEditor object. These methods will also initiate, update, or delete the belonging CustomEditorElement objects. This approach allows for fewer mistakes and inconsistencies in the data.

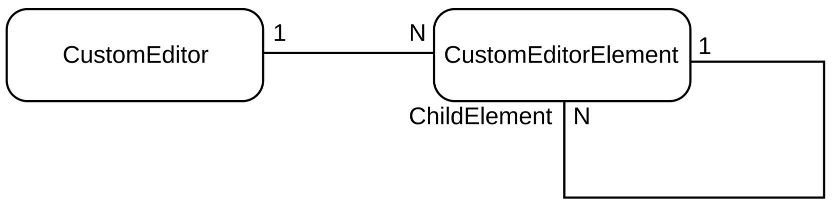


Figure 4.7: ER diagram of the custom view components

Table 4.2: Status types for custom editor views

Status	Visible for	Allowed to update
PRIVATE	owner	owner
PUBLIC	all users in the project	users with create, update, delete permission

5 Design and Implementation

This chapter explores the design and implementation based on the developed architecture. The technologies employed for the implementation are in line with those generally utilized in QDAcity. The frontend of the application is built with React[1] and a JavaScript framework, while the backend service operates on Google App Engine[2] and is developed in Java 8.

5.1 Task Navigation

Regarding the task navigation element, we implemented frontend and backend changes, which are presented in this section.

Backend

For each editor view that can be disabled, a state is stored in the project entity. Initially, the project settings also controlled the editor settings. Therefore, we first removed all attributes related to the editor settings from the project settings API method and DTO. Then, the relocation of the editor settings required a separate API method as well as a separate DTO, which is visible in Figure 5.1.

Frontend

The new navigation element is located at the top of the page in the form of a tab header. Figure 5.2 displays the UI of this new tab header. This header includes all elements that were previously located in the project section of the coding editor. For the implementation of the task navigation, we will focus on the tabs at the center of the header. When accessing a coding editor of a new project the coding and the editor tabs are displayed in the header. When the editor tab is disabled it is displayed in gray and does not react on hovering by changing its color. We describe the disabling logic for the editor tab in Chapter 4.1. Instead, a tooltip gives information on why the view is not supported.
The user may enable other views in a drop-down menu that is located to the

[1]https://react.dev/
[2]https://cloud.google.com/appengine

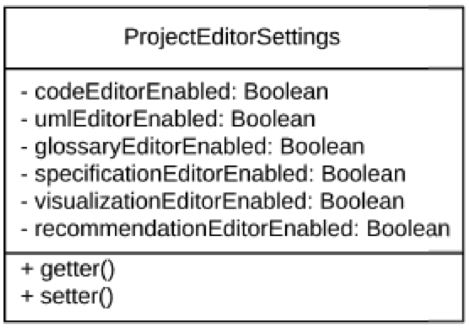

ProjectEditorSettings
- codeEditorEnabled: Boolean
- umlEditorEnabled: Boolean
- glossaryEditorEnabled: Boolean
- specificationEditorEnabled: Boolean
- visualizationEditorEnabled: Boolean
- recommendationEditorEnabled: Boolean
+ getter()
+ setter()

Figure 5.1: UML diagram of the editor settings DTO

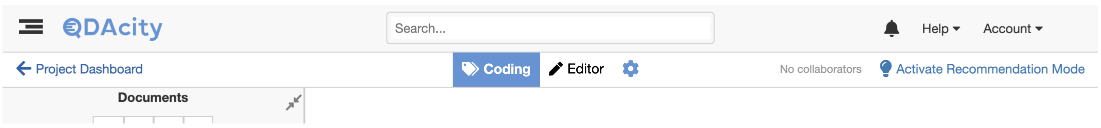
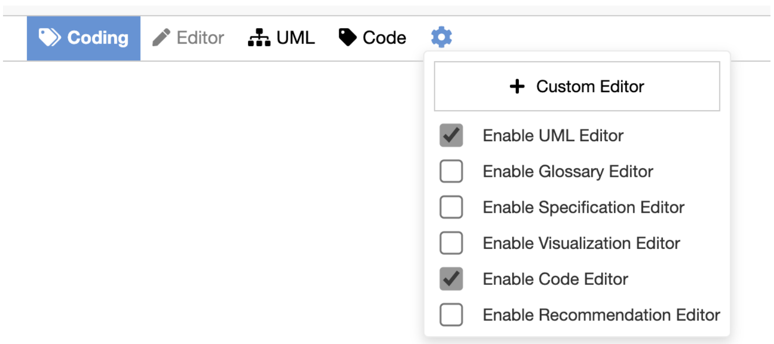


Figure 5.2: UI of the tab header

right of the tabs using a configuration icon. The drop-down menu is displayed in Figure 5.3. When selecting or deselecting an editor from the list the tabs will



- [✓] Enable UML Editor
- [ ] Enable Glossary Editor
- [ ] Enable Specification Editor
- [ ] Enable Visualization Editor
- [✓] Enable Code Editor
- [ ] Enable Recommendation Editor

Figure 5.3: UI of the tab configuration drop-down

appear or disappear instantly. However, the new settings state will only be stored when closing the drop-down menu to reduce traffic.
The navigation element has been developed to support smaller User Interfaces. The small navigation UI can be seen in Figure 5.4. When the window size is not big enough to display all tabs in one row, the tab header does not include the tab titles. Then, only the tab icons are shown and allow for navigating between views.

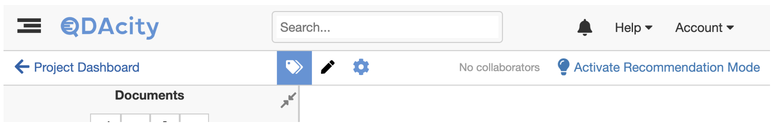
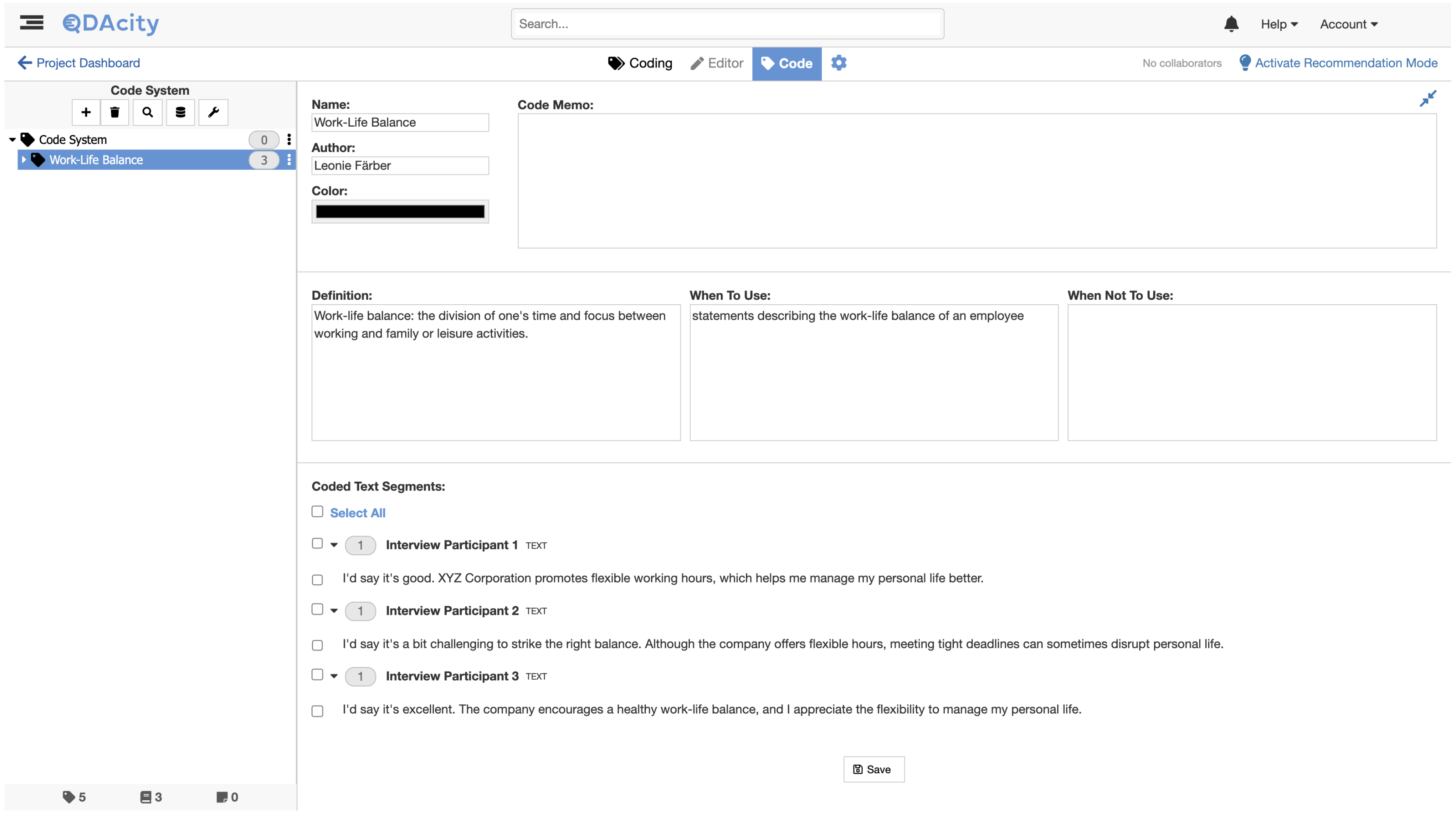


