Intercoder Evaluation Metrics in QDAcity

MASTER THESIS

Vishwas Anavatti

Submitted on 5 July 2022

Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik
Professur für Open-Source-Software

Supervisor:
Dr. Andreas Kaufmann
Prof. Dr. Dirk Riehle, M.B.A.

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, 5 July 2022

License

Erlangen, 5 July 2022

Abstract

In qualitative research, Intercoder Agreement (ICA) represents the extent to which two or more researchers share the interpretation of the data. The ICA assessment adds investigator triangulation to a project and helps researchers validate their findings.

QDAcity is a cloud-based application which provides a platform for qualitative data analysis by allowing researchers to collaborate and analyse the data by defining codes mutually and code the same documents independently. The researchers can assess their coding using three ICA metrics: F-Measure, Krippendorf's Alpha and Fleiss' Kappa, which can be generated with an evaluation unit as 'paragraph' for selected documents.

In this thesis, we extend the existing metrics by providing 'sentence' as another evaluation unit and an option to calculate agreement on a subset of the codesystem, which enables collaboration on larger projects. Additionally, we present the design and implementation of a new ICA metric called Agreement Queries, which generates results based on agreement types such as Code Occurrence, Code Frequency and Code Intersection Percentage.

Contents

List of Figures

List of Tables

List of Codes

Acronyms

API Application Programming Interface

CSV Comma Separated Value

GAE Google App Engine

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

ICA Intercoder Agreement

LOC Lines Of Code

NLP Natural Language Processing

PDF Portable Document File

QDA Qualitative Data Analysis

RTF Rich Text File

UI User Interface

UML Unified Modeling Language

XLS Microsoft Excel Spreadsheet

1 Introduction

The concept of Intercoder Agreement (ICA) has a wide range of applications in various domains of study, with qualitative research being one of them. Data classification into preset categories is a typical task in qualitative research. Codes are often values assigned to a nominal or ordinal property. By asking two individuals, referred to as coders, to independently perform this classification with the identical set of data, the agreement of this classification process can be established. By performing this task, these two individuals have just completed an ICA exercise, which will result in two categorizations of the identical data. ICA refers to the degree to which these two categorizations agree (Gwet, 2014).

ICA in general is widely known as agreement between different coders on how to code the same data. In a collaborative environment, ICA assessment yields numerous benefits for qualitative research as it helps to improve the communication and transparency of the coding process (O'Connor & Joffe, 2020). Researchers use the ICA measure to communicate, validate and ensure, they have a shared interpretation of the data.

QDAcity is a web application for Qualitative Data Analysis (QDA) which provides a collaborative environment for researchers to conduct qualitative research. The tool allows researchers to analyse ICA using three statistical methods such as F-Measure, Krippendorff's Alpha and Fleiss' Kappa. The ICA can be measured in various other techniques and researchers would like to assess results in different aspects. Therefore, the goal of this thesis is to enhance ICA evaluation approaches in order to give researchers with a diverse set of alternatives for analysing ICA.

In this thesis, we analyse the current state of QDAcity, identify the problems and define the thesis objectives in chapter 2. chapter 3 states the functional and non functional requirements. In chapter 4 we evaluate the libraries and present the architecture, while our implementation solution is discussed in chapter 5. We discuss the implementation challenges in chapter 6 before evaluating the provided solution in chapter 7. Scope of future work is discussed in chapter 8 and conclusion to the thesis is made in final chapter 9.

2 QDAcity

In this chapter, we analyse the current state of QDAcity, identify the problems and derive the objectives for the thesis.

2.1 Analysis of Current State

QDAcity runs on Google App Engine (GAE)[1] and provides a platform for collaborative research on the qualitative data. In QDAcity, a user can create a project and add other users as collaborators. The users can then create text documents and upload PDF, RTF and audio files for analysis. Codes are defined mutually by users in an iterative process which forms a codesystem. The codesystem is a hierarchical structure of codes and captures concepts, categories, their properties and interactions (Kaufmann & Riehle, 2019). These codes are then used to annotate the data in the document to identify the pattern or categorize the content. The process of annotating the data is called 'coding'. After coding, a revision must be created for further analysis which is explained as follows.

Figure 2.1 shows the overview of project dashboard. It consist of 'Project Description', 'Project Stats', 'Intercoder Agreement bar charts', 'Todos', 'Users' and 'Revision History'. As our main focus is ICA which is part of a revision, we explain the contents of a revision as numbered in the screenshot.

1. Revision: Multiple revisions can be created for a project which is a snapshot of the project. *Revision 0* is the first revision of the project and a typical revision contains revision info, generated reports and validation projects.
2. Validation Projects: Users of the project can click on 'recode' button to create a validation project which is a clone of the revision without the applied codings. The section contains the validation project list in the name of users who has recoded the revision.
3. Reports: A user can click on 'Create Report' button to create an ICA report. The section contains list of generated reports for the revision.

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

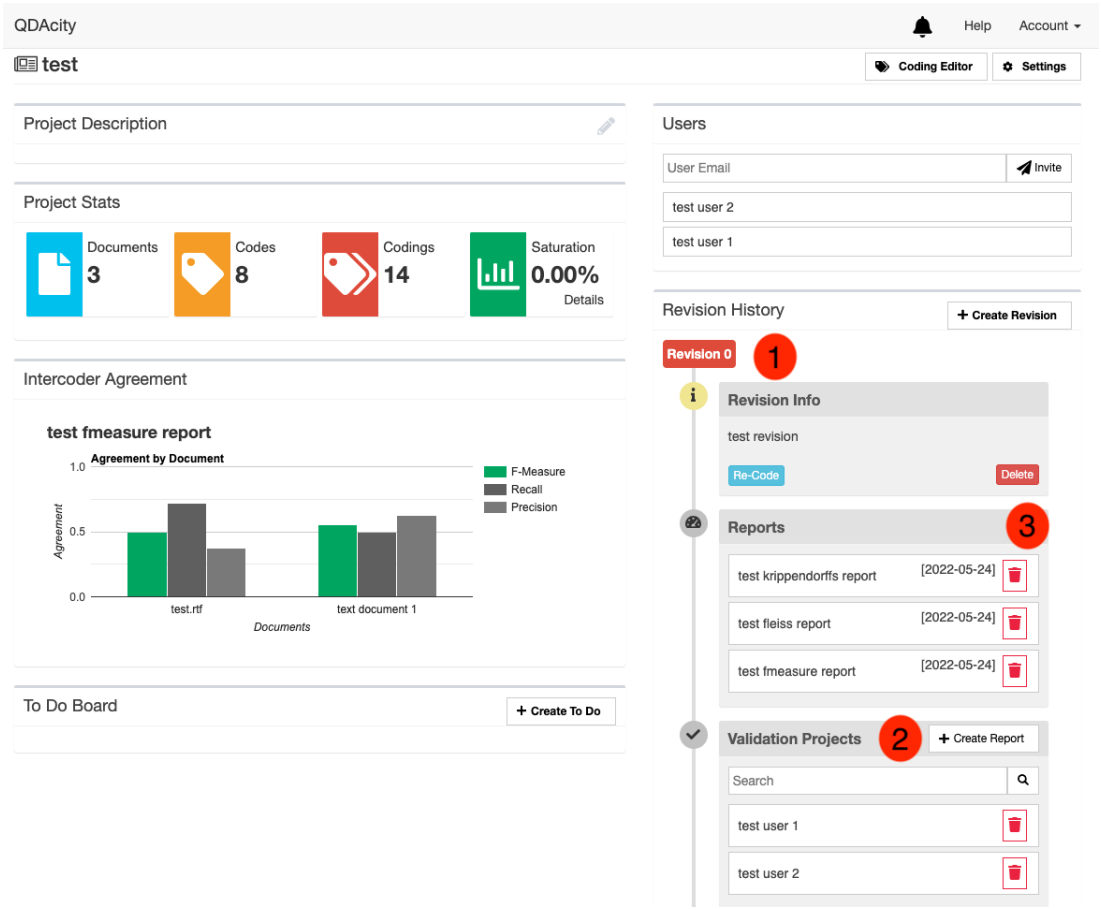


Figure 2.1: Project Dashboard overview in QDAcity

ICA report can be generated for a revision using statistical methods such as F-Measure, Krippendorf's Alpha and Fleiss' Kappa using 'paragraph' as an evaluation unit for selected documents. This report assists researchers in identifying differences in their codings, discussing the reasons for the differences and improving coding so that they arrive at the same conclusion.

2.2 Problem Statement

QDAcity has three metrics for ICA assessment and limitations with current metrics are

1. **Evaluation unit**

   Figure 2.2 shows the report creation modal. The user can provide title, choose one of the three metric as evaluation method and select documents. Only 'paragraph' is supported as unit of coding (evaluation unit).

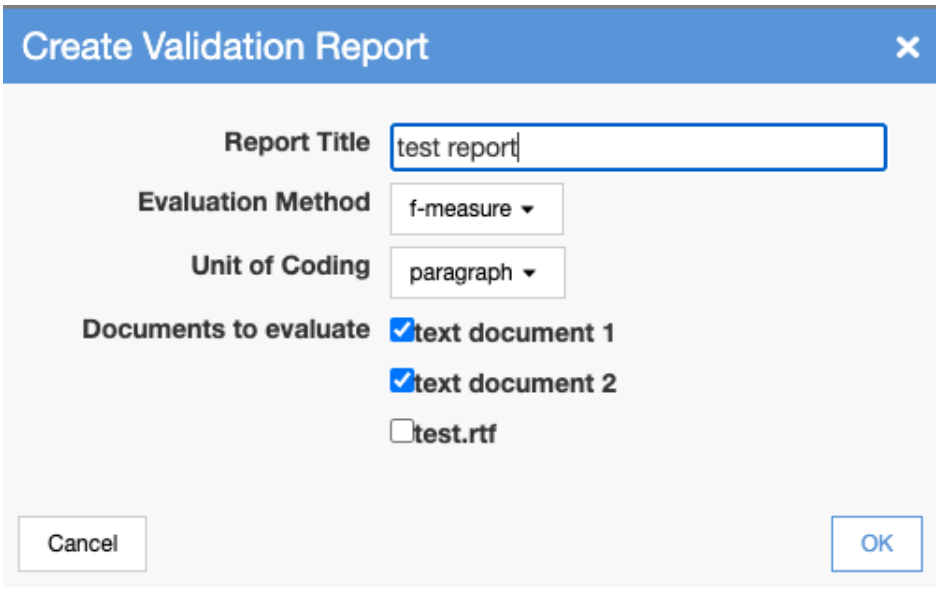
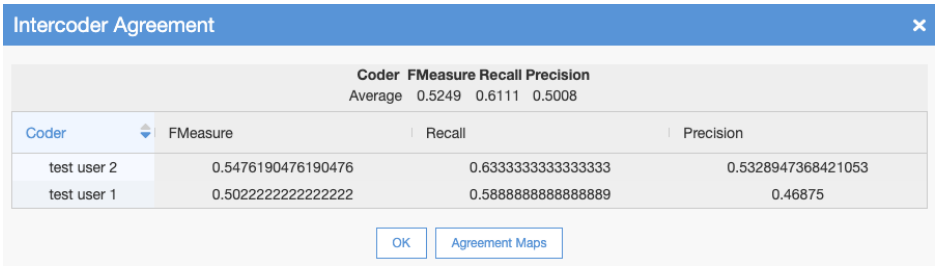


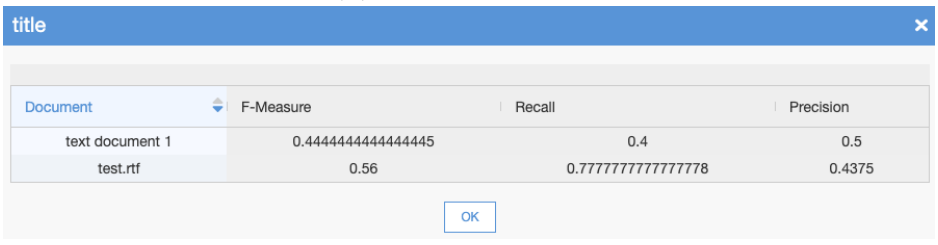
Figure 2.2: Validation report creation modal dialog

The 'paragraph' as only evaluation unit does not provide the users with many options to assess their coding. Though the user interface provides an option to select the sentence during report creation, the implementation for the same is not handled in the backend. Thus, QDAcity requires an extension to the evaluation unit.

2. **ICA by code**



(a) F-Measure report

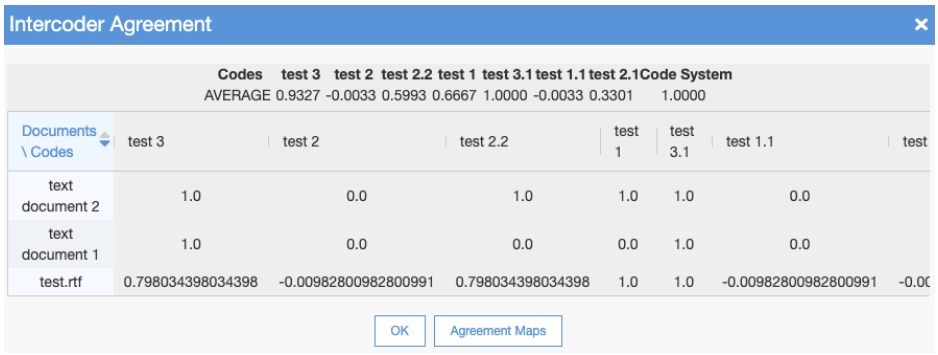


(b) F-Measure result per document for a coder

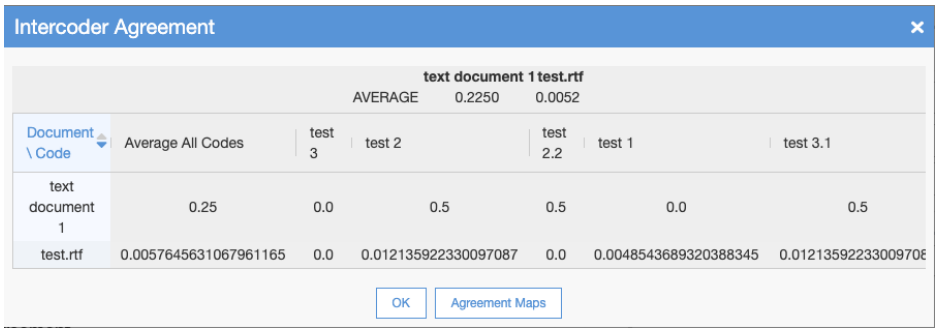
Figure 2.3: Overview of F-Measure report

Figure 2.3 shows the F-Measure report in a modal. The results of the each coder is displayed in the Figure 2.3a. On selecting the coder, the result per document can be viewed for that coder as shown in Figure 2.3b.

Figure 2.4 shows the Krippendorff's Alpha and Fleiss' Kappa report in a modal. Both the report displays result for selected documents and whole codesystem. The Krippendorff's Alpha report shows code average and Fleiss' Kappa report shows document average in the header.



(a) Krippendorff's Alpha report



(b) Fleiss' Kappa report

Figure 2.4: Overview of Krippendorff's Alpha and Fleiss' Kappa report

The report is generated by considering all of the codes in the codesystem. However, in large projects with over 100 codes, multiple users would prefer to do ICA using a subset of the codesystem rather than requiring each user to become familiar with over 100 codes in the codesystem. The F-Measure report allows the user to view agreement by document and the user would like the same flexibility to view agreement by code. As a result, there is a need to provide an option to select a subset of the codesystem during creation, as well as the ability for the user to view agreement per code for the selected subset of codes from the codesystem.

3. **User Interface (UI)**

   QDAcity requires a UI modification of ICA for the following reasons:

   - Figure 2.3b displays result per document but provides no information on which coder the document result is displayed for.
   - The Figure 2.4a and Figure 2.4b shows average of codes and documents in the header respectively. The results are cramped which makes it difficult for users to interpret the result.
   - The results are shown in a modal which is not very intuitive as it restrict the amount of information that could be shown and requires too much user interaction.
   - The labels in the table header are hard coded in English, as a result it is not possible to change labels as per user language preference.

4. **Intercoder evaluation metrics**

   The QDAcity has three statistical metrics for ICA measurement which calculate results based on established mathematical models. There are several other well established methods like Holsti's method (Holsti, 1969), Scott's pi (Scott, 1955), Cohen's kappa (Cohen, 1960) which user would be interested to use or user would like to define his own criteria to analyse ICA in different aspects. Therefore, QDAcity requires the addition of a new evaluation metric.

2.3 Objectives

From the four identified problems we derived the following objectives for this thesis:

- Provide 'sentence' as another evaluation unit.
- Provide an option to select subset of codesystem while creating ICA and the ability for users to view agreement per code in F-Measure report.
- Provide an intuitive UI for ICA where user can view all results at one place.
- Provide an implementation of new metric where users can analyse ICA based on their own criteria.

2.4 Agreement Queries

To provide a new metric, we looked at our competitor MAXQDA[2] which is commercially available platform for QDA. In MAXQDA, the users can do a teamwork and analyse their codings by generating ICA results. The ICA is measured on three types of agreement such as code occurrence in the document, code frequency in the document and minimum code intersection rate of X% at the segment level[3].

We analysed the use case of providing such a metric in QDAcity as it could provide flexibility for researchers to define their own criteria and analyse the results conveniently with potentially large group of intercoders. Therefore, inspired by MAXQDA we came up with the idea of implementing 'Agreement Queries'. The name Agreement Queries represents that we are querying the agreement results based on different agreement types and parameters which the user may configure. As a result, it gives users a wide option to analyse results based on their own criteria. This implementation also provides scope for future development as agreement types can be extended to analyse the results in various ways. The Agreement Queries is similar to ICA in MAXQDA but there are some differences which are discussed in section 5.5.

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

[3]https://www.maxqda.com/help-max18/teamwork/problem-intercoder-agreement-qualitative-research

3 Requirements

In this chapter, we define the requirements for this thesis. The requirements are categorized into functional and non-functional requirements. All the requirements are defined using Rupp (2014) templates.

3.1 Functional Requirements

The terms used in the requirements are defined in the Table 3.1. The definition of the requirement keywords *shall*, *should* and *will* are as follows.

- Shall - The requirement that has to be fulfilled.
- Should - The requirement that is important but not required for the proper operation of software.
- Will - The requirement is desirable but not required.

The functional requirements are defined as follows:

1. QDAcity shall provide new UI for ICA.
    1.1. QDAcity shall provide an ICA button for each revision in the project.
    1.2. QDAcity shall provide two sidebar menus 'Report' and 'Agreement Queries' in the UI.
    1.3. QDAcity shall provide 'Report' as default menu on navigating to ICA and list generated reports if any or show appropriate message and provide option to create a report.
    1.4. QDAcity shall provide the results in tabular form and the ability for the user to sort the table results in ascending or descending order based on column values.
    1.5. QDAcity shall be able to display the codes in the hierarchical structure according to codesystem in the result table containing codes.

Term	Definition of the Term
user	The physical person who is using the QDAcity application
documents	The data structure that has the potential to be used for QDA
codes	The unique set of values to represent documents data
coder	The user who can code the documents
project	The place where users can collaborate, add documents, define codes mutually and code the documents
result	The calculation outcome of ICA for the existing metrics
coder result	The calculation outcome of ICA for the existing metrics for a coder
document result	The calculation outcome of ICA for the existing metrics for a document per coder
code result	The calculation outcome of ICA for the existing metrics for a code per document
report	The ICA generated for the existing metrics containing consolidated result
agreement queries	The new metric to generate agreements based on various agreement types
agreement query	The single calculation outcome of agreement queries
baseline coding	The coding of individual user project or the main project used as basis for comparison to calculate the agreement query
compared codings	The codings of individual users project or the main project that are used to compare the result with baseline coding
coder agreement	The calculation outcome of agreement query metric for a coder whose coding is selected for comparison
document agreement	The calculation outcome of agreement query metric for a document per coder
code agreement	The calculation outcome of agreement query metric for a code per coder
notification feature	The feature of QDAcity used to update users with important information

Table 3.1: Definition of the requirement terms

1.6. QDAcity shall be able to validate the following for report and agreement queries and throw corresponding errors.

- At least one document is selected

- At least one code from the codesystem is selected
- Report title is not empty
- At least one compared coding is chosen for agreement queries
- If "Code Agreement by Intersection Percentage" is selected as agreement type for agreement queries, the "minimum intersection percentage" value must not be empty and must be between 20-100

2. QDAcity shall be able to allow only authorized users to create ICA from projects.
3. QDAcity shall be able to allow only authorized users to view ICA from projects and validation projects.
4. Report
   - 4.1. Report Creation
     - 4.1.1. QDAcity shall provide the user the ability to generate report with sentence as an evaluation unit using all three currently supported metrics.
     - 4.1.2. QDAcity shall provide the user the ability to generate report for a subset of the codesystem using all three currently supported metrics.
   - 4.2. Report View
     - 4.2.1. QDAcity shall provide a report that includes the evaluation unit, evaluation method, report name, creation date and creator name entered during creation as an overview.
     - 4.2.2. F-Measure Report
       - 4.2.2.1. QDAcity shall provide a section to display documents and codes that were selected during creation.
       - 4.2.2.2. QDAcity shall provide the user the ability to select documents to view the agreement per document and to select codes to view the agreement per code.
     - 4.2.3. Krippendorff's Alpha and Fleiss' Kappa Report
       - 4.2.3.1. QDAcity shall provide a result table that includes evaluated documents in the first row and codes in the first column.
       - 4.2.3.2. QDAcity shall provide the average of each document and code agreement at the row and column ends of the result table.

4.2.3.3. QDAcity should provide the user the ability to export table results into a CSV document.

5. Agreement Queries

5.1. Agreement Query Creation

5.1.1. QDAcity shall provide the user the ability to select only one baseline coding in order to provide basis for the ICA assessment with other codings.

5.1.2. QDAcity shall provide the user the ability to select multiple compared codings in order to assess ICA of many coders with baseline coding.

5.1.3. QDAcity shall provide the user the ability to select documents and subset of the codesystem.

5.1.4. QDAcity shall provide three agreement types: “Code Occurrence”, “Code Frequency” and “Code Agreement by Intersection Percentage”, with the ability for the user to choose only one of the three.

5.1.5. If “Code Occurrence” is selected as agreement type, QDAcity shall provide the user the ability to mark “Consider True Negatives” in order to consider true negatives codes as agreed or not.

5.1.6. If “Code Agreement by Intersection Percentage” is selected as agreement type, QDAcity shall provide the user the ability to enter a “minimum intersection percentage” in order to define agreement criteria.

5.2. Agreement Query view

5.2.1. On initial viewing, QDAcity shall provide an agreement query that includes the baseline coding, agreement type and creator name as an overview and coders agreement.

5.2.2. QDAcity shall provide the user the ability to select a coder agreement to view the documents and codes agreement for that coder.

5.2.3. QDAcity should provide a search box in the documents and codes agreement, in order to filter the table by document and code name.

5.2.4. QDAcity shall provide the user the ability to select a code agreement, in order to view the codings of baseline and compared coder for that code in a two column comparison view which is resizable.

5.2.5. QDAcity should provide the user the ability to select a document agreement along with code agreement, in order to compare the

codings of baseline and compared coder for that combination of document and code in a two column comparison view.

5.2.6. QDAcity shall provide the user the ability to export the coder, document and code agreements tables into a CSV document.

5.2.7. QDAcity shall provide the user the ability to create new agreement query.

5.3. QDAcity shall be able to delete the previously created agreement query when initiating a new one.

6. QDAcity shall be able to notify the initiating user with the QDAcity notification feature upon successful creation of report or agreement query.

3.2 Non-Functional Requirements

Figure 3.1: Characteristics of quality software based on ISO/IEC 25010

The non-functional requirement is about the quality of the software. According to ISO-250101[1] standards, the quality software comprises of eight characteristics as shown in the Figure 3.1.

Functional Suitability

1. The ICA implementation shall be fully functional and user shall be able to generate and view ICA reports and agreement queries.
2. The newly added functionalities should not break the previous implementation of ICA and shall support all previous use cases of the ICA.

Performance Efficiency

3. The ICA metrics subsystem shall be horizontally scalable without relying on the main backend system with a more appropriate hardware configuration.

[1]https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

Compatibility

4. The ICA implementation shall be compatible with the existing data and data model.

Usability

5. The UI implementation of ICA shall follow the colour scheme of QDAcity.
6. The text containing UI elements shall adhere to the app's localization strategies.
7. As user initiates the report or agreement query creation, QDAcity shall provide a visual feedback message to the user.

Reliability

8. QDAcity shall perform as expected and throw or log errors for all the possible error occurrences.

Security

9. QDAcity shall be able to allow only authorized and authenticated users to perform the generation and viewing of ICA.

Maintainability

10. The implementation of evaluation unit extension should provide developers the ability to further extend the evaluation unit using pre-defined interface.
11. The implementation of agreement types in agreement queries should provide developers the ability to extend the agreement type and add new functionalities.
12. The implementation shall include tests for all API endpoints and added functionalities with Lines Of Code (LOC) coverage of at least 90%.
13. The implementation shall include acceptance tests for the newly added and modified functionalities.

Portability

14. The ICA feature shall be able to work as intended in widely used browsers such as Google Chrome and Mozilla Firefox.

4 Architecture

In this chapter, we first discuss the sentence detection, evaluation of libraries and architecture change to find codings in a sentence. Later a general architecture of push queues and agreement queries are discussed.

4.1 Sentence Detection

As we would like to provide 'sentence' as another evaluation unit for the current metrics, the detection of sentences in the text documents becomes a challenge. The sentence detection is not a straight forward process because it depends on many factors. Natural Language Processing (NLP)[1] is the branch of artificial intelligence which gives computers the ability to extract meaning from natural language data. Thereby, we discuss why NLP is required and which NLP libraries could be used for our purpose.

4.1.1 Sentence Detection - an NLP Problem

The detection of sentence seems easy as one could imagine splitting paragraph at period ('.') could result in the sentences. The period sign is quite ambiguous as it can have several functions. Consider an example

Mr. President, the call with vice president is scheduled at 8.30 p.m.

The example shows the ambiguity of the period sign as it can be at the end of a sentence, or can be used in abbreviations, or in acronyms and initialism, or in numbers. The use of ambiguous punctuation characters has to be resolved in a sentence detection to determine if the punctuation character is a true end-of-sentence marker (Schweter & Ahmed, 2019).

The semantics of a sentence is present in its interpretation. An English speaker would understand the sentence and be able to perform the individual activity like "Pass the bal". However, sentences can be too ambiguous in some cases and

[1] https://www.ibm.com/cloud/learn/natural-language-processing

their meaning can only be deduced from context. Typically this makes NLP a complex task for a machine because it will be missing the context[2].

The process of deciding the beginning and end of the sentences is known as Sentence Boundary Disambiguation. This process is heavily researched under NLP. Hence, we need an NLP library which is trained with machine learning models to detect the sentences in a text document.

4.1.2 Evaluation of Libraries

Our application is built on Java and there are several NLP libraries available for Java. Our primary purpose in using an NLP library is to identify sentences, so we must assess each library's ability to do so. Other factors to consider when choosing a library include the license, documentation and whether the library is actively maintained.

Another important aspect of the library is that it should provide the position of sentences in a paragraph rather than just the content and the reason is explained in detail in subsection 5.1.3. All evaluated libraries are listed in Table 4.1. Considering all the criteria, the decision was made to use Apache OpenNLP[3] as it has a permissive license, very active project status and provides the functionality to find the sentence position.

Criteria	**Stanford CoreNLP**	**Apache OpenNLP**	**LingPipe**
License	GNU General Public License v3.0	Apache License 2.0	Proprietary
Documentation	Good documentation with demo example	Detailed documentation, many implementation examples available	Detailed documentation with code demo
Project status	Active	Active	No status available
Sentence position detection	No	Yes	No

Table 4.1: Evaluation of Java NLP libraries for sentence detection

For sentence detection, OpenNLP library requires a pre-trained model. The models are language dependent and perform well only when the model language matches the language of the text document. To select a trained model, we must first understand the language of the text document. As a result, another library that can detect the language of the text document is required.

[2]https://xperti.io/blogs/java-natural-language-processing/
[3]https://opennlp.apache.org/

The Table 4.2 lists all of the libraries that have been evaluated for language detection. Lingua[4] and Apache Tika[5] are good candidates for language detection because they have a permissive license, an active project and detailed documentation. According to Lingua documentation, it outperforms Apache Tika in language detection but due to the fact that Lingua consumes very high memory and Apache Tika being product of Apache software foundation[6] having large number of contributors and QDAcity already using the Tika library for document parsing, the decision was made to use Apache Tika.

Criteria	**Apache OpenNLP**	**Apache Tika**	**Lingua**	**optimaize**
License	Apache-2.0	Apache-2.0	Apache-2.0	Apache-2.0
Documentation	Good documentation with demo example	Good Javadoc and demo examples available	Detailed documentation with code demo	Demo code available but document is not detailed
Project status	Active	Active	Active	Not Active
Disadvantages	Needs training data file to provide input to the machine learning algorithms	Detects only 18 of 184 standard languages standardized by ISO 639–1	Consumes very high memory	Not suitable for short text

Table 4.2: Evaluation of Java libraries for language detection

4.1.3 Architecture

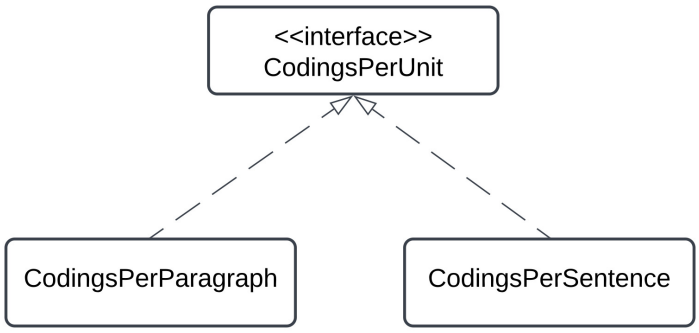


Figure 4.1: Codings per unit architecture

[4]https://github.com/pemistahl/lingua
[5]https://tika.apache.org/
[6]https://www.apache.org/

We would like to add 'sentence' support to the evaluation unit. Extending the evaluation unit to a sentence helps to find the codings used in each sentence. Previously, only *CodingsPerParagraph* class was used to find codings within a paragraph. We outlined the architecture in the Figure 4.1 to make it more generic and extensible for finding codings for various units. *CodingsPerParagraph* is the class that finds the applied codings per paragraph and *CodingsPerSentence* is the class that finds the applied codings per sentence. Both classes implement the *CodingsPerUnit* interface.

The current metrics calculation implementation requires code Id of the codings per paragraph which are obtained from the *CodingsPerParagraphCalculator* class which implements the *DocumentVisitor*. The *CodingsPerParagraphCalculator* overrides the *visit()* method which provides the required code Ids of the codings per paragraph using *CodingsPerParagraph* class.

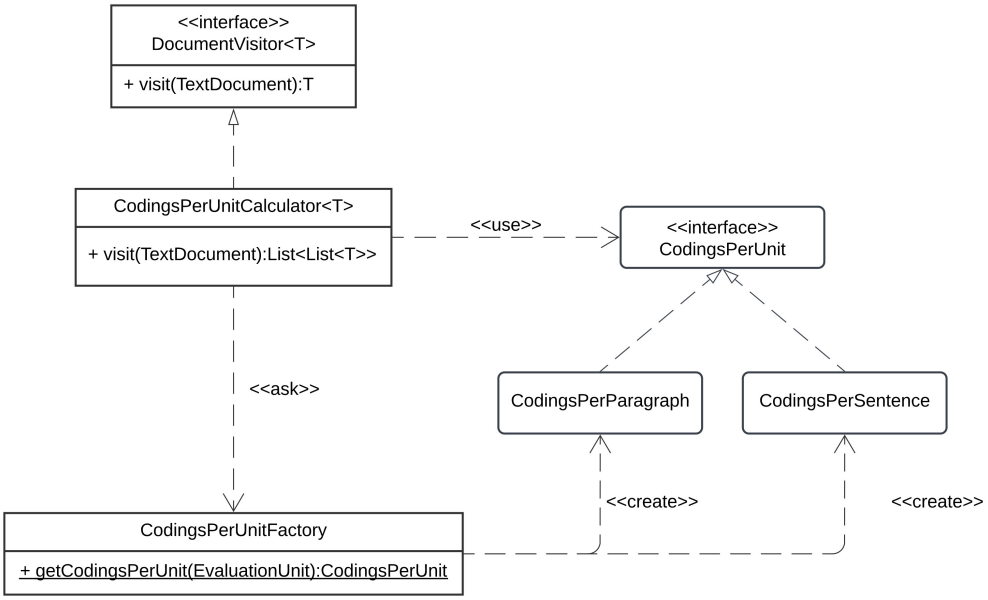


Figure 4.2: Strategy with factory pattern UML diagram of codings per unit

We modified the current implementation to make it generic for any evaluation unit by applying two design patterns. The factory pattern defines an interface for creating an object but allows subclasses to select which class to instantiate, whereas the strategy design pattern enables runtime selection of an algorithm's behaviour.

Figure 4.2 represents the strategy with factory design pattern for finding codings per unit. The *CodingsPerParagraphCalculator* class has been renamed to *CodingsPerUnitCalculator*. When *visit()* is called, the factory pattern is used to create the class object, i.e. *CodingsPerUnitFactory* class is used to create the

object based on the evaluation unit and the strategy pattern is used to run the algorithm corresponding to the object, i.e. at runtime, the algorithm corresponding to the created object is selected, which gets the codings per unit.

4.2 Task Queues

Task Queues are used to perform tasks asynchronously outside of a user request. The task can be described as a unit of work such as "write object to datastore". The tasks are added to a task queue which gets executed later by GAE worker services. There are two types of Task Queues

- **Push Queues:** The tasks are enqueued into the queue and GAE executes them on the handler. It can be scaled automatically or manually and tasks are deleted by App Engine after processing.
- **Pull Queues:** The tasks are enqueued into the queue which needs to be pulled by worker service by leasing tasks. It requires manual scaling and task deletion is done explicitly.

Push queues are used extensively to generate the ICA results. As a result, we explain push queues in detail. Figure 4.3 shows the push queue workflow. A queue.yaml file is used to name and configure the push queues. Push queues carry out tasks by sending HTTP requests to App Engine worker services and requests are processed at a constant rate. When a task fails, the service retries it by sending a new request if retry is configured. We must provide a handler for each task we use. A single service can have multiple handlers for different types of tasks, or different services can be used to manage different task types.

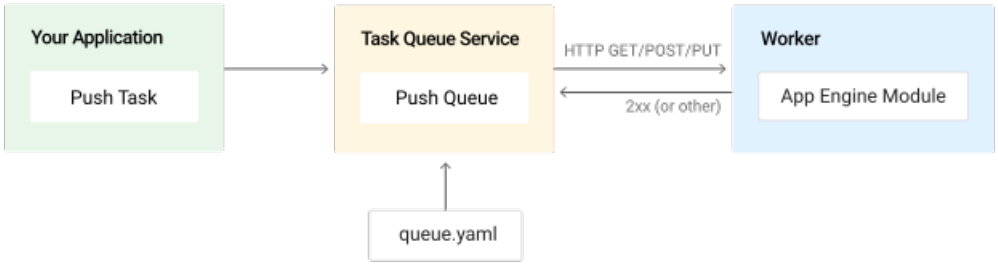


Figure 4.3: Workflow of push queue (Google, 2022)

When a worker service receives a push task request, it must handle the request and send an HTTP response before a deadline defined by the worker service's scaling type. Our application is set up to run push queues with automatic scaling, which should finish the task in 10 minutes. The task's success is indicated by an

HTTP status code ranging from 200 to 299, while any other value indicates the task's failure. When a task fails, it can be configured to retry before the deadline (Google, 2022).

4.3 Agreement Queries

The new metric implementation should allow researchers to define criteria and analyse agreements at the coder, document and code level with a potentially large group of intercoders. The criteria can be defined in terms of an agreement type and we provide three different agreement types: code occurrence, code frequency and code intersection percentage. To provide an implementation for each of the agreement types, we have created the following architecture.

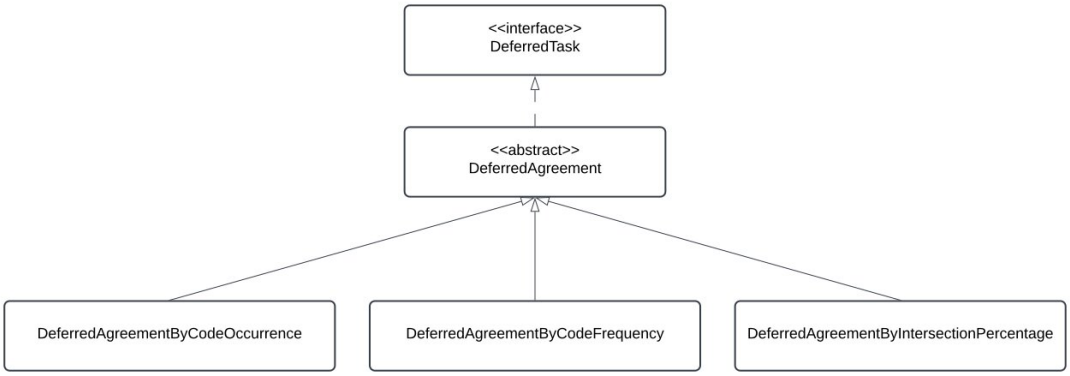


Figure 4.4: Agreement queries architecture

Figure 4.4 outlines the architecture for generating agreements based on different agreement types. The implementation class for each of the agreement types code occurrence, code frequency and code intersection percentage is called *DeferredAgreementByCodeOccurrence*, *DeferredAgreementByCodeFrequency* and *DeferredAgreementByIntersectionPercentage* respectively. All classes are represented by the abstract class *DeferredAgreement* which implements the *DeferredTask*. To run the tasks asynchronously and generate agreements for large group of intercoders simultaneously, we utilized the *DeferredTask* interface provided by the App Engine task queue to configure the tasks into push queue.

5 Implementation

This chapter discusses the implementation of the architecture described in Chapter 4 as well as the frontend refactoring required for ICA. The QDAcity uses Java for the backend and JavaScript and React for the frontend. The implementation is discussed in detail, beginning with the provision of sentence as an evaluation unit. Followed by the implementation of ICA by code and the new metric Agreement Queries.

5.1 Evaluation Unit - Sentence

The provision of sentence as an evaluation unit is to find the applied codings per sentence for generating ICA reports. Previously, the ICA report was generated for text documents with evaluation unit as paragraph by finding applied codings per paragragh. Because only the evaluation unit 'paragraph' is supported, the existing metrics algorithms obtain the codings used in each paragraph from the *CodingsPerParagraph* class. The algorithm must obtain the applied codings for each sentence in order to extend the metrics to sentence as an evaluation unit. As a result, the modification is required only where the applied codings are obtained, with no modification required in the metrics calculation.

To find applied codings in a sentence, it is necessary to detect sentences in a paragraph. We use the Apache OpenNLP library to detect sentences. The library contains pre-trained models[1] for different languages. The models are language dependent and they will only function properly if the model language matches the language of the input text. QDAcity supports English and German as localized languages and this thesis focuses solely on sentence detection in these two languages.

5.1.1 Language Detection

The enum class for *LanguageModel* is shown in Figure 5.1. The supported languages English(en) and German(de) are added as enums and the corresponding

[1] http://opennlp.sourceforge.net/models-1.5/

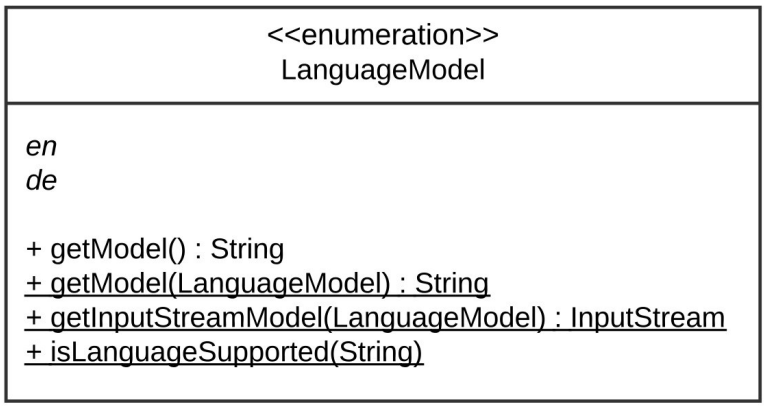


Figure 5.1: UML class diagram of *LanguageModel*

pre-trained models 'en-sent.bin' and 'de-sent.bin' are kept in the project resources path (src/main/resources). The enum class includes the following methods:

