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

JULIAN LEHRHUBER
MASTER THESIS

PDF SUPPORT FOR QUALITATIVE RESEARCH IN THE CLOUD

Submitted on 20 May 2021

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

Erlangen, 20 May 2021

License

This work is licensed under the Creative Commons Attribution 4.0 International license (CC BY 4.0), see https://creativecommons.org/licenses/by/4.0/

Erlangen, 20 May 2021

Abstract

Effective Qualitative Data Analysis (QDA) using software tools relies on the range of supported document types to work with. The Portable Document Format (PDF) standard is widely known and used because of its versatility. Therefore, the support of PDF documents in QDA software is essential.

The cloud-based QDA tool 'QDAcity' only supports Rich Text Format (RTF) documents. In this thesis, we design and implement PDF support for QDAcity. Since the current state of QDAcity does not allow to easily extend the range of supported document types, our implementation is required to allow this in the future. The main challenge however is to design a coding mechanism, that can handle different document types. Using the coding mechanism, researchers annotate segments of the qualitative data to extract a theory. To implement such a coding mechanism, we analyzed the implementation of different QDA tools and evaluated various implementation strategies for the cloud.

Different types of documents require different types of codings, such as area codings that can be used for image data. Our implemented coding mechanism therefore can be extended by future coding types and already includes support for area codings.

Contents

1 Introduction **1**

2 QDAcity **2**

3 Previous work **4**
3.1 Comparison with other QDA software 4
3.1.1 Analysis of coding mechanism design 4
3.1.2 Handling editable documents with an abstract coding mechanism 5
3.2 Evaluation of different strategies to integrate PDF support 5
3.2.1 Backend based approach 6
3.2.2 Frontend based approach 7
3.3 Findings 9

4 Requirements **10**
4.1 Functional requirements 10
4.2 Non-functional requirements 11

5 Architecture **12**
5.1 Document Support 12
5.1.1 Backend 12
5.1.2 Frontend 13
5.2 Coding mechanism 14

6 Implementation **16**
6.1 Document Support 16
6.1.1 Backend 16
6.1.2 Frontend 19
6.2 Coding mechanism 21
6.2.1 Side-effects of new coding mechanism 24
6.2.2 Synchronizing coding updates across clients 26
6.2.3 Migration of existing text documents 27

7 Unforeseen challenges **30**
7.1 Use SUPERCLASS_TABLE inheritance with DataNucleus 30
7.2 Use external codings with Slate 31

8 Evaluation **33**
8.1 Functional requirements 33
8.2 Non-functional requirements 36

9 Future work **39**
9.1 Remove embedded coding keys from RTF documents 39
9.2 Realtime-service integration for saving documents 39
9.3 Store documents externally 40
9.3.1 Lazily fetch documents 40
9.4 Coding previews for area codings 40

10 Conclusion **42**

Appendices **44**
Appendix A Evaluated Java PDF libraries 44
Appendix B Evaluated JavaScript PDF libraries 46

References **47**

List of Figures **48**

List of Code Snippets **49**

Acronyms

API Application Programming Interface.

DOM Document Object Model.

HTML Hypertext Markup Language.

ID Identifier.

JDO Java Data Objects.

JSON JavaScript Object Notation.

MIME-Type Internet Media Type.

PDF Portable Document Format.

PMF Persistence Manager Factory.

QDA Qualitative Data Analysis.

RE Requirements Engineering.

RTF Rich Text Format.

SDK Software Development Kit.

SVG Scalable Vector Graphics.

UI User Interface.

UML Unified Modeling Language.

1 Introduction

Qualitative Data Analysis (QDA) describes a process in which researchers combine relevant data from different sources to uncover their essence (Bazeley, 2013; Caudle, 2004). The sources consist of unstructured qualitative data like interviews, on-site observations, and even audio or image data. During data analysis, researchers annotate segments of the data with information that abstracts the meaning of the data.

To support the QDA process, various software tools are available. Since the QDA process is not limited to certain types of documents, software tools have to support several different document types. QDAcity, which is a cloud-based QDA application, currently only supports Rich Text Format (RTF) documents. As PDF documents are used in many environments, it is important to support PDF documents in a QDA application. To enable PDF support for qualitative research in the cloud, the goal of this thesis is to extend QDAcity with support for PDF documents. From a software engineering perspective, the goal of this thesis raises multiple questions:

- How can we integrate PDF support into QDACity?
- Can we reuse existing functionality from the RTF document support for this feature?
- How can we implement support for multiple document types, that can be extended in the future?

To answer these questions, we first present the current state of QDAcity in chapter 2. In chapter 3, we analyze the implementation of PDF support in different software tools. The analysis allows us to develop concepts on how best to integrate PDF support in a cloud application. With the findings of the analysis, we formulate the requirements for our implementation (chapter 4). Afterward, we present the software architecture (chapter 5) and implementation of our solution (chapter 6). Before evaluating our solution in chapter 8, we point out unforeseen challenges to help future developers (chapter 7). We close this thesis with information about future work (chapter 9) before drawing a conclusion (chapter 10).

2 QDAcity

QDAcity-RE is a method for domain modeling in Requirements Engineering (RE), which is based on QDA. The method is performed by sampling stakeholders, interviewing them, correlating other materials, and performing QDA on the materials to derive the *code system*. The code system represents an abstraction of the qualitative data. This process is done iteratively, which incrementally refines the code system until saturation is reached. The code system is a hierarchical structure consisting of *codes*, that are defined by the user. Codes capture concepts, categories, their properties, and interactions (Kaufmann & Riehle, 2015, 2019).

Figure 2.1 shows a screenshot of the coding editor of QDAcity, the tool that implements this method. Using this coding editor, a user can perform the QDAcity-RE method. The view consists of three parts, which are numbered accordingly in the screenshot:

1. Documents list: A project can consist of several documents, which are accessed via this document list.

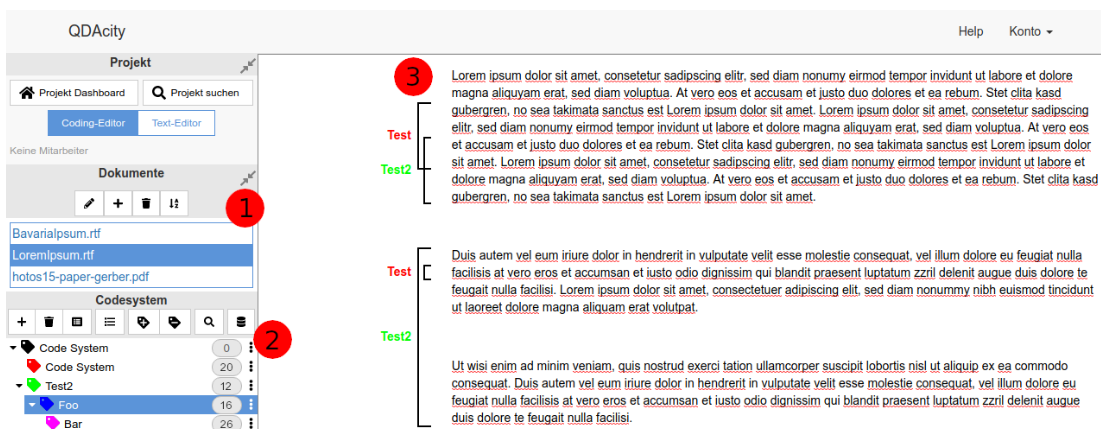


Figure 2.1: Coding editor view in QDAcity

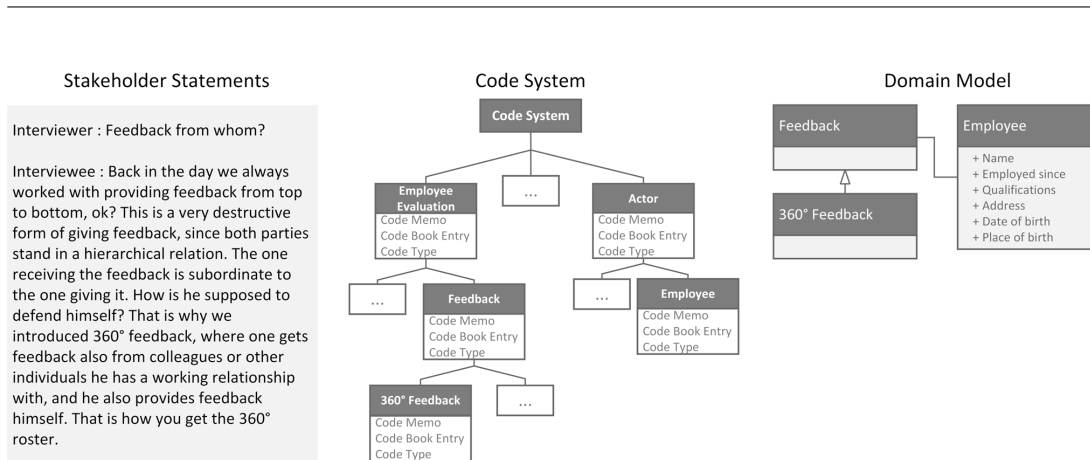


Figure 2.2: Code system example by Kaufmann and Riehle (2019)

2. Code system view: Allows the user to build a tree-based structure of codes. For each code, several properties as e.g. when to use the code can be specified.
3. Editor with the so-called *coding brackets* on its left side: Allows the user to code and edit a document.

The coding brackets visualize, where codes are applied in the document. Coded document segments are called *codings*. Additionally, when clicking on a coding bracket, the coded text gets highlighted.

The code system can be used to derive models describing different views on the domain. Kaufmann and Riehle (2019) deducted a study in which they derived a Unified Modeling Language (UML) class diagram from the code system. The coding editor of QDAcity includes an UML editor for this purpose. Figure 2.2 shows, how the code system bridges the gap between natural language text containing stakeholder information and an analysis domain model. The user builds the code system by coding one or more text segments, therefore linking the text segments to a code. The code system then can be used to derive analysis domain models in e.g. UML.

3 Previous work

In a student project preceding this thesis, we analyzed multiple aspects of different implementations for PDF support in QDAcity. This helped to determine the tasks to accomplish the goals of this thesis. The following sections cover a comparison of QDAcity with other QDA software tools and an analysis of their implementation of PDF support. Afterward, we evaluate different strategies to implement PDF support.