Figure 5.4: UI of the tab header for small page sizes

5.2 Code Editor View

The implementation of the code editor view required no new backend functionality. All required features can be implemented using existing API methods and data models.
The frontend GUI (Graphical User Interface) is displayed in Figure 5.5. The interface is split into three rows. The first contains the code properties and the code memo. This section can be collapsed by clicking the icon on the right. The collapsing feature provides users who do not want to edit the properties or the memo with a more space-efficient view. In the collapsed state, a user may see but not edit the code name and a shortened preview of the code memo.
The codebook entry fills the second row of the code view. The three different elements "definition", "when to use" and "when not to use" are arranged vertically. The third row contains a coding overview. This overview allows the user to read all codings belonging to a code sorted by documents. Additionally, the coding overview provides quick access to a coded data segment. When hovering over a coded segment, an arrow icon appears. On clicking this icon the coded segment is displayed in the coding view. This access to the coding in the coding view permits the user to quickly change the scope of the coding as well as read surrounding text to get context on a coding. The coding overview also permits the user to collapse documents to shorten the codings list and focus on specific segments. Moreover, there is an option to select codings or documents. When a document is selected or deselected, the same is done for all codings in said document. In addition, it is possible to select or deselect all codings in the list with one click. When at least one coding is selected a menu appears on the top of the section. This menu provides three actions that a user can perform for selected codes: move, copy, and delete. Figure 5.6 shows the UI for copying and moving codings. When copying or moving codings a drop-down menu allows the selection of a code. This is the code that the selected codings will be moved or copied to. When deleting codings a confirm dialogue opens, requesting the user to verify the action.
At the bottom of the page, a save button allows the user to save changes made in any input field. This button will communicate the state of the data by displaying an "unsaved changes" message when at least one field has been updated. When storing the changes the message disappears. Figure 5.7 shows the save button UI when there are unsaved changes.



Code System
- Code System 0
  - Work-Life Balance 3

Name: Work-Life Balance

Author: Leonie Färber

Color:

Code Memo:

Definition: Work-life balance: the division of one's time and focus between working and family or leisure activities.

When To Use: statements describing the work-life balance of an employee

When Not To Use:

Coded Text Segments:

Select All

1 **Interview Participant 1** TEXT

I'd say it's good. XYZ Corporation promotes flexible working hours, which helps me manage my personal life better.

1 **Interview Participant 2** TEXT

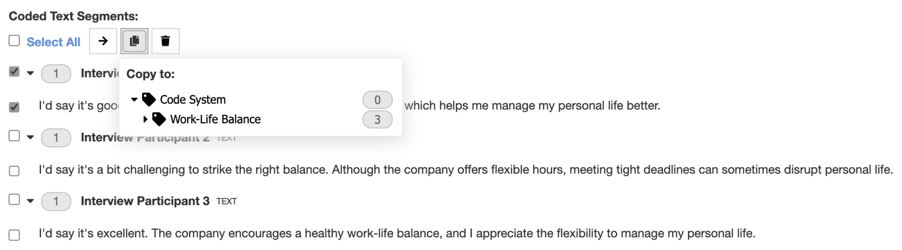
I'd say it's a bit challenging to strike the right balance. Although the company offers flexible hours, meeting tight deadlines can sometimes disrupt personal life.

1 **Interview Participant 3** TEXT

I'd say it's excellent. The company encourages a healthy work-life balance, and I appreciate the flexibility to manage my personal life.

Save

Figure 5.5: UI of the code editor view



I'd say it's goo which helps me manage my personal life better.

I'd say it's a bit challenging to strike the right balance. Although the company offers flexible hours, meeting tight deadlines can sometimes disrupt personal life.

I'd say it's excellent. The company encourages a healthy work-life balance, and I appreciate the flexibility to manage my personal life.

Figure 5.6: UI of the coding overview menu

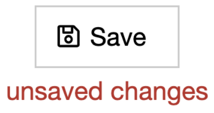


Figure 5.7: Save button UI for unsaved changes

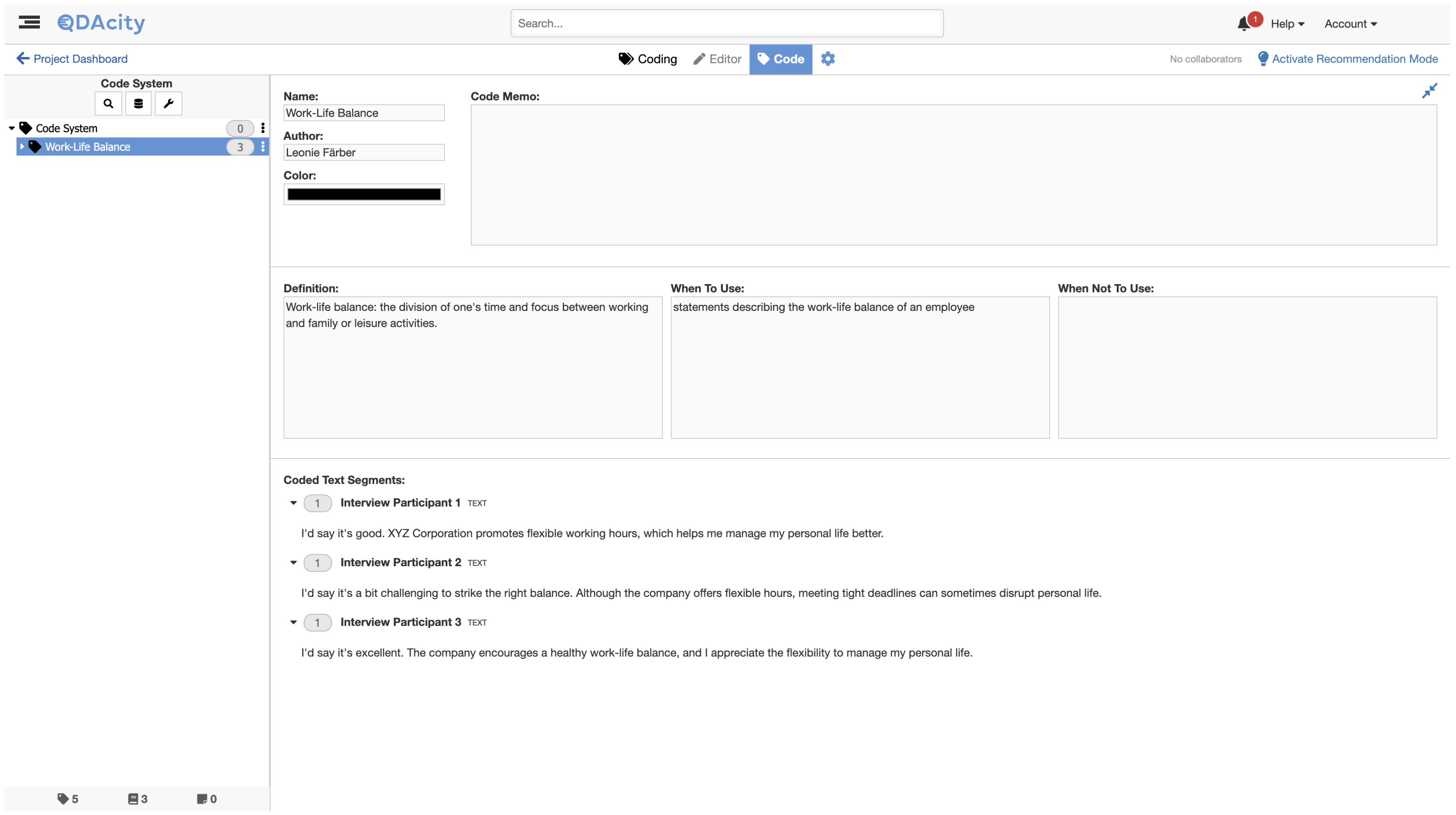
The implementation of the code editor view requires the support of some other existing QDAcity features. When the recommendation mode is enabled, any changes made in the input fields should be stored as recommendations and not overwrite the current data. Therefore, when saving changes in the recommendation mode, the corresponding recommendation dialogue is opened. Another feature that has to remain consistent in the coding editor is the use of permissions for different collaborator roles in a project. All updates in the coding editor view require permission to create, update, or delete elements in the coding editor. The save button and coding overview functionality check the user permissions accordingly. If the permissions are not given, the input fields are disabled, the save button is hidden and the coding overview does not allow the selection of documents. The disabled UI can be seen in Figure 5.8. A change to the UI of the code system is that the improved code system will not display the code view button in the code editor view. If the code view footer is enabled in other views, it does not appear in this view since it would provide redundant information.

