Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

GREGOR FENDT
MASTER THESIS

META MODEL VISUALIZATION FOR QDACITY

Submitted on 22.05.2022

Supervisors: Julia Krause, M.Sc.
Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

_____

Bamberg, 22.05.2022

License

_____

Bamberg, 22.05.2022

Abstract

The web app QDAcity supports the process of qualitative data analysis (QDA). Among other things QDAcity can aid users during the coding process. A unique feature of QDAcity is that the user can classify codes of their code system using a meta model. The meta model helps in structuring the code system and expressing semantic relationships between codes.
This thesis develops an interactive visualization of the meta model of QDAcity. The visualization could help in overviewing, communicating, and manipulating the meta model.

Contents

1 Introduction

1.1 Motivation

QDAcity[1] is a web application to support qualitative data analysis (QDA). QDAcity's code system is an artifact emerging from a coding process. Researchers discover concepts in the qualitative data and label them with a code. When new codes get created in QDAcity, they get sorted into the hierarchical code system. Furthermore, the codes can be classified using the code system language (CSL). The CSL describes a meta model and restricts the application of codes. It is a unique way of classifying concepts and expressing semantic relationships between them. In QDAcity a user can classify a code by associating them with elements of this meta model. The goal of the code system is to make it possible to create formal models and other artifacts out of qualitative data, and the meta model was created to support this process, by acting as a single unified model (Kaufmann, 2021). A researcher structuring codes using the CSL supports the derivation of other models, like conceptual or behavioral models from his code system.
Computer assisted qualitative data analysis software (CAQDAS) like QDAcity is commonly used to support researchers with the documenting, structuring, and analyzing during the QDA process. The usage of computers can be an efficient and effective tool for QDA. CAQDAS not only facilitates the research process and makes the process less error-prone; it is also enabling new efficient ways of creating visualizations. A visualization could make it possible "to describe, document, and share complex data in a clear and precise manner" (Pokorny et al., 2018, pp. 170–171). Furthermore, interactive visualizations can allow the users to manipulate the data to express their own ideas or help them in discovering the data or new concepts in the data. However, most CAQDAS doesn't provide a qualitative visualization of their code system. Often qualitative researchers transform qualitative data into quantitative segments, making the visualization process more objective, but at the same time losing the ambiguity and subtleties, which might be a big value of qualitative data (Henderson & Segal, 2013, p. 68). The CSL of QDAcity provides the possibility to create an interactive visualization, which pictures concepts and relationships in way that surpasses the expression of tables or other quantified visualizations.

[1] https://qdacity.com/

QDAcity can benefit from an interactive meta model visualization in multiple ways. Visualizations offer new ways of overviewing and thinking about data. During the analysis the researcher can use the meta model visualization to overview his code system more easily and it might help him in discovering new concepts and interrelationships. Furthermore, the user can edit the meta model inside the visualization. Another benefit could be to use the visualization as a communication tool. An important part of QDA is to make the inference process and the analysis results transparent to other people like colleagues or stakeholders in the study. In QDAcity the user could filter and arrange the codes of his codes system, such that he could represent his code system or highlight certain ideas more easily to other people, which aren't as familiar with his code system. Visualizations could also be beneficial to document the coding process and they might capture the general development of the code system over time.

This thesis develops an interactive qualitative visualization of QDAcity's meta model. The visualization enables the user to overview, investigate, and manipulate the meta model. Design decisions are made by looking at possible visualization methods, how other tools visualize their code system and the goals of QDAcity and the meta model. QDAcity's meta model offers unique capabilities to express the concepts and semantic relationships of a code system. There exists a lot of different CAQDAS, but we couldn't find another web app, which offers a qualitative visualization of the concepts and relationships of its code system.

1.2 Thesis Structure

At first, Chapter 2 explains some important aspects of qualitative research and mainly focuses on QDA. Then it gives an overview of QDAcity and explains the meta model. Chapter 3 then maps out possible benefits, visualization methods and guidelines of the visualization, by relating it to other research.
In Chapter 4 the requirements of the implementation of the visualization are listed.
After that in Chapter 5 we compare the qualitative visualizations of different CAQDAS.
We use a graphic library to assist us with implementing a visualization. That's why Chapter 6 inspects different libraries and chooses the best fitting one.
Then Chapter 7 illustrates the concept and design of our visualization.
Chapter 8 gives a brief overview of the architecture and Chapter 9 presents the final implementation.
In Chapter 10 we evaluate the requirements of Chapter 4.
Finally, Chapter 11 concludes the thesis, presents the results, and lists some improvements and outlooks.

2 Fundamentals

Chapter 2.1 begins with explaining the fundamentals about qualitative research, focusing on the analysis part of qualitative research. Then Chapter 2.2 is about the web-application QDAcity, the QDAcity-RE method, which QDAcity is based on, and presents important features of QDAcity.

2.1 Qualitative Research

Qualitative research engages in the gathering, organization, and interpretation of qualitative data and is applied in a variety of fields, as for example sociology, medicine, economics and psychology (Flick, Kardorff, & Steinke, 2010, pp. 3–5). Qualitative data is non-numerical data and deals with meanings, in contrast to quantitative data there are no explicit rules to analyze them, but rather broad guidelines (Bryman & Bell, 2011, p. 571). Data in a qualitative research study can be relatively unstructured and of different data types (Hammersley, 2013, p. 12). Depending on the studied phenomena, different data collection methods are used. Data can be for example gathered through interviews, focus groups, observations, documents, or other artifacts (Austin & Sutton, 2014, p. 438). The combination of different data collection methods results in a wide range of data formats. The focus can often be on textual data, but some tools supporting qualitative research offer support for multiple data types, like audio, video, image, or social media data[2]. Qualitative research tries to systematically understand processes, patterns, and structural features in qualitative data, to gain an understanding of a complex phenomenon. In this context the abstract term phenomena stands for the concept being researched. In contrast to quantitative research methods, qualitative research focuses more on subjective perspectives about the phenomena (Flick et al., 2010, p. 5). This is also the reason why avoiding subjectivity in qualitative research might be impossible, as the research topics are complex and happening in the “real” world, not under experimental conditions (Hammersley, 2013, pp. 11–14). It is common to acknowledge subjectivity in qualitative research, without labeling it as “non-rigorous” research (J. Bettis & A. Gregson, 2001)

[2] QDA tools like ATLAS.ti and MAXQDA support various types of data:
https://doc.atlasti.com/ManualWin.v9/Documents/DocumentsSupportedFileFormats.html
https://www.maxqda.com/help-mx20/import/data-types

2.1.1 **Qualitative Data Analysis (QDA)**

QDA is the process of describing, classifying and interconnecting phenomena with concepts created by the researcher (Graue, 2015, p. 8). Creswell represents the QDA process as a spiral, which can be seen in Figure 1 (2013, p. 182). Qualitative researchers start at the bottom of the spiral with data collection, and then move in a non-linear way through the procedures, often circling back to previous ones. The process stops when an account or narrative about the data is created.

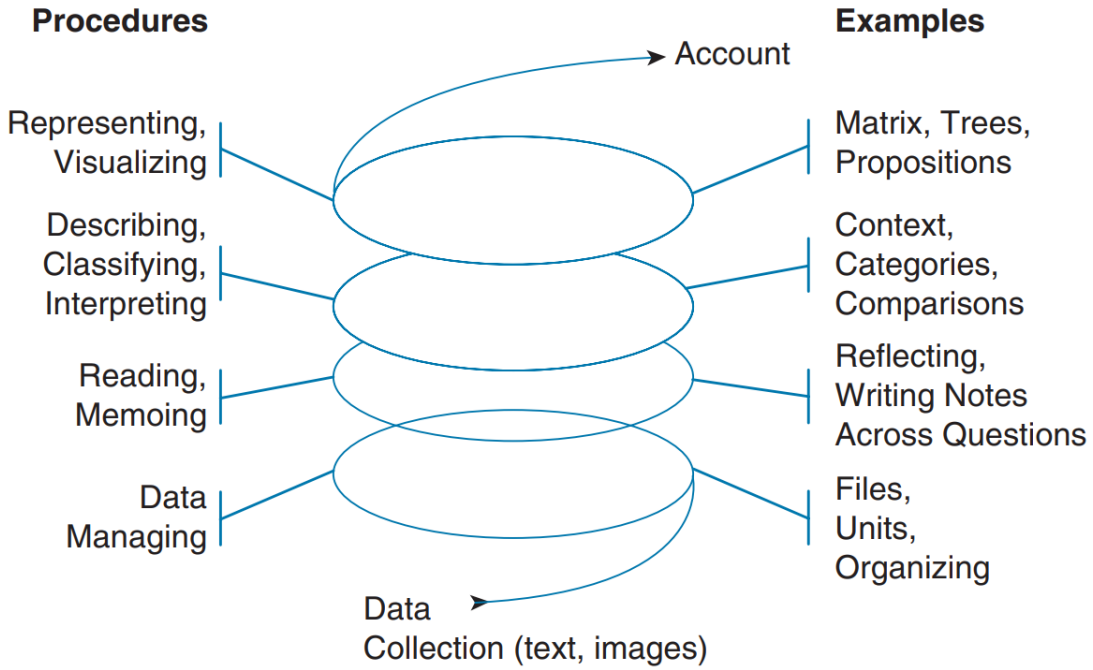

Figure 1: Data Analysis Spiral
(Copied from "Qualitative inquiry and research design", by Creswell, 2013, p.183)

To describe a complex phenomenon the data collected through various methods needs to be organized, structured, and analyzed. After collecting and organizing the data the researchers starts reading through the data and looking for initial categories (Creswell, 2013, pp. 182–188).

Memoing assist the researchers in connecting raw collected data to the abstract concepts, which explain the data (Birks, Chapman, & Francis, 2008). Memoing is the procedure of creating notes, which capture the researcher's thoughts and decisions during the analysis process. Birks et al. listed multiple ways memos support the QDA process (2008). Firstly, memos document the decision making of the researcher to make their conclusions transparent. Researchers also use memos to "articulate, explore, contemplate and challenge their interpretations when examining data" and thereby help the researcher on the path from "the concrete to the conceptual" (Birks et al., 2008, pp. 70–71). Miles and Huberman declare memos to be "one of the most useful and powerful sense-making tools" of QDA (2009, p. 72). For them it only comes secondary to communicate the results to other people, for example researchers or stakeholders in the study. This makes memoing an integral part of the analysis process and more than a tool for just capturing ideas (Corbin & Strauss, 1990, p. 10).

The next step in the spiral is to describe, classify and interpret the data. This is done by using ,the “heart of [QDA]”, the coding process (Creswell, 2013, p. 184). When discovering the same concept in different data sections the researcher emphasizes their relationship by tagging them with the same code. Using the coding process qualitative data gets iteratively sorted into codes, by looking for similarities and differences in the data (Walker & Myrick, 2006, p. 549). Then multiple of codes can be clustered together under a new code, these categories are not directly associated with data sections (Graue, 2015, p. 10). Finally, themes, which are more abstract concepts describing the data, can be formed from categories and concepts. Using the coding process researchers gradually discover concepts, categories, or themes in the data and link them to codes (Austin & Sutton, 2014, p. 439). The analyzed themes try to summarize the qualitative data without losing important aspects of their origin (Seers, 2012). The researcher interprets the data by creating themes and organizing them into larger interpretations (Creswell, 2013, p. 187). During the analysis process the number of codes grow and codes change. To record the current state of the codes a codebook can be used. The codebook lists all the created codes and further properties like a description of the code; conditions on when to or when not to use a code; examples of data that best represent the code (Saldaña, 2013, pp. 24–25).
The last circle of the spiral is the representation and visualization of the data. As this thesis is about visualizing analyzed results, this topic is covered in Chapter 3.2 about related work.

Due to the subjectivity in QDA it might be especially important to validate results to ensure the rigor of the research. There exist different strategies to validate and ensure credibility in the analyzed results (Creswell, 2013, pp. 250–253). One of these techniques to ensure rigor is the audit of the decision trail. The whole process of QDA is made transparent by documenting everything starting from the sources of data and the methods of collecting them, to the decision-making process behind the interpretations made by the researcher (Long & Johnson, 2000, p. 35). To retrace and understand the conclusions made by the researcher it is especially important to document in QDA.

All the QDA processes could be done on paper, however computer programs can support the researcher in various ways. A tool supports the time-consuming documenting process necessary in QDA. Additionally, software helps with organizing and locating data; it is easier to restructure and change codes; software allows the user to browse through the data and following associations to related data (Creswell, 2013, pp. 201–202). For this thesis the usage of software enables us to create interactive visualizations. Visualizations can be created by the software, and the user can interact with the visualization by changing the design or manipulating the data the visualization is based on. Though software tools can also have disadvantages. Researchers need to learn the correct usage of a new tool and more importantly they are restricted to the features and capabilities of the software.

2.2 QDAcity

QDAcity[3] is a single-page web application. It is a tool supporting the QDAcity-RE method but can be used to generally assist with QDA.
Figure 2 shows the dashboard of a project in QDAcity. The project dashboard gives an overview over the selected project. A QDAcity user can create multiple projects and invite other QDAcity users to work on them collaboratively. From the dashboard the coding editor of the project can be reached, there the user can upload documents and start coding. The codes, documents and all the other data in QDAcity is backed up in the cloud.
Chapter 2.2.1 will first give a general explanation of the QDAcity-RE method, on which QDAcity is based on. Then chapter 2.2.2 and chapter 2.2.3 take a closer look at the concepts code system and code system language, how they are defined in the method and how they are implemented in QDAcity.

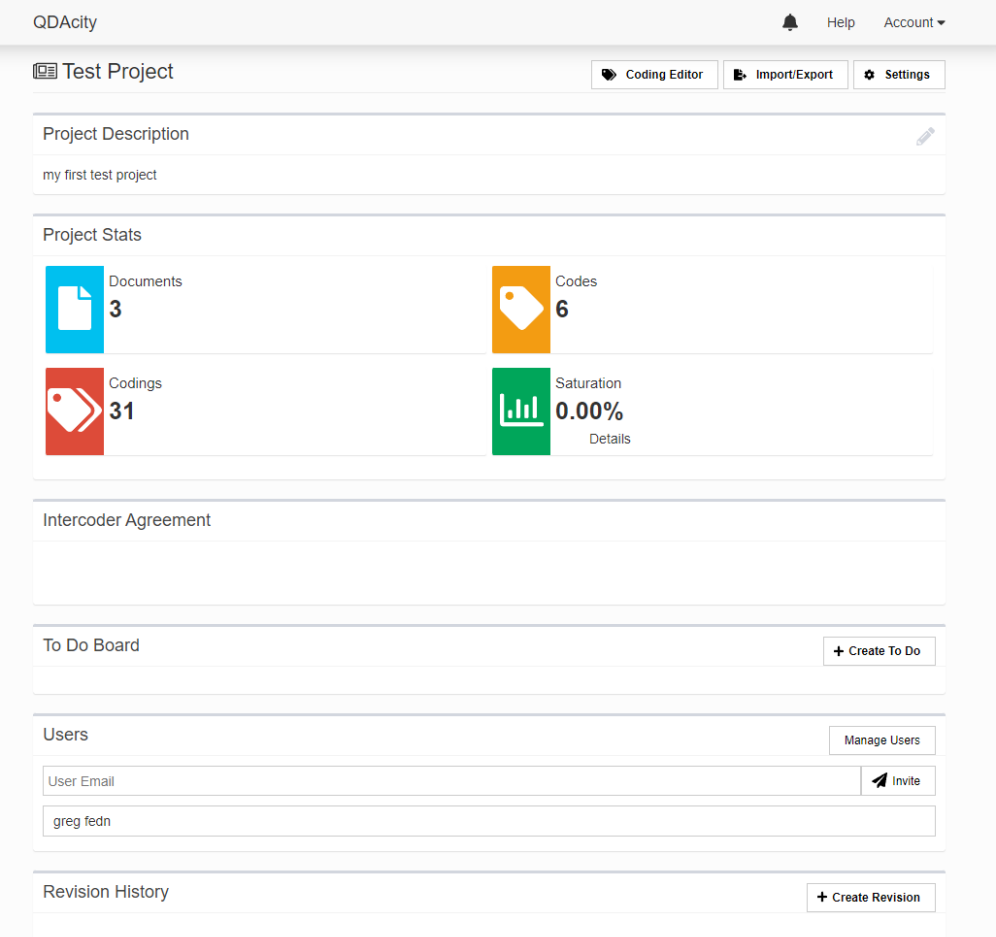

Figure 2: Project Dashboard of QDAcity

[3] https://qdacity.com/

2.2.1 **QDAcity-RE Method**

In requirements engineering (RE) the creation of domain models is highly dependent on the experience of the analyst (Kaufmann, 2021, pp. 1–2). During domain analysis, data collected from different stakeholders needs to be conceptualized to create a model, of the domain the system will operate in. The domain model is used to get more insight about the system, and it communicates the analyzed results to the stakeholders. Consequently, domain models should represent the domain correctly and be comprehensible.

The analysis of domain models struggles with problems, which are similar to the ones QDA tries to solve (Kaufmann, 2021, pp. 2–4). QDA methods try to extract relevant information out of large amounts of unstructured qualitative data, then interpret the data, and finally form abstract concepts, on which the theory is based. Therefore, the QDAcity-RE method adapts an approach to QDA to improve the RE process of domain modeling.

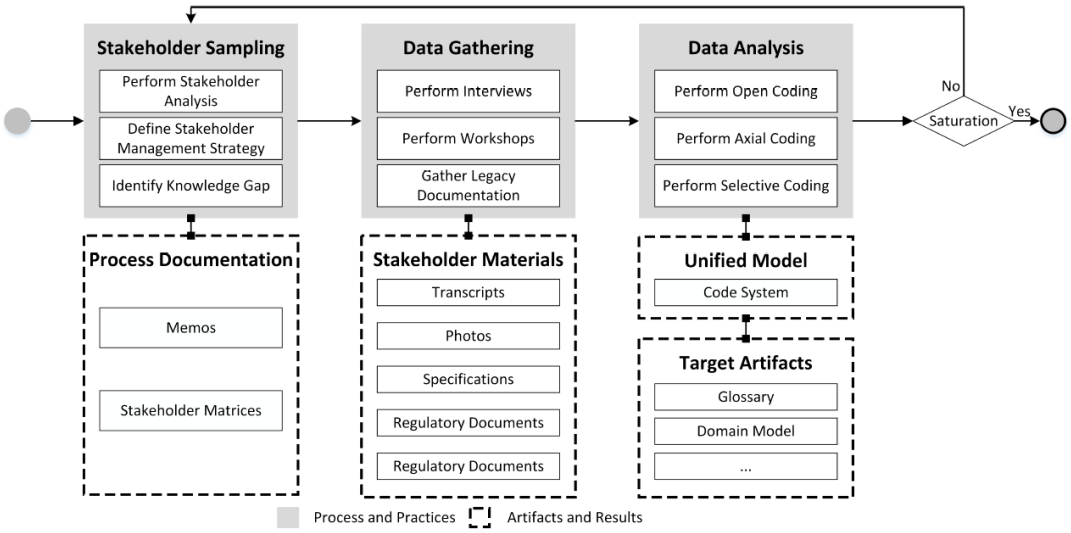

Figure 3: The QDAcity-RE Process for Structural Domain Modeling
(Copied from "Domain Modeling Using Qualitative Data Analysis", by Kaufmann, 2021, p.59)

In Figure 3 the three steps making up the QDAcity-RE method can be seen. At the start of the process a broad interview guideline gets created (Kaufmann, 2021, pp. 57–73). Then in iterative cycles the process passes through its steps till a certain saturation is reached. The saturation is reached when another cycle wouldn't add any significant information. In the first step of a cycle stakeholders get sampled. The stakeholders who best can discuss the questions of the current interview guideline are selected. After that the stakeholders' knowledge can be gathered through interviews. Then the collected data gets analyzed by utilizing the coding process; a general approach to coding was explained in the Chapter 2.1.1 about QDA. The coding guidelines change as well during every cycle to fill in gaps and correct inconsistencies.

The main artifact resulting from this iterative refinement is the code system. It is used as a model to bridge the gap between the data in natural language gathered from stakeholders and more structured artifacts, like for example the domain model. The following subchapters will expand further on the code system.

The time needed for carrying out of the QDAcity-RE method was significantly increased by doing it manually and not having software supporting it. The development of QDAcity was motivated to create a software tool, which helps with the process and documentation of the QDAcity-RE method.

2.2.2 **The Code System**

The code system is the main artifact of QDAcity, it can be created by the QDAcity-RE method, however the user can also choose to use another coding approach. The goal of the QDAcity-RE method is to act as a unified model, a common denominator between models (Kaufmann, 2021, pp. 59–61). This allows other models, like for example a conceptual domain model, to be derived from it. Depending on the usage of QDAcity, the code system can also act like most code systems found in other CAQDAS. The feature of QDAcity, facilitating the derivation of other models and thus differentiating QDAcity from other tools, is the code system language (CSL), which will be explained in the next subchapter.

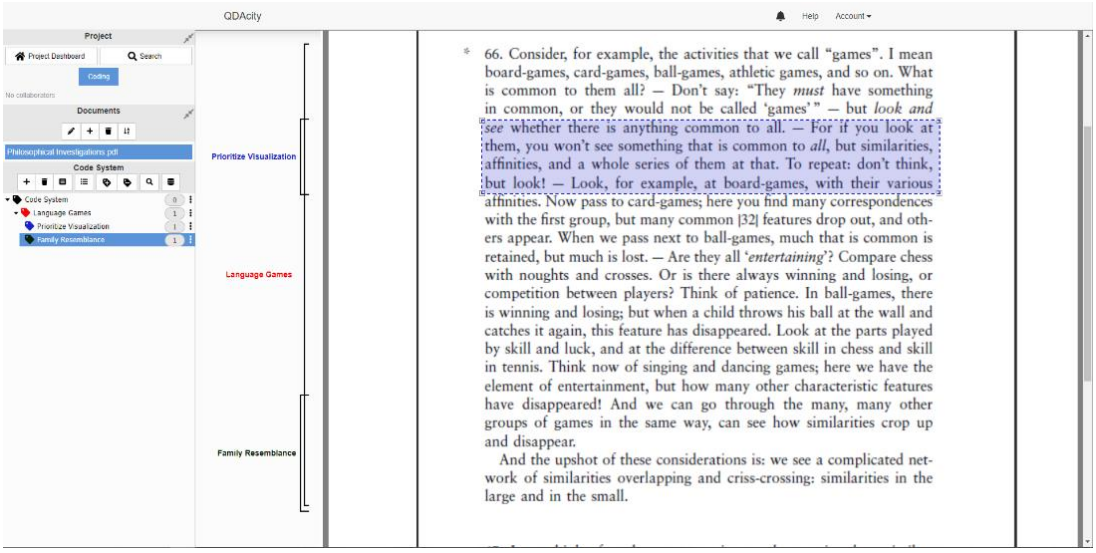

Figure 4: The Coding Editor is open, and one PDF file is uploaded.
The blue section of the text is assigned to the code 'Prioritize Visualizations'.
Two other codes are assigned to the text overlapping each other.

To start creating a code system in QDAcity, it is first needed to create a new project and then go to the Coding Editor of the project. In Figure 4 the Coding Editor is open, a document was already uploaded, and insightful sections of the document were tagged with codes. The codes can be found on the left-hand side sorted into the hierarchical code system of QDAcity.

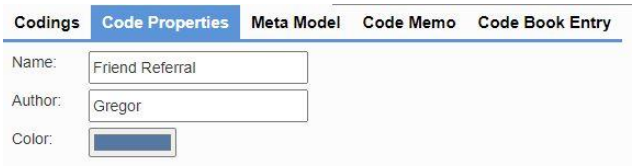

Figure 5: The Code Properties menu can be opened in the Coding Editor. These are the properties of the code 'Friend Referral'. Right now, the tab 'Code Properties' is selected.

The codes of QDAcity can have multiple properties, in Figure 5 the properties of one code are opened. The first tab "Codings" lists all the text sections associated with the code. The tab "Code Properties" specifies the name, author, and an assigned color of the code. The next tab "Meta Model" is connected to the CSL explained in the next subchapter. In the "Code Memo" tab the user can capture the meaning of the code and his decision process behind creating the code. In Figure 6 the last tab "Code Book Entry" is selected, it has, like its name implies, some typical code book properties.

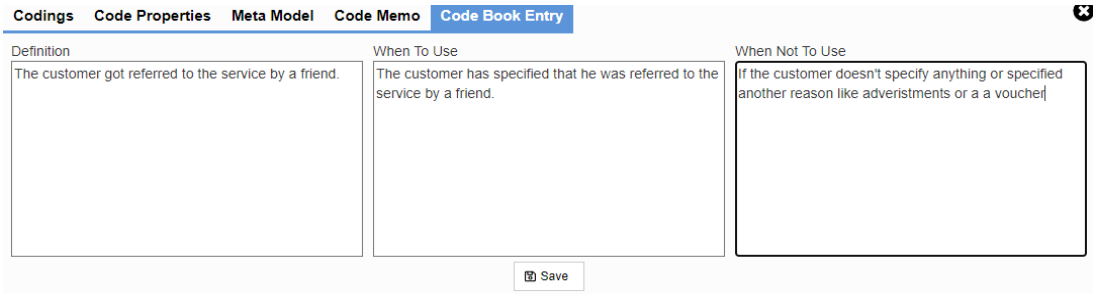

Figure 6: The tab 'Code Book Entry' of the code 'Friend Referral' is selected.
It includes the definition of the code and directions on when to apply the code.

2.2.3 Code System Language and the Meta Model

The code system language (CSL) structures the coding process by restricting how codes can be applied and structured (Kaufmann, 2021, pp. 68–73). Creating a coding framework was one goal of the CSL, another one was to support the vision of the code system being a unified model. A unified model makes it possible to derive other models from it and might enable more challenging goals, like using multiple independent QDAcity projects to come to conclusions, which are out of the scope of the individual projects alone.
The CSL describes a meta model, which was designed to derive structural and certain types of behavioral models from it (Salow, 2016, p. 9). Which means to fulfil its goal of being a unified model, the meta model still might need to be adapted. Another vision of the meta model is that it can be extended by researchers to represent their domain and making it possible to share their domain specific meta model extension with colleagues (Kaufmann, 2021, p. 71).
This thesis implements an interactive visualization of the current meta model of QDAcity.

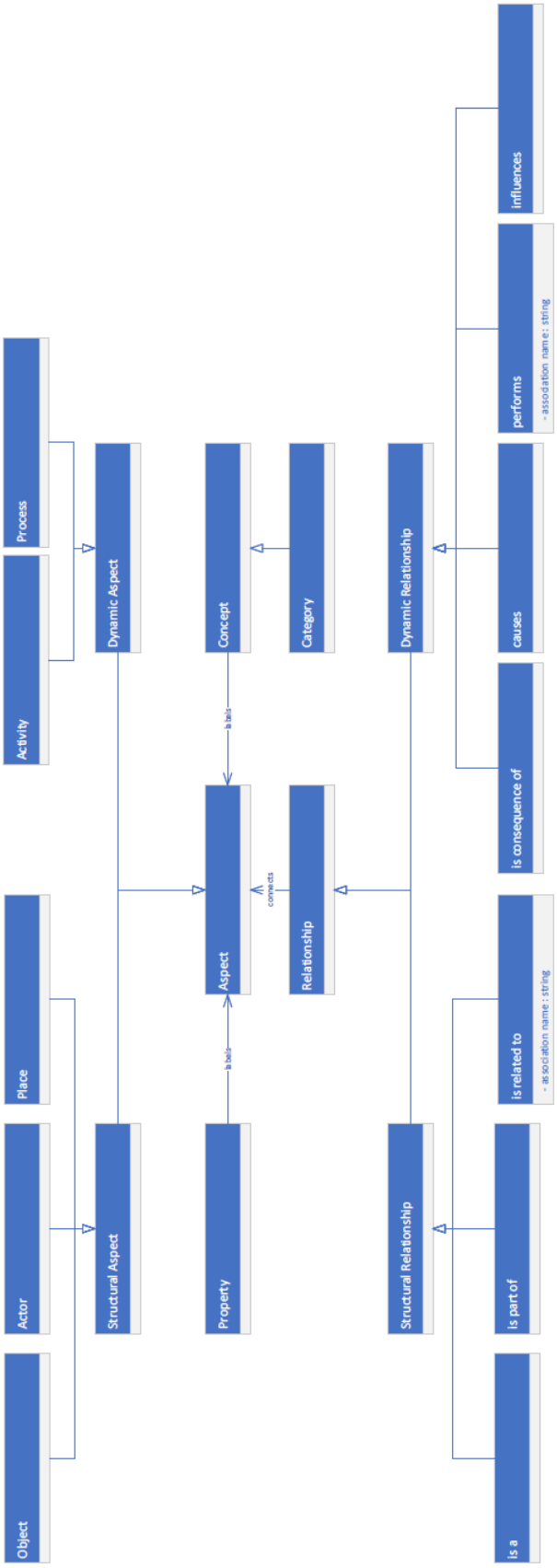

Figure 7: The Meta Model of QDAcity
(Copied from "Domain Modeling Using Qualitative Data Analysis", by Kaufmann, 2021, p.70)

The meta model is displayed in Figure 7. The elements of the meta model give the user of QDAcity the possibility to express and structure their codes more detailed than in most QDA tools. The meta model consists of two main groups, codes can either be Aspects or Relationships (Kaufmann, 2021, p. 71).
Codes labeled as an Aspect can be seen as the "common" codes created during any coding approach. They can be further classified as a Structural Aspect (Object, Actor, Place) or as a Dynamic Aspect (Activity, Process). Aspect codes can additionally be classified by a Label element of the meta model which is either a Property, Concept or Category.
In other tools memos are often the only way of expressing more complicated relationships between codes (Kaufmann, 2021, p. 68). In QDAcity relationships between codes can be Relationship codes themselves (Kaufmann, 2021, p. 71). This makes it possible to associate text sections with relationships. Relationships codes can either be structural (is a, is part of, is related to) or dynamic (is consequence of, causes, performs, influences).

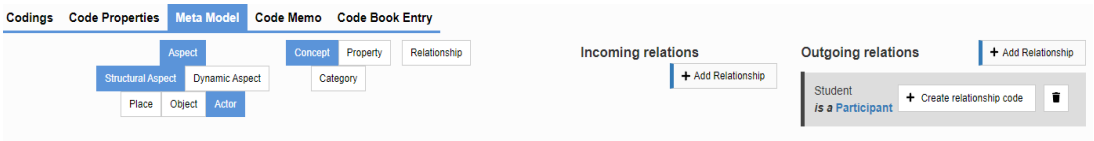

Figure 8: The Meta Model Tab in the Code Properties Menu.
On the left-side the elements of the meta model associated with the code are chosen and on the right-side relationships between codes can be created

As mentioned in the previous chapter in the Code Properties Menu of a code there is a tab called "Meta Model'. Figure 8 shows the contents of this tab for an example code. In this tab a code can be associated with elements of the meta model. Additionally, relationships between codes can be created, which aren't codes themselves. Relationships can either be codes themselves or are just defined as relations between codes without being a code. In the first case text sections can be tagged with the relationship code. Both relationship types have a meta model element associated with them.

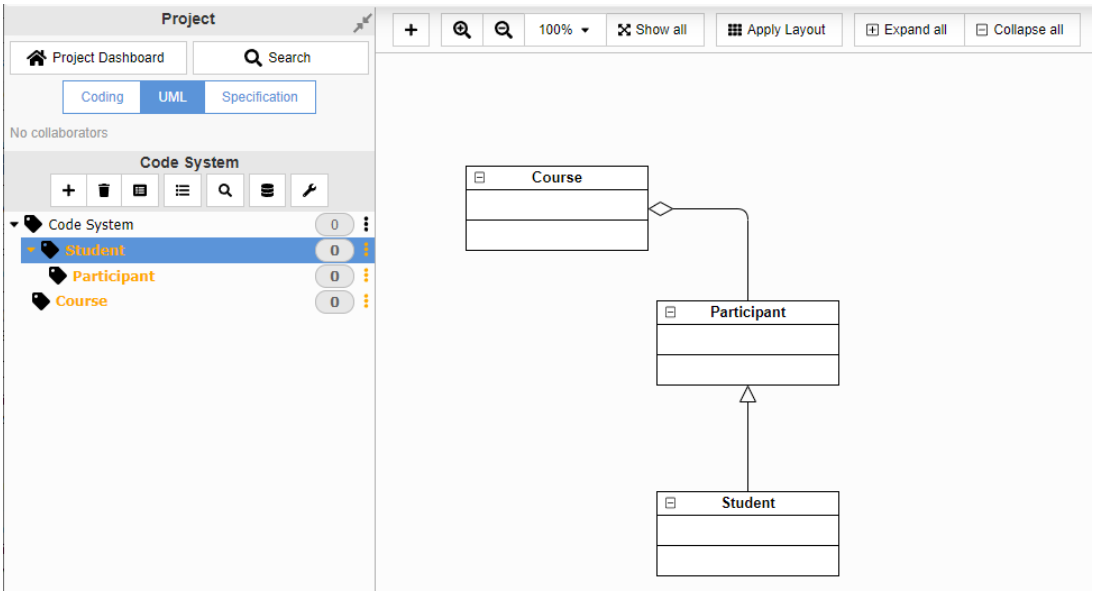

Figure 9: Coding Editor with the UML Editor View selected.
The different views can be selected at the top left of the Coding Editor, right now "UML" is selected.

In the Project Dashboard of a QDAcity project views can be selected, which can then be chosen in the Coding Editor. In Figure 9 the Coding Editor is open, and three views can be selected, with the UML Editor being the current view.
The UML editor uses a mapping developed by Salow, which allows to derive a UML class model from the Meta Model (2016, p. 21). The editor was implemented in QDAcity by Loos and had some similar goals as this thesis (2017). Some of the goals were to visualize the mapping from Meta Model to UML model and that updates of the code system change the visualization (Loos, 2017, pp. 13–14). The UML editor however doesn't want to visualize the totality of the Meta Model; therefore, it isn't possible to create, manipulate or visualize all the Meta Model's elements.
In this thesis we want to create an interactive visualization of the whole Meta Model, to allow the user to overview and manipulate every aspect of it.

3 Related Work

This chapter first lists possible benefits of a visualization, then it presents different visualization methods and lastly some guidelines for interactive visualizations are explored. Card, Mackinlay and Shneiderman define *information visualization* as "[t]he use of computer-supported, interactive, visual representations of data to amplify cognition" (2007, p. 7) This might be a computer-centric view on visualization; however, it fits our thesis perfectly.

3.1 General Benefits of a Visualization

In this chapter we will look at general benefits of visualizations and how the visualization of the meta model would benefit QDAcity.

In general, it can be said that information visualization can amplify cognition in several ways, like enhancing the recognition of patterns, encoding info in a manipulable medium and reducing search for information (Kerren, Stasko, Fekete, & North, 2008, pp. 1–6). But quantifying the exact benefits of an information visualization system is a hard task. In neurophysiological research the biological foundations of visual thinking are little developed and there is also no concrete model what happens when people think in terms of mental images. However there exists several theories how visualizing supports humans with the processing of information. Visualizations give humans another way of understanding and reasoning about problems, than if they would just use numbers and words (Swedberg, 2016, pp. 254–255). In a review of multiple studies Vekiri presents three different theories on how visualizations could benefit the processing of information (2002, pp. 300–302). The theories' focuses are on different aspects of processing visualizations but are not in conflict with each other. One theory focuses on the claim that processing graphics is less demanding than text and graphics are therefore more effective in communicating complex content. The other two theories argue that the positive effects of visualizations come from visual information being represented separately from verbal information. Card et al. list a couple of ways how visualizations can amplify cognition (2007, pp. 15–17). Firstly, visualizations can increase the memory and processing resources of the user, by storing information externally in the visualization without the user having to hold them in working memory. Secondly, they can speed

up searches by grouping and visually relating information. Patterns in the data can get recognized through clustering or common visual properties. Lastly, visualizations enable the user to make inferences, as the user can monitor and manipulate the visualization.
Henderson and Segal also argue that visualizations can reduce information, provide structure, and might help to get to new levels of understanding (2013). However, they also warn about oversimplifying or misrepresenting the data and advise on educating people on how to interpret visualizations.

There isn't much research done on visualizing code systems and the benefits of code system visualizations. Assumed benefits of the meta model visualization can be explained by the general benefits of visualizations. Multiple possible ways the meta model visualization could support QDAcity can be discerned.

The first obvious benefit could be for the user to have a different way of interacting with the meta model. In QDAcity the CSL can only get manipulated through opening the menu of each code. Overviewing the meta model in its totality, might make it easier to grasp and manipulate it. Interacting with the code system through a visualization sounds far more accessible than searching through the hierarchical structure of the code system and looking up their CSL properties one by one.

A researcher using QDAcity could use the meta model visualization to discover new insights. Through a literature review Yi et al. identified four possible processes how people can get insights through InfoVis systems (2008). The four processes are *Provide Overview*, *Adjust, Detect Pattern* and *Match Mental Model*. The user could get new insights by overviewing an InfoVis system and paying more attention to areas he doesn't yet fully understand. A visualization could help with gaining insights by making it possible for the user to adjust the level of abstraction and/or the range of selection, which makes it easier to filter out uninteresting parts. The user could detect patterns in the visualization by purposefully searching for them or discovering them passively. Lastly, visualizations can reduce the difference between the own mental model and the data. Those processes not only show how people possibly create new insights using InfoVis, but they could also be used to create guidelines for designing InfoVis systems.

Visualizations can be used "to describe, document, and share complex data in a clear and precise manner" (Pokorny et al., 2018, pp. 170–171). Another way to utilize this is to communicate the results or the analytic process of QDA to other people like the researcher's colleagues or stakeholders of the study. A visual overview of the code system could facilitate communication by picturing the whole code system or focusing on specific parts. To communicate the code system and the associated meta model it makes sense to present the codes and their relationships visually and not as a list.
A goal of QDAcity was to support the documentation of the QDAcity-RE method. To retrace or audit the decision trail of a researcher it would be nice to have an overview over the development of the code system over time. The meta model visualization could be captured and saved at different stages of the QDA process to help with the transparency of the research.

3.2 Visualization Methods in QDA

Visualizations of qualitative data can be used at various stages of qualitative research, for example maps can be used to gather data from research participants (Wheeldon & Faubert, 2009). Henderson & Segal roughly classify visualizations by the segments they break their data down into (2013). In textual qualitative data visualizations could display words, sentences, themes, or whole narratives. After collecting data, illustrations based on words or sentences like word clouds, phrase nets and word trees can be used. All of them don't require an analysis process beforehand.

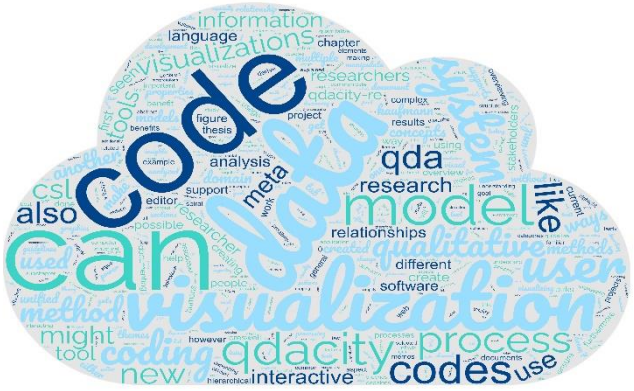

Figure 10: Word Cloud
A world cloud of the previous chapters of this thesis.
(Created with https://www.wortwolken.com/)

Maps, spectrum displays, sentiment analysis or matrices could be used to capture themes or narratives found in the data. It is easy to see that before the meta model can get visualized it is necessary to have created codes and typed them in accordance with the CSL. To present the elements of the meta model and their relationships between them, map-like visualizations present themselves offering more freedom and possibilities for the researcher to present their code system.
Matrices could also depict a network and in large dense graphs, without prior knowledges about the data set, users can be faster to find out certain information compared to network graphs (Keller, Eckert, & Clarkson, 2006). Though, it seems a matrix would be worse at making complex associations across multiple nodes and the matrix's interactivity, and customizability would be restricted compared to a network graph. ATLAS.ti[4] and MAXQDA[5] are two QDA software tools and they offer map-like code system visualizations. For their representations, ATLAS.ti uses the term "network" and MAXQDA uses the term "concept map" and "info graphic". In Chapter 5 their code system visualizations are described, and their differences highlighted. The next subchapters will look at three different map-like visualizations, which could be considered to convey the meta model: networks, concepts maps and mind maps.

[4] https://atlasti.com/
[5] https://www.maxqda.com/

3.2.1 **Networks**

A network can be generally defined as a collection of nodes, which are connected by a collection of edges (Newman, 2018, p. 1). ATLAS.ti for example, uses networks to describe their underlying code system. Figure 11 shows their networks visualization, which depicts among other things codes and the semantic relationships between them.

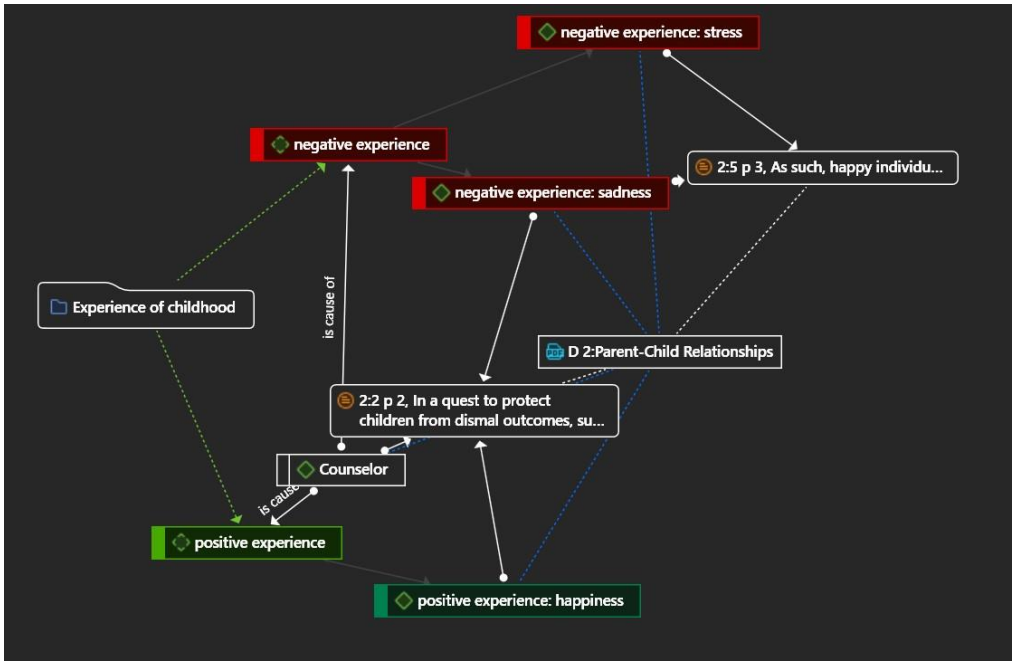

Figure 11: Example Network in ATLAS.ti.
All codes are marked by the green rhombus icon.
Semantic relationships between codes can be recognized by the lables on top of the edges, for example betweeen the codes “positive experience” and “Counselor”.

In QDA networks could not only be used as interactive code system visualizations, but they could also be analyzed to discover information about the network and specific parts. In the paper “Network Analysis for the Visualization and Analysis of Qualitative Data” (2018), Pokorny et al. developed a reproducible method to create a network out of already created codes. The network should display the interrelations of codes, and network analysis methods can be used to gain information about its content. In their method the codes are displayed as the nodes and the edges between them represent the chronological location of the codes in the data. They visualize the chronological location as follows, when codes overlap in the data, they have a bidirectional edge connecting them and if they’re just located next to each other, they share a directional edge. Edge weights were increased if the same code pairs were repeated. Then in their example they sized the nodes depending on the number of edges connected to them and their corresponding edge weight. The nodes are differently colored depending on their associated node cluster. A node cluster can be determined how well nodes are connected among each other. Figure 12 displays one of the example networks they created. But the approach of Pokorny et al. to create edges is data-driven, they are constructed using predefined criteria and not like in

QDAcity's meta model, in which the relationships are defined by the researchers. There is a good chance QDAcity users working on the same data will analyze different relationships. If we would create a network out of those code systems, it would result in different networks. The resulting networks could still offer new perspectives on the data and aid assistance in the analysis process. However, another question is how network analysis could be used meaningful on these networks. Are metrics like number of nodes, graph density, degree of each node meaningful for the use case of our thesis? And which definition for metrics like node and edge weight would make sense? Since the CSL properties are set using a researcher-driven approach, the interpretation of the network's metrics should probably be in the researcher's hand as well. But it is interesting to keep in mind, how network metrics could influence the visualization of the network and how certain metrics like the weight of nodes and edges can be determined.

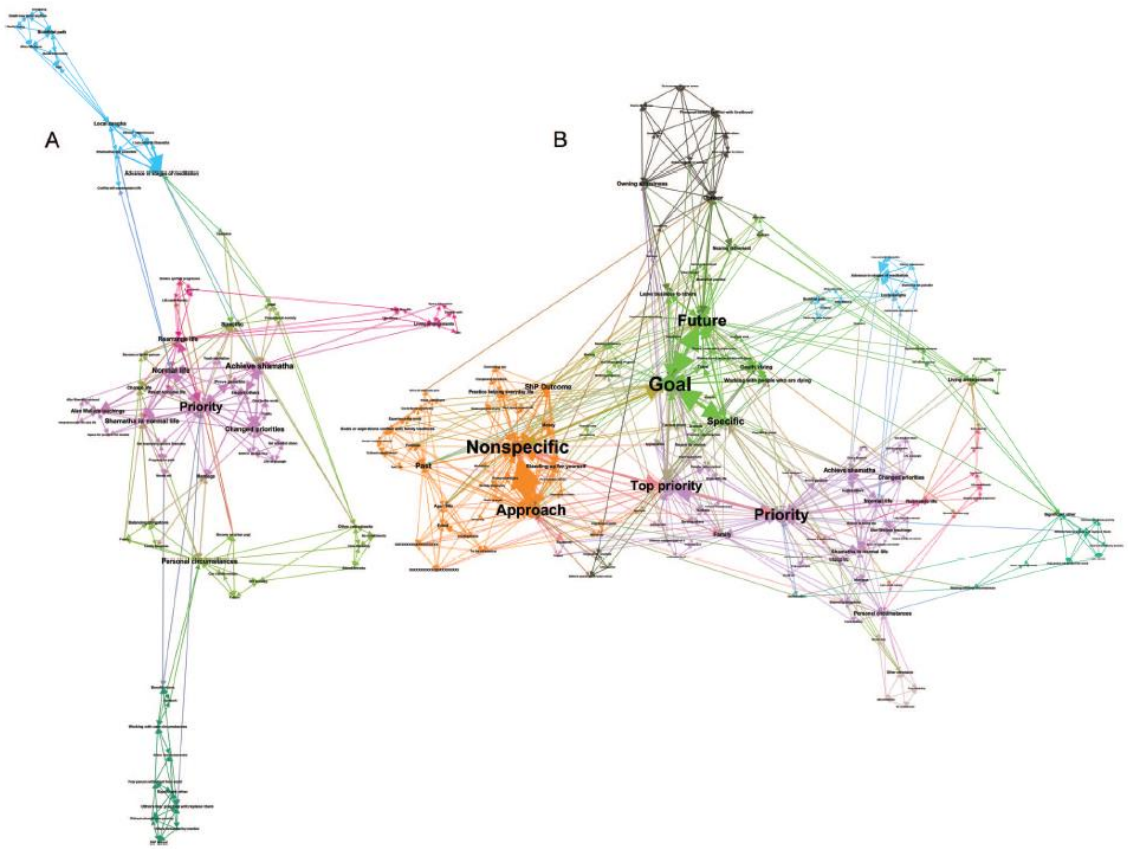

Figure 12: (A) Codes created from one participants data visualized as a network graph
(B) Codes created from all participants data visualized as a network graph
Importance of nodes and weights are calculated by using network metrics and elements of higher importance are bigger in size. Clusters of well-connected nodes are differently colored.
(Copied from "Network Analysis for the Visualization and Analysis of Qualitative Data", by Pokorny et al., 2018, p.175)

3.2.2 **Concept Maps**

Another map-like visualization are concept maps, they are generally hierarchically structured and try to explain a concept top-down. In Figure 13 a simple concept map explaining the concept “Bread” is displayed. Concept maps start with the concept needing to be explained and then connect it with subordinate concepts and those can have subordinate concepts as well (Davies, 2011). They are generally more structured and more formal than mind maps and don’t embed pictures or much text. Concept maps convey the main concepts, the relationships between them and the domain to which they belong. Concept maps are often used to elicit and visualize knowledge (Coffey, Hoffman, & Cañas, 2006). Daley argues, that this visualization method is useful for framing research, reducing data, discovering and analyzing themes (2004). The CSL in QDAcity doesn’t necessarily structure the codes in a hierarchical way, so the code system wouldn’t always follow this strict definition. However, non-hierarchical concept maps exist as well (Davies, 2011). For example, MAXQDA classifies their visualization as a concept map and their preset templates often have only one of the project’s elements in their focus (Verbi GmbH, 2022, p. 471). Having said this, some templates analyze the context of multiple elements, and their visualization can always be used freely, meaning the user can decide the structure.

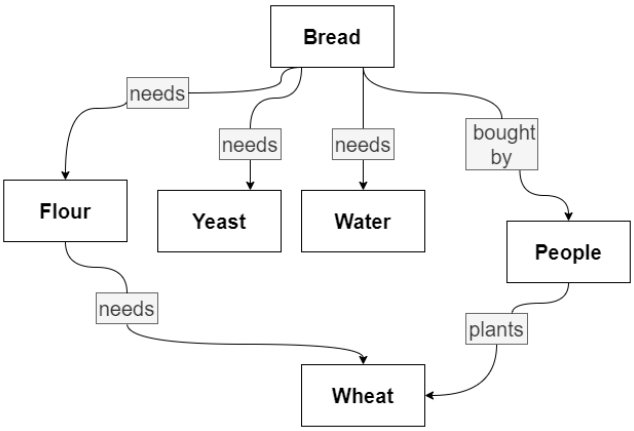

Figure 13: A simple hierarchical concept map.
The concept “bread” is at the top of the hierarchy
and gets explained through its relationships with other concepts.
Created with https://app.diagrams.net/

3.2.3 **Mind Maps**

Mind maps are similar to concept maps, in as far as they also focus on one concept and visualize a network out of connected concepts or rather ideas, but their form is less structured. Images, symbols, diagrams, or other objects can be used within the map and a concept can be connected to any other concept (Davies, 2011). Mind maps can be used as a tool for brainstorming and idea generation and benefit from similar advantages as the other map-like visualizations (Lin & Faste, 2011). In QDA they can be used to memo, develop ideas or visualize associations between codes and data sources (R. Mammen & R. Mammen, 2018, p. 2). A disadvantage of mind maps is that they use simple associations and not complex relationships with labels (Davies, 2011, p. 282). This makes the typical mind map bad at visualizing the meta model's complex relationships.

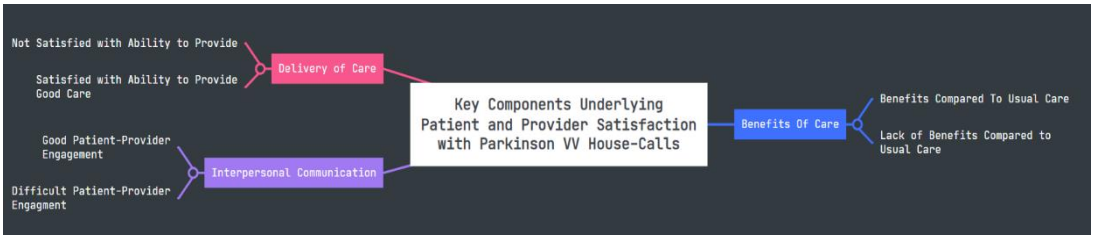

Figure 14: Example Mind Map
(Based on "Beyond concept analysis: Uses of mind mapping software for visual representation, management, and analysis of diverse digital data" by R. Mammen & R. Mammen, p.8;
Created with www.mindmeister.com/)

3.2.4 **Discussion**

There is not much research on how to visualize a code system. The above presented visualizations could be adapted to depict the meta model. Map-likes visualizations allow to portray objects and their relations and give their creator options to express qualitative ideas in their design. Visualization methods will influence the visualization's features. Some methods fit better to our thesis' goal. Chapter 5 looks at the visualization methods of other CAQDAS and then Chapter 7 justifies the decision on a visualization method.

3.3 Guidelines for the Meta Model Visualization

Munzner presents a model for visualization design, which divides the process into four levels (2009). The levels of the model are depicted in Figure 15, each level has their own threats to the validity of the visualization and errors on a higher-level cascade downwards.

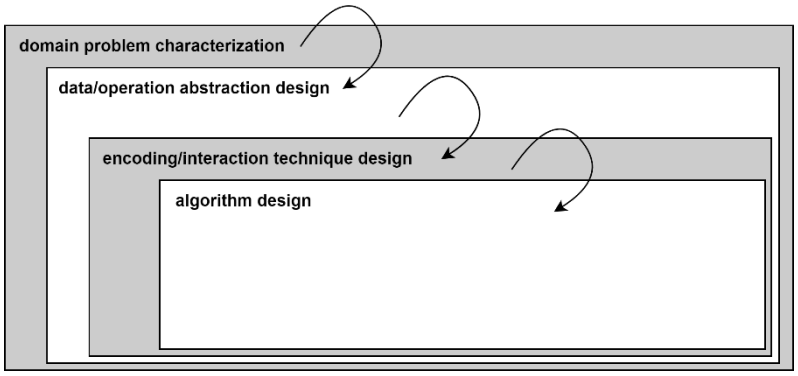

Figure 15: Nested Model of Visualization Creation
(Based on "A Nested Model for Visualization Design and Validation" by Tamara Munzer, p.922;
Created with https://app.diagrams.net/)

On the first level, the domain needs to be understood to learn about the problems and tasks a user faces. It is important to characterize the problems correctly since the visualization's job is to aid with those. In general, it is important to ask if a visualization assists the user with certain tasks or problems. A. Carr (1999) suggests to consider visualizations if "there are large amounts of data, the user goals are not easily quantifiable, and there are no simple algorithms to accomplish the goals". In QDAcity we assume the benefit of a meta model visualization because of multiple reasons already metioned in Chapter 3.1. The user never can interact with the whole meta model but must change its properties through each code. Possible insights of the researcher through overviewing and inspecting the model are not feasible through an algorithm. Lastly communicating the code system of QDAcity to other people might be easier by using a visualization than a code list. Obviously, it would have been better to elicit the requirements of the visualization through interacting with QDAcity's users.

The second level is about mapping the data and problems of the domain to more abstract data and operations a computer could better deal with. In our case the meta model is already an abstract model and is used in QDAcity. There is no need to further transform the meta model data into another data type. Additionally, to the abstraction of data the domain-specific operations should also be described more generic and abstract. Shneiderman (1996) presents seven abstract tasks, which can be considered for the meta model visualization . The volume of codes can grow big in a qualitative research project. It is therefore necessary to have features, which support the user in exploring and getting insights about the visualization. These seven tasks are: *overview,*

zoom, filter, details-on-demand, relate, history and *extract*. In QDAcity the user should have the possibility to *overview* the entire meta model visualization and *zoom* in on certain elements of interest. It should be possible to *filter* out uninteresting elements, to enable the user to focus on interesting parts of the meta model. More detailed information about elements and groups of elements are *displayed on demand*. *Relationships* between elements should be visible. A *history* of actions should be saved to enable undo, redo. It would be useful to be able to *extract* only parts of the meta model and save it to a file. Yi et al. (2008) present some procedures, how people could gain insights from InfoVis systems. Their discovered procedures support some of those tasks. One way of benefiting the gain of insight was providing an overview over the dataset. Another one was exploring the dataset by adjusting "the level of abstraction and/or the range of selection". Selecting the range of a dataset is another way of saying, filtering the data set. Carr (1999) emphasizes that the ability of the user to navigate and zoom around in visualization doesn't replace the act of eliminating unnecessary information by filtering. The meta model visualization implements those abstract operations to support the processes of information seeking and gaining insights.

The third level now wants to find a design with the right kind of abstraction for the user of the system. A visual encoding of the meta model needs to be designed which expresses the CSL properties of the codes and their relations.

In the last level, a time and memory effective algorithm to visualize the meta model should get implemented. To not exceed the scope of a master thesis, we will look for a fitting graphic library to support our task in Chapter 6.

4 Requirements

This chapter specifies the requirements that should be met by the visualization implementation of this thesis. In Chapter 10, they are used to evaluate the thesis' results.

4.1 RQ1: Visualizing the Code System

The user can visualize all the codes of a QDAcity project.

4.1.1 RQ1.1: Visualizing the Meta Model and the CSL

If codes associated with Meta Model elements are depicted in the visualization, the user can recognize the Meta Model properties of a code. This means, every Meta Model element is depicted using different visual properties. Relationships between codes are displayed, even if the relationships aren't codes.

4.1.2 RQ1.2: Visualizing only parts of the Code System

The user can choose to visualize only certain parts of the code system by drag- and dropping selected codes into the visualization.

4.2 RQ2: Changing the Code System using the Visualization

In QDAcity the user can manipulate the code system and the meta model using the visualization.

4.2.1 RQ2.1: Creating and Deleting Codes

Codes can be created and deleted from within the visualization.

4.2.2 RQ2.2: Changing Meta Model Properties

The Meta Model properties of codes can be changed inside of the visualization and the codes within the visualization update their appearance correspondingly.

4.3 RQ3: Customization of the Visualization

The QDAcity user can customize the visualization.

4.3.1 RQ3.1: Creating Independent Nodes and Edges

Nodes and edges not representing codes can be created.

4.3.2 RQ3.2: Changing Visual Properties of Nodes and Edges

Visual properties of nodes and edges can be changed. Changing the color of a node connected to a code doesn’t change the color in the code system.

5 CAQDAS Comparison

This chapter presents the visualizations of different computer-assisted QDA software, which can be abbreviated to CAQDAS. The chapter's focus is on qualitative visualizations, mainly map-like visualizations, which depict code systems. The presented CAQDAS also offer other representations such as tables, matrices, or word clouds but they are not of interest. CAQDAS exists as web applications or downloadable programs, as open source, free or proprietary software. Even though there exists a lot of different CAQDAS, they often don't offer qualitative visualizations like maps or networks. Non-proprietary tools with relevant visualizations were not found.