3.1 Comparison with other QDA software

MAXQDA[1] is a commercially available program to perform QDA. It supports various types of documents including PDF documents, which makes MAXQDA a reference for a possible implementation of PDF support in QDAcity. However, there are some key differences compared to QDAcity:

- QDAcity is a web application, while MAXQDA runs locally.
- QDAcity aims for collaborative editing. MAXQDA allows working in teams, but project files have to be split and merged afterward to simultaneously work on a single project (Gerson, 2016).

Despite these differences, the implementation of MAXQDA for supporting PDFs is still interesting. In particular, the management for PDF codings in MAXQDA can provide information for the implementation in QDAcity. The following section outlines how a coding mechanism as integrated in MAXQDA could be implemented.

3.1.1 Analysis of coding mechanism design

Since we do not have access to the source code of a QDA software tool that supports PDF documents, we can only make assumptions about possible coding mechanism implementations. The supported document types of MAXQDA

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

include various office document, image, audio, and video formats[2]. As these document standards are very diverse, it is safe to assume that the coding data is stored apart from the documents. Otherwise, QDA tools could either (1) alter the document standards for the supported document types or (2) convert the documents in a proprietary format in order to embed coding information. However, both options seem significantly more challenging than designing a coding mechanism, whose data can be stored apart from the documents.

An abstract coding mechanism that can handle multiple different document types would require the following information in every coding:

- A reference to the document in which the coding is applied
- A reference to the code to which the coding belongs.

The above set of attributes could be extended by unambiguous references to the coding start and end positions dependent on the coding type. For text codings, additional attributes could be the *index* of the paragraph and the *offset* of characters within the paragraph for both start and end positions. For image codings, additional attributes could be the x and y coordinates as well as *width* and *height* of the coding. For audio and video data, *timestamps* could be used.

3.1.2 Handling editable documents with an abstract coding mechanism

The major challenge of a coding mechanism as outlined in section 3.1.1 is handling editable documents. Since the coding references are stored apart from the document, the application has to track coding positions when the user edits the document. If codings were embedded in the documents, the coding positions would automatically be adjusted by the user editing the document. However, we consider extending the implementation of an abstract coding mechanism to be less time-consuming and cleaner than to fit different document types to a specific coding mechanism.

3.2 Evaluation of different strategies to integrate PDF support

Based on the previous findings, we considered different implementation approaches for PDF support in QDAcity. The following sections will evaluate these implementation approaches.

[2] https://www.maxqda.com/help-mx20/import/data-types

3.2.1 Backend based approach

The idea is to use existing frontend code while extending the backend to convert PDF documents to Hypertext Markup Language (HTML). Since QDAcity already supports HTML documents, this would minimize the effort for changes in the application. Furthermore, the existing coding mechanism could be reused, which is closely tied to the HTML document. We analyzed various libraries for PDF processing but found they lack the desired functionality. A brief overview of all evaluated libraries can be found in appendix A.

Although many libraries allow the extraction of text and media, most of them are not able to extract the layout of the document correctly. A promising library is Apache PDFBox[3] because it is open-source, feature-rich, actively developed, and well documented. However, in an exemplary Java project, it shows that this library has difficulties working with e.g. multi-column documents like scientific papers.

Since Apache PDFBox provides several sample binaries for various functions such as text extraction, the question arose whether these binaries could be easily improved. The responsible class for the position of extracted text was found to be *TextPositionComparator.java*[4]. The comparator uses the text position coordinates to sort the found text in the order from top to bottom and from left to right, analogous to how a human would read a single-column left-to-right document. Consequently, it is possible to write a custom comparator that can handle multi-column layouts by comparing e.g. pixel padding between words and lines. This idea however was discarded because:

- Differentiating the columns may be feasible, but implementing a comparator that can handle every possible layout seems unreasonable and perhaps not even possible.
- Recognition of specific text passages relies on the text formatting, too. Particularly in scientific papers, a wrong representation of the former block set alignment makes it difficult to recognize certain text passages.
- Many PDFs embed tables or other figures as vector images or some kind of non-pixel-images, in which text can actually be recognized as text. A distinction between continuous text and text in these elements also seems challenging.

Since all these aspects would be major drawbacks concerning our goals, the backend-based approach was discarded.

[3]https://pdfbox.apache.org/

[4]https://github.com/apache/pdfbox/blob/trunk/pdfbox/src/main/java/org/apache/pdfbox/text/TextPositionComparator.java

3.2.2 Frontend based approach

The second implementation strategy to support PDFs was to extend the frontend implementation while also adjusting the backend application to support multiple document types. The goal is to design a backend architecture that allows the upload of supported documents and can be extended to support additional document types. Furthermore, the frontend needs to be equipped with an additional editor view to display PDFs and allow coding of such. However, most important is to decouple the existing coding mechanism from the HTML documents, as we will not be able to include the coding marks in the PDF document for several reasons:

- The inclusion of coding marks in the PDF document would require to implement a non-standard extension to the PDF standard.
- PDFs can become large. When storing coding data inside the document, the entire document has to be sent to the server to issue an update. Since QDAcity's goal is to enable collaborative work on a document, this step has to be performed as soon as coding data has changed. Additionally, collaborative work on a document requires not only to send the document to the server but also to all other users for them to see the changes.
- Inserting codings into the PDF leads to more processing work. This has to be done on the client, as it would not only have to parse the PDF when it was updated but also create a new PDF when changing codings.

A better alternative for the backend application is to provide an endpoint to process codings for different document types, that could be designed similar to the implementation outlined in section 3.1.

In order to display PDFs in the browser and work with them in the desired way, the possibilities had to be analyzed. During the evaluation of JavaScript libraries for PDF processing and rendering, we discovered that there are only a few options for this task. All evaluated libraries are listed in appendix B. Eventually, the decision to use PDF.js[5] was made. It is provided under an open-source license, is actively developed by Mozilla, and is also included in their browser (Firefox[6]). Therefore, it can be considered reasonably stable and safe for use in a production application. It is even used in larger projects such as Nextcloud[7].

PDF.js is focused on rendering a PDF into a web page and not enabling to edit it or create new PDF documents. Furthermore, PDF.js is not a pre-built and ready-to-use PDF viewer, but a toolkit to build a custom PDF viewer. However, a sample viewer implementation is also included, which is equipped with many

[5]https://mozilla.github.io/pdf.js/

[6]https://www.mozilla.org/de/firefox/

[7]https://nextcloud.com

(a) Example for good overlaying performance

(b) Example for worse overlaying performance

Figure 3.1: PDF.js overlays the text of a PDF transparently (highlighted blue here) on top of the text that is rendered into the canvas

features such as zooming and displaying page thumbnails. PDF.js' approach on rendering PDFs can be split into at minimum 3 steps:

1. Create HTML container elements for the PDF pages.
2. Render each PDF page to a separate canvas and insert those into the page containers. This includes the entire content of a PDF page (images as well as text).
3. Extract text passages, create HTML elements for the passages and overlay the canvases in the page containers with the text passages.

PDF.js modifies the overlaid text to be transparent because its format does not perfectly match the text rendered in the canvas underneath. This way the text is still selectable via mouse, but the user always sees the correctly formatted text. Figure 3.1 shows a comparison of the performance of this mechanism by coloring the transparent text overlay in blue.

A big advantage of PDF.js regarding this mechanism is that it can easily be improved using an own implementation. The implementation used in figure 3.1 is the default implementation provided by Mozilla.

Additional but optional PDF.js rendering steps are to e.g. also extract PDF annotations (notes) and overlay those on top of the canvas. During testing, the clean HTML structure of the rendered PDF was found to allow easy modification and extension with various features. The capabilities of PDF.js as well as the easy customization of it gave it the advantage over other evaluated libraries.

3.3 Findings

With the insights from section 3.1 and section 3.2 we decided to extend QDAcity so that PDF.js is used to display PDFs in the frontend. The backend will be enabled to handle coding data similar to how the mechanism is outlined in section 3.1. The reasons for this decision are:

- Other QDA software tools prove the approach to be viable by seemingly implementing a coding mechanism using absolute references.
- Equipping the frontend as well as the backend with interfaces for easy expandability increases the ability to support additional document types in the future.
- No conversion of PDF documents into a different format has to be performed. Therefore, this should result in the best possible user experience.
- The approach unifies the coding mechanism. Since all documents can use the same coding mechanism, no special handling for PDF coding data has to be implemented.
- The impact of the coding mechanism on network traffic can be reduced as it will no longer be necessary to transmit the entire document.
- PDF.js allows easy customization. Therefore, it is possible to implement a custom PDF viewer that contains only necessary features.

4 Requirements

This chapter defines the requirements that have to be implemented in this thesis. We categorize the requirements into functional and non-functional requirements. In order to define explicit, complete, and testable requirements, all requirements were built using the templates by Rupp (2014).

4.1 Functional requirements

1. QDAcity shall provide the user the ability to code PDF documents collaboratively using the web browser.
   (a) QDAcity shall provide the user the ability to import PDF documents.
   (b) QDAcity shall be able to keep the layout of the PDF.
   (c) QDAcity shall be able to synchronize codings across multiple clients, in order to provide collaborative coding support.
2. The coding mechanism of QDAcity shall be able to handle current RTF documents as well as PDF documents.
   (a) The coding mechanism should be able to handle different coding types, in order to e.g. enable coding of non-text areas of PDFs.
   (b) The coding mechanism of QDAcity shall be able to handle editable documents.
   (c) The coding mechanism of QDAcity shall ensure coding references to be unambiguous, to allow an external coding mechanism.
   (d) The coding mechanism of QDAcity should not embed coding references into the content of the documents, in order to be independent of the document and be used for different document types.

4.2 Non-functional requirements

1. The implementation should provide developers the ability to extend the range of supported types of documents using pre-defined interfaces.
2. The introduced source code shall be documented extensively inside the source code and the Gitlab wiki.
3. The implementation shall include tests for modified and added functionality of QDAcity.

5 Architecture

This chapter outlines the software architecture. First, the necessary changes to support different document types are highlighted. The architecture of the overhauled coding mechanism is described afterward.

5.1 Document Support

5.1.1 Backend

As evaluated in section 3.2, the frontend-based approach requires changes in the frontend as well as in the backend. To support different document types, the backend must be extended to allow uploading and managing of these documents. Additionally, the frontend has to include logic to handle all document types accordingly.