5.3 Custom Editor View

The following section explores the implementation of the custom editor view. First backend changes are discussed before going into depth on developed frontend elements.

Backend

The data model for storing the custom editor views follows the architecture described in Chapter 4.3. It is displayed in Figure 5.9. The CodingEditor entity stores the ID, the project ID, the user ID of the creator, an isEnabled field, and the status of the type statusType. These fields aid in accessing views.



Name: Work-Life Balance

Author: Leonie Färber

Color: ■

Code Memo:

Definition: Work-life balance: the division of one's time and focus between working and family or leisure activities.

When To Use: statements describing the work-life balance of an employee

When Not To Use:

Coded Text Segments:

1 Interview Participant 1 TEXT

I'd say it's good. XYZ Corporation promotes flexible working hours, which helps me manage my personal life better.

1 Interview Participant 2 TEXT

I'd say it's a bit challenging to strike the right balance. Although the company offers flexible hours, meeting tight deadlines can sometimes disrupt personal life.

1 Interview Participant 3 TEXT

I'd say it's excellent. The company encourages a healthy work-life balance, and I appreciate the flexibility to manage my personal life.

Figure 5.8: UI of the disabled code editor view

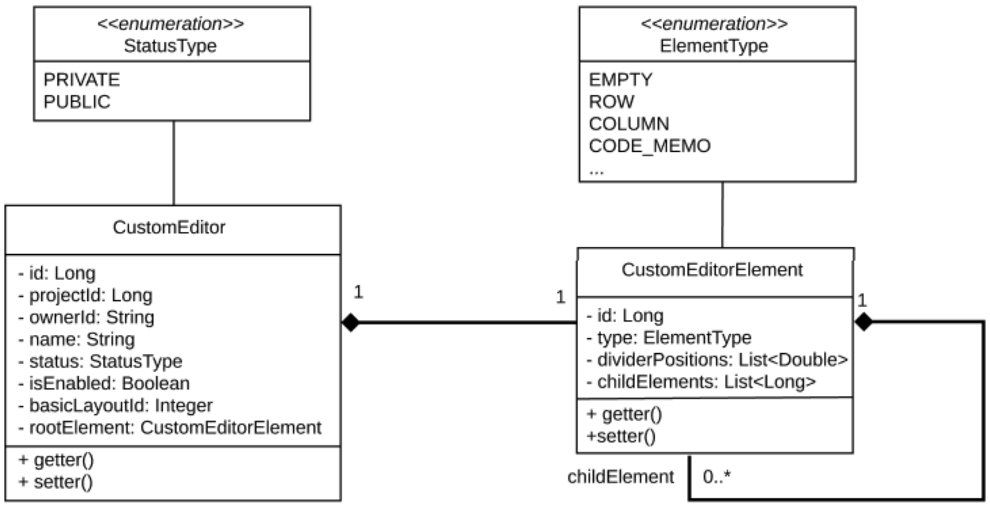


Figure 5.9: UML diagram of the custom view data model

The statusType enumeration contains the values "PUBLIC" and "PRIVATE" and determines the visibility of a view. The entity also includes the fields name, basicLayoutID, and rootElement. These are concerned with the content of the view. The rootElement contains a CodingEditorElement which is the outer element of the view's layout.
The CodingEditorElement entity stores an elementType property, a list of child element IDs, and a list of divider positions. The elementType declares the content of an editor element. This type can either indicate that an element is empty, that it is a row or a column, or that it contains a coding editor element. All possible values of this property can be seen in Table 5.1. The lists childElements and dividerPositions are only used for elements of the type "ROW" or "COLUMN". For these element types the childElements determine the content of the row or column while the dividerPositions determine their size. The number of children determines how many rows or columns are displayed.
For transferring data to the frontend a DTO is used for each entity. The objects differ from the data model displayed in Figure 5.9 in how they store references to coding editor elements. Instead of the ID or an array of IDS, a single CodingEditorElementDAO or an array of them is stored. Therefore, accessing an entire coding editor view in the frontend without having to load singular elements is possible. The API operations available for a custom editor view are the following:

- initCustomEditor

- getCustomEditorsForProjectAndUser
- renameCustomEditor
- updateCustomEditor
- updateIsEnabled
- deleteCustomEditor

The **initCustomEditor** method returns the DTO for a new coding view that is initialized with default values. The status is set to PRIVATE, it is named "Custom" and the basicLayoutId is 1 which refers to a one-element layout. Additionally, the method initializes a CustomEditorElement that is referred to in the rootElement property. This element is of the type "EMPTY" and therefore has no child elements or divider positions.

When loading the custom views for a coding editor the **getCustomEditorsForProjectAndUser** method can be used. This method returns all views that belong to the corresponding project and are visible to the current user. The ownerID and the status of a custom view determine the visibility. Views with the statusType PUBLIC are visible to everyone in the project, while PRIVATE views can only be seen by the owner of the view.

The methods **renameCustomEditor**, **updateCustomEditor**, and **updateIsEnabled** enable a user to update the properties of a custom view. **RenameCustomEditor** will only update the property name whereas **updateCustomEditor** edits multiple properties. These include the name, status, basicLayoutId and all necessary CustomEditorElements that are referenced as the root or child element of this view. The **updateIsEnabled** allows the separate updating of the isEnabled property. This property will only be set in the editor settings and thus we must handle it separately from the other properties that we may set in the custom view.

The last method, **deleteCustomEditor**, allows the removal of a custom view from the database. Consequently, all related CustomEditorElements are deleted as well.

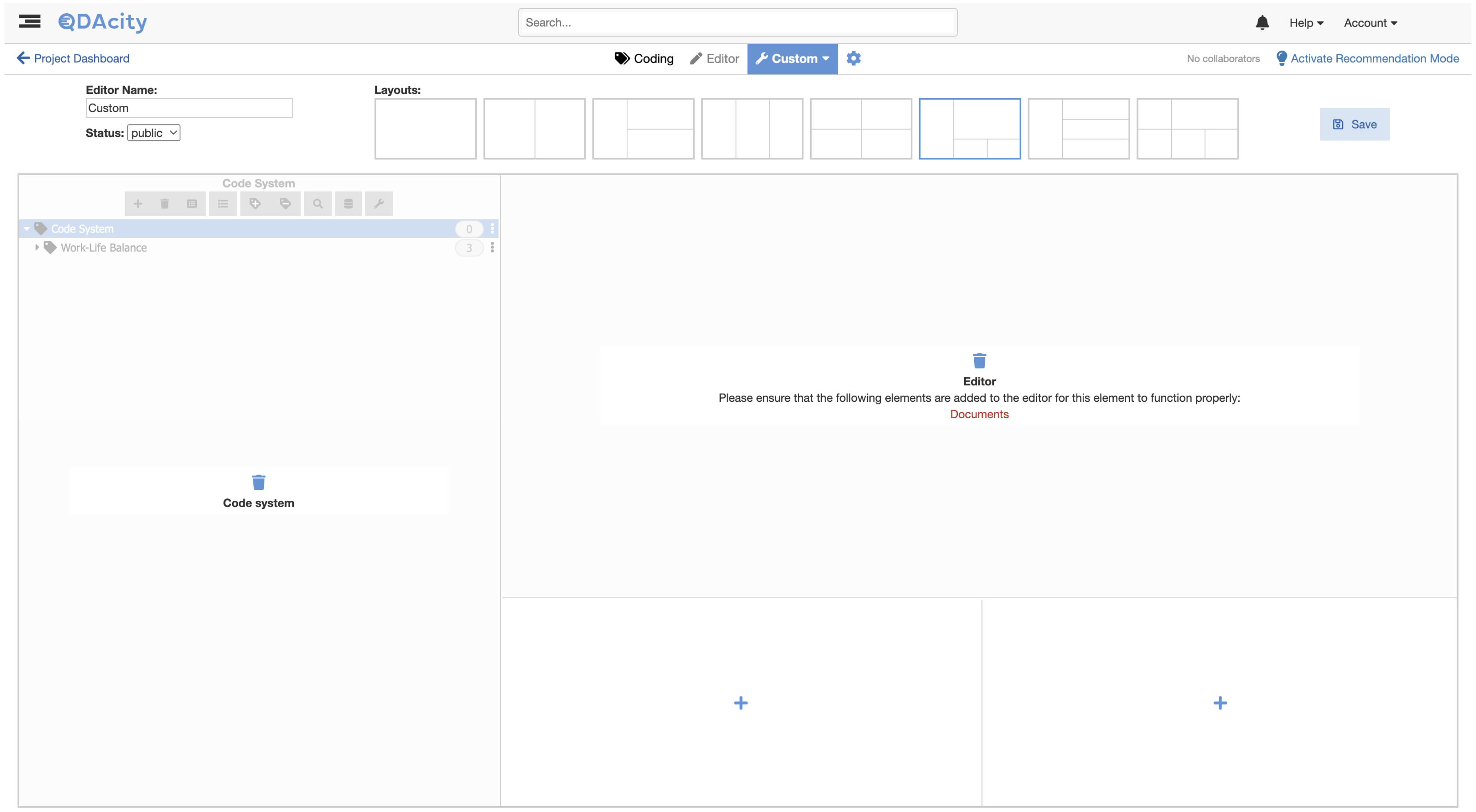
All API methods apart from getCustomEditorsForProjectAndUser check whether the user has permission to create, update, and delete elements in the coding editor. Additionally, all methods perform a check on whether the user is an authenticated QDAcity user. We added unit tests for the API methods that verify the correct authentication check as well as appropriate behavior for basic use cases.