CAQDAS assists researchers in sorting, structuring, and analyzing data and documenting the whole process. Going through the process of QDA and documenting it would be far more time consuming without the help of software tools. Furthermore, software helps with finding the data connected to elements of the code system, for example the memos connected to codes, themes or documents (Creswell, 2013, p. 202). It is easier for the researcher to look up and inspect certain data elements in detail. However, for our thesis the most important advantage of CAQDAS is the creation of interactive visualizations. Prior to using computers in QDA, it was possible for researchers to draw maps or networks, but it was more cumbersome. Moreover, software allows the user to interact with the visualization and it is far easier to manipulate a visualization using software, than redrawing it. When code systems get visualized in CAQDAS it is possible for the user to interact with it. A user could manipulate code properties, add, or delete codes or create new relationships between codes. CAQDAS provides the opportunity to visually represent certain codes or themes discovered in the qualitative data.

This chapter compares the qualitative visualizations of three tools. The CAQDAS ATLAS.ti and MAXQDA both offer map-like visualizations for their code systems. The visualization of Quirkos also sorts their code into a 2D canvas but doesn't have any relationships visualized as edges. We will examine how the visualization is integrated into the tool, which elements do they depict and how do they visualize those elements. Also, explore the ways the user can customize and interact with the visualization.

5.1 MAXQDA

MAXQDA[6] offers three commercial versions of their product for Windows and macOS, to analyze a wide variety of data formats including audio, video, and social media data. The versions differ in the amount of analysis methods available to the user, however all three versions offer a concept map visualization called MAXMaps. The information of this subchapter is obtained by using the trial version of MAXQDA and using the MAXQDA 2022 Manual (Verbi GmbH, 2022).

The code system of MAXQDA, seen in Figure 16, is similar to QDAcity's code system, it is hierarchically structured, meaning there is one root code, that can have subcodes, which in turn can have subcodes again. When a researcher creates a new code, he must decide where in this tree structure the code fits in.

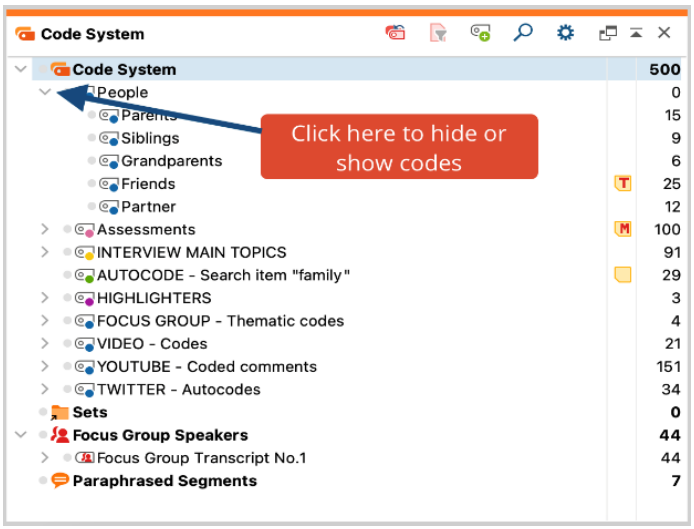

Figure 16: The code system in MAXQDA
The root code 'Code System' is at the top of the hierarchy and consists of 500 coded segments. One of its subcodes 'People' does not hide its subcodes, among them are for example the codes 'Partner', 'Friends' and 'Grandparents'.
Copied from 'MAXQDA 2022 Manual', by Verbi GmbH, 2022, p.183

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

The visualizations of MAXMaps are unrestrictive and can be customized in various ways. After importing an element of the project into the visualization it has a default appearance consisting of an icon and a label, but all its properties, like the label, icon, and color, can be changed. Elements in the map can either be actual objects from a MAXQDA project, like codes, texts, memos or documents, or elements that are not connected to any objects in the project, which can be texts, imported images or basic shapes. If an element is connected to a project object, it can only be inserted once into the map, though modifications of those objects in the map don't modify the underlying code. For example, renaming a code in the map would not rename the code in the code system. Both of those types of elements can get connected by lines to indicate a relationship, but these relationships only exist inside of a single map. Creating visualizations to represent themes can benefit a lot from the free form of MAXMaps. Using other geometric objects and pictures leaves the researcher a lot of options to visualize their analyzed concepts. Another use case of MAXMaps is to display the associations between elements of the project, for example the associated text sections of a code could be imported. Through the associations of codes with other codes, concept maps can be created. The idea of exploring the associations of certain elements of a MAXQDA project is further supported by the option to select a template, when creating a new map. For example, when the focus is on a single code, a map can be created with either all this code's subcodes, all memos, which are attached to the code, or some data segments associated with the code. Another example can be seen in Figure 17, the 'Single-Case Model' shows a document in the center of the map and visualizes all its associated codes, coded segments, and memos.

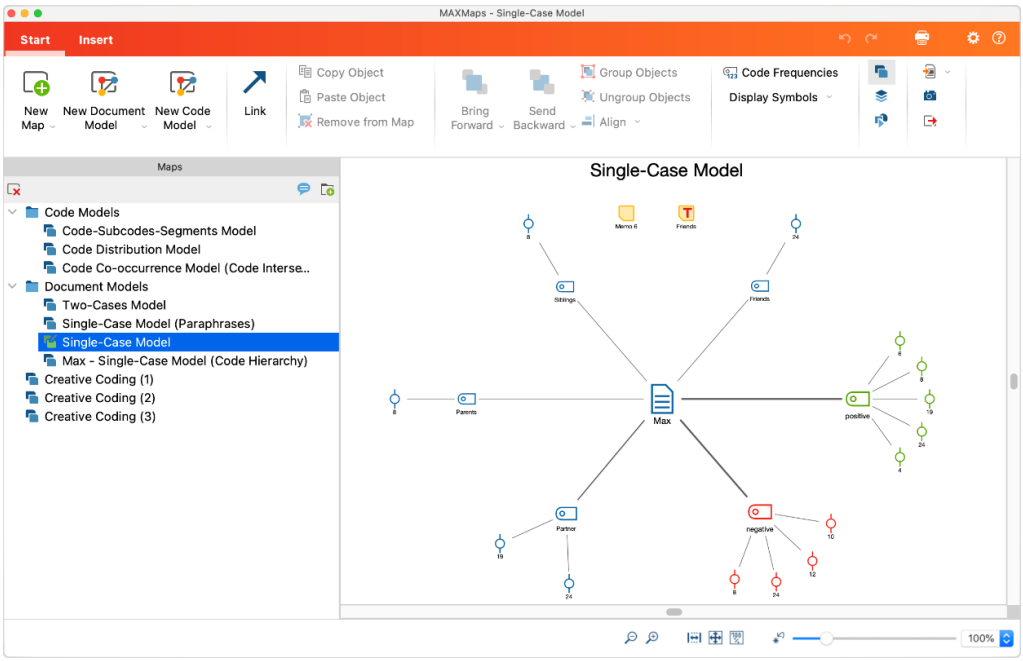

Figure 17: The MAXMaps interface
On the left side all created maps are listed. At the top, new maps can be created, elements can get inserted and manipulated and undo/redo options can be found. Located at the bottom are the zooming functions and a button to focus on the first object.
Copied from 'MAXQDA 2022 Manual', by Verbi GmbH, 2022, p.442

The interface of MAXMaps is seen in Figure 17, it has its own window, where maps can get created and managed. Link Mode can be entered by clicking on the Link icon on the top of the interface or using a keyboard shortcut. Then two elements can be connected by clicking on one and dragging a line to another one. The appearance of those lines and their labels can be changed. Some attributes like color, thickness, font size can be chosen, and the line convey a directional or bidirectional relationship.
MAXMaps supports the information seeking tasks by Shneiderman (1996). A user can create an overview by dragging all codes into the visualization and it also offers a button to fit all the elements of the visualization on the screen. Zooming and panning is possible in every map. It can be said that filtering is done by the user choosing the elements of interest. By right-clicking on elements it is possible to examine details about the selected elements in the project. The user can view all the associations of an element by right-clicking on it and selecting to import all the associations of a certain type. MAXMaps keeps a history of actions and thus supports the undoing and redoing of actions. The map gets automatically saved every minute and when the map interface window is closed. The map can be copied to the clipboard or exported in different formats, including SVG, JPG and PNG.

5.2 ATLAS.ti

ATLAS.ti[7] is proprietary software to analyze textual, audio, video, graphical, geo data and social network comments. Different versions run on Windows, macOS, iOS, Android and in the Browser. The desktop versions are mainly the same, while the Web version misses a lot of features, including their network visualization. To gather the information about ATLAS.ti, their trial version and their windows manual from 2022 was used (Dr. Susanne Friese, 2022).

Contrary to the code system in MAXQDA and QDAcity, codes in ATLAS.ti are not sorted into a hierarchical tree structure but are kept in a flat code list. Codes don't need to be sorted into a hierarchy right away, but it is still recommended to sort them into a hierarchical catalogue, when it gets hard to overview the codes in the project. This is achieved by creating folders, categories and subcodes.

Figure 18: The code system in ATLAS.ti
Two folders are at the top level of the code list.
The folder "Experience of childhood" is expanded and contains two codes, which in turn have subcodes themselves. For example, the code "positive experience" has two subcodes "happiness" and "supportiveness".

[7] https://atlasti.com/

The user can visualize data of ATLAS.ti projects as a network. Multiple networks can get created; an example network is displayed in in Figure 19. Among other things, codes, quotations, memos, other networks, and documents can be added to a network. Same as with MAXMaps the networks can be used to explore a research project, discover new insights, and visualize findings. Contrary to MAXMaps, actions like renaming, changing the color, or changing the comment of a code, changes those attributes in the project. The networks always visualize the underlying project, if one changes anything in a network the changes apply to the whole project and thus other networks as well. Another difference is that they define two kinds of relationships in their networks, first-order and second-order relationships. Second-order relationships are associations between objects and have no semantic intention, for example a code is linked to its quotations and the quotations can be linked to their source documents. The concept of first-order relationships does not exist like this in MAXQDA. First-order relationships are semantic relationships between codes or between quotations. When created they are also then created for the project and could appear in other networks, if for example the related codes appear in there.

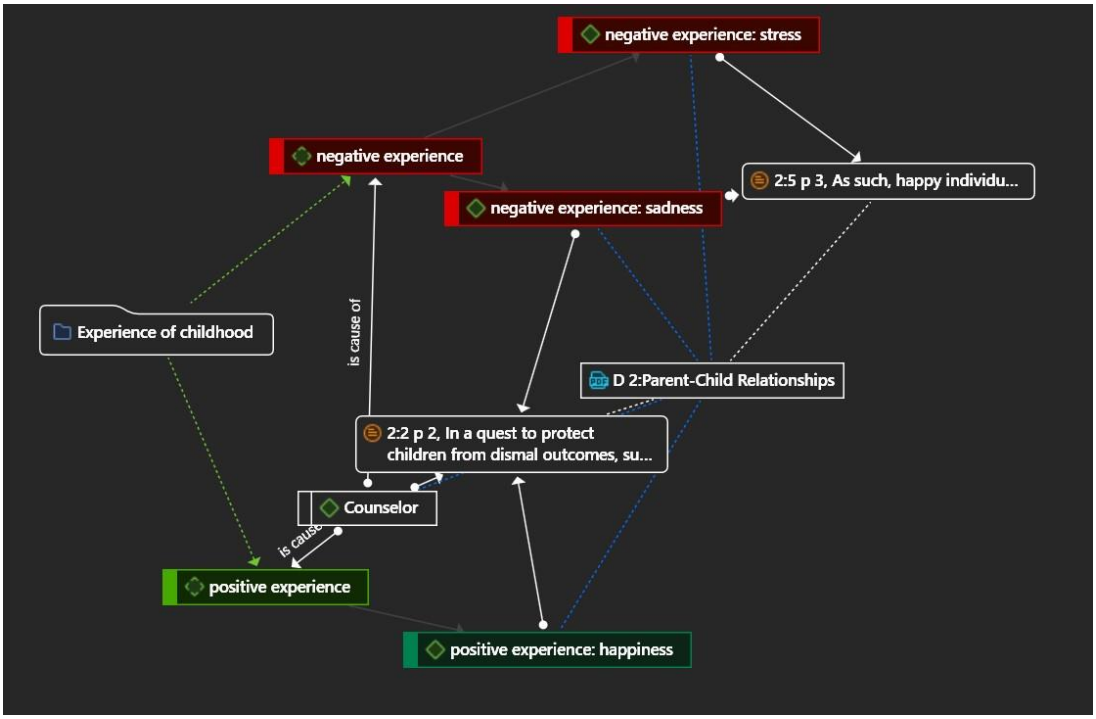

Figure 19: A network in ATLAS.ti
The network consists of a code group, colored codes, quotations and one document.
The links display the associations or semantic relationships between codes.

Nodes can get connected to each other by dragging a line form a certain spot of a node to another node or like in MAXQDA by clicking on a button and then clicking on the nodes, which should be connected. When connecting two codes the user must decide on the relationship he wants to use. The default relationships can be seen in Figure 20. Additionally, one can create his own relationships, then properties like a name and color must be chosen and if the relationship is symmetric or asymmetric.

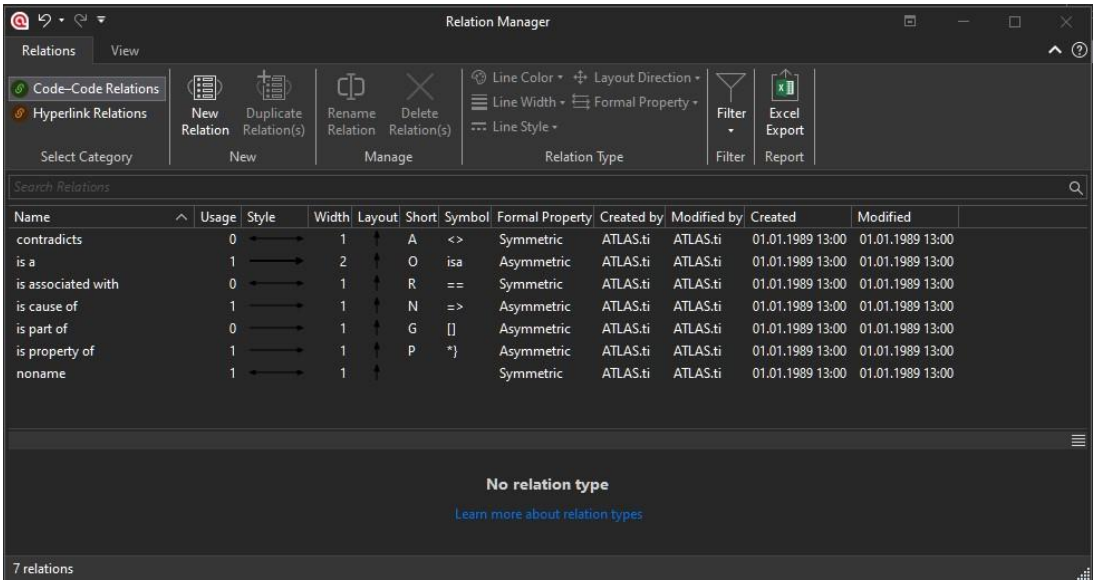

Figure 20: The relation manager with its default relations
The default relations are listed here, they could be changed, or new relations could be created.

Like MAXMaps, ATLAS.ti implemented the information seeking tasks defined by Shneiderman (1996). The data of a project can get overviewed by adding all the codes into a network. In the networks the user can zoom and pan around to focus on certain parts. Only certain parts of the project can get displayed, by choosing which ones to add to a networks. Details of certain elements can be obtained by clicking on them. All relationships between elements are portrayed in networks. Actions can be undone and redone and the network can be exported to file formats including PNG, JPG and BMP.

5.3 Quirkos

Quirkos[8] offers a version for Windows, macOS, Linux and a version running on the internet. Quirkos exclusively supports the qualitative analysis of textual data.

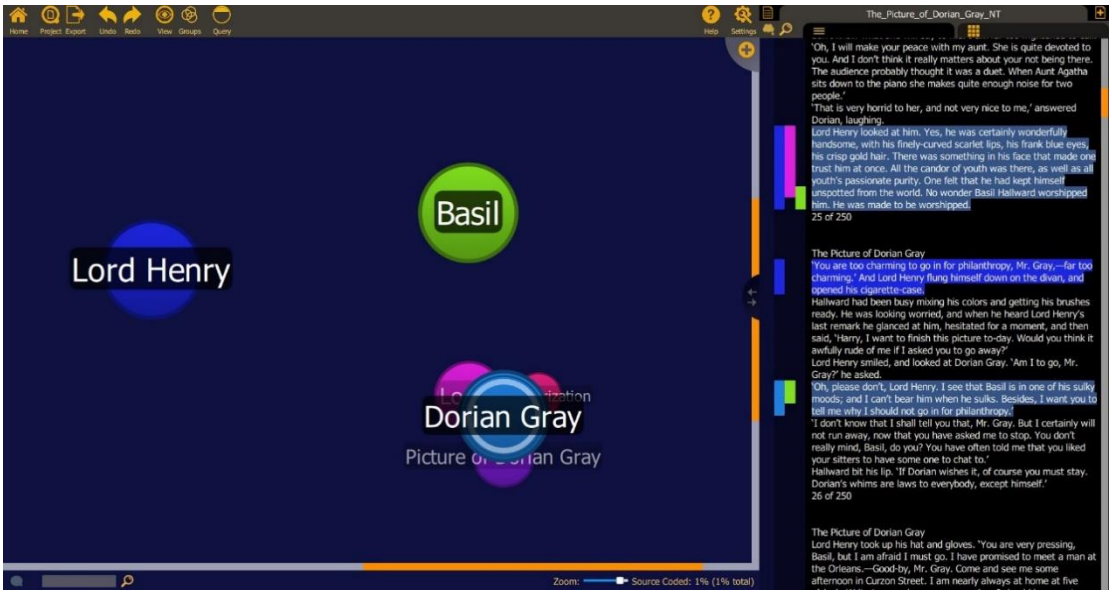

Figure 21 The coding editor of Quirkos.
The visualization of the codes is on the middle-left side.
On the right side is the text annotated with codes.

It differs from the other two tools since the visualization is incorporated during the coding process. The visualization in Quirkos is next to the text, as can be seen in Figure 21. Codes can be created by clicking on a button or opening a context menu on the canvas. Once a code is created, text sections can be dragged onto them, to annotate those sections with the code. The more annotations the code has the bigger it grows.
Codes can be dragged onto each other to group them together. Hovering over a node shows the ratio of the code's annotations to total annotations. A code can be edited by right-clicking on them and choosing 'edit'. In Figure 22 a code's properties menu is shown, in there the code's title, description and color can be changed and the code can get associated with non-hierarchical groups.
Their visualization also doesn't have the possibility to create semantic relationships between codes.