Figure 5.1 outlines the architecture to support multiple types of documents. Previously, only RTF documents were supported by the implementation. As the RTF files are converted to HTML, the class implementing the document type was generically named *TextDocument*. We kept the name *TextDocument* for the new document support architecture. The class to represent PDF documents is called *PDFDocument*. Both types are represented by the abstract document type *BaseDocument*.

In order to inform the client about which types are supported, the *DocumentType* enum is advantageous. The enum lists constants for all document types and

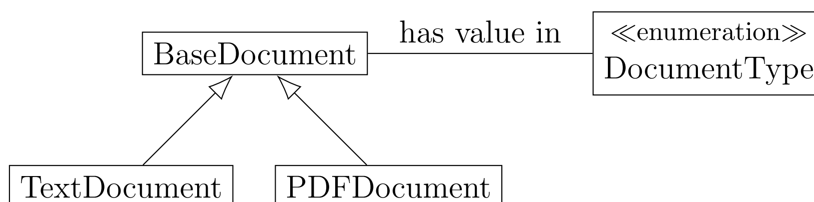


Figure 5.1: Architecture of backend document support

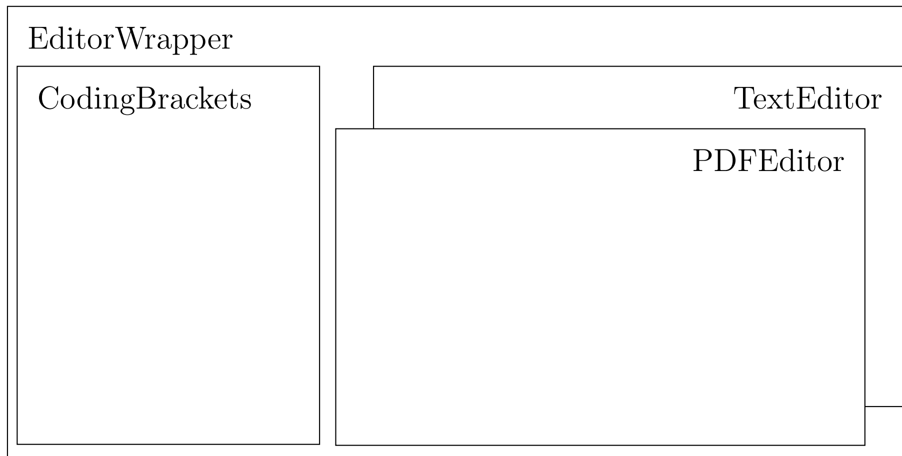


Figure 5.2: Architecture of frontend file support

associates them with a Internet Media Type (MIME-Type) and the corresponding document type implementation. Using this enum, the server provides the client with a list of MIME-Types that can be used to filter the file picker dialog for only supported documents.

The outlined architecture has many advantages:

- It allows to hide document type specific implementation details from the client. While RTF documents are editable, PDF documents are not. This difference is handled by the specific implementation of the document type but abstracted through *BaseDocument*.
- Since every document that passes the Application Programming Interface (API) is an instance of *BaseDocument*, the client can expect all objects to have a certain set of minimum properties. This can be advantageous for parts of the client, that only need e.g. the title of the document. To differentiate between the document types, API objects carry a *type* property. Using the *type* property, the editors can filter for the desired document type in all received objects and access document type specific properties.
- Implementation of further document types is straightforward. After creating an additional document type class that extends from *BaseDocument* and defining a new *DocumentType* enum constant, it can be handled by the API. The API will not have to be changed, as basic document handling is already in place.

5.1.2 Frontend

Figure 5.2 presents the architecture of the frontend document support feature. Previously, only *TextEditor* existed that also held *CodingBrackets*. Because the

coding brackets will be necessary for every editor, *CodingBrackets* was moved out of *TextEditor*. In order to abstract from all different document types, *EditorWrapper* was introduced. The *EditorWrapper* follows the Facade design pattern (Gamma, 1995) as it defines an interface that has to be implemented by all specific editors. If the application outside of *EditorWrapper* has to get data from the editor, it will call the appropriate method on *EditorWrapper*. The *EditorWrapper* then passes the call to the currently active specific editor implementation. The Facade design pattern allows us to hide the complexity of the editors from the application outside of the coding editor.

Analogous to the existence of different document type implementations in the backend, the frontend will use different editor implementations according to the type of the document. Which editor implementation is currently active is determined by the *type* property of the document object.

In addition to document type handling, the *EditorWrapper* also implements logic to display the coding brackets. The coding bracket feature is part of the interface that has to be implemented by all editors. The *CodingBrackets* require values for their height and offset in the vertical dimension. By including this feature into the *EditorWrapper* interface, the specific editor is required to convert coding data to the appropriate bracket values. Outsourcing the conversion logic into the specific editor implementation allows every editor to define a different set of coding properties.

When expanding the range of supported documents in the future, the outlined architecture only requires the implementation of a new editor that implements the interface of *EditorWrapper*.

5.2 Coding mechanism

Previously, the coding mechanism was closely tied to the documents. As QDAcity only supported RTF documents that were converted to HTML, it was possible to design a coding mechanism that embeds the codings inside the document. Hence, codings inside the document were represented by pseudo HTML elements. Embedding the codings inside the document had the following advantages:

- The coding positions were unambiguous by design.
- Editing the document did not have side-effects on codings. When adding text in the range of an applied coding, the coding range was automatically increased.
- Synchronizing the document across clients also synchronized the codings.
- No separate handling for codings on the backend was needed.

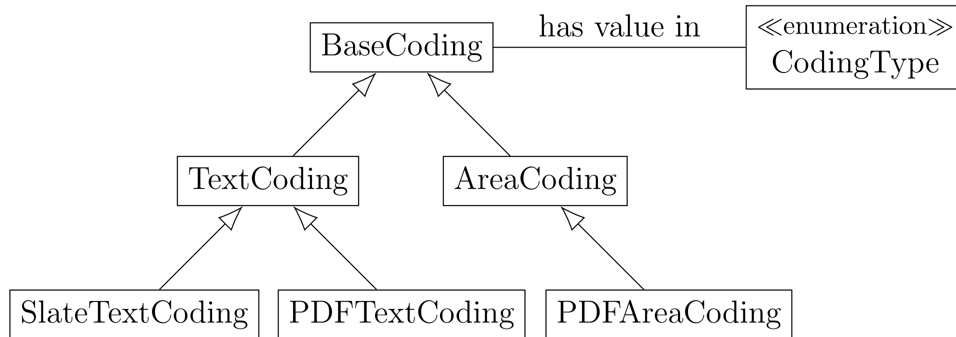


Figure 5.3: Architecture of the coding mechanism

However, this coding mechanism also had major disadvantages:

- Non-HTML documents could not use the coding mechanism.
- Documents have to be parsed by the client to e.g. display the count of applied codings in the code system.
- To get all codings for a project, e.g. to display the coding count in the code system, all documents of the project have to be fetched from the server.

Because of the above disadvantages, we decided to implement a new coding mechanism that allows to handle the coding of different document types. Figure 5.3 shows the proposed coding mechanism architecture. The core idea is similar to the architecture of supporting different document types. For every type of coding, a coding class has to be implemented. Since both *TextEditor* and *PDF-Editor* require text codings, a further abstraction level called *TextCoding* was implemented. This allows the *TextEditor*, which is using the *Slate* editor, to use its own coding object structure while the *PDFEditor* can do likewise. But PDFs also require to allow codings based on area selections over e.g. images. For this purpose, *PDFAreaCoding* was implemented based on *AreaCoding*, which allows future codings to use *AreaCodings* as parent implementation.

All coding types are abstracted through *BaseCoding*, which allows managing all kinds of coding objects by the backend. To differentiate between the coding types, the objects include the *type* property, which holds the appropriate constant of *CodingType*. The *CodingType* enum lists constants for every coding type and maps it to the specific coding implementation.

The abstraction allows the client to always access shared coding properties like e.g. its ID or the ID of the document the coding was applied to. At the same time, the specific editor can access all document type specific coding properties that are needed to work with the document.

6 Implementation

In this chapter we describe in detail, how the components outlined in chapter 5 are implemented. The key aspects of the components and their interaction will be explained. Since QDAcity is a Java-based Google App Engine application that uses JavaScript and React for the frontend, the necessary modifications were implemented using these technologies.

6.1 Document Support

6.1.1 Backend

Figure 6.1 shows the class layout of the feature to support multiple document types. Since all documents will be stored in a database, each type of document has to be serializable. The abstract class *BaseDocument* implements the *Serializable* interface and all shared properties between documents. These properties were taken from the previously existing *TextDocument*, which was the only supported document type. However, the *text* property was modified. The property's data type previously was *Text*[1], as it only needed to contain HTML text. In order to store the document data regardless of its form, the data type of the *text* property was changed to *Object*. Thus, every document class can store its content data inside *text*, as long as it is serializable.

The instance methods of *BaseDocument* were taken from the previous *TextDocument*, too. As the datatype of *text* changed, so were the return type of *getText()* and the parameter type of *setText()*. Furthermore, *BaseDocument* implements two abstract methods to be overwritten by child classes:

- *isEditable()*: This method allows to specify if documents of this type are editable by the user. The return value will be output to the API, so that the client can read this property and allow or deny editing of the document accordingly. Hence, the control over which type of documents appear editable to the user is in the backend.

[1] *com.google.appengine.api.datastore.Text*

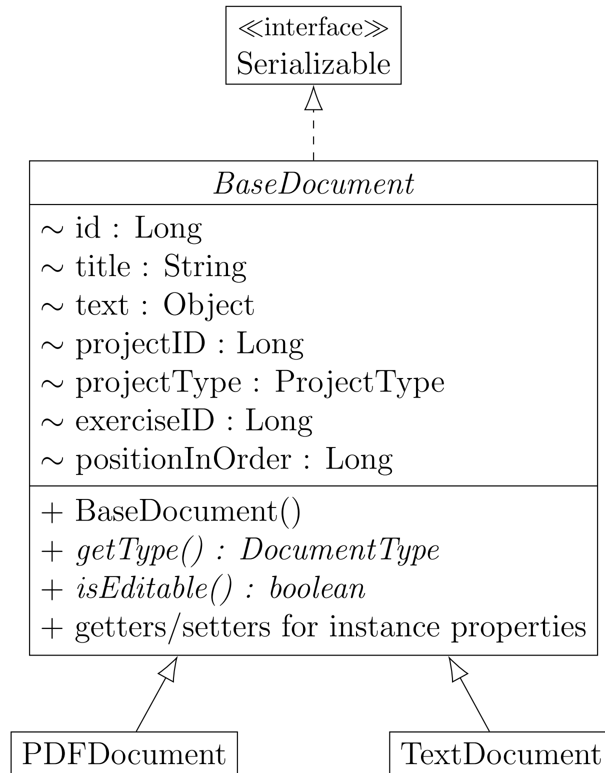


Figure 6.1: UML class diagram of the document architecture

- *getType()*: This method is required to return the enum value of *DocumentType* according to the document class. Since the returned value will also be output to the API, the client can easily differentiate between documents of different types.