Table 5.1: Element types in custom editor views

EMPTY
ROW
COLUMN
CODE_MEMO
CODE_PROPERTIES
CODE_SYSTEM
CODEBOOK_ENTRY
CODING_EDITOR
CODINGS_OVERVIEW
DOCUMENTS
EDITOR
GLOSSARY
SPECIFICATION
UML

Frontend

A user can create a new custom editor view in the editor tab settings. The button for adding a new view is located at the top of the list which can be seen in Figure 5.3. When adding a new custom view the coding editor will automatically switch to this view.
The UI for the custom editor view can be split into two separate interfaces for the different modes of the custom view. We developed the work and the edit mode in the architecture in Chapter 4.3. When a user creates a new custom view, initially the work mode interface is displayed. Figure 5.10 shows this interface. The editor mode offers the option of changing most of the view's properties. The menu on top allows the user to name the custom editor and set its status. Status modification is only possible for the owner of the view. Otherwise, the property appears disabled. Next to these two properties, a list of eight different layout options is displayed. These layout options were referred to as "basic layouts" in Chapter 4.3. The selected layout is framed in blue and is displayed in the lower section. This section displays the full layout that will be available when storing the view and switching to the work mode. Each element of the layout contains a plus icon if empty or a preview of its contained element. The previews will display the name of an element as well as a message informing the user about dependencies as discussed in Chapter 4.3. The preview also contains a trash icon that allows the user to delete an element. In the background, a disabled preview image of the element will be displayed at a lower opacity. A click on the plus icon in empty elements opens a drop-down menu displaying all elements that a user can possibly add to the layout. All elements can only appear once in a layout as discussed in the architecture section located in Chapter 4.3. The size of elements can be changed by dragging and dropping dividers between elements. When done with configuring a custom view the user can store the changes by clicking the save button in the top right corner. Then the custom editor view switches into the work mode.
An example of a custom editor in the work mode can be seen in Figure 5.11. The work mode of a custom view displays only the configured layout. Instead of the disabled element previews, this interface contains elements that a user can interact with. These elements are now fixed in size. For further modification, the user must access the edit mode.
A drop-down menu that the user can find in the tab of the custom view allows access to further functionality. This drop-down is displayed in Figure 5.12. The menu contains an option to return to editing the layout, renaming the editor, which will open a separate dialogue, or deleting the view. Figure 5.13 shows the dialogue interface for renaming a custom view. The user must confirm the deletion of an editor view in another dialogue window.
All custom editor views can be enabled and disabled as other coding editor views in the editor tab settings.

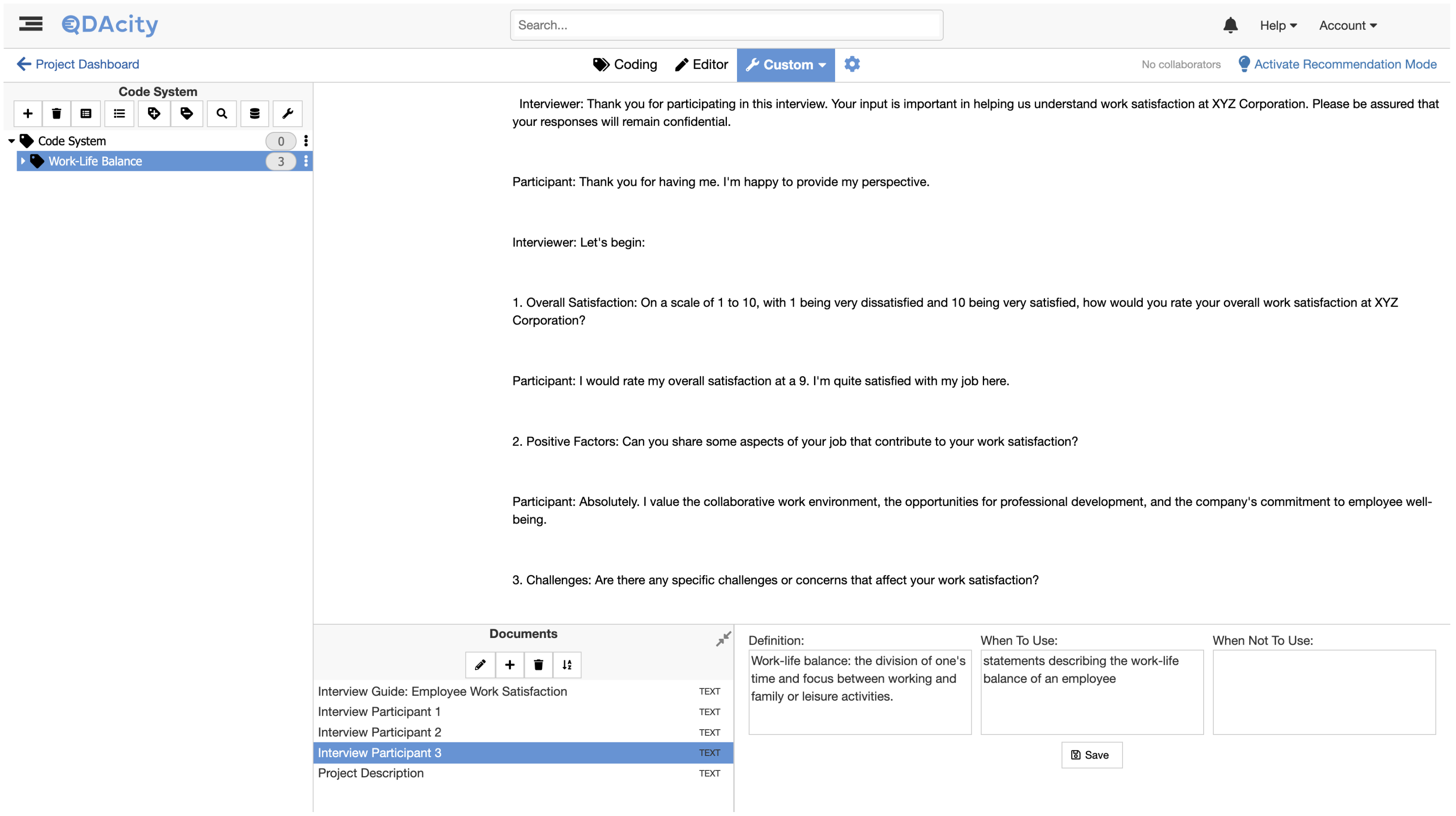


Editor Name: Custom

Status: public

Please ensure that the following elements are added to the editor for this element to function properly:
Documents

Figure 5.10: UI of the custom editor view in the edit mode



Interviewer: Thank you for participating in this interview. Your input is important in helping us understand work satisfaction at XYZ Corporation. Please be assured that your responses will remain confidential.