[8] https://www.quirkos.com/

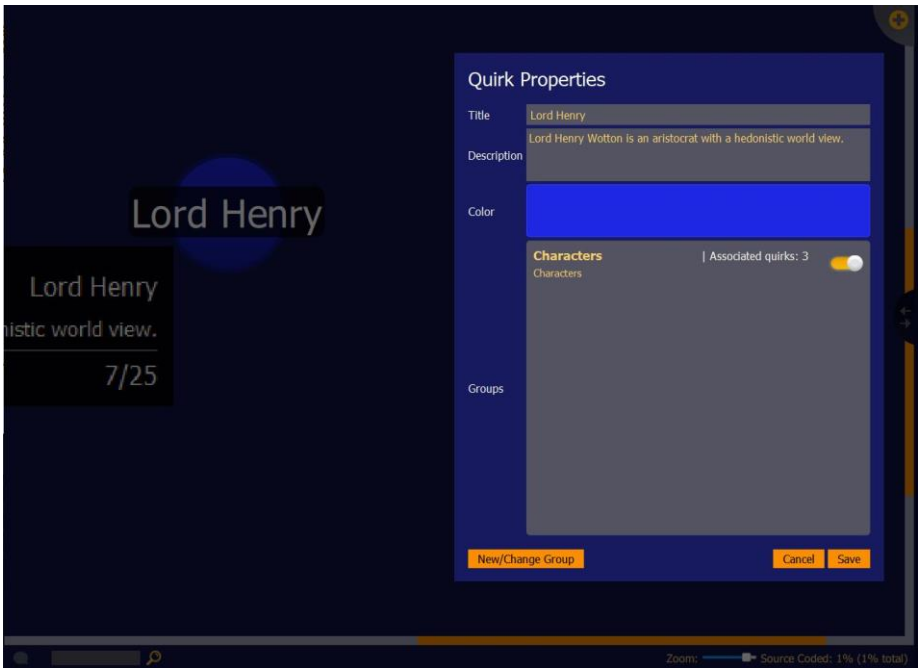

Figure 22: 'Quirk Properties'
In the properties menu of a code their title, description, color, and associated groups can be changed

The visualization doesn't support all of Shneiderman's (1996) tasks. The visual representation always gives an overview of the whole code system, and it is also possible to zoom into certain parts. More detailed info about codes can be obtained by clicking on them, it supports undo and redo, and the visualization can be extracted as a file. Besides the groupings there are no relations in the Quirkos visualization. More importantly the visualization always portrays the whole code system, and it isn't possible to only visualize the parts of interest.

5.4 Discussion

Quirkos is an outlier compared to MAXQDA and ATLAS.ti, its whole code system is managed by its visualization. It doesn't structure codes using a code hierarchy or group hierarchy, it just uses the visualization. However, the program doesn't support any relationships or any filtering. A researcher using Quirkos would always have the visualization on their screen while coding, this might give them a better overview over their code system than researchers working only with code lists while coding.

MAXQDA and ATLAS.ti visualizations are closer together than Quirkos. They visualize the objects of their project using icons and elements. Both have the option to visualize links, which reflect associations like the association between a code and its coded segments. They have different options to change the layout of their visualization, objects can be dragged onto the visualization, and even more similarities can be found.
In contrast to MAXQDA, the visualization of ATLAS.ti always depict the properties of the underlying project. Changing the properties of an element changes that element and creating a new element creates a new element in the whole project including other networks. Considering that, the networks of ATLAS.ti are close to the goals of our meta model visualization. Networks are used to display the existing associations and semantic relationships between the objects of a project. For the meta model visualization, we also want to visualize and manipulate the existing meta model properties of codes and the relationships between those codes.
In comparison, MAXQDA separates the visualization from the project and gives the user more freedom in designing their visualization. Once a code is inserted into the visualization, renaming, or recoloring it, doesn't change its properties anywhere else than in the map. MAXQDA maps are not restricted to the underlying project, the appearance can be manipulated more freely. The possibility of MAXMaps to use geometric shapes, texts and to import images or icons, might make it better at expressing concepts and themes of a code system.

6 Visualization Library Comparison

This chapter decides on the library which will be used for the visualization of the meta model in QDAcity. Four different graphic libraries get compared to decide for the best fitting one for our use case. The focus is on "higher-level" React libraries. React libraries have the advantage to be easily integrable into QDAcity, which is implemented in React. "Higher-level" library means a library fixated on visualizing graphs, networks, maps, etc. If the goal is to create a higher quality visualization, one could either implement their own visualization functionalities with a "lower-level" data visualization library like D3[9], but this would take too much time, or use a library that already supports some of the wanted visualization features.
Some of the requirements for the library are some basic canvas features like zooming and panning. The code system could consist of hundreds of codes, if all of those would be visualized the user would need to zoom and pan around to get an overview and work on certain sections. The library should already have nodes and edge components and those components should also be customizable. The user should be able to adapt the visualization in some ways, like rearranging the codes, changing the color or other visual features of nodes and edges, adding, and deleting codes. For a libraries documentation it would be important to have some guidance on how to achieve those features. In QDAcity drag and dropping codes into the visualization should be possible, for this reason it would also be nice for the library to have integrated layout algorithms, to sort the dragged in nodes and not just dropping them all on one spot.
The four libraries in this section were tested by trying to implement some of those features using the documentation and general help found on the internet[10], to estimate how easy it is to find information how to implement features and at the same time estimating the size and activity of the community. However, it must be noted that this process is subjective. To rely on facts as well, some stats of the github repositories and npm packages are getting compared in the last subchapter.
All the following libraries use the MIT-License[11] making it legal to modify and use them in commercial software.

[9] https://d3js.org/
[10] e.g github issues, www.stackoverflow.com or other forums
[11] https://opensource.org/licenses/MIT

6.1 ExcaliDraw

Excalidraw[12] is by their definition a “virtual whiteboard for sketching hand-drawn like diagrams”. In Excalidraw, to create diagrams, all kinds of geometric forms, text, imported pictures and hand drawn lines are used. Excalidraw can get used on their website or it can get integrated into a website. The reason why Excalidraw is considered as a graphic lib is, that they created a npm package[13] exposing an API. With the API Excalidraw elements can get created programmatically, making it possible to visualize the meta model.
However, while trying to create a node representation it was already hard to connect the rectangle object to the text object, so when dragged around they would move together. This could be fixed by using the same id for them, but it seemed like a workaround. There isn’t much documentation found about the API and concluding from the github issues, their weekly downloads, and the general questions asked online it seems not a lot of people use Excalidraw by using the API. Another problem that came up was, when trying to connect the arrows to the nodes, so they would resize, when moving one of the connected nodes. The arrows had properties for connecting the objects, but after trying for some time and not finding any help in the documentation it still couldn’t be done. One could calculate the resizing of the arrows every time a connected node is moved, but this would be additional work. It seems like Excalidraw isn’t the right package to create an interactive visualization of the meta model and would just take up too much time implementing some basic features.
Having said this, ExcaliDraw has some interesting features for creating a mind map, like drawing onto the canvas and creating unique objects.

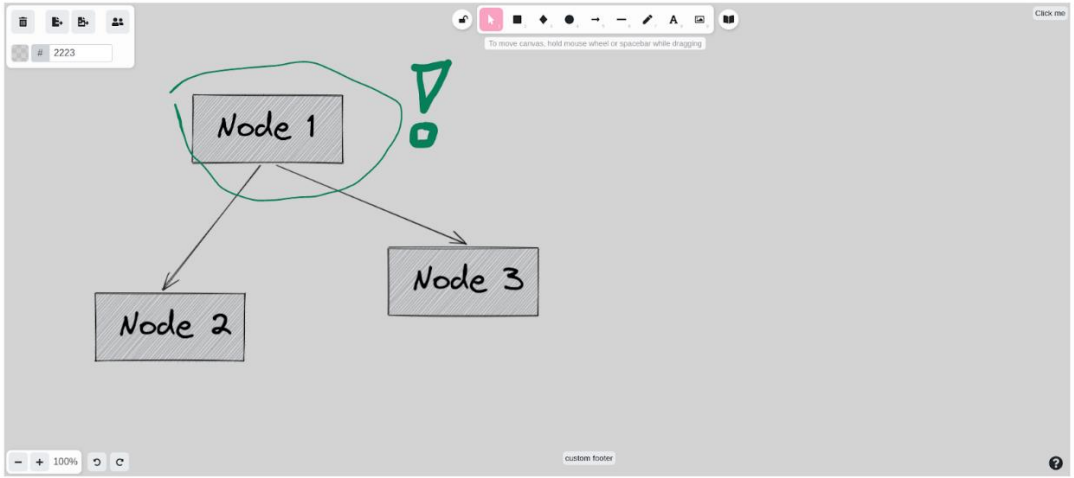

Figure 23: Integrated ExcaliDraw React Component
Three nodes connected by arrows and green hand drawn lines can be seen.

[12] https://github.com/excalidraw/excalidraw
[13] https://www.npmjs.com/package/@excalidraw/excalidraw

6.2 Beautiful React Diagrams

'Beautiful React Diagrams'[14] is a small React library, it is used to build diagrams consisting of nodes and edges. The user can drag nodes around in the canvas and can connect nodes at certain points, which are called ports. Elements can be styled using css, and custom nodes can be implemented. Figure 24 shows a custom node, it is a html element making it possible to use other html elements inside of it, like for example a button or a form. The nodes and edges can be manipulated, by changing the state given to the diagram component.
Since the library doesn't have a lot of features the documentation is small and the implementation of basic features was also easy. It would be possible to roughly visualize the meta model using this library and the user could create their own visualization by choosing different node customizations and dragging the nodes around. The library has no built-in zooming or panning, making the basic implementation a fixed size on the screen. The edges don't have a lot of customization features. Edges can have their own label, but they don't have any indication of flow. Therefore, it is hard to implement a directional relationship, which is needed to properly visualize the relationship codes.

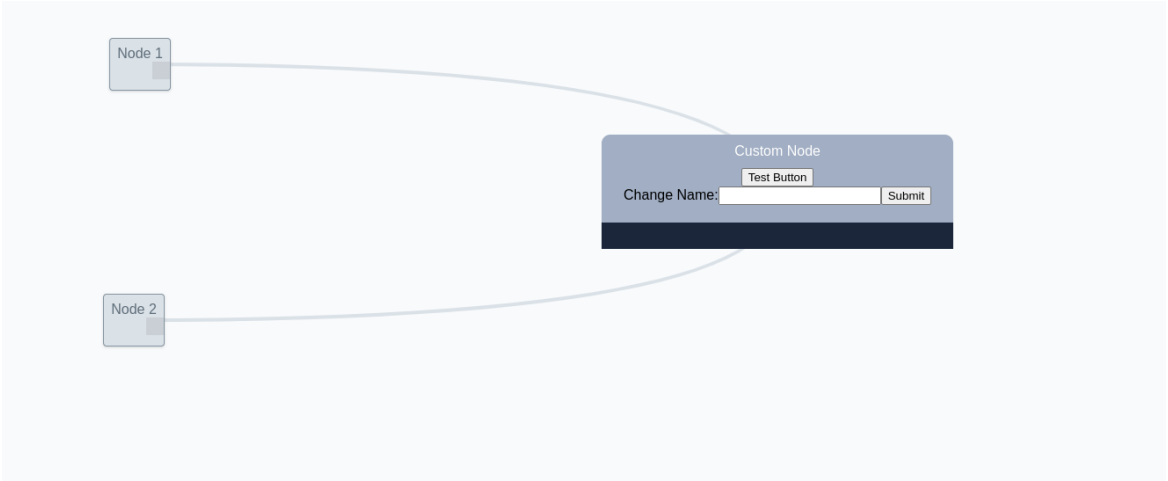

Figure 24 Integrated 'Beautiful React Diagrams' Component
Two nodes are connected to a custom node. The Custom Node has a button, a form to change his name, and it's at the bottom of the node port is dark blue. Ports of different nodes can be used to create new connections.

[14] https://github.com/antonioru/beautiful-react-diagrams

6.3 Graphin

The React toolkit Graphin[15] is based on the graph visualization library G6[16], which was created by the same company. Besides zooming and panning Graphin has a lot of other built-in features. Nodes and edges can be easily customized using various options. Nodes can use icons that are made available or own icons can be imported. Edges can be dashed, with halos or animated in different ways. Multiple nodes can be selected, dragged around, and deleted. Additionally, a component package can get downloaded, to use components like a context menu, or tool tip for the graph elements, a mini map, a hull, which creates an area around multiple nodes visually grouping them together, or a legend, which makes it possible to differentiate or select certain types of nodes. A toolbar component supporting undo and redo actions can be integrated. Graphin also has multiple integrated tree-graph and network layout algorithms. Using those the graph or only sections of the graphs can be reshaped. Behavior of the graph when clicking on a node, hovering on a node, or hovering an edge and more can be changed.
With the many functionalities and additional components Graphin provides, it seems possible to create a visualization of the meta model with features, which would take time to implement using other libraries.
However, parts of the documentation on their website are in chinese. Comments in their code are partly chinese, a lot of github issues are in chinese. When looking up how to customize nodes, beyond their available forms, they link to G6 documentation, which is only in chinese.

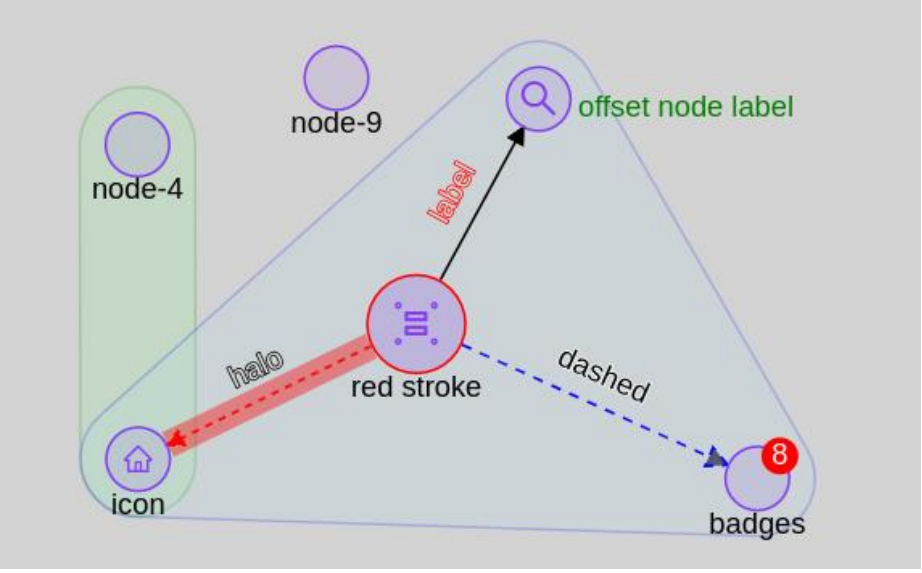

Figure 25 Integrated Graphin Component
Multiple nodes in a green and/or blue hull can be seen. Some nodes use Graphin icons.

[15] https://github.com/antvis/Graphin
[16] https://github.com/antvis/G6

6.4 ReactFlow

ReactFlow[17] is a React component to create graphs using nodes and edges. Like the other libraries they can be customized. To change their appearance CSS[18] rules can be defined, some preset options can be selected, or a custom element can be created. A custom node is depicted in Figure 26. Custom nodes and edges are html elements making it possible to render other components or html elements inside of them. The canvas can zoom and pan, nodes can be dragged around, and new connections can be created by dragging connections between nodes. Handles are the sections of the nodes, which edges can get connected to. Edges can have arrows at their end, but custom SVG[19] elements can be created and used at the start and end of edges. ReactFlow provides some components like a mini map and a control bar, but Graphin offers far more. Layouting algorithms are also missing, the documentation recommends using a third-party library for layouting. The ReactFlow component can be given a lot of different properties to customize behavior.
React Flow seems to have enough customizability to visualize the meta model and create own interactive features.

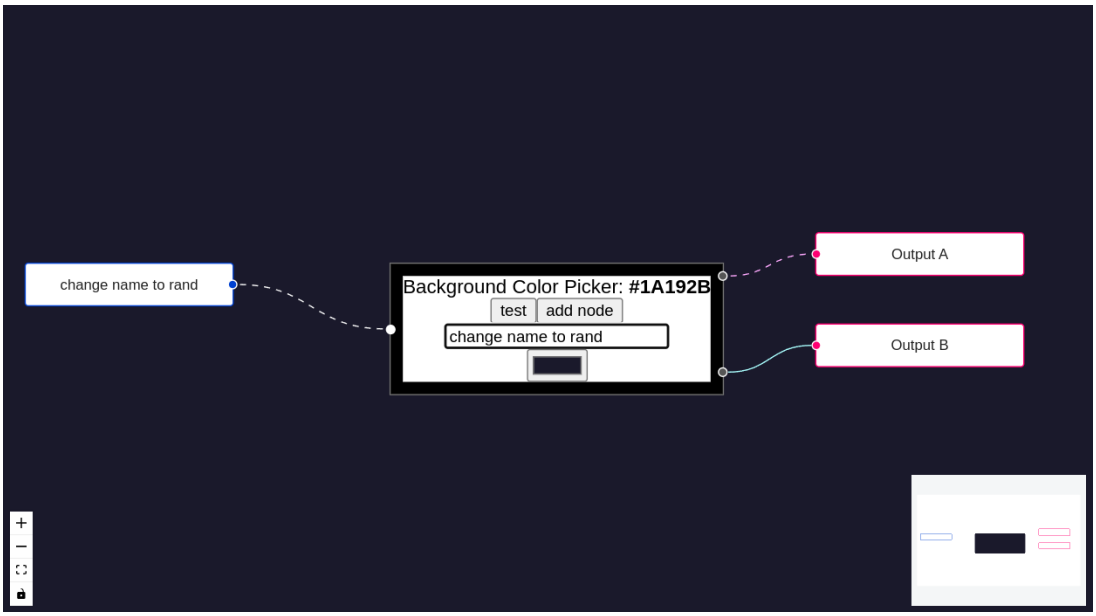

Figure 26 Integrated ReactFlow Component
A custom node with a color picker element, which changes the background and a form element which changes the name of the left node.

[17] https://github.com/wbkd/react-flow
[18] Cascading Style Sheets
[19] Scalable Vector Graphics

6.5 Discussion

In the upcoming subchapter the libraries get compared using the experience gained from implementing some basic features and some stats from their npm packages and github repositories, which can be seen in Table 1 and Table 2. These stats are used to estimate the activity and popularity of a project, which in turn may implicate a higher quality. The size of the packages is important since the user must download and run the QDAcity page in their browser. The bigger this downloaded bundle gets the longer it takes to download and compile all those files, making QDAcity slower and creating a worse experience for the user.

Npm Package	ExcaliDraw v. 0.11.0	Beautiful React Diagrams v. 0.5.1	Graphin v. 2.6.3	ReactFlow v. 10.1.0
Monthly Downloads	35.1 K	3.1 K	8.2 K	440.5 K
Monthly Growth	+13.3 %	+4.7%	+6%	+12.4 %
Package Size	11.1 MB	184 kB	1.1 MB *	1.2 MB
Dependencies	1	6	5	6

Table 1: Npm statistics from the 16.04.2022.
Monthly downloads and monthly growth are taken from https://moiva.io/.
* The size of their component package was added
(https://www.npmjs.com/package/@antv/graphin-components)

Github Repository	ExcaliDraw	Beautiful React Diagrams	Graphin	ReactFlow
Stars	28.6k	2.4k	671	10.4k
Total Open Issues	511	41	54	82
Closed Issues Last Month	27	0	1	41
New Issues Last Month	20	0	2	22
Merged Pull Requests Last Month	35	0	9	7
Open Pull Requests Last Month	27	2	3	4

Table 2: Github statistics from the 16.04.2022.
The statistics are taken from their github repositories and their associated insights page.
The monthly statistics are from the period 16.03 – 16.04.2022.

ExcaliDraw is the biggest package out of the examined libraries. ExcaliDraw seems like a popular and active project. It has a lot of downloads and github stars, but it seems that their npm package, which provides an API is less popular than their github projects, if one compares their github stars to their downloads. In the last month they merged new pull requests and solved some issues. However, the resolved issues are not about the ExcaliDraw API, but the application itself. Which is consistent with the implementation experience, that ExcaliDraw isn't used a lot programmatically. Hence, the popularity and activity are probably not meaningful for our use case. There were already problems, when trying to connect edges to nodes and dragging them around. ExcaliDraw can be ruled out, as it would be too complicated to create an interactive visualization.

Beautiful React Diagrams is the smallest sized package and has the least activity. A library, which doesn't add new features and just tries to remove bugs could be used for our use case. New features could be added to the visualization by implementing them ourselves. But the library is already missing some basic features like zooming and panning around in the canvas. For this reason, it doesn't seem to make sense to use this library, when basic features that are needed will take up a lot of time to implement them and other libraries have those features built-in.

Graphin has a similar size as ReactFlow if their component package is added to their total size. In the last month there were not a lot of issues dealt with but merged a similar amount of pull requests as ReactFlow. Their github stars seem low compared to their npm package downloads and the other projects. Graphin has a lot of inbuilt features, more than any of the other libraries. Inbuilt components and layouting are included. A visualization using Graphin has a lot of features starting out and time spent on the visualization can be focused on adding more features, instead of implementing basic ones. However, their documentation is partly in Chinese as well as their community seems to be mainly Chinese. Some documentation couldn't be found in English, some issues and code are in Chinese, and they seem to just focus mainly on the Chinese market. Because of the partly Chinese documentation it is hard to say how exactly Nodes can be customized and if edges between nodes can be created by the user dragging a connection.

ReactFlow seems popular and active. During the writing of the thesis, they released a new major version going from version 9 to version 10. The size of ReactFlow is similar to Graphin's size. It has less features than Graphin, no layouting and no components like an inbuilt context menu, tooltip, or legend. Their new sub flow feature[20] in version 10, which groups nodes together in a node, is worse than the hull feature[21] of Graphin, if used to just group nodes visually together. Because the sub flows can't overlap and right now also don't automatically extend when dragging a node to the edge of a sub flow. But ReactFlow's nodes can have handles, which allow the user to drag edges to another handle and it's easy to create custom nodes by following the documentation. In general, the relevant stuff to implement basic features is found in the documentation and they also offer examples on implementing different features.

ReactFlow is probably the best library to use for our use case out of those four libraries. ExcaliDraw was rejected on its own and Beautiful React Diagrams just has too few features compared to Graphin and ReactFlow. Graphin or ReactFlow could be used to create an interactive visualization. However, a documentation totally written in English is valued more than additional features. Furthermore, it is crucial to understand the source code of the library and not to miss out on comments written in Chinese.

[20] https://reactflow.dev/docs/examples/sub-flows/

[21] https://graphin.antv.vision/en-US/components/built-in/hull

7 Concept and Design

This section describes the concept and the design decisions of the meta model visualization in QDAcity. The chapter is divided into four main parts: the visualization method, the integration location, the visual encoding and the operations and features of the visualization. The visualization method and the way of integrating the visualization impact the types of features and operations of the visualization. Most of the thesis is about the visualization methods and the features of interactive visualizations and this will get used to justify decision. Nevertheless, eliciting requirements from users and then assessing the design decisions through usability tests is far better, but also time consuming. We bypass this deficit a bit by looking at other CAQDAS and relying on a guideline for abstract operations.
For the visual encoding of the meta model elements, we rely even more on assumptions. The objective is to make all the different elements distinguishable and visually pleasing, though if later on a better solution is found, it doesn't take much work to change the meta model elements design.

7.1 Visualization Method

The decision on which visualization method to use gives the design of the visualization some overarching concept. The depicting of the meta model's elements and the features of the visualization will be influenced by this choice. Different viable visualization methods were presented in Chapter 3.2 and some methods used by CAQDAS were shown in Chapter 5.
The main goal of our visualization is to give the user a visual way of interacting with the whole meta model. To use the visualization as a way of communicating ideas has a bit less of a priority. Therefore, we decided to design the meta model visualization as a network visualization or rather a node link diagram, in which we will restrict the customization options of nodes and edges a bit.
Totally unrestrictive infographics or mind maps, which allow to change the design of every element and to add foreign objects like pictures or shapes, might be better at illustrating ideas. However, the overview of the meta model might get obstructed if all the elements of the visualization can get changed freely. The user should be able to customize the model but should still be able to recognize the different meta model elements. In QDAcity it might not just be the user making it harder for themself to

recognize meta model properties, when multiple people are working on a project, too much freedom in designing the visualization makes it extremely hard when other people can manipulate the visualization in any way they want. Another benefit of having some common structure is, that the meta model properties can be recognized in different visualizations, and QDAcity users can share their visualizations between each other and understand them faster.
Defining the visualization as node link diagram is useful as codes can be linked through the CSL to an arbitrary number of other codes. There also exists no inherent hierarchy when using the meta model, so it doesn't make sense to classify the visualization as a concept map or a tree structure.
The meta model visualization provides the user with a new interface to interact with the meta model. This is supported by restricting customization options, but we also want to support the illustration of ideas to a certain extent. To facilitate the presentation of their meta model, the user can choose the set of codes he wants to visualize, the user can change certain properties of the nodes and relationship like their color and the user can arrange the codes on a 2D canvas.

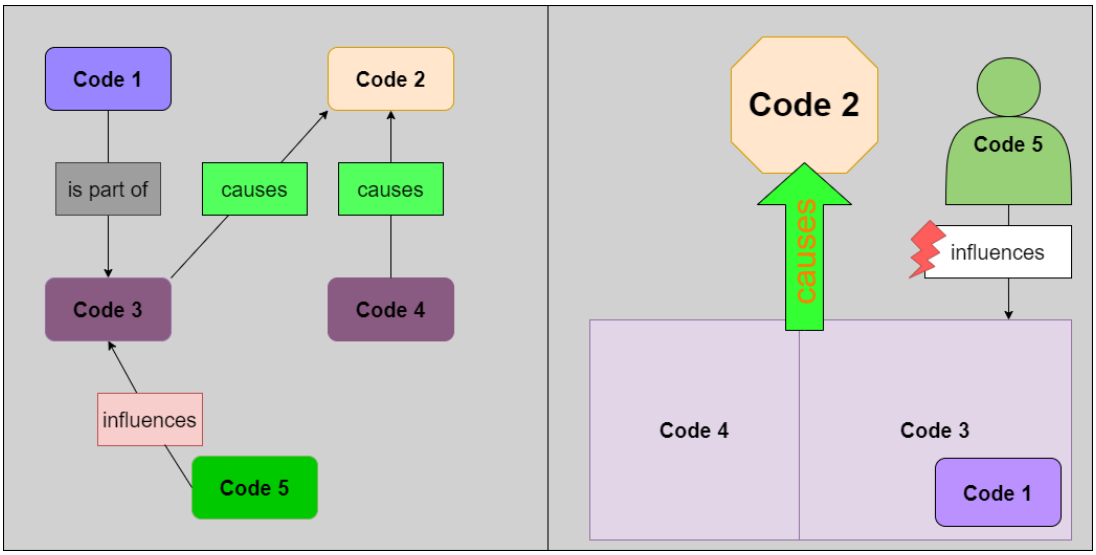

Figure 27: Two different Visualization Methods
The node link diagram on the left side facilitates the recognition of meta model properties.
The looser form of the info graphic on the right side makes the visualization overall more expressive.
(Created with https://app.diagrams.net/)

7.2 Integration Location

Where in QDAcity should the visualization get integrated in? The location in QDAcity influences the use case of the visualization. Two different locations are presented, their differences are highlighted and the decision for one gets justified in this subchapter.

As seen in Figure 28, the meta model visualization could be integrated into the Coding View of the Coding Editor. Additionally, to the code system and an open document, which are normally present, there would be space for a visualization. The user would have the meta model visualization open during the coding process. While discovering concepts and tagging text sections with codes the user could already see their code system developing. The user might be able to drag text sections into the visualization onto a node of a code to tag them. To use the network in this way the user might automatically get more familiar with the meta model and its visualization, since it will always be present while coding. Always having the visualization around might help with discovering insights and new relationships between concepts. On the other hand, connecting the visualization with the coding process in this degree might be disadvantageous. Users might not want to use the visualization and we also didn't test if a visualization of the meta model is beneficial for the user. Furthermore, it would make less sense to portray parts of the code system and more reasonable to always portray the whole code system. Another disadvantage is that the implementation of this possibility might be more difficult. Adding the visualization to the Coding View, the network would need to share the space with a document.

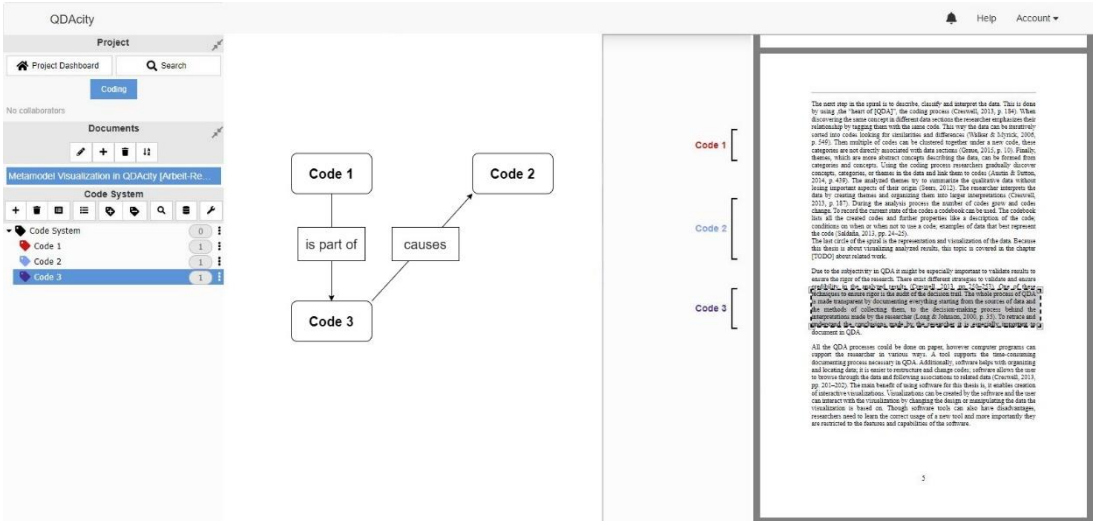

Figure 28: Meta Model Visualization integrated in the Coding View
The meta model visualization is between the code system on the left and the open document on the right
(Created with https://app.diagrams.net/)

In Figure 29 an easier possibility to fit the visualization into QDAcity is depicted, the visualization can be made into another view of the coding editor. The coding editor has already multiple views, which can be selected in the project dashboard and then can be switched to in the coding editor. The visualization could just be another view the user can activate and then can swap to if they want to use the network. We choose this location for the visualization since it fits the goal of visualizing only parts of the meta model better. The choice is also superior, if we want to extend the visualization in the future and let users create multiple visualizations. Moreover, the visualization can take up more space as it doesn't need to share it with a document.

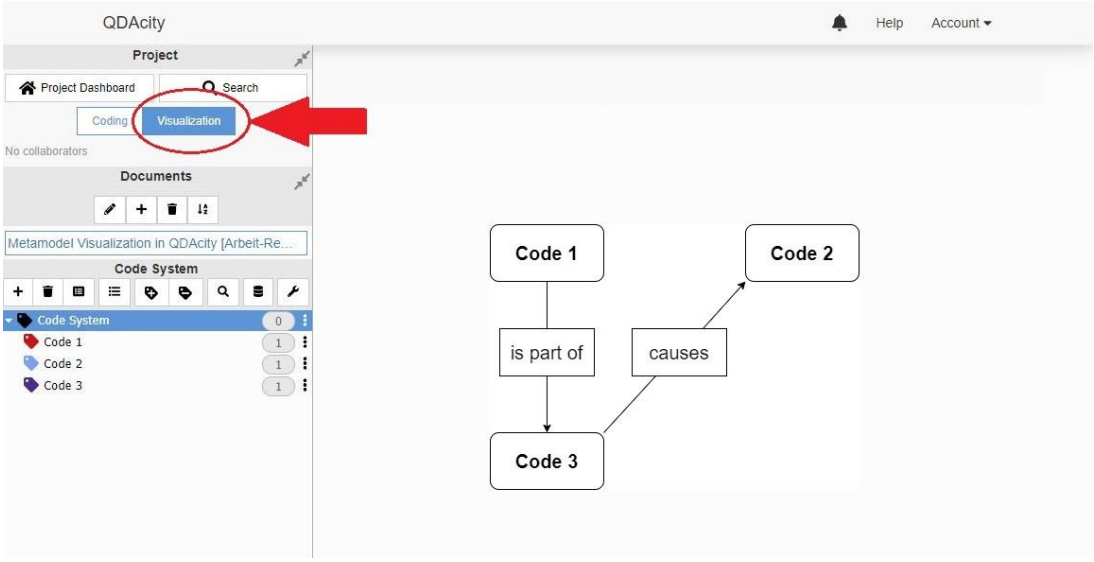

Figure 29: Meta Model Visualization integrated as a View
After clicking on the view "Visualization" in the coding editor the meta model visualization, seen here on the right, gets opened.

7.3 Visual Encoding of the Meta Model

The objective of finding a visually representing the elements of the meta model is to make the different elements recognizable while being visually pleasing. The design of the nodes and edges can be easily changed if needed.
The meta model, explained in Chapter 2.2.3, can describe two different kinds of codes. Codes can be classified as an Aspect or as a Relationship.

7.3.1 Aspect Codes

In QDAcity if a code is an Aspect code it can also be associated with a Label element of the meta model. Counting the parent elements an Aspect code can be associated with 8 different elements, namely: *Aspect, Structural Aspect, Dynamic Aspect, Object, Actor, Place, Activity, Process.* The three Label elements are *Property, Concept* and *Category*.
To visualize the meta model, we use the graphic library ReactFlow. In ReactFlow nodes can have handles, which allow the user to create new edges by dragging a line from one handle to another one.
For our Aspect codes all the necessary properties, which we want to fit into our node, are their associated Aspect and Label element if they have one, a custom color of the code, the handle element of ReactFlow and the name of the code.

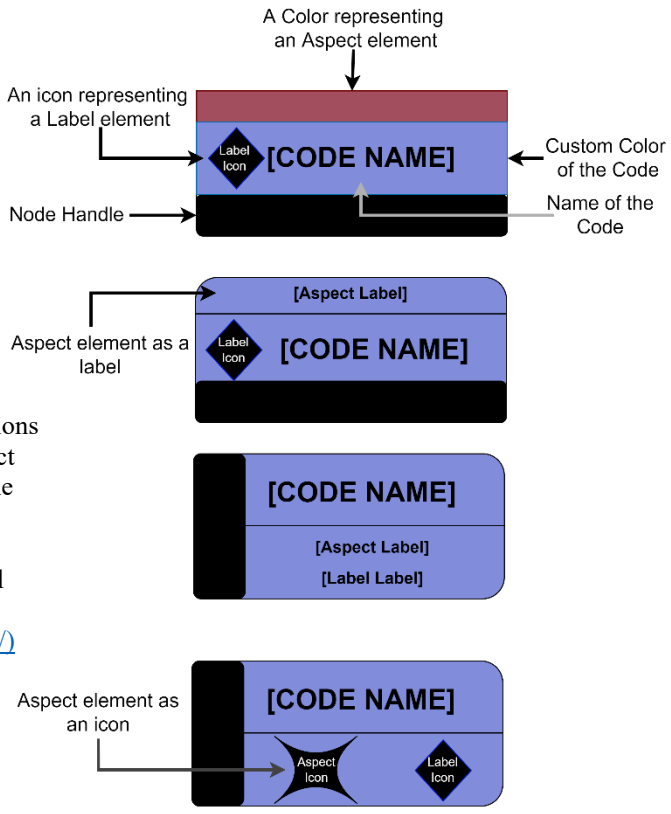

Figure 30: Aspect Code Design Variations
Different design variations of an Aspect code. The main differences between the variations are where the handle is positioned and how the meta model properties for the Aspect and the Label elements are visualized.
(Created with https://app.diagrams.net/)

We experimented with a few different designs, Figure 30 presents some of those. The name of the code should be the most present feature, the handle should be big enough to make clicking on it easier. The most thoughts were spent on the representation of the Aspect and Label elements. For the Aspect element we thought about using different icons, different colors, different node forms, additive frames or just labels. Since we want the nodes to take on the color associated with the code and give the user the possibility to change the color, representing the Aspect element as another color would then make the code have two different colors. This would then probably lead to the colors being less informative and confusing. The thought however was to visually differentiate not only between two elements of the meta model but also between their parent elements. For example, all the Dynamic Aspects would get associated with warm colors and all the Structural Aspects with colder colors. This approach can also be used if we would use forms, frames, or icons. However, we weren't sure if using 8 different visual representations for the Aspect elements would be beneficial for the user. If the user can't easily discern those designs or must look up which design is associated with which label, then it is less effective then a simple label. For this reason, we first used the simplest solution and represented the Aspect element of a code as label.
Contrary to the decision for the Aspect elements, we decided to use icons to represent the Label elements. It is easier for the user to get to know three icons than 8 icons and using two labels could clutter the design.
Figure 31 depicts an example code of the final design.

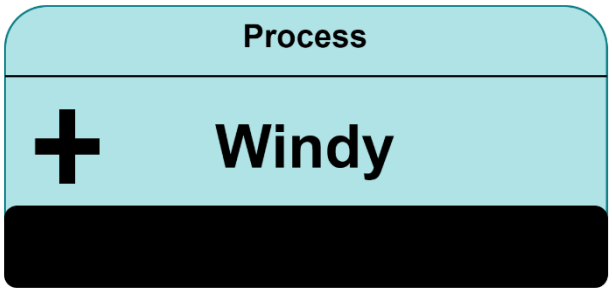

Figure 31: Example of the Aspect Code Design
The code has the name "Windy", a plus icon on the left to associate it with the Label element "Property" and a label on top to associate it with the Aspect element "Process". The handle to connect the code with other codes is at the bottom of the node

7.3.2 **Relationship Codes**

The total Relationship elements add up to 10 total elements: *Relationship, Structural Relationship, Dynamic Relationship, is a, is a part of, is related to, is consequence of, causes, performs, influences.* A new opportunity is available when using relationships as visual representations. Not only the label of an edge can be used to communicate a meta model element, also the edge can be visually changed. For example, different arrows or forms can be added to the end or start of an edge and the edge itself can be dashed or take different paths e.g., the path could be a Bézier curve. However, we will stick to our design decision of the Aspect code and rely on a label to represent a Relationship element.
Another design decision comes up with Relationship codes. Should the Relationship codes be visualized as a label on top of an edge or should they be visualized as another node. If it is visualized as a node the Relationship code can be moved around the canvas independently from the two codes a Relationship code is connecting. We want all codes to exist as nodes, this includes Relationship codes.
In Figure 32 presents an example of the final design of a Relationship code and the difference between visualizing Relationship codes as edge labels or as nodes.

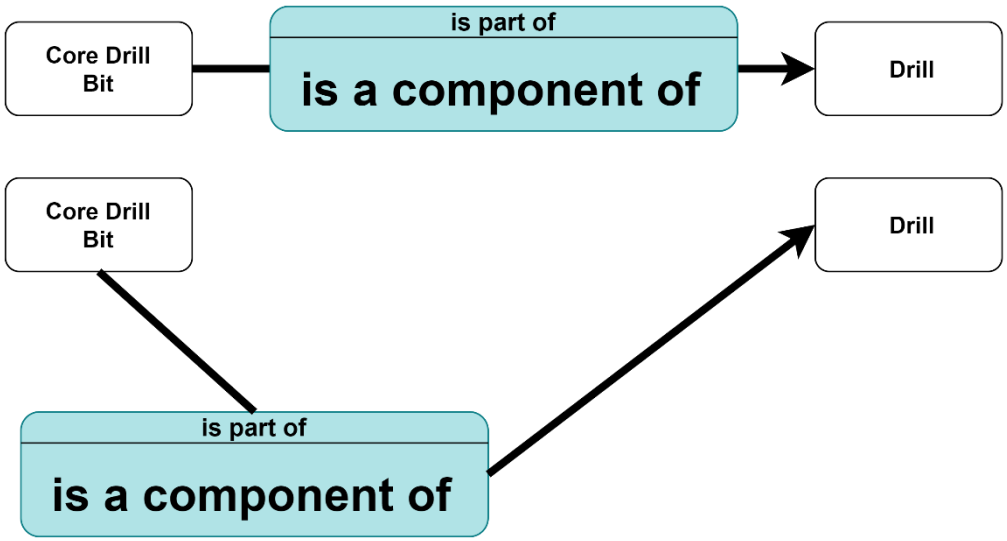

Figure 32: Relationship code represented as an edge label and as a node
The name of the Relationship code is “is a component of” and is associated with the Structural Relationship “is part of”.
The Relationship code on top is visualized as the label of an edge and can’t be independently moved around, it only moves if one of the connected codes gets moved.
The Relationship code on the bottom is a node itself and can be moved independently.

7.4 Interactivity methods and operations

Users can interact with visualizations in different ways. The interactivity methods important to us are dragging and dropping elements into the visualization, using a context menu, hovering over elements and a toolbar.
We want the QDAcity user to be able to drag codes from the code system into the visualization. They then would appear in the visualization and visualize their meta model properties. If the user drags a code, which has subcodes hierarchically below, they can choose to visualize the single code or all the code's subcodes as well.
The context menu enables the user to have specific operations depending on the context element they clicked on. Right-clicking on a code, an edge, the canvas, or a selection of visualization elements presents the user with different operations to choose from. Some examples for context specific operations are right-clicking on a node, the user can choose to delete the node; right-clicking on an edge, the user can choose another arrowhead for the edge; right-clicking on the canvas, the user can create a new node at this position; right-clicking while having selected multiple elements, the user can choose another layout for those elements.
Hovering with the mouse above different elements of the visualization also can have different effects. We want to display more information about the element below the mouse pointer. When the user hovers over nodes or edges, he gets access to information, which isn't visually represented. For example, if the mouse is above a node associated with a code, the description and memo of the code could get visible.
The toolbar has operations concerning the totality of the visualization. Operations like exporting the visualization to a file, changing the layout of the visualization, centering the visualization to have all the elements in view and undo/redo operations.

Shneiderman's (1996) abstract operations, presented in Chapter 3.3, can be implemented using those methods of interactivity.
An overview of the visualization is achieved through being able to zoom all the way out and then having all the elements of the visualization in view. The toolbar has a button to center the whole visualization in the view, which is a faster way of getting an overview.
Zooming in and out should be possible at different factors. The user controls the zooming and panning to get the exact view he wants.
The operation of filtering is not the same as selecting the desired codes out of the code system. It would be better to have quicker filtering options, maybe a legend at the top of the visualization. The user could create certain groups of codes and then select if they are visible or not. This reduces the amount of visible information fast. Another way of filtering the meta model visualization could utilize a slider. Codes could either be automatically divided into levels of abstraction. A code's level of abstraction is determined by its level in the hierarchical code system. Or the user assigns the codes with different importance levels. When moving the slider codes associated with certain levels fade out.
One way of getting more details about a code could be hovering above it and then

displaying additional information. The user could also get more information by right-clicking on an element opening the context menu and then selecting a menu, which shows even more information. Maybe it is also worth grouping the details of multiple codes together.
One of the main goals of the visualization is visualizing the complex relationships between codes. Another way to view the relationships of elements, could be selecting a group of elements and then opening another table to see different details about their relationship, for example which codes overlap with each other.
Undoing and redoing actions could be done with two buttons in the toolbar or by using common shortcut key combinations like “CTRL+Z” and “CTRL+Y”.
Extracting the visualization as an image or as a data structure would be best fitted into the toolbar.

8 Architecture

This chapter outlines the architecture of the meta model visualization. Figure 33 portrays the main components of the implementation. The frontend of QDAcity was implemented using React[22], a JavaScript framework and the backend service is running on the Google App Engine[23]. All the main components portrayed in Figure 33 are integrated into the frontend and therefore are implemented using React.

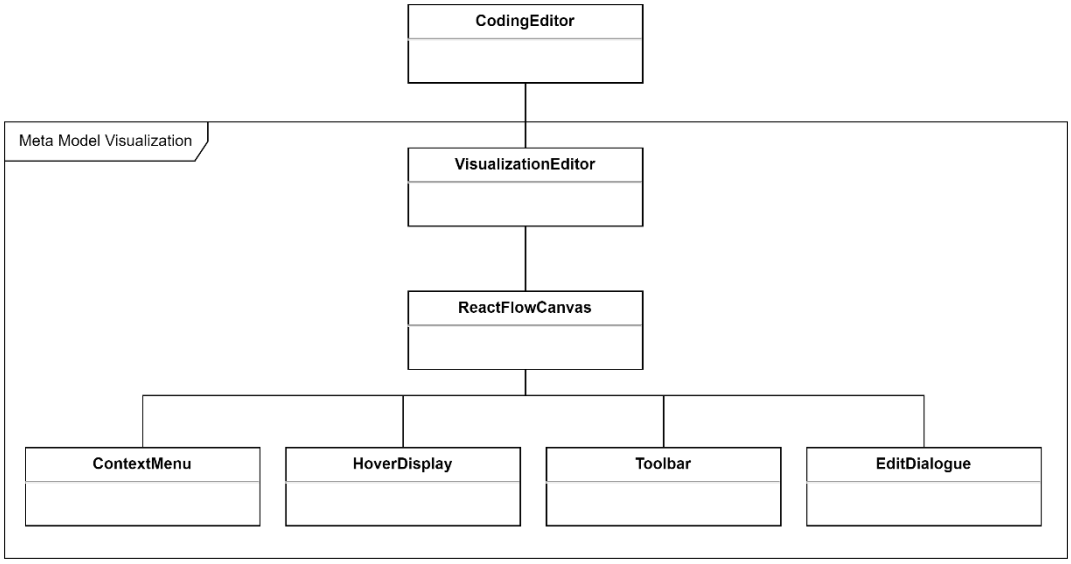

Figure 33: Component Structure of the Meta Model Visualization

The VisualizationEditor is the parent component of the visualization. Since we decided on integrating it as a view, the CodingEditor conditionally renders it like all the other views. A view only gets rendered, when the related setting got activated in the settings menu of the project dashboard. When the VisualizationEditor is rendered, it loads the meta model and the saved visualization from the backend and then afterwards renders the ReactFlowCanvas.

[22] https://reactjs.org/
[23] https://cloud.google.com/appengine

The ReactFlowCanvas implements the graphic library ReactFlow[24] and with its help visualizes the meta model. ReactFlow is a node-based graphic library and for this reason ReactFlow uses a collection of nodes and a collection of edges to represent their visualization. The ReactFlowCanvas component keeps track of all the nodes and edges of the visualization and implements functions to change them. To add codes to the visualization they can get dragged in from another component. The code's properties are requested from the backend and to associate a code with a meta model element we need to use the meta model provided by the VisualizationEditor. After receiving the code's properties and associated meta model elements we can render it in the visualization. When code change the CodingEditor calls an update function inside its child component ReactFlowCanvas. The ReactFlowCanvas then checks if due to the update anything inside the visualization should change.

The four components ContextMenu, HoverDisplay, Toolbar and EditDialogue can access the information inside of the visualization and are able to manipulate it. The toolbar is used to save the nodes and edges of the visualization in the backend. The other three components are context specific. The event handlers residing in the ReactFlowCanvas render them conditionally and give them information about the context element. Context elements can be the canvas, a node, an edge, or multiple elements. The component ContextMenu offers a different context menu for each context element. The component HoverDisplay display different information if the context element is a node or an edge. The component EditDialogue can get activated through context menus and can visually change nodes or edges in different ways.

[24] https://reactflow.dev/

9 Implementation

The following chapter covers the implementation details of the meta model visualization. The React components of the frontend are all class components and to define their CSS attributes in a simple and dynamic way the styled-components[25] library is used. Mostly the newly created components of this thesis will uphold this standard. However, ReactFlow uses functional components and some of the library's features can only be used when hooks are utilized. Class components can't use hooks, though we could mostly avoid this problem.

9.1 ReactFlowCanvas

When the CodingEditor gets initialized, it acquires the settings of the project from the backend. The settings contain which views are activated by the user. If the user activated the visualization editor, the CodingEditor renders the VisualizationEditor component. Then the user can choose a button to switch to the view of the visualization editor. After the VisualizationEditor gets mounted it loads the meta model entities from the backend and then renders the ReactFlowCanvas with the meta model entities as its properties. Then the ReactFlow component will get rendered, it is shown in Figure 34. The canvas of ReactFlow has a grey background and takes up nearly all the space of the browser. Like in the other views, the code system stays on the left side and if the user chooses to open the Code Properties Menu, it pops up on the bottom of the browser. The ReactFlow component renders the nodes and edges passed to it. Event handlers are passed to ReactFlow to update ReactFlow's internal state. Without event handlers only the viewport gets updated when the user zooms or pans around in the canvas. Basic event handlers to select, move and remove nodes and edges are passed to ReactFlow. We use two of the three plugin components ReactFlow offers. The first one is the mini map at the bottom right, it displays an overview of nodes in the viewport and close to the viewport. It is possible to assign colors to the nodes in the mini map, in our case we colored them with the same colors they have in the canvas. The other plugin component is the background component, and it gives the canvas a striped, gray background.

[25] https://styled-components.com/

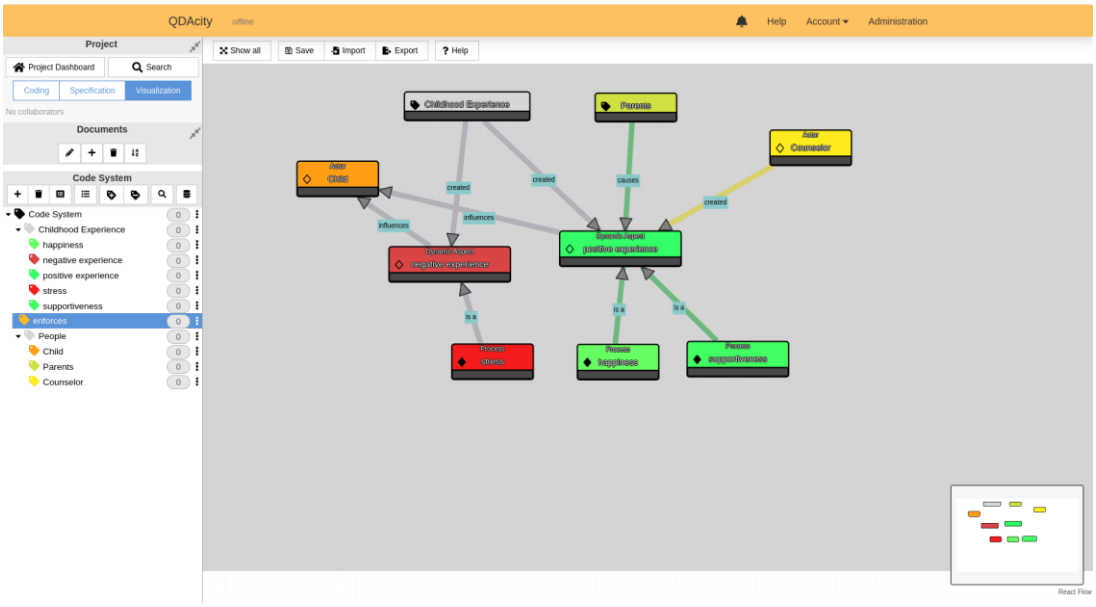

Figure 34: Visualization View

9.2 Nodes and Edges

ReactFlow offers the feature to create custom nodes and edges. Custom nodes render their inner HTML code and thus make it possible to render any content or functionality. Our custom nodes and edges are presented in Figure 35. We implemented two types of custom nodes. The first type is used for nodes not associated with codes, codes who don't have meta mode properties and codes which have Aspect or Label elements of the meta model associated with them. The second type is used for Relationship codes. The first type of node always has a name and a handle to create connections with other nodes. If the node is associated with a code, it has an icon on the left side. The normal tag is used for codes, which aren't classified with a Label element of the meta model. The plus icon symbolizes the Property element, the empty diamond icon symbolizes the Concept element and the filled in

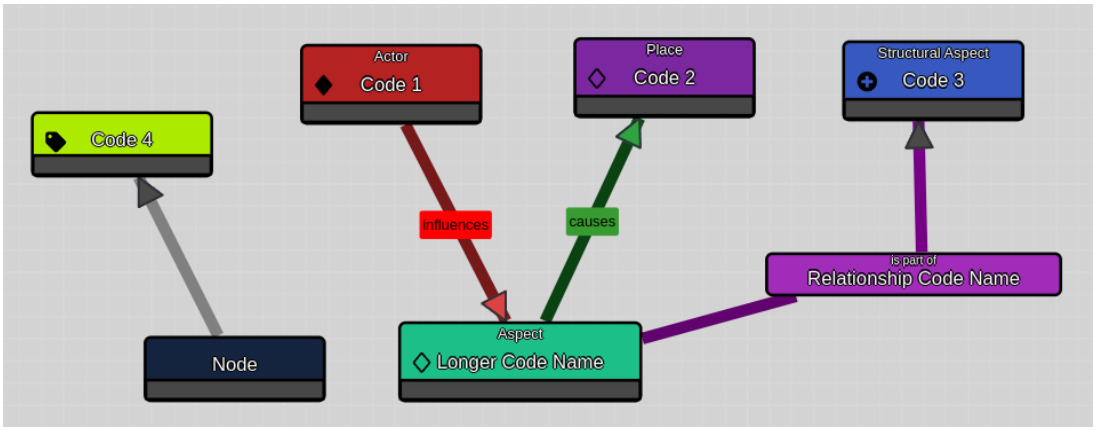

Figure 35: Custom ReactFlow Nodes and Edges

diamond symbolizes the Category element of the meta model. If a code has an Aspect element it is denoted at the top of the node. The custom nodes used for Relationship codes also depict their Relationship element of the meta model at the top. However, they don't have an icon and no handle so they can't be connected to other nodes. Relationship codes can only have edges to the source and target code of the relationship.
Two types of custom edges were created. One for depicting the relationship between all nodes and one for depicting the relationship between the source or target node and the Relationship code. The Relationship code edge can't have a label and the edge from source code to Relationship code doesn't have an arrow. For custom edge the path of the edge needs to be declared. ReactFlow offers a few functions to calculate the straight, step or Bézier path. Normally edges use the path from one handle to another, but in the ReactFlow documentation there is an example on how to implement floating edges[26]. When trying to implement this component as a class component we couldn't get away from using the *useStore( )* hook to get the position of the source and target node. Therefore, we had to wrap all the custom edges in a functional component which passes on the results to a class component. The arrow at the end of an edge is a custom SVG element. It is possible to create own SVG elements and visualize them somewhere on the edges. Fittings positions are often the end and start of an edge.

9.3 Features

A series of event handlers can be passed on to ReactFlow. The event handler *onConnect( )* is, for example called when a user drags a connection from one node handle to another. We use multiple event handlers to enable context specific actions.

9.3.1 **HoverDisplay**

We designed the HoverDisplay to give more details about a node on demand. ReactFlow has event handlers for the mouse entering and leaving a node, but we can't get the position of the node in the canvas when using styled-components. Therefore, the custom nodes activate the HoverDisplay and send their own React Ref to the HoverDisplay. The Ref of the node then is used to calculate the position of the HoverDisplay. If the node gets dragged around, the HoverDisplay moves with it. Additionally, the HoverDisplay doesn't change size and makes it able to get detailed information about nodes even if they are small, as seen in Figure 36. The HoverDisplay shows the name of a code, its meta model properties and from its codebook entry it shows the description of the code.

[26] https://reactflow.dev/docs/examples/edges/floating-edges/

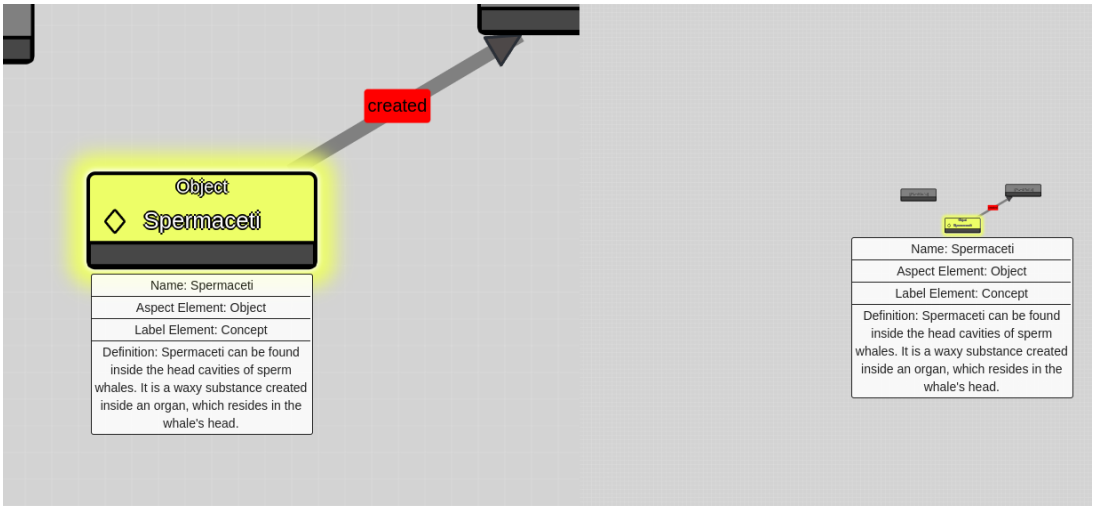

Figure 36: HoverDisplay zoomed in and zoomed out
On the left side the gray neighboring nodes can't even be seen and on the right side it isn't possible to read their names. But when hovering over a node we can still get information about the node.

9.3.2 **ContextMenu**

Some event handlers get called when the user right-clicks on different elements of the ReactFlow canvas. There are different context menu event handlers for single nodes and single edges, for a selection of elements and for the canvas itself. The ContextMenu component gets rendered at the position of the mouse, when right-clicking any of those elements. Different options depending on the element present themselves to the user. The context menu gets information about the context and the element of the context. For example, the node context menu gets information about the node and maybe about a code, if one is associated with the node.

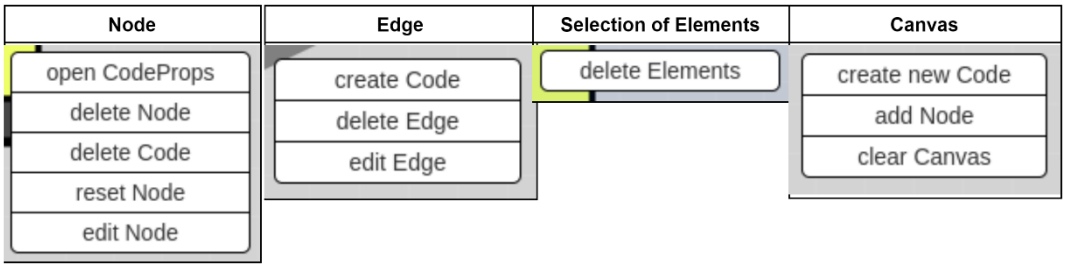

Figure 37: Context Menus of different Elements
The context menus of different elements are listed. From left to right: Node, Edge, Selection of Elements and Canvas

From the node context menu, the Code Properties Menu can be opened. In the Code Properties Menu, the meta model properties of a code can be changed. If the meta model properties or any other properties are changed, the *codeUpdated( )* function of the ReactFlowCanvas component is called from the CodingEditor. Then the ReactFlow component checks if the updated code is in the visualization and if it is, it just deletes the old node and adds a new node with the changes.

Using the context menus nodes, edges and multiple elements can get deleted or the whole canvas can get cleared. Those changes only affect the visualization, but codes can also get deleted from the backend database from within the visualization. Then the codes disappear from the visualization and from the code system. On the opposite side, new codes can be added through the canvas context menu. The node of the code gets added at the exact position of the canvas, where the context menu was opened. This is only possible by passing a Ref of the ReactFlow instance to the ContextMenu component and then through the mouse position and the current viewport of the visualization the position on the canvas is determined.
The node and context menu have the option to edit their element. When clicking on “edit Node” or “edit Edge” a context specific EditDialogue opens.

9.3.3 **EditDialogue**

From the EditDialogue component visual properties of nodes and edges are changed. In the edit dialogue of a node its name and color are customizable. The color of the arrow, the label, and the edge itself is changed in the edge edit dialogue. The EditDialogue can change the state of the nodes and edges of the ReactFlowCanvas by getting passed an update function. Updating the nodes and edges is separated from updating the codes. In Figure 38 the two different edit dialogues are shown.
The color of the arrowhead can be changed but is restricted to a few options right now and doesn’t use a color picker, as in the other options. The custom arrowhead needs to get passed to the ReactFlow component with a color. To enable a color picker, new custom SVG arrowheads would need to get created and managed.

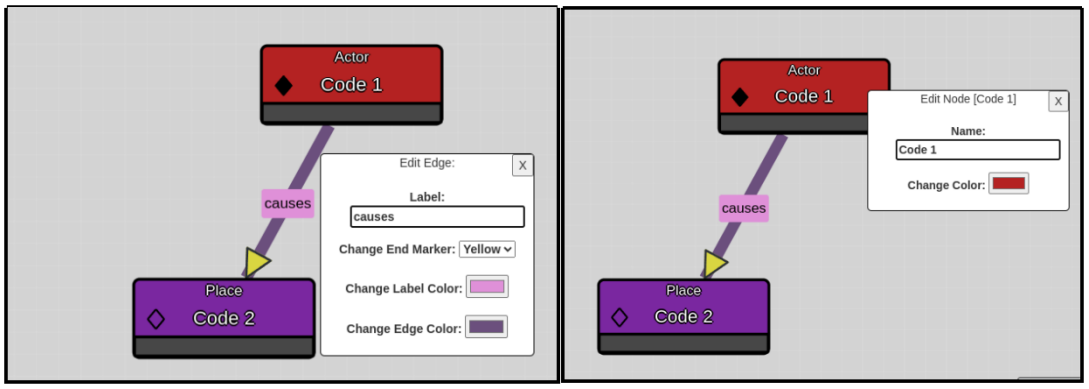

Figure 38: Edge and Node Edit Dialogue
The edit dialogue of the edge is on the left and the edit dialogue of the node is on the right.

9.3.4 **Toolbar**

The toolbar of the VisualizationEditor copies its looks from the UMLEditor toolbar. In Figure 39 the buttons of the toolbar are shown. The first button “Show all” changes the viewport to have all the nodes of the visualization in view. The “Import” button can import the data structure of a meta model visualization, and the “Export” button can export the nodes and edges. This is only done by opening a pop-up window and pasting in the nodes and edges representing a visualization or copying the current data of the visualization. The “Help” button gives some instructions on how to use the visualization. The backend endpoint is not yet implemented to make the “Save” button

Figure 39: Toolbar
The toolbar with its five buttons.

work.

9.3.5 **Drag and Drop**

From the list of codes of the code system, code can be dragged into the ReactFlowCanvas component. QDAcity uses the React DnD[27] library t drag and drop elements from one component to another. Dragging the code into the ReactFlowCanvas triggers the function *dropCodeIntoCanvas(codeMainID)*. It first gets the code’s properties by calling *getCodeByCodeId(codeMainID)*, which it receives from the CodingEditor. Then if the code has subcodes, it asks the user wether all the subcodes should get imported as well. If the user only wants to import exactly one code, *addCode(code, isRecursive)* is called only once. First it checks if the code is in the visualization and therefore shouldn’t be imported another time. If it isn’t in the visualization yet, the meta model entity id is used to look up the corresponding meta model elements from the meta model which gets passed down by the VisualizationEditor.
If the code has no meta model properties or it is an Aspect code, a certain type of custom node is created. If it is a RelationshipCode, a RelationshipCodeNode is created. If added nodes have associated relations to other codes, they get added as edges. The user also might choose to import all the subcodes of the dragged-in code. Then *addCode(code, isRecursive)* gets called for every subcode as well.
Using React DND we didn’t manage to call the *onDrop(event)* event handler of ReactFlow. As a result, we can’t determine at which position to code was dropped into the canvas. We place the dragged-in nodes randomly in a certain area to avoid stacking all of them on one position.

[27] https://react-dnd.github.io/react-dnd/about

10 Evaluation

In Chapter 4 the requirements of the meta model visualization got set up. Now it is time to check every one of them and use them evaluate the resultant implementation. A better evaluation approach is to observe and interview the people using the implemented system. We didn't have time to evaluate our system this way and therefore we must rely on some assumptions when evaluating the thesis. There exist several threats to the validation of this thesis. Maybe there was no need for a visualization in QDAcity and creating one doesn't help to solve any problems (Munzner, 2009, pp. 923–924). The interactive operations we chose to implement might not help the target users. Our visual encoding might not be effective at communicating the meta model. Lastly, the implemented system might be slow and memory inefficient.

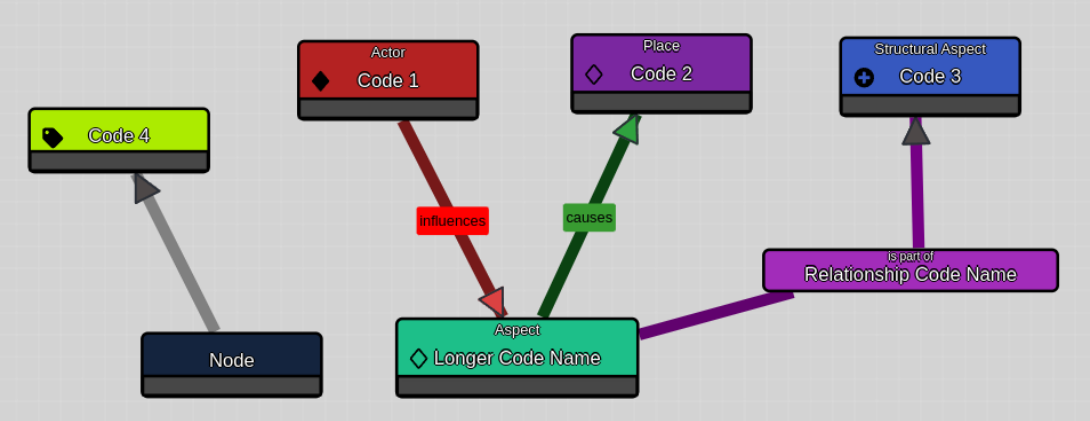

Figure 40: Visualization of the Code System

10.1 Check RQ1: Visualizing the Code System

The user can visualize the whole code system by dragging the root code into the visualization and choosing to include all subcodes. It doesn't matter if a code has Meta Model properties or not, all codes get visualized.

10.1.1 Check RQ1.1: Visualizing the Meta Model and the CSL

Nodes with associated codes always visualize their current Meta Model properties. All the different Meta Model elements have different visual features. The Aspect and Relationship elements are represented as labels above the code name. The Label elements are depicted as icons. Relationship codes can be further distinguished, as they have a different node design. "Non-code" relationships are visualized as edges with labels.

10.1.2 Check RQ1.2: Visualizing only parts of the Code System

It is possible to only include certain parts of the code system, as every code can be dragged into the visualization by oneself. Or the user can choose to visualize certain groups of codes by dragging in the code hierarchically above all the other codes and choosing to include its subcodes.

10.2 Check RQ2: Changing the Code System using the Visualization

10.2.1 Check RQ2.1: Creating and Deleting Codes

In the context menu of a code, the specific code can get deleted from the code system and the node gets deleted form the visualization.
In the context menu of the canvas, a code can be created and given a name. It gets added to the code system and appears in the visualization.

10.2.2 Check RQ2.2: Changing Meta Model Properties

The codes Meta Model properties can be changed by using the Code Properties Menu. First the user needs to toggle the menu in the context menu of a code and then he can change the Meta Model properties. However, this is inside of the Visualization view but not directly inside of the visualization. This means, the requirement isn't totally fulfilled. When a code gets updated the CodingEditor will call the update function in the ReactFlowCanvas. If the code is inside the visualization the corresponding node will change its appearance.

10.3 Check RQ3: Customization of the Visualization

10.3.1 Check RQ3.1: Creating Independent Nodes and Edges

Independent nodes are created through the canvas context menu. They appear in the canvas at the position the context menu was opened.
Independent edges can be created by dragging a connection between the handles of two nodes. This makes it possible to connect any node with another, besides the nodes of relationship codes, since they don’t have a handle

10.3.2 Check RQ3.2: Changing Visual Properties of Nodes and Edges

In the EditDialogue of nodes and edges different color properties can be selected. For nodes it is only possible to choose the background color and for edges it is the color of the edge, the label, and the arrow at the end of the edge. Changing the colors in an EditDialogue only changes the visualization and not the code system

11 Conclusion

11.1 Improvements

Some useful improvements for the visualization are:

Saving the Visualization:
There is no backend functionality yet to save the visualization. This is a much-needed feature. It should be possible to save the created visualization and load it when opening the visualization view.

Multiple Visualizations:
The visualization doesn't have to portray the total code system. Only parts can get depicted in the canvas. The user creates a qualitative visualization to convey certain concepts. Since the visualization is used like this it seems sensible to make it possible to create and save multiple visualizations for one project.

More Customization Options:
The changeable visual properties of nodes and edges are only a few. Users should be able to control the look of the visualization to a higher degree. Especially the relationships should get more options. The path an edge takes from node to node should be customizable. There also should be more elements at the beginning and end of edges.

Visualize more Elements:
Other elements from a project could also be visualized. Such as memos or documents.

Save Multiple Versions of a Visualization:
Make it possible to save different versions of a visualization. Version control could be used to document the development of the code system and the Meta Model.

Filtering out elements:
Give the user more options to filter uninteresting information fast. Nodes can be deleted from the canvas, but this approach is slow. The filter options should be determined by questioning and observing target users of QDAcity.

Exporting an Image:
Showing the visualization to other people is easier if the visualization can be exported as an image with the press of a button. Unfortunately, ReactFlow doesn't have an inbuilt feature like this.

Keeping an Action History:
By saving the history of user actions, it is possible to implement undo and redo functions.

Using Metrics to Manipulate the Visualization:
Metrics can be used to change the appearance of the visualization. For example, the size of nodes could be determined by the amount of text sections associated with the code, or/and by the number of edges connected to the code.

11.2 Results

In this thesis we created a visualization for the meta model of QDAcity. We have shown that the use case of a visualization changes depending on the visualization method and where it gets integrated in QDAcity. A less structured visualization method didn't seem viable to use it as an editor of the code system. Integrating it into the Coding View is an interesting idea to pursue, however it conflicted with our goal to only visualize parts of the code system. Our goals were to create a way of manipulating the meta model and at the same time creating a qualitative representation the code system. We can be certain that there is a lot of room for improvement, since the area of visualizations is broad, and we only assumed the needs of the users. Still, we fulfilled our goal of creating a meta model visualization. It enables a new way of interacting with the meta model and code system. A visual representation of the code system might help the user immensely in overviewing and manipulating it.

The thesis results can be a basis to further develop this visualization, test how QDAcity's users interact with a visualization or give ideas to pursue other forms of visualizations for QDAcity.

References

Austin, Z., & Sutton, J. (2014). Qualitative research: Getting started. *The Canadian Journal of Hospital Pharmacy*, *67*(6), 436–440. https://doi.org/10.4212/cjhp.v67i6.1406

Birks, M., Chapman, Y., & Francis, K. (2008). Memoing in qualitative research. *Journal of Research in Nursing*, *13*(1), 68–75. https://doi.org/10.1177/1744987107081254

Bryman, A., & Bell, E. (2011). *Business research methods* (3rd ed.). Cambridge, New York NY: Oxford University Press.

Card, S. K., Mackinlay, J. D., & Shneiderman, B. [Ben] (Eds.) (2007). *The Morgan Kaufmann series in interactive technologies*. *Readings in information visualization: Using vision to think* ([Nachdr.]). San Francisco, Calif.: Morgan Kaufmann.

Carr, D. (1999). Guidelines for designing information visualization applications. In *ECUE'99 : 01/12/1999 - 03/12/1999*. Retrieved from https://www.diva-portal.org/smash/record.jsf?pid=diva2:1013047

Coffey, J. W., Hoffman, R., & Cañas, A. (2006). Concept Map-Based Knowledge Modeling: Perspectives from Information and Knowledge Visualization. *Information Visualization*, *5*(3), 192–201. https://doi.org/10.1057/palgrave.ivs.9500129

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

Creswell, J. W. (2013). *Qualitative inquiry and research design: Choosing among five approaches* (3rd ed.). Los Angeles: SAGE Publications.

Davies, M. (2011). Concept mapping, mind mapping and argument mapping: What are the differences and do they matter? *Higher Education*, *62*(3), 279–301. https://doi.org/10.1007/s10734-010-9387-6

Dr. Susanne Friese (2022). ATLAS.ti 22 Windows - User Manual. Retrieved from https://atlasti.com/manuals-and-documents

Flick, U., Kardorff, E. von, & Steinke, I. (Eds.) (2010). *A companion to qualitative research* (Repr). London: SAGE.

Graue, C. (2015). Qualitative data analysis. *International Journal of Sales, Retailing & Marketing*, *4*, 5–14. Retrieved from https://www.circleinternational.co.uk/wp-content/uploads/2021/01/ijsrm4-9.pdf#page=9

Hammersley, M. (2013). *What is qualitative research?* London: Bloomsbury.

Henderson, S., & Segal, E. H. (2013). Visualizing Qualitative Data in Evaluation Research. *New Directions for Evaluation*, *2013*(139), 53–71. https://doi.org/10.1002/ev.20067

J. Bettis, P., & A. Gregson, J. (2001). *The why of research: Paradigmatic and pragmatic considerations*. Retrieved from http://www.bwgriffin.com/gsu/courses/edur7130/readings/bettis_gregson_paradigm.pdf

J. Daley, B. (2004). *Using concept maps in qualitative research*. University of Wisconsin -

Milwaukee. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.133.6537&rep=rep1&type=pdf

Kaufmann, A. (2021). *Domain Modeling Using Qualitative Data Analysis*. Erlangen: Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU). Retrieved from https://nbn-resolving.org/urn:nbn:de:bvb:29-opus4-167369

Keller, R., Eckert, C. M., & Clarkson, P. J. (2006). Matrices or Node-Link Diagrams: Which Visual Representation is Better for Visualising Connectivity Models? *Information Visualization*, *5*(1), 62–76. https://doi.org/10.1057/palgrave.ivs.9500116

Kerren, A., Stasko, J. T., Fekete, J.-D., & North, C. (Eds.) (2008). *Lecture Notes in Computer Science*. *Information Visualization*. Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-70956-5

Lin, H., & Faste, H. (2011). Digital mind mapping. In D. Tan, S. Amershi, B. Begole, W. A. Kellogg, & M. Tungare (Eds.), *Proceedings of the 2011 annual conference extended abstracts on Human factors in computing systems* (p. 2137). New York, NY: ACM. https://doi.org/10.1145/1979742.1979910

Long, T., & Johnson, M. (2000). Rigour, reliability and validity in qualitative research. *Clinical Effectiveness in Nursing*, *4*(1), 30–37. https://doi.org/10.1054/cein.2000.0106

Loos, F. (2017). *A Visual UML-Editor for QDAcity*. Erlangen: Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU). Retrieved from https://osr.cs.fau.de/wp-content/uploads/2017/11/loos-2017-arbeit.pdf

Miles, M. B., & Huberman, A. M. (2009). *Qualitative data analysis: An expanded sourcebook* (2. ed., [Nachdr.]). Thousand Oaks, Calif.: SAGE.

Munzner, T. (2009). A nested model for visualization design and validation. *IEEE Transactions on Visualization and Computer Graphics*, *15*(6), 921–928. https://doi.org/10.1109/tvcg.2009.111

Newman, M. E. J. (2018). *Networks* (Second edition). Oxford: Oxford University Press. Retrieved from https://ebookcentral.proquest.com/lib/kxp/detail.action?docID=5447663

Pokorny, J. J., Norman, A., Zanesco, A. P., Bauer-Wu, S., Sahdra, B. K., & Saron, C. D. (2018). Network analysis for the visualization and analysis of qualitative data. *Psychological Methods*, *23*(1), 169–183. https://doi.org/10.1037/met0000129

R. Mammen, J., & R. Mammen, C. (2018). *Beyond concept analysis: Uses of mind mapping software for visual representation, managment, and analysis of diverse digital data.*

Saldaña, J. (2013). *The coding manual for qualitative researchers* (2nd ed.). Los Angeles: SAGE.

Salow, S. (2016). *A Metamodel for Code Systems*. Erlangen: Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU). Retrieved from https://osr.informatik.uni-erlangen.de/wp-content/uploads/2016/10/salow-2016-arbeit.pdf

Seers, K. (2012). Qualitative data analysis. *Evidence Based Nursing*, *15*(1), 2. https://doi.org/10.1136/ebnurs.2011.100352

Shneiderman, B. [B.] (1996). The eyes have it: a task by data type taxonomy for information visualizations. In *IEEE Symposium on Visual Languages, 1996. Proceedings* (pp. 336–343). IEEE / Institute of Electrical and Electronics Engineers Incorporated. https://doi.org/10.1109/VL.1996.545307

Swedberg, R. (2016). Can You Visualize Theory? On the Use of Visual Thinking in Theory Pictures, Theorizing Diagrams, and Visual Sketches. *Sociological Theory*, *34*(3), 250–275. https://doi.org/10.1177/0735275116664380

Vekiri, I. (2002). What Is the Value of Graphical Displays in Learning? *Educational Psychology Review*, *14*, 261–312. Retrieved from https://www.researchgate.net/publication/30845217_What_Is_the_Value_of_Graphical_Displ

ays_in_Learning

Verbi GmbH (2022). MAXQDA 2022 Manual. Retrieved from https://www.maxqda.com/help-mx22/welcome

Walker, D., & Myrick, F. (2006). Grounded theory: An exploration of process and procedure. *Qualitative Health Research*, *16*(4), 547–559. https://doi.org/10.1177/1049732305285972

Wheeldon, J., & Faubert, J. (2009). Framing Experience: Concept Maps, Mind Maps, and Data Collection in Qualitative Research. *International Journal of Qualitative Methods*, *8*(3), 68–83. https://doi.org/10.1177/160940690900800307

Yi, J. S., Kang, Y., Stasko, J. T., & Jacko, J. A. (2008). Understanding and Characterizing Insights: How Do People Gain Insights Using Information Visualization? In E. Bertini, A. Perer, C. Plaisant, & G. Santucci (Eds.), *Proceedings of the 2008 conference on BEyond time and errors novel evaLuation methods for Information Visualization - BELIV '08* (pp. 1–6). New York, New York, USA: ACM Press. https://doi.org/10.1145/1377966.1377971