The *DocumentType* enum however serves more purpose than to just signal the type of document to the client. Figure 6.2 shows the properties and values of the *DocumentType* enum.

- *mimeType*: The MIME-Type is used to map uploaded documents to the correct document class. Compared to specifying the filename ending, it allows using the enum value for all filename endings belonging to the same type of document.
- *documentClass*: This property is used to derive the desired document class from the MIME-Type of the uploaded document.
- *mimeTypes*: This static map serves as a cache to quickly gather the enum value corresponding to a given MIME-Type. This map is useful for when the server e.g. receives a newly uploaded document to decide which document implementation to use based on the MIME-Type of the document.

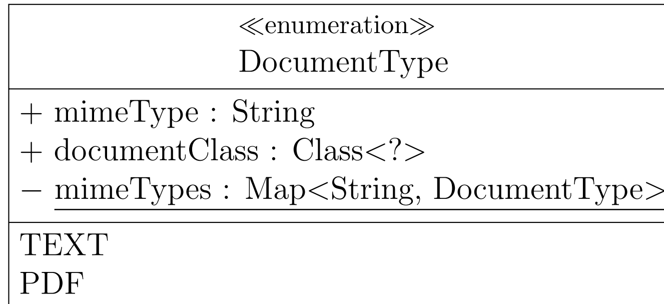


Figure 6.2: UML class diagram of the *DocumentType* enum

Consequently, every *DocumentType* enum value is bound to a MIME-Type and a document class. Using the static property *mimeTypes* and the additional endpoint route *getAllowedMimeTypes()* for *UploadEndpoint*, the client is able filter for supported files in the 'open file' dialog.

The *DocumentType* enum also helps to deserialize a JavaScript Object Notation (JSON) object sent by the client into the correct document implementation. As mentioned earlier, all document objects exchanged through the API are instances of *BaseDocument*. The received JSON objects could therefore be easily deserialized into a *BaseDocument*. But as *BaseDocument* is an abstract class, no instances of it can be created. Furthermore, the dynamic type of these deserialized objects would also be *BaseDocument*. Hence, we would lose the information about which type of document was sent.

For deserialization of abstract objects into the correct child instance, a custom *TypeIdResolver* is needed. Figure 6.3 shows that *DocumentTypeIdResolver* and *CodingTypeIdResolver* are children of the abstract *OwnTypeIdResolver*. This prevented code duplication for the custom *TypeIdResolver* that is also necessary for deserializing codings which will be addressed later. The purpose of *DocumentTypeIdResolver* is to derive the corresponding document class from the value of the *type* property of the received JSON document object. Afterward, the document class is used to deserialize the received JSON into an object of this document class.

With the *DocumentType* enum, the addition of a future document type is seamless:

1. Create a new child class of *BaseDocument*
2. Add an additional enum value for that document type, mapping a MIME-Type and the document class.

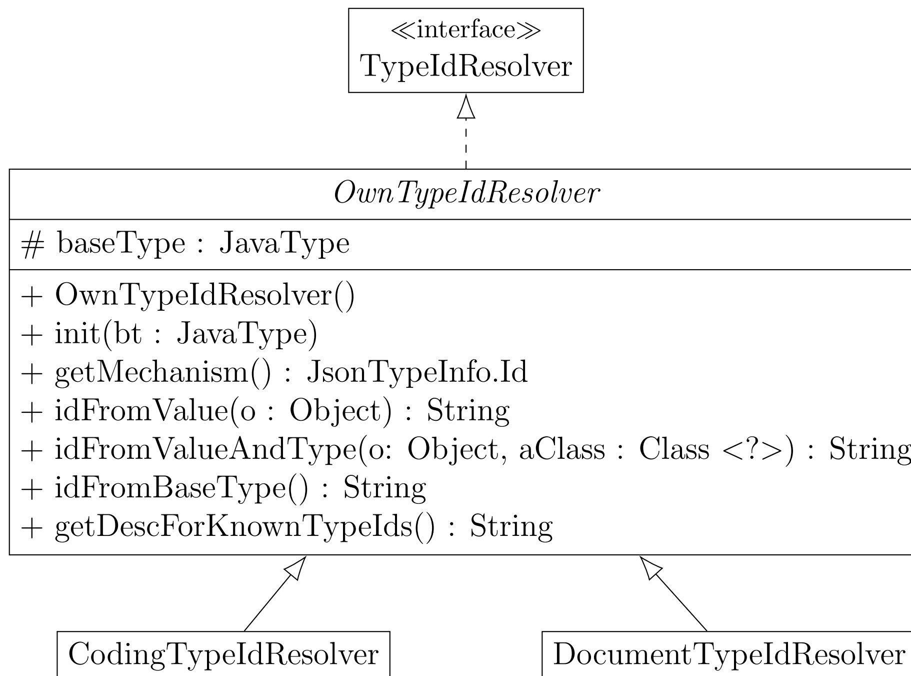


Figure 6.3: Custom *TypeIdResolvers* help to deserialize JSON objects of a common base type into the correct child implementation

6.1.2 Frontend

To support multiple document types in the frontend, we decided to implement the *EditorWrapper* facade. The *EditorWrapper* has many purposes:

- Abstracts from different editor implementations. The facade implements necessary methods for QDAcity to interact with documents
- Handles the coding brackets across different editors
- Triggers the request to save a document.

For the application outside the coding editor to interact with a document, the *EditorWrapper* implements the following methods:

- *getDocumentContent*: This method can be used by QDAcity to get the content of the currently active document.
- *addCodingForSelection*: When the user presses the button to add a coding in the code system toolbar, this method is called.
- *removeCodingForSelection*: When the user presses the button to remove a coding in the code system toolbar, this method is called.

- *activateCodingInEditor*: When the user triggers a function that is supposed to highlight the coding in the document, this method is called. QDAcity e.g. provides the user with a table of codings that are present in a project. If the user clicks on a coding in this list, the appropriate document with the highlighted coding will be displayed.

All of these methods act as bridges between QDAcity and the specific editor implementations. Hence, the *EditorWrapper* expects the editor implementations to implement these methods. This allows every editor implementation to handle calls to these methods differently, depending on the type of document.

In order to handle the coding brackets, the *EditorWrapper* expects the editors to implement one more method: *emitCodingBracketData*. As outlined in section 5.2, each document may require different kinds of coding objects. To calculate the height and top offset of all coding brackets for the active document, the *EditorWrapper* calls this method from the active editor. The specific editor will then calculate the coding bracket properties and emit them to the *EditorWrapper*. Then, the *EditorWrapper* can pass the list of coding brackets to the *CodingBrackets* component, which will render the coding brackets.

The *EditorWrapper* registers event listeners that will trigger a re-evaluation of the coding bracket data:

- When a different document is selected
- When codings are changed
- When entering the edit-mode of a document (the editor might render a toolbar on top of the document, which requires the top offset of the coding brackets to be adjusted properly)
- When the window size is changed, as text then might flow differently.

In the case of editable documents, the coding bracket should reflect changes in the codings immediately. If a user e.g. adds or modifies text that affects a coding in its position and/or length, this should be immediately reflected by the coding bracket. We decided not to send an update request every time a coding has changed, because it would increase the number of database transactions and network traffic. Also, intermediate coding updates are not of use for the server as long as the document is not yet saved. Instead, we implemented a caching functionality into the *EditorWrapper* shown in figure 6.4.

When the user edits the document, the editor is supposed to pass changed codings to the *EditorWrapper*. The *EditorWrapper* saves these changed codings in a local list. The *EditorWrapper* then passes the original list of codings merged with the local list of updated codings to the editor (step three in figure 6.4). As shown in the diagram, this results in steps three through four being the same as steps one

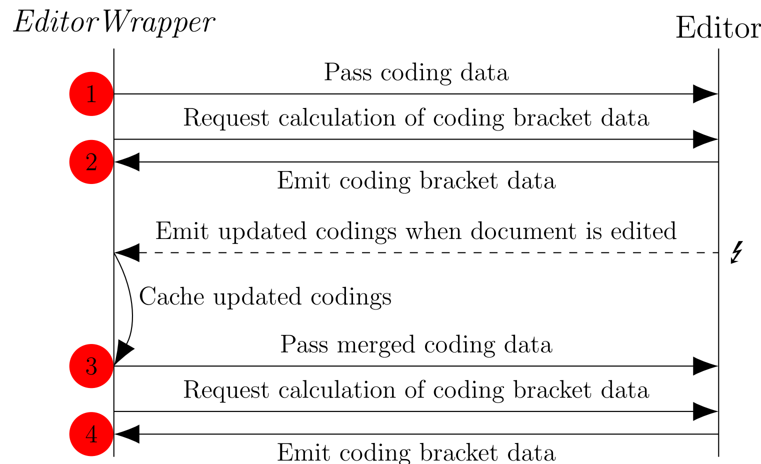


Figure 6.4: Sequence diagram of *EditorWrapper* coding caching functionality

through two. Hence, we are able to re-use existing logic of the editors to calculate the coding bracket data. The cache will be flushed when the document is saved.

When to save a document is also controlled by the *EditorWrapper*. Previously, the document was only saved when switching documents, leaving the edit-mode, or leaving the coding editor. Concerning usability, this implementation was flawed because users had to know how to trigger a save of the document. For example, it could happen that a user closed the browser tab or window expecting QDAcity to have saved the document, but it has not. In the current implementation, the *EditorWrapper* registers a listener for the *blur*-event on the current editor's Document Object Model (DOM) node. The *blur*-event will be triggered every time the editor loses focus. When the *EditorWrapper* receives the *blur*-event, it will trigger a request to save the document and possibly changed codings. Hence, a document will be saved every time the user might stop working on a document. In order to save all changed codings alongside the updated document, the *EditorWrapper* just attaches the cached list of codings to the save request.