Participant: Thank you for having me. I'm happy to provide my perspective.

Interviewer: Let's begin:

1. Overall Satisfaction: On a scale of 1 to 10, with 1 being very dissatisfied and 10 being very satisfied, how would you rate your overall work satisfaction at XYZ Corporation?

Participant: I would rate my overall satisfaction at a 9. I'm quite satisfied with my job here.

2. Positive Factors: Can you share some aspects of your job that contribute to your work satisfaction?

Participant: Absolutely. I value the collaborative work environment, the opportunities for professional development, and the company's commitment to employee well-being.

3. Challenges: Are there any specific challenges or concerns that affect your work satisfaction?

Interview Guide: Employee Work Satisfaction	TEXT
Interview Participant 1	TEXT
Interview Participant 2	TEXT
Interview Participant 3	TEXT
Project Description	TEXT

Definition:	When To Use:	When Not To Use:
Work-life balance: the division of one's time and focus between working and family or leisure activities.	statements describing the work-life balance of an employee	

Figure 5.11: Example UI of the custom editor view in the work mode

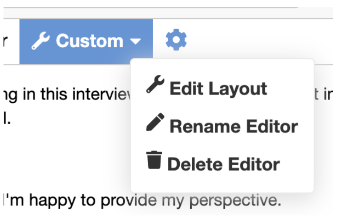


Figure 5.12: Drop-down UI of a custom view tab

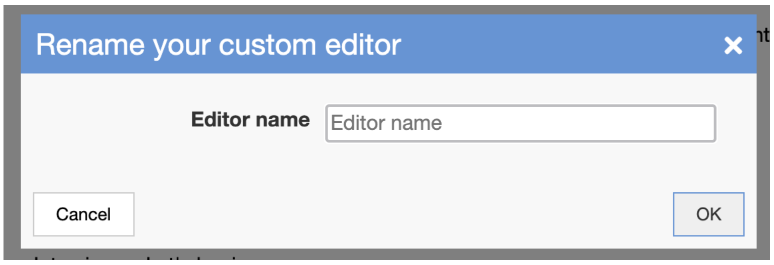


Figure 5.13: Dialogue window for renaming a custom editor view

5.4 Other Features

Some of the implemented changes affect more than one view. We removed the add and remove coding buttons from the code system for all views except coding and custom. These buttons only offer functionality if the coding editor element is present in the view. Therefore, they are unnecessary for all other tasks-views. Additionally, we implemented a code system statistics that can be seen in figure 5.14. We added the statistics to the left sidebar in all views but the custom view.

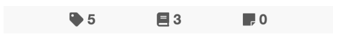


Figure 5.14: UI of the code system statistics

The custom view does not receive the statistics since it does not necessarily have a sidebar containing the code system. These statistics display information on how many codes the code system contains and how many codes have a memo, codebook entry, or meta-model information added to them. The meta-model information statistics will only be displayed if there is at least one code containing it. Therefore, users who do not use the meta modeling do not receive unnecessary information. This statistics section allows for a better overview of the entire code system. We first planned the feature for the code editor view to support the task of codebook refinement. However, since the code system is a central part of all non-customizable views, an overview can be beneficial wherever the code system is present.

6 Evaluation

This chapter assesses the alignment between the formulated requirements and the actual implementation of QDAcity. Moreover, the developed GUI was analyzed in a heuristic Evaluation according to Nielsen (Nielsen, 1995).

6.1 Requirements Assessment

The following section evaluates the realization of the requirements formulated in Chapter 3.

6.1.1 Coding Editor

Req 1.0 is satisfied

Req 1.0 - All task views in the coding editor should not include any functionality or information that is unrelated to their task.
Elements that do not have any functionality for a task view, such as the buttons for adding and removing codings were removed from the respective views. Additionally, we only allowed elements that are related to the tasks in new task views. The code view footer is disabled for the code editor view. For the custom editor view two different modes have been developed to better divide the tasks of working in a custom view and editing its content.

Req 2.0 is satisfied

Req 2.0 - The coding editor shall provide the user with the ability to easily switch between task views.
We eliminated the dependency between the selected document type and the navigation elements as far as possible. By restructuring the coding editor and adding a tab header to the page, the task navigation element is clearly separated from the editor content and can be accessed efficiently.

Req 2.1 is satisfied

Req 2.1 - The coding editor shall provide the user with the option of quickly adding and removing displayed task views.
The disabling and enabling of task views can be done in the new tab header. Moving this functionality into the tab header allows the user to modify the displayed editor views without leaving the coding editor.

Req 2.2 is satisfied

Req 2.2 - The display of the task views should be space-efficient for smaller window sizes.
Each view has an icon associated with it. When the page size does not allow for all tab titles and icons to be displayed, the tab titles are removed. Then the icons can be used to navigate between task views.

Req 2.3 is not satisfied

Req 2.3 - The display of the task views should provide the user with the ability to change the order of the tasks.
The order of the tab elements cannot be modified. This feature was omitted due to time constraints and other priorities.

6.1.2 Codebook Refinement

Req 3.0 is satisfied

Req 3.0 - The coding editor shall provide the user with the ability to select a view that supports the task of codebook refinement.
The user has access to the new code editor view. This view provides an interface to perform the task of codebook refinement.

Req 3.1 is satisfied

Req 3.1 - The codebook refinement view shall provide the user with the ability to see and modify the code properties, code memo, and codebook entry of a selected code.
The codebook refinement view allows the user to see and modify all information assigned to a code. The information includes the code properties, the code memo, and the codebook entry. Additionally, a user can prioritize the codebook entry by decreasing the size of the other two elements.

Req 3.2 and 3.3 are satisfied

Req3.2 - The codebook refinement view shall provide the user with the ability to see all codings of a selected code as well as quickly switch to a specific coding in the coding view.
Req 3.3 - The codebook refinement view should provide the user with the ability

to move or copy codings from one code to another as well as delete codings efficiently.
The codebook refinement view displays all codings in the coding overview section. An arrow icon allows for quick access to the coding in the coding view. Additionally, in the coding overview a user may copy, move, and delete codings.

Req 3.4 is not satisfied

Req 3.4 - The codebook refinement view should allow the user to write comments for codings.
We did not implement the feature of adding comments on codings. During the development of concepts and architecture, it became clear that this feature should be implemented on a bigger scale. A user should be able to write comments not only in the code editor but rather the functionality should be present in the coding editor as well. The implementation of this feature should be guided by further research and will require more time than what was available for this thesis.

6.1.3 Custom View

Req 4.0 is satisfied

Req 4.0 - The coding editor shall provide the user with the ability to add custom editor views to the selectable task views.
The user has the option of adding custom editor views to the coding editor. These views can be added in the editor tab settings.

Req 4.1 and 4.2 are satisfied

Req 4.1 – The custom editor views shall provide the user with the ability to name each view.
Req 4.2 - The custom editor views shall provide the user with the ability to change the visibility of each view in the project.
The custom editor view allows the user to attach a name to it. Additionally, the user may change the visibility by updating the status of the view. Both can be done in the edit mode of the view.

Req 4.3 is satisfied

Req 4.3 - When selecting a custom view, the user should be able to choose which elements out of the available coding editor elements the view should display.
The custom editor allows the user to add most of the elements in the coding editor. Some elements such as the visualization element are not available since they are still being developed.

Req 4.4 is partially satisfied

Req 4.4 - The custom editor should be highly modifiable regarding the size and

layout of its components.
The custom editor view allows for free size modification of elements. However, the view does not support the development of custom layouts. Instead, the user can select out of eight layout structures. This approach is beginner-friendly and can be extended in future work. More information on how the custom editor can be developed further can be found in the outlook given in Chapter 7.

6.2 User Evaluation

To evaluate the developed GUI based on usability and clarity, a user test was performed.

6.2.1 Procedure

The user-test procedure was based on the heuristic evaluation after Nielsen (Nielsen, 1995). In heuristic evaluation, experts called evaluators assess interfaces based on their usability and identify problems by performing a set number of tasks. An observer is present at all times and may answer any questions that arise throughout the process.

Evaluators

A minimum requirement for possible evaluators was previous experience in QDA. Therefore, the number of available, suitable evaluators was slightly limited. For this thesis, only two evaluators performed the user test. This number differs slightly from Nielsen's suggestion of acquiring a minimum of three evaluators to better fit the time limitations of this thesis. Out of the two selected evaluators, both had worked with MaxQDA which allowed for more comparison between tools. Moreover, one evaluator had limited experience in working with QDAcity.

Example Project

A small-scope example project was set up to provide the test environment. The project included a project outline, an interview guideline, and some interviews with company employees. The documents were coded using a limited set of codes. Prior to the test, all editor views were disabled if possible.

The evaluators then performed the following three tasks:

1. Locating an editor view that provides information on a selected code
2. Dissolving a code, including moving all codings to another code and writing a short memo text

3. Creating a customizable editor view containing the coding editor and the code memo

After each task, the observer initiated a discussion that aimed at answering the following questions:

- Which steps posed a problem and why?
- Was the GUI intuitively usable?
- Is the GUI efficient in a daily workflow?
- What improvements could be made?

6.2.2 Results

While we can make some improvements in the areas of usability and clarity, the developed GUI was generally evaluated as very useful for a task-focused QDA workflow. The following sections describe the user feedback given on the performed tasks. The full evaluations can be found in the appendix sections A and B.

Locating an editor view that provides information on a selected code

Initially, both evaluators struggled with locating the tab header. They expected a view related to code information to be located in the code system. Even when finding the tab elements the configuration icon was not seen as configuring the tab elements but rather general settings. Part of this problem might be caused by the test environment since a new user could normally take the time to perform the tutorials provided by QDAcity. In these tutorials, the tab header is used to switch between the coding and the editor view. This might help a new user to quickly locate and use the tab functionality.
Possible improvements for the tab header would be adding more color to the tabs or having at least three tabs displayed as the standard tabs to make the tab header look more clickable. Furthermore, an evaluator proposed changing the configuration icon to a drop-down icon. However, the suggestion is incompatible with the display of the custom editor tabs since the drop-down icon is already used here. Therefore, there might be two drop-down icons directly next to each other in the header but have different purposes which could confuse users.
Generally, the evaluators assessed the tab header as very useful and even preferred it to the GUI provided in MaxQDA. The switching of task views using one click allows for an efficient workflow.

Dissolving a code, including moving all codings to another code and writing a short memo text

The evaluators expected functionality on moving codings to be located in the

code system. After searching in the code editor, discovering the coded segments section was no problem. Both evaluators wished for a drag-and-drop option on codings to drag them directly into the code system view and possibly user feedback on moving codings. However, an evaluator also considered, that user feedback in the form of a confirm dialogue after every action might be tedious when moving codings to multiple codes. A compromise for this issue might be an undo option.
The save button in the code editor should be more visible, either by making it more colorful, moving it further up the page, or displaying a save button in each input field. An evaluator considered the code memo as a secondary element of the view and would prefer a smaller display located further down the page. Other proposed improvements for this view include indenting the coded segments on different levels to clearly distinguish documents and codings. Furthermore, there should be an option to quickly see the surrounding text of the codings to give the user more context when necessary. This functionality could be provided in a specific view where the user is also allowed to document code themes. Additionally, linking and viewing deleted codes would be useful, to allow a user to revisit old codes and make code deletion non-permanent. Moreover, more comment space would improve communication with collaborators on developing codes. One evaluator would also prefer a longer editor name and considered that the tab title "Codebook" might define the view better than just "Code".
Both evaluators regarded the code editor as very beneficial for performing QDA. The clear structure of the view and the limited use of color allows for easy use. One evaluator stated on opening this view while performing the previous task that this interface was lacking in MaxQDA.

Creating a customizable editor view containing the coding editor and the code memo

When facing this task, locating the button for adding a custom editor was difficult. Again, this issue might not come up with users who had more time to discover the tab header functionality. However, it could be improved by providing a tutorial for creating a custom editor. Another problem was understanding the meaning behind the status property. The keywords used here should define the visibility more clearly. The resizing functionality and the save functionality are clear to use. The text of the warning message for dependent elements might be hard to read depending on the element displayed in the background. More user feedback on adding dependencies in the form of turning added elements green in the dependency messages would be nice. Loosing previously selected Elements when changing the layout leads to re-adding multiple elements which is tiresome. Elements should remain selected when switching between layouts.
It would be an improvement to enable size modification after the layout is saved. Switching back and forth between editing the layout and working to modify sizes would be inefficient. Dragging and dropping elements in the layout would make

the configuration process more efficient. Allowing the user to completely customize the layout would be a beneficial extension of the current functionality.
The reaction on whether the evaluators would use this view for their QDA workflow was mixed. One evaluator was certain that the custom editor view is very valuable as it provides the option of layout customization that is currently missing from the rest of the view. The other evaluator was content with using the standard views but imagined the custom editor view to be useful for more complex tasks.

Further comments

The naming and icons of the coding editor tabs could be improved. Size modification should be allowed in as many coding editor interfaces as possible. Moreover, the size modification option for the coding brackets in the coding editor view should be more visible.
Further tasks that could be supported are team polls, commenting, and task distribution in teams.

6.2.3 Implementation of Issues

After evaluating the user feedback some changes were made to the implemented GUI.
The design of the save button in the code editor view is now bigger and more colorful to increase its visibility. Figure 6.1 shows the new UI. For this, a newly developed button design was used, which was not yet available when implementing the code editor view. The warning message for dependent elements in the coding

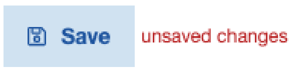


Figure 6.1: New save button design

editor is now displayed on a white background to improve legibility. Moreover, the dependencies that are added to the layout are displayed in green. As proposed by an evaluator, the "status" text in the custom editor view was changed to "visible for" and the selectable options are "me" or "everyone in this project". Another improvement in the custom editor view is keeping the elements on changing the layout. When selecting a new layout with the same or a bigger number of possible elements, all added elements remain in the new layout. When changing to a layout with fewer possible elements, the maximal possible number of elements is kept in the layout. Finally, the code tab has been added to the standard tabs that cannot be disabled. Including the code view in the standard views not only increases the visibility of the tab header by always displaying a minimum of three tabs, but it also establishes the code editor view as a standard view for QDA. Further improvements can be made in future work. Chapter 7 explores some possibilities of integrating more user feedback in the UI.

7 Future Work

This chapter explores possible areas of future work. The concepts in this section are mainly based on the performed research or the feedback from the user test.

Code Editor View

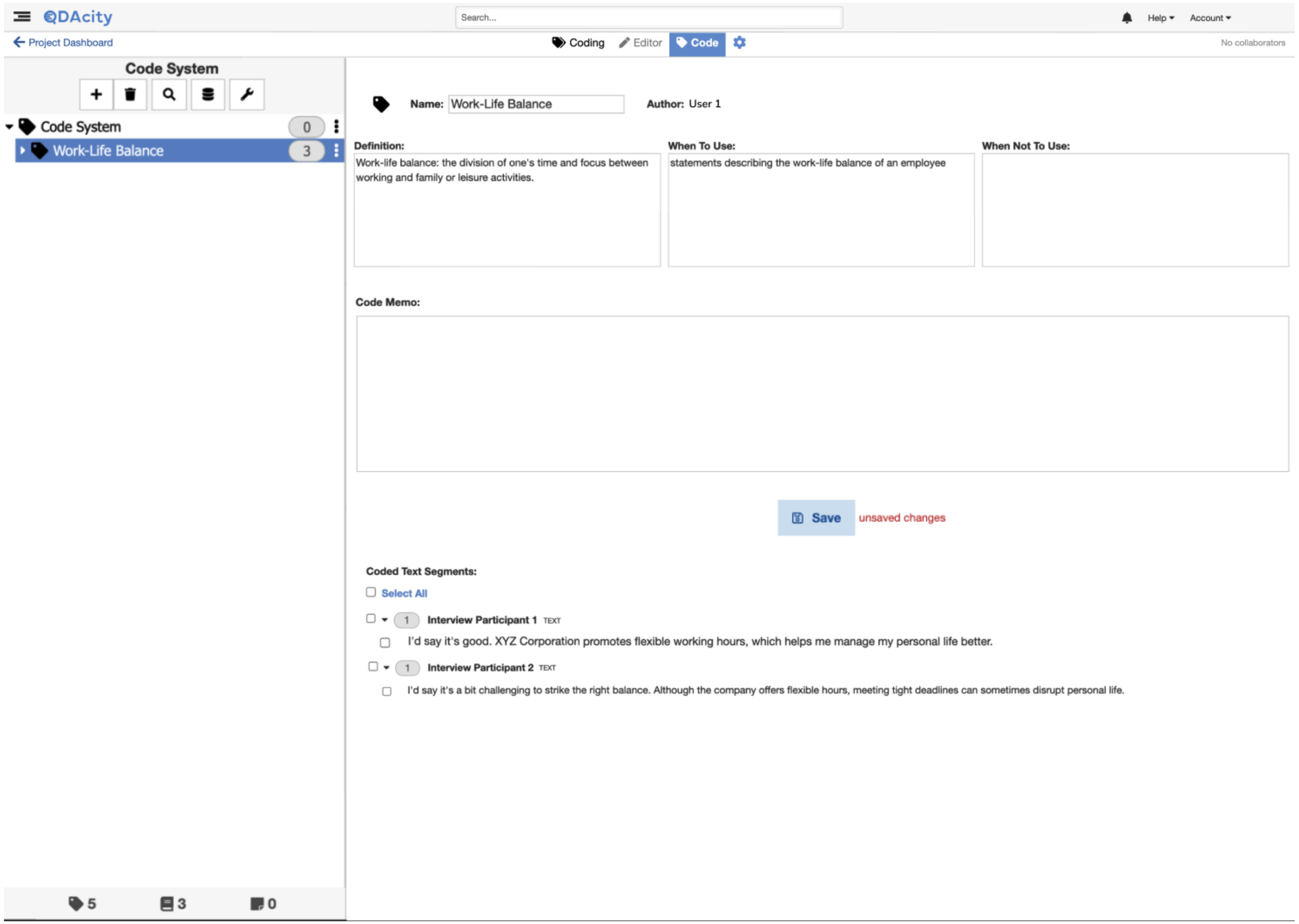
The code editor view can be further developed based on the feedback from the user evaluation. A frame of a possible improved interface can be seen in Figure 7.1. An important point is the restructuring of the coding overview, the coded segments can be indented to better mark the difference between documents and codings. Dragging and dropping codings into codes in the code system should also be allowed.
The input fields for code information can also be improved. The author of a code should not be modifiable but rather just displayed. The color of the code can be changed directly in the tag symbol. The code memo is now located beneath the codebook entry to better state the importance of the codebook.

Custom Editor View

The custom editor view can be further developed in the face of layout modification as discussed above. Therefore, an interface allowing for complete modification should be developed. A proposal for this can be seen in the frame in Figure 7.2. In this frame, new columns and rows can be added to all sides of the layout. Each element allows the addition of a row or column. This editor frame also allows for drag-and-drop of fields as well as the content of the fields. Additionally, allowing for size modification of elements in the work mode of the editor view would improve the custom editor view further.

Analysis Tools

As mentioned before, performing QDA in QDAcity can be optimized by supporting more analysis elements. The visualization of relationships between codes and theories is a huge benefit that CAQDAS tools can provide. QDAcity's functionality in this area can still be expanded and some features such as the comparing of codes should be revised to improve their performance.



Name: Work-Life Balance Author: User 1

Definition:
Work-life balance: the division of one's time and focus between working and family or leisure activities.

When To Use:
statements describing the work-life balance of an employee

When Not To Use:

Code Memo:

unsaved changes

Coded Text Segments:

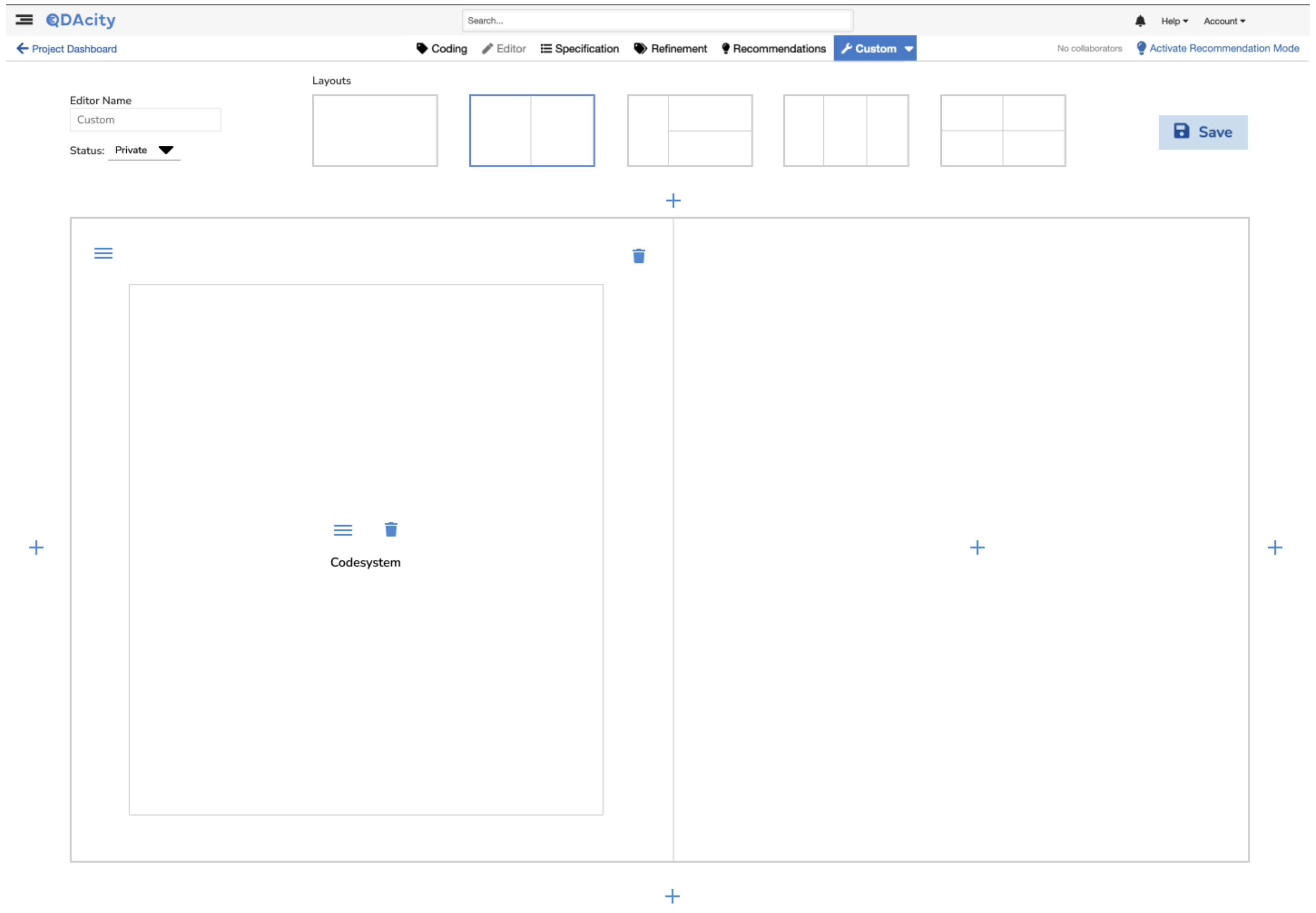
1 Interview Participant 1 TEXT

I'd say it's good. XYZ Corporation promotes flexible working hours, which helps me manage my personal life better.

1 Interview Participant 2 TEXT

I'd say it's a bit challenging to strike the right balance. Although the company offers flexible hours, meeting tight deadlines can sometimes disrupt personal life.

Figure 7.1: Frame of a possible future code editor view



Editor Name: Custom

Status: Private

Figure 7.2: Frame of a possible future custom editor view

8 Conclusion

The aim of this thesis was to develop a task-focused editor for QDAcity. The editor was supposed to enable efficient task-switching as well as support most standard QDA tasks.
First, we analyzed different approaches to qualitative research and their implications for QDA, going in-depth on the exemplary strategy of grounded theory (GT). Then different tasks were derived from this overview. In addition, we analyzed two other computer assisted qualitative data analysis software (CAQDAS) tools and evaluated their strategy for providing functionality for the developed tasks. Afterward, QDAcity was introduced, giving a quick overview of existing features in the coding editor.
In Chapter 3 we stated 15 requirements for the new task-focused editor interface. Then an architecture was developed that supports the given requirements. The architecture focuses mainly on three parts: the creation of a task-navigation element, a code editor view, and a custom editor view.
Then the implementation of the architectural concepts was described. Additionally, some changes were implemented across multiple editor views. This chapter especially went in-depth on the developed data models and user interfaces.
We evaluated the implementation based on the requirements formulated in Chapter 3. 12 requirements were satisfied while one requirement was partially satisfied and can be further developed in future work. Two requirements were not met because of time limitations for this thesis. We also performed a user test. The test aimed at evaluating the usability of the developed interfaces. All three main elements, the tab header for navigation, the code editor view, and the custom editor view were seen as useful by the evaluators. Some improvements could be made, of which a couple were already implemented.
We also made some suggestions for future work. The recommended improvements target the areas of the code and custom editor as well as analysis tools.
This thesis developed a task-focused editor interface for QDAcity, giving users quick access to different task views. We also achieved the implementation of two new views that support the task of codebook refinement and the customization of an editor interface. Therefore, QDAcity is now better equipped to enable a task-focused workflow.

Appendices

A Evaluation of QDA-Expert 1

The following section presents a user evaluation of the developed UI translated into English.

1. Locating an editor view that provides information on codes

Initially, locating the tab header posed a problem since the evaluator expected any functionality related to codes in the code system. Specifically, in the toolbar or a context menu. The evaluator discovered the code view bottom panel this way. Even after locating the tabs, the configuration icon did not seem related to adding more editor views but rather was expected to lead to general settings such as text size.
To improve the issue of locating the view, the evaluator would prefer a drop-down icon rather than the current configuration icon. Renaming the editor view to “Codebook” rather than “Code” would clearly define its task. Generally, the tab header is helpful and efficiently placed at the top of the page but might not be instantly noticed by first-time users. The evaluator prefers it compared to the functionality in MaxQDA.
The code editor view will help refine the code system and is missing in MaxQDA where you have to navigate to each code individually to modify it. The evaluator would add an interface to the code editor view where code themes can be documented.

2. Dissolving a code, including moving all codings to another code and writing a short memo text

Initially, the evaluator expected the codings as elements of the code system. After further consideration, this idea could not be implemented efficiently. Locating the coded segments section was uncomplicated when searching in the code editor view. The evaluator would prefer to drag and drop codings. Selecting and moving codes was swift. The evaluator expected a confirm dialog on moving codings but also considered that it might be tedious. A compromise might be an undo option. When merging two codes showing both code entries would be helpful to compare and contrast. Linking a deleted code that was merged would be helpful in understanding the development of the code system at a later point. This would also allow the user to delete codes non-permanently by having the option to revisit old codes. The evaluator expected a context menu for deleting codes.
The coded segments view could be improved by indenting elements on different levels so that the distinction between documents and coding is clear. The understanding of codings would be better if there was an option to show surrounding text for each coding. This would give the user context without having to switch

the view to the coding editor. This could be done in a specific task view that could then also allow for theme definitions and summaries. Another possibility would be to show more context to a code when hovering over it. The arrows allowing to switch to the coding in the respective documents are only noticeable when working in the coded segments section. Additionally, the save button should be more visible if there are unsaved changes. Moving the button closer to the text input fields would also improve its visibility. Another option would be to have a save icon in each input field. The evaluator considers the code memo as a secondary element and would prefer the codebook entry in its position. The code memo could be moved to a speech bubble that is used to make a new note. Collapsing the first section does not have the wanted effect since changing the color of a code should remain possible.

3. Creating a new customizable editor view containing the coding editor and the code memo

Initially, the evaluator was surprised that QDAcity offers customizable interfaces since most of the views are static. When looking at the tab configuration settings, the create editor button could be located at the end of the list rather than at the beginning. The status message in the new custom editor view could be longer and thus easier to understand. The different options could be renamed to "visible for me" / "visible for the team". The evaluator found the resizing functionality, save button, and warning of missing elements message instantly. The custom tab drop-down options are also easily identifiable. The text of the warning message was hard to read over the coding editor element and the evaluator would prefer a set of missing elements for the entire custom editor layout. When a required element has been added to the custom editor the warning message should show this element in green rather than black. The selected elements should stay in the custom editor on updating the layout and an option of moving elements from one field to another would be more efficient.
Resizing the elements is very helpful and makes the provided layouts work for multiple purposes. Enabling the user to define their own layouts would be a beneficial extension of the current functionality. It would also be nice to have more options for layout and size modification in the other editor views. Further improvements for the custom editor view would include allowing drag-and-drop for modification but the current element for adding elements is very space-efficient and intuitive.

Further comments

The tab header is very intuitive to use after spending some time with the tool and differs quite a bit from comparable QDA tools. A full-screen view of only the selected editor view might be beneficial for focused work that does not require switching between editor views. The naming of tasks in the tab header should be improved to make them more cohesive. The tag- and tags-icon of the coding and the code editor view are too similar. Building a custom view is very helpful for the user. Size modification in the custom editor views should always be allowed. In the coding editor view, the size modification is not visible.

B Evaluation of QDA-Expert 2

The following section presents a user evaluation of the developed UI translated into English.

1. Locating an editor view that provides information on codes

The evaluator was looking for a code editor view in the three dots code menu and the code system toolbar and found the code view bottom panel instantly. Finding the tab configuration icon required a hint even though using the tabs for navigating back to the coding editor view was clear.
The tab bar was not seen as clickable because of a lack of colors. Including at least 3 tabs in the standard view makes it easier to notice the tab options. Aligning the tabs to the left would also make them look more like a selectable menu. Generally, the tab header is very efficient. Switching the editor view with one click is key for a good workflow.

2. Dissolving a code, including moving all codings to another code and writing a short memo text

When moving codings a drag-and-drop option would be very intuitive. Initially, dragging and dropping a code into another code seemed like the best solution for merging codes. However, that turned the code into a new subcode. After receiving a hint of manually moving the codings to a different code, the evaluator found the coded segments view easily. User feedback on moving the codings would be helpful to know that the changes were made. It was unclear whether moving the codings required pressing the save button. Generally, saving changes can be forgotten easily.
The coded segments view does not receive instant attention at the end of the page. The save button should be more visible. Generally, it is positive that the editor has a clear structure. The limited use of color makes using it feel more relaxed than other tools. Further improvements would include more comment space to communicate with colleagues on developing codes.

3. Creating a new customizable editor view containing the coding editor and the code memo

The use of the tab configuration icon for adding a custom editor view is intuitive but the button was not found which could be improved by making it look more clickable. Selecting a layout, adding elements and saving the layout was clear. The custom tab drop-down with further functionality was also found easily when wanting to edit the layout after saving it. The warning message for missing elements was noticed but should be more extensive. It should clearly note all steps that should be taken to solve the issue of missing elements. After further hints selecting a bigger layout for more elements was clear but the evaluator did

not know immediately where to find the missing elements.
Customizing the size of elements is clear but should also be possible after saving the layout. Switching back and forth between the edit and work mode of the view to change sizes would be frustrating. Changing sizes accidentally while working should not be an issue. For this user test the button for adding a custom editor view could have been more prominent but in a normal workflow it might not be as important. Generally, making the button look more clickable would suffice. The editor might be helpful for complex tasks but the evaluator is content using standard views for now.

Further comments

Further tasks for the coding editor that the evaluator would benefit from are a view for team polls, more commenting functionality, more options for adding notes, and a view that can be used for distributing tasks in a team. The size modification in the coding editor view is not visible.

References

Campbell, J. L., Quincy, C., Osserman, J., & Pedersen, O. K. (2013). Coding in-depth semistructured interviews: Problems of unitization and intercoder reliability and agreement. *Sociological Methods and Research*, *42*(3), 294–320. https://doi.org/https://doi.org/10.1177/0049124113500475

Corbin, J. M., & Strauss, A. (1990). Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative Sociology*, *13*(1), 3–21. https://doi.org/https://doi.org/10.1007/BF00988593

Creswell, J. W., & Creswell, J. D. (2018). *Research design: Qualitative, quantitative, and mixed methods approaches* (5th ed.). SAGE.

Dey, I. (1993). *Qualitative data analysis: A user friendly guide for social scientists.* Routledge. https://doi.org/https://doi.org/10.4324/9780203412497

Flick, U. (2009). *An introduction to qualitative research* (Fourth edition). Sage Publications.

Graue, C. (2015). Qualitative data analysis. *International Journal of Sales, Retailing and Marketing*, *4*(9), 5–14.

Mayer, I. (2015). Qualitative research with a focus on qualitative data analysis. *International Journal of Sales, Retailing and Marketing*, *4*(9), 53–67.

Merriam, S. B., & Tisdell, E. J. (2015). *Qualitative research : A guide to design and implementation.* John Wiley; Sons.

Miles, M. B., & Huberman, A. M. (1994). *Qualitative data analysis: An expanded sourcebook* (Second edition). Sage Publications, Inc.

Nielsen, J. (1995). How to conduct a heuristic evaluation. *Nielsen Norman Group*, *1*, 1–8.

Sophisten. (2016). *Schablonen für alle fälle* (Third edition). https://www.sophist.de/fileadmin/user_upload/Bilder_zu_Seiten/Publikationen/Wissen_for_free/MASTeR_Broschuere_3-Auflage_interaktiv.pdf

Stol, K.-J., Ralph, P., & Fitzgerald, B. (2016). Grounded theory in software engineering research: A critical review and guidelines. In *Proceedings of the 38th international conference on software engineering* (pp. 120–131). Association for Computing Machinery.