6.2 Coding mechanism

The coding mechanism is inspired by the findings in section 3.1. Building a coding mechanism using external, but unambiguous references allows to design a coding mechanism to support multiple different type of documents. External references are those that not embedded as e.g. marks in the document itself. Figure 6.5 shows the structure of several coding classes.

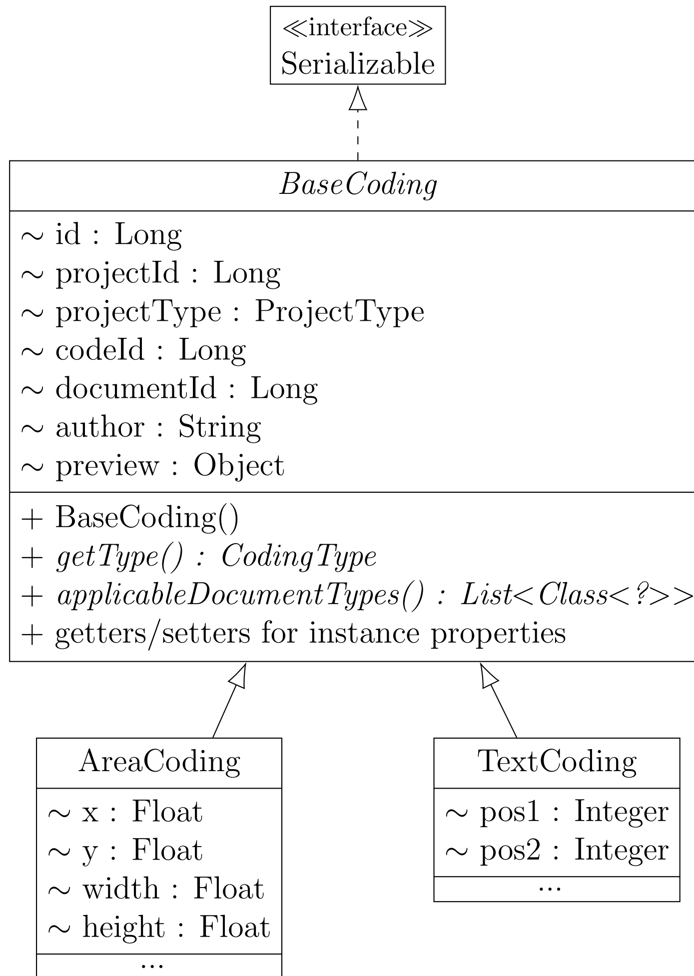


Figure 6.5: UML class diagram of the coding architecture

Since each coding object will be stored in a database, it has to be serializable. Apart from the *Serializable* interface, *BaseCoding* also implements properties and methods to be used by all codings. Every coding needs:

- *id*: Unique identifier
- *projectId*: Reference to the project
- *projectType*: The type of the project
- *codeId*: Reference to the code to which the coding belongs
- *documentId*: Reference to the document in which the coding was applied
- *author*: The name of the user who created the coding
- *preview*: Attribute to hold a preview of the coded document segment.

Since the *projectId* and *projectType* would be indirectly specified through the *documentId*, because each document belongs to a project, the properties could be omitted. However, we decided to include the properties in the coding data, as filtering for specific codings is easier if the properties are directly accessible.

Because a specific coding type is only applicable to certain type of documents, *BaseCoding* implements the abstract method *applicableDocumentTypes()*. A child implementation is required to overwrite this method and return a list of applicable document classes. In the current scenario, each coding class will only apply to a single document type. However, we decided to define this method to return a list of document types, as a more abstract coding type might be used for multiple document types.

For PDF support in QDAcity, two types of codings are needed: text codings and area codings. Text codings are used for user-selectable HTML text, while area codings are used for rectangular selections that can be created over e.g. images. This enables the user to code both text and areas inside a PDF. The *AreaCoding* and *TextCoding* classes in figure 6.5 represent these two types of codings. However, they are not directly used for PDFs. Because PDFs are paginated documents, we instead decided to build a more granular structure of coding classes. This enables to derive further coding classes from direct children of *BaseCoding*. Hence, the *AreaCoding* and *TextCoding* are used as parent implementations for RTF codings and PDF codings of both types.

The *AreaCoding* consists of four properties: x and y coordinate, width, and height. These properties are stored as floating-point values because they are relative to the document. Depending on e.g. the width of the rendered PDF, an area coding is placed in a certain spot. Using absolute pixel values for positioning would render the area coding in another spot when the document width is changing. This applies to all other properties of area codings. Concerning area codings in PDFs, the single additional property required is the page number.

The *TextCoding* consists of two properties: *pos1* and *pos2*. These properties describe the start and end position of a coding. They were named generically because they are supposed to be used by different child implementations which might have different naming conventions for these properties. The library Slate for example, which is used in QDAcity to work with RTF documents, uses the naming convention 'anchor' and 'focus' to describe the start and end position of a text selection.

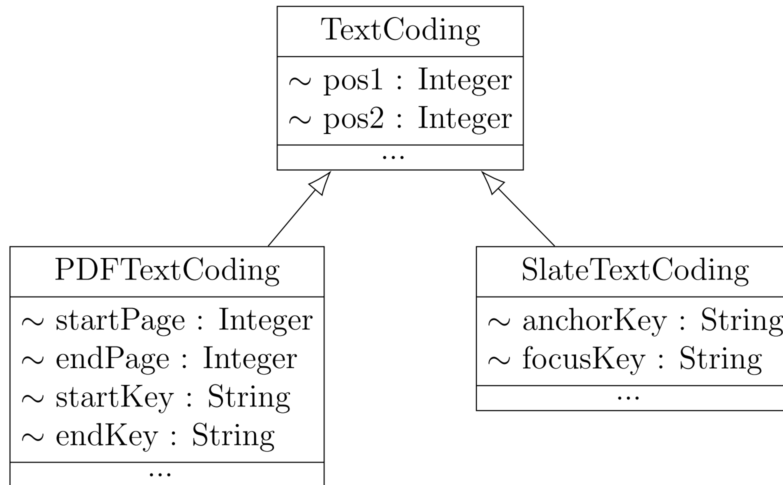


Figure 6.6: UML class diagram of *TextCoding*

Figure 6.6 shows the class structure for *PDFTextCoding* and *SlateTextCoding*. A coding in Slate requires two properties per start and end position of a coding:

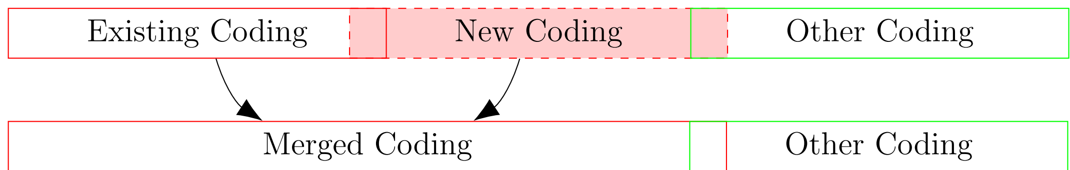
- Coding start: *anchorKey* & *anchorOffset*
- Coding end: *focusKey* & *focusOffset*.

The keys are used to reference a specific text node in the document. Combined with the offset values, it is possible to reference an exact position in the document. *Pos1* and *pos2* from *TextCoding* are reused as *anchorOffset* and *focusOffset*.

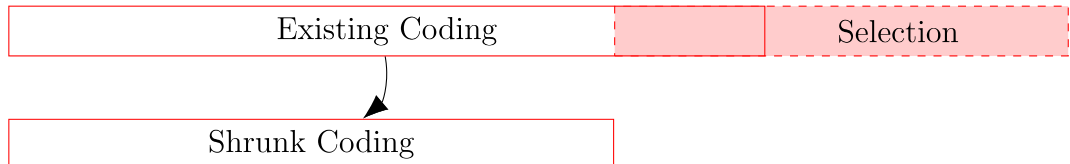
The *PDFTextCoding* defines *startKey* and *endKey*, too. As with Slate, these keys reference certain text segments within a PDF. However, PDF.js does not apply keys to the HTML text elements by default. Therefore, the implementation of PDF.js for generating the HTML text overlay layer had to be extended. Each HTML element that directly contains PDF text is assigned a key stored in the properties of the coding. The keys are numeric and increment for each text element.

6.2.1 Side-effects of new coding mechanism

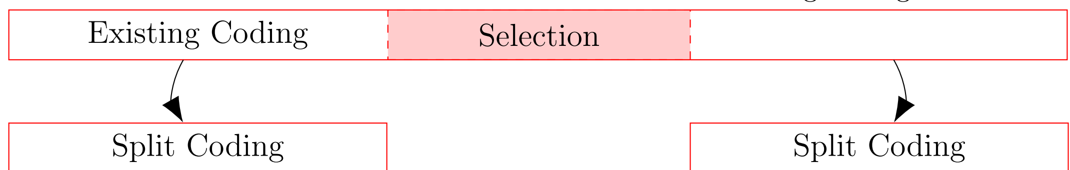
Because the new coding mechanism no longer uses HTML elements that are embedded into the documents to describe codings, coding boundaries are more difficult to compare. However, we need to compare coding boundaries when the user applies a new coding or removes a coding. When adding a coding, the coding could be merged with an existing coding of the same code if their ranges overlap. Merging codings allows the user to extend the range of an existing coding. When removing a coding, an existing coding of the code could be split or shrunk. These operations require comparing the boundaries of existing codings with the range of



(a) Codings are merged when adding a coding if they belong to the same code (denoted through border color) and are overlapping.



(b) A coding is shrunk when removing a coding if the existing coding overlaps the selection and the selection is set to the same code as the existing coding.



(c) A coding is split when removing a coding if the existing coding surrounds the selection and the selection is set to the same code as the existing coding.

Figure 6.7: Illustration of coding merging and splitting mechanism

the selected text. Figure 6.7 illustrates the different scenarios for coding merging, shrinking, and splitting.

We use the DOM Range API[2] to get the current text selection and to construct codings based on it. As the API provides the method *compareBoundaryPoints*, we can implement a frontend utility class to compare codings with the current text selection. For this purpose, the implemented class *TextCodingUtils* requires following parameters:

- *code*: The code of the coding to apply/remove
- *range*: The DOM range of the current text selection
- *allCodings*: All codings of the document
- *allRanges*: A key-value mapped list of coding IDs to the DOM range of the coding.

The specific editors are by design not required to hold the DOM range representation of the codings for the editor to work. However, the editors must now convert the codings to DOM range objects before passing them to *TextCoding-*

[2]https://developer.mozilla.org/en-US/docs/Web/API/Range

Utils. But since this conversion is the opposite of when a new coding object is created from the current selection's DOM range, every editor must be able to handle this conversion.

The *code* parameter was primarily introduced to prevent code duplication. The parameter is used by the utility class to filter for affected codings in the *allCodings* list. This filtering would have to be done by every editor if it was not included in *TextCodingUtils*. By filtering for affected codings in *allRanges*, the utility class will get the coding IDs of all codings whose boundary points have to be compared. With the list of coding IDs, the utility class now can get all DOM ranges from *allRanges*, that were converted by the editor beforehand.

The comparison that takes place in *TextCodingUtils* checks for the cases illustrated in figure 6.7 using the *compareBoundaryPoints* method. During the comparison process, the utility class will construct three lists of DOM ranges:

- *codingsToCreate*: List of DOM ranges of codings to create. If the detected case requires new codings to be created, this list gets populated.
- *codingsToUpdate*: List of DOM ranges of codings to update. If the detected case requires existing codings to be updated, this list gets populated.
- *codingsToDelete*: List of DOM ranges of codings to delete. If the detected case requires new codings to be removed, this list gets populated.

Depending on the detected case, *TextCodingUtils* computes the resulting DOM ranges based on the given DOM ranges of the codings. The utility class will then insert the new DOM range with the codings ID in the appropriate list. The three lists are used to construct a single batch, where every item is assigned with a constant of `CREATE`, `UPDATE`, `DELETE`. This batch is returned to the editor instance afterward. The editor loops over the batch and handles the creation of new coding objects as well as update and deletion of existing coding objects. Since every returned item holds the new DOM range, each editor can re-use logic that is already present to convert a DOM range into a coding object. With the *TextCodingUtils* relying on DOM range objects, the utility class is usable for every future editor implementation.

6.2.2 Synchronizing coding updates across clients

In order to collaboratively code a document, QDAcity implements a realtime-service. The realtime-service acts as a man-in-the-middle that forwards requests from connected clients to the server. After receiving the answer from the server, the realtime-service broadcasts this answer to all connected clients. In the legacy coding mechanism, the realtime-service was already able to synchronize coding changes across clients. Because the codings were embedded in the document, the entire document had to be serialized and sent through the realtime-service.

The new coding mechanism no longer embeds coding data in the documents. To handle the new coding mechanism, the realtime-service had to be refactored. Since we implemented an endpoint method that can receive coding batches to process, the realtime-service has to handle this method. The refactored code consists of the addition of:

- *CODING.BATCH* message
- *CODING.BATCHED* event.

The *CODING.BATCH* message is used to send coding updates from the client to the realtime-service. These updates can include the creation, update, and deletion of codings. The realtime-service uses the message to decide how to proceed with the request, as other parts of QDAcity might need additional steps to be performed. In our case, the realtime-service forwards the request to the server without performing additional steps. The received answer will be broadcasted to all connected clients using the *CODING.BATCHED* event. In the event listener for the *CODING.BATCHED* event, the client will merge the processed coding batch into its local state. Hence, every user will receive coding updates performed by a different client.

6.2.3 Migration of existing text documents

To use the new coding mechanism with existing documents, the documents had to be migrated. The implemented migration routine had to solve the following issues:

- Extract codings from text documents
- Create new coding entities with appropriate properties in the database
- Build a new text document that no longer includes coding HTML elements.

We used JSOUP[3] to parse the HTML of original text documents. Extracting coding data and building a new document using an iterative approach proved to be complex and challenging to implement. The main problem using this approach was to keep track of the applied codings for a given text segment. As coding HTML elements could be deeply nested, an upward search inside the DOM tree would be necessary to gather the applied codings. If codings were overlapping, the HTML coding elements were split in order to (1) describe the part of the text that only has one of the codings applied and (2) describe the part of the text that has both codings applied. However, the new coding entities that have to be created have to describe the entirety of a coding. Hence, a coding that is split into many parts in HTML should be detected and stored as one single coding.

[3]https://jsoup.org

```
1 <coding id="8" code_id="2" title="Test22" author="JL">
2     <coding id="7" code_id="3" title="Test" author="JL">
3         Lorem ipsum dolor sit amet, consetetur sadipscing
              elitr, sed diam nonumy eirmod tempor invidunt
              ut labore et dolore...
4     </coding>
5 </coding>
```

Code (6.1) Excerpt of original text document DOM tree

```
1 <span codingkey="2">
2     Lorem ipsum dolor sit amet, consetetur sadipscing
          elitr, sed diam nonumy eirmod tempor invidunt ut
          labore et dolore...
3 </span>
```

Code (6.2) Excerpt of migrated text document DOM tree

Figure 6.8: Visualization of original and migrated text document HTML excerpts

A better approach proved to be the Visitor design pattern (Gamma, 1995) that is already implemented in JSOUP[4]. The Visitor design pattern allows us to traverse the DOM tree of the text documents. Therefore, JSOUP requires to implement two methods in our visitor:

- *head*: This method is called when an HTML node is first visited (when traversing down the DOM tree).
- *tail*: This method is called when an HTML node is last visited (when traversing up the DOM tree).

Using these methods, we were able to implement a very compact and easy-to-understand algorithm to migrate existing documents. To handle codings that were split into many parts in the HTML structure, we used a hashmap that maps coding IDs to temporary coding entities. When a coding element with an unmapped coding ID is visited, we create a new empty coding entity and store it in the hashmap. We decided to store an empty coding because there still could be a nested coding that we have not visited yet. Hence, the direct child of an HTML coding element cannot be used to set the properties of the entity. Instead, we extend the empty coding entities every time a text node is visited.

Figure 6.8 shows an excerpt of an original text document and the equivalent migrated document excerpt. To keep track of the codings applied to a given text

[4] https://jsoup.org/apidocs/org/jsoup/select/NodeVisitor.html

segment, we use a stack. While traversing down the DOM tree, every time a coding element is visited we push its ID to the stack. Analogous we pop coding IDs from the stack when traversing up the DOM tree. With the stack, we are able to collect all coding IDs of applied codings for a certain text segment. Hence, when a text node is visited, we modify all coding entities referenced by the stack to include the current text segment.

To build the migrated text document, we have to remove all coding elements. The new text document is built as a copy while traversing the original document. Every visited node except coding elements is added to the new text document. As our introduced codings for text documents (*SlateTextCoding*) require coding keys for text segments, the text nodes are wrapped in a single *SPAN* element containing the coding key. Effectively, this results in a copy of the text document where text nodes are unwrapped from any coding elements. Instead, text nodes are surrounded by single *SPAN* elements containing the coding key for the text segment.

7 Unforeseen challenges

During implementation, some unforeseen challenges had to be overcome. To help future developers extend QDAcity, this chapter highlights the implemented solutions to these challenges.

7.1 Use `SUPERCLASS_TABLE` inheritance with DataNucleus

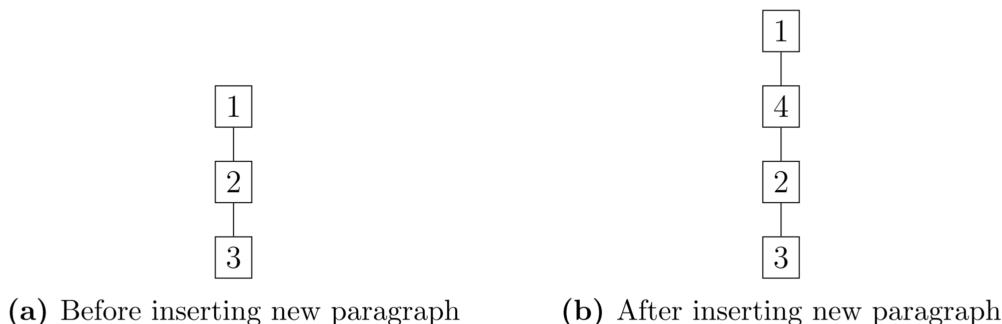
For the implementation of *BaseDocument*, *BaseCoding* and their child implementations, we required to use the Java Data Objects (JDO) inheritance strategy `SUPERCLASS_TABLE`. This inheritance strategy allows to store all objects of a common supertype in a single table. It is necessary to use a single table for all objects because scattering them into separate tables would require extensive queries to e.g. get a list of all documents or codings. With this inheritance strategy, however, we noticed unexpected behavior of QDAcity. When accessing the application for the first time after it was booted, no documents would show in the documents list. But after uploading a document, all existing documents of the same type were visible again.

As we could not find a reason for this behavior in any documentation, we debugged the server application in depth. We were able to find the problem in DataNucleus[1], which is used as a middleware to persist JDO objects in the Google Datastore. DataNucleus caches the child types of a supertype by remembering the relationship in a hashmap. For each supertype, a child type is added to this hash map after it was first instantiated[2].

Therefore we have to instantiate each child type that is persisted via `SUPERCLASS_TABLE` once to force DataNucleus to add it into its hashmap. Because every child type is mapped in the *DocumentType* and *CodingType* enums, we can loop over the enum values to instantiate a dummy object of every child type. We decided to implement this loop statically in our Persistence Manager Factory (PMF)

[1] https://www.datanucleus.org

[2] MetaDataManager.java#L1704 & MetaDataManager.java#L1734



(a) Before inserting new paragraph **(b)** After inserting new paragraph

Figure 7.1: Slate paragraph keys

singleton class, which we use to retrieve persistence manager instances for the database. Hence, the fix will be applied before the first database transaction occurs. Additionally, using the enums will apply the fix for future document and coding types. When however extending QDAcity with a new feature that also uses `SUPERCLASS_TABLE`, the code in PMF has to be extended to include all child types of the feature.

7.2 Use external codings with Slate

As mentioned in section 6.2, Slate uses keys to identify text nodes in a text document. The simplest form of a text node is a paragraph. But when e.g. applying text formatting, the formatted region is represented by a separate text node. We use the node keys in *SlateTextCoding* as reference to the start of the concerning text node. In order to enable editable documents, the key of a text node must not change. But when implementing the coding handling for editable documents, we found that Slate's keys for text nodes are not persistent. If e.g. a text document has three paragraphs, Slate numbers them with keys one through three. When the user now inserts a new paragraph after text node one by pressing enter, the new paragraph will be assigned key four. Figure 7.1 shows the resulting scenario.

After storing the modified document and reloading it, Slate will assign all keys in incremental order again. Hence, text node four will be text node two, text node two will be text node three et cetera. This inconsistency in keys introduces problems to our coding objects, that are stored separately. In this case, all codings of the former text node two will be shown in text node four. An attempt to modify our implementation of Slate's (de)serializer showed that our version of Slate did not allow full recursive control over all Slate nodes. Hence, we were not able to get the generated keys and persist them to the generated HTML, nor load keys from the HTML into Slate.

Updating Slate to the newest version (0.60) enabled full control over the document (de)serialization. The updated version however no longer uses keys to reference a text node inside the document. Instead, Slate nodes are referenced using paths. Although we could use paths instead of keys in *SlateTextCoding*, we decided against it. When using paths, every subsequent coding to a newly created paragraph would require an update. A mass update of codings could quickly introduce large requests and many database updates, depending on the document size and coding count. Therefore we decided to stick with the idea of persistent keys for text nodes. To implement this feature, it was necessary to extend Slate's (de)serialization mechanism and implement a plugin for Slate.

Our implementation initializes the next assignable text node ID of our plugin by parsing the highest text node ID from the HTML document. If a new text node is inserted, we hook into Slate's *apply* function and assign new text node IDs incrementally. The *apply* function is called by Slate when any operation is going to be applied to the document. During document serialization, every assigned text node key is persisted into a *SPAN* element that wraps the text of the node. During deserialization, we load the text node keys again and assign them to the appropriate text node.

8 Evaluation

In chapter 4, functional and non-functional requirements that have to be implemented in this thesis were defined. In this chapter, we evaluate the implemented solutions based on these requirements.

8.1 Functional requirements

Req 1: QDAcity shall provide the user the ability to code PDF documents collaboratively using the web browser

Req 1.a: QDAcity shall provide the user the ability to import PDF documents

This requirement is fulfilled. In order for the user to import PDF documents into QDAcity, some features had to be extended:

- The backend must support PDF documents.
- The upload modal must allow the user to pick PDF documents.

As described in section 6.1, an extensive class hierarchy was implemented for the backend to support PDF documents. The class hierarchy consists of *BaseDocument*, *TextDocument* and *PDFDocument*. *BaseDocument* was implemented as an abstract class for other document types to inherit from. *BaseDocument* already defines the main properties and methods that are shared by each document type. The former *TextDocument* was refactored to comply with the new class hierarchy.

To support PDF documents in QDAcity, the *PDFDocument* class was added. Using the *DocumentType* enum, the document class to instantiate is determined by the MIME-Type of the uploaded document. The enum is also used to pass a list of supported MIME-Types to the client. The list of MIME-Types is used to filter the file-picker dialog for supported files. Before this thesis, no filtering was used for the file-picker dialog.

Req 1.b: QDAcity shall be able to keep the layout of the PDF
This requirement is fulfilled. Our implementation does not alter the PDF document. Since the frontend uses PDF.js to render the content of the document on HTML canvases, the layout of the PDF is kept. Figure 8.1 shows the layout of a complex PDF in QDAcity compared to the Evince[1] PDF viewer.

Figure 8.1: PDF layout in QDAcity (left) vs. Evince PDF viewer (right)

Req 1.c: QDAcity shall be able to synchronize codings across multiple clients, in order to provide collaborative coding support
This requirement is fulfilled. In order for QDAcity to synchronize codings across multiple clients, parts of the realtime-service had to be refactored. Now, the realtime-service forwards a batch of codings to create, update and delete to the server. The realtime-service broadcasts the answer of the server to all connected clients. Hence, every client receives coding updates that were issued from a different client.

Req 2: The coding mechanism of QDAcity shall be able to handle current RTF files as well as PDF files

Req 2.a: The coding mechanism should be able to handle different coding types, in order to e.g. enable coding of non-text areas of PDFs
This requirement is fulfilled. For the coding mechanism to support different coding types, an extensive class hierarchy was implemented. The class hierarchy consists of *BaseCoding*, *TextCoding*, *AreaCoding*, *SlateTextCoding*, *PDFTextCoding* and *PDFAreaCoding*. Intermediate coding types were implemented to provide a base implementation for future codings types to inherit from.

[1]https://wiki.gnome.org/Apps/Evince

Figure 8.2: Visualization of support for different coding types

BaseCoding implements all main properties and methods that are shared by child implementations. Using this class hierarchy allows for every document type to use multiple different types of codings. Hence, both text, as well as area codings, can be used for PDF documents. Since the new coding mechanism replaces the old coding mechanism, RTF documents were migrated to support the new coding mechanism. Figure 8.2 shows a text coding (red) and an area coding (green) applied to a PDF document.

Req 2.b: The coding mechanism of QDAcity shall be able to handle editable documents
This requirement is fulfilled. The new coding mechanism no longer embeds coding marks into the document. Hence, we implemented an endpoint that can receive the updated document as well as updated codings. Currently, only RTF documents are editable in QDAcity. Our implementation requires the client to keep track of codings to update. We used built-in functionality of Slate to keep track of codings to update during user interaction.

Req 2.c: The coding mechanism of QDAcity shall ensure coding references to be unambiguous, to allow an external coding mechanism
This requirement is fulfilled. In order for external codings to work, coding references have to be unambiguous. Our implementation for unambiguous codings uses multiple properties for different coding types. For text codings, every para-

graph is assigned a key. We use this key in combination with an offset to specify the exact start and end positions of a coding. The implementation is similar to the Range Web API[2] that is used to describe ranges of text (e.g. a user selection) on a website.

For area codings, we use floating-point values to describe height, width, x and y coordinates relative to the document. The use of floating-point values is important, as all properties have to be described using percentages. With percentage-based values, area codings are going to be shown in the same location even if the displayed width of the document changes.

Additionally, PDF codings include a page index property. For PDF text codings, a page index has to be stored for the start and end positions of the document. Since area codings cannot span multiple pages, only a single page index is required.

Req 2.d: The coding mechanism of QDAcity should not embed coding references into the content of the files, in order to be independent of the document and be used for different document types
This requirement is fulfilled. As outlined in section 6.2, all coding properties are stored in a database using an extensive class hierarchy. However, we still had to embed some coding information in the document for RTF documents. In section 7.2, we justified the decision not to reference paragraphs using their index or the path provided by Slate, as this would require coding mass updates when editing the document. Instead, we assign keys to the paragraphs of an RTF document and use these keys in the text codings. Hence, when editing the document, only minimal coding updates have to be performed.

If resources allow in the future, embedding these keys could be replaced by e.g. using paragraph indexes. Afterward, coding references are truly no longer embedded in documents.

8.2 Non-functional requirements

Req 1: The implementation should provide developers the ability to extend the range of supported types of documents using pre-defined interfaces

This requirement is fulfilled. To support different types of documents as well as different types of codings, extensive class hierarchies were implemented. For documents, the *BaseDocument* class serves as parent class for future document types. Analogous, the *BaseCoding* class serves the same purpose for codings.

[2]https://developer.mozilla.org/en-US/docs/Web/API/Range

Additionally, the enums *DocumentType* and *CodingType* were implemented. For new document and coding types, those enums have to be extended by a single value. Afterward, all common issues like e.g. (de)serializing child types for use by the API are already handled.

In the frontend, the *EditorWrapper* that follows the Facade design pattern can be extended by new editor implementations. Hence, the *EditorWrapper* defines an interface to be implemented by all editors. Using the *EditorWrapper* facade, complex editor logic is hidden from the application outside of the editors. Additionally, there are multiple but limited methods available for the application outside of editors to communicate with the current editor.

The *EditorWrapper* also handles the coding bracket, that is used by all editors. Hence, not every editor has to render its own coding bracket. Instead, the *EditorWrapper* will receive normalized coding bracket data from the editors via the defined interface.

All above solutions were implemented targeting the ease to extend these features.

Req 2: The introduced source code shall be documented extensively inside the source code and the Gitlab wiki

This requirement is fulfilled. All newly added code is documented in the source code. For Java source code, the Javadoc[3] documentation style is used. For JavaScript source code, the JSDoc[4] documentation style is used. The issues addressed in chapter 7 were documented extensively including references to e.g. online sources. Furthermore, complex code snippets include explanatory comments.

Documentation of the implemented features was added to the Gitlab wiki of the code repository. We created a wiki page for the document support and coding mechanism feature.

Req 3: The implementation shall include tests for modified and added functionality of QDAcity

This requirement is partly fulfilled. Since the goal of this thesis was to extend existing functionality, most tests were already existing. The existing tests were refactored to comply with the new implementation. However, we were not able to fully refactor tests in some cases as they involve the calculation of complex metrics that are not part of this thesis. These tests have to be refactored in the future.

[3]https://www.oracle.com/java/technologies/javase/javadoc.html
[4]https://jsdoc.app

In order to use existing documents with the new backend architecture and coding mechanisms, the documents had to be migrated. We implemented a migration job using the Visitor design pattern. A parameterized JUnit test was implemented to check the migrator for edge cases and bugs. The parameterized test allowed us to test against multiple different documents and compare the migrated document to the expected document. Additionally, the input parameters can be adjusted to force the handling of edge cases like:

- Dismissing of codings that belong to already deleted codes
- Dismissing of legacy HTML snippets in documents like Scalable Vector Graphics (SVG) containers

The test consists of 9 documents to migrate:

- 1 document of which all codings should be dismissed
- 1 document of which only part of the codings should be dismissed
- 1 document of which the legacy SVG container should be stripped
- 2 documents that include extensive and overlapping use of font styling (bold, italic, underline, font family, and font size)
- 4 anonymized documents from the QDAcity production application.

An anonymized document has all text contents replaced with random characters of the same length. Only metadata stored in HTML attributes was left unchanged.

9 Future work

Although QDAcity now supports PDF documents, there are still aspects to improve. This section highlights milestones for future work on QDAcity.

9.1 Remove embedded coding keys from RTF documents

As noted in section 7.2, we still embed some coding information in RTF documents. If resources allow in the future, the embedded coding keys can be removed in favor of storing e.g. paragraph indexes in the codings. We expect this change to have minor impact on the new coding mechanism. Regarding the *SlateTextCoding* class, the *anchorKey* and *focusKey* properties will have to be refactored. The *TextEditor* in the frontend also has to be modified to work with the chosen paragraph references instead of the coding keys.

9.2 Realtime-service integration for saving documents

When handling editable documents, the new coding mechanism requires attaching updated codings to document update requests. These coding updates will currently not be synced with other users, as the document is also not synced upon saving. In the future, document modifications synced across multiple users would improve collaborative work.

Because of this missing feature, the backend currently has to perform integrity checks for coding add, update, and delete requests. Without these integrity checks, users working on the same document could store incompatible codings. If e.g. *user1* and *user2* work on the same document, while *user1* edits the document and *user2* changes codings, the codings modified by *user2* could be incompatible with the updated document.

9.3 Store documents externally

Before this thesis, documents were stored inside the Google Datastore[1]. This has not changed for the support of PDF documents. However, the Google Datastore only allows storing entities with a maximum size of 1MB[2]. Especially for PDF Documents, this limit can be exceeded easily. Hence, the new backend architecture for handling different documents has to be adjusted to e.g. store files in Google Cloud Storage[3].

We expect this change to have no impact on the new coding mechanism. However, the *BaseDocument* class will have to be refactored in order to use a different storage for documents. The frontend will also have to be refactored in order to fetch the documents from a different storage.

9.3.1 Lazily fetch documents

Currently, all documents of a project are fetched from the server when opening the coding editor. In the past, this was necessary to parse the number of codings and the coding previews out of the documents. With the new coding mechanism, it is no longer required to fetch all documents at once. Instead, documents could be fetched on-demand, e.g. when the user opens a document. We consider this milestone to be a sub-task of migrating documents to a different storage, since both tasks involve changes considering document handling.

9.4 Coding previews for area codings

QDAcity provides users with a list of coding previews. For text codings, this feature was migrated to the new coding mechanism. For area codings, this feature is currently missing.

To implement coding previews for area codings, more research is needed on how to get a screenshot of the coded area. Since the backend architecture for the new coding mechanism already supports storing different kinds of previews, no backend modifications are expected. However, the previews must not exceed 1MB in size, as they are stored inside the Google Datastore. For very large area codings, this could be a limitation. Since coding previews are not intended to include the entirety of a coding, we see no advantages in storing coding previews in a separate storage. Therefore, you could either compress or crop coding previews to meet the 1MB limit.

[1]https://cloud.google.com/datastore/

[2]https://cloud.google.com/datastore/docs/concepts/limits

[3]https://cloud.google.com/storage/

The coding preview list in the frontend of QDAcity will require some changes to handle different coding preview types. Since the preview list shows all codings of a code across multiple documents, it has to know how to handle different preview types by itself. This issue could be resolved in multiple ways:

1. Let the preview list convert the coding previews into HTML snippets by comparing the *type* property of the coding. This however requires extending the logic of the preview list every time a new coding type is added to QDAcity.
2. Implement coding classes in JavaScript and use the Factory design pattern (Gamma, 1995) to cast the received JSON objects into specific coding instances. The coding classes could all implement a method to convert the coding-specific preview into HTML snippets that can directly be inserted into the preview list.
3. Already store HTML representations of the coding preview in the coding entity. This strategy however might not be possible with every coding type that might be added in the future.

10 Conclusion

QDAcity now allows users to upload and do qualitative research with PDF documents in the cloud. Analogous to the already supported RTF documents, codings are synced across multiple browser sessions to allow collaborative coding. To enable users to code text as well as non-text areas of PDF documents, text and area codings were implemented.

To successfully implement this feature in QDAcity, we first had to analyze how to integrate a PDF viewer in QDAcity. In chapter 2, we described the purpose of QDAcity and how it already enables users to do qualitative research with RTF documents. We showed the frontend layout and explained all components of the coding editor, which is the central component to do qualitative research.

Before designing our implementation architecture, we compared QDAcity with other QDA software tools in chapter 3. Additionally, we researched available libraries to process PDF documents and compared their pros and cons. We used the gathered implementation details to evaluate different strategies on how to integrate PDF support in QDAcity.

In chapter 4, we formulated functional as well as non-functional requirements. These requirements were used in chapter 8 to evaluate our implementation against them. The requirements were built using the templates by Rupp (2014) in order to define explicit, complete, and testable requirements.

We described the planned implementation architecture in chapter 5. For both supporting multiple document types as well as multiple coding types, we designed extensive class hierarchies that can be easily extended in the future. In the implementation chapter (chapter 6), we explained the implemented classes and methods in detail. Since the original coding mechanism of QDAcity was not usable with PDF documents, we had to design a new coding mechanism. The new coding mechanism now no longer relies on coding information that is embedded in the content of the documents. Instead, the new coding mechanism stores coding data separate from the document. Since this change involved side-effects and the need for a document migration, we also addressed these topics in section 6.2.1 and section 6.2.3 respectively.

As we were facing some unforeseen challenges during implementation, we included chapter 7 in this thesis. This chapter is supposed to help future QDAcity developers that e.g. might extend the range of supported document or coding types. Additionally, we justified the previously unplanned step of updating the Slate text editor. After evaluating our implementation against the defined requirements, we highlighted milestones for future work on QDAcity in chapter 9.

The resulting PDF support was implemented using widely adopted design patterns. In order to extend the range of supported document types in the future, extensive class hierarchies were designed. Future document and coding types can inherit from the pre-defined base classes and do not need changes in the endpoint implementations. To render different document types in the frontend, the Facade design pattern was used. Using the Facade design pattern, we implemented different editors to handle a specific document type.

Appendix A Evaluated Java PDF libraries

Lib	License	Pro	Con
Adobe PDF Library[1]	Proprietary, non-free	“Adobe Quality” (Compatibility, Features, etc.)	Needs license agreement, which is granted on a case-by-case basis
Apache PDFBox[2]	Apache	Feature rich (pdf creation, text extraction, signing, printing, preflight, form filling, save as image, split & merge) License Actively developed (latest release late 2019)	None in respect to the usage of the library
iText[3]	Proprietary/AGPL	Feature rich PDF Software Development Kit (SDK) Actively developed	License: AGPL is based on GPL2, which is incompatible with Apache. Apache is only compatible with GPLv3, if the final product is licensed under GPLv3[4] Data extraction is closed source and needs a license (pdf2Data)

[1] https://www.adobe.com/devnet/pdf/library.html
[2] https://pdfbox.apache.org
[3] https://itextpdf.com/en
[4] http://www.apache.org/licenses/GPL-compatibility.html

JPedal[5]	Proprietary/GNU LGPL	Actively developed Feature set seems equal to Apache PDFBox	Full version only available through commercial license
OpenPDF[6]	GNU LGPLv3 / MPLv2.0	Fork of iText 4.2 extensive JavaDoc documentation “good” license	Presents itself as not as big of an library as e.g. PDFBox, as the only reference is the Git repository
PDFTron Systems[7]	Proprietary	Tailored for enterprise PDF software, hence comprehensive PDF SDK	Needs a license agreement

[5]https://www.idrsolutions.com
[6]https://github.com/LibrePDF/OpenPDF
[7]https://www.pdftron.com

Appendix B Evaluated JavaScript PDF libraries

Lib	**License**	**Pro**	**Con**
PSPDFKit[8]	Proprietary	Very full featured and easy to use PDF editor for the browser	Needs license
PDF-LIB[9]	MIT	Free to use, can edit PDFs active project	PDFs can only be created/modified programmatically, as there is no User Interface (UI) for integrated for those tasks
PDF.js[10]	Apache 2.0	Free to use (by Mozilla), can parse and render PDFs active project	PDF-reassembly obviously not possible (hence the need of PDFassembler), no editor included. Library is called "terrific" by PDFassembler
PDFTron WebView (PDFNet)[11]	Proprietary	Seems to have an extensive documentation and to be very feature rich	No demo available, needs license

[8] https://pspdfkit.com
[9] https://pdf-lib.js.org
[10] https://mozilla.github.io/pdf.js/
[11] https://www.pdftron.com/api/web/PDFNet.html

References

Bazeley, P. (2013). *Qualitative data analysis: Practical strategies*. Sage.

Caudle, S. L. (2004). Qualitative data analysis. *Handbook of practical program evaluation*, *2*(1), 417–438.

Gamma, E. (1995). *Design patterns: Elements of reusable object-oriented software*. Pearson Education India.

Gerson, J. (2016). Work as a team - Understanding MAXQDAs Teamwork options. Retrieved November 4, 2020, from https://www.maxqda.com/work-as-a-team-understanding-maxqdas-teamwork-options

Kaufmann, A. & Riehle, D. (2015). *Improving traceability of requirements through qualitative data analysis*. Gesellschaft für Informatik eV.

Kaufmann, A. & Riehle, D. (2019). The QDAcity-RE method for structural domain modeling using qualitative data analysis. *Requirements Engineering*, *24*(1), 85–102. https://doi.org/10.1007/s00766-017-0284-8

Rupp, C. (2014). Requirements Templates — The Blueprint of your Requirement, 6.

List of Figures

2.1 Coding editor view in QDAcity 2
2.2 Code system example by Kaufmann and Riehle (2019) 3

3.1 Demonstration of PDF.js text overlaying accuracy 8

5.1 Architecture of backend document support 12
5.2 Architecture of frontend file support 13
5.3 Architecture of the coding mechanism 15

6.1 UML class diagram of the document architecture 17
6.2 UML class diagram of the *DocumentType* enum 18
6.3 UML class diagram of custom *TypeIdResolvers* 19
6.4 Sequence diagram of *EditorWrapper* coding caching functionality 21
6.5 UML class diagram of the coding architecture 22
6.6 UML class diagram of *TextCoding* 24
6.7 Illustration of coding merging and splitting mechanism 25
6.8 Visualization of original and migrated text document HTML excerpts 28

7.1 Slate paragraph keys 31

8.1 PDF layout in QDAcity vs. Evince PDF viewer 34
8.2 Visualization of support for different coding types 35

List of Code Snippets

6.1 Excerpt of original text document DOM tree 28
6.2 Excerpt of migrated text document DOM tree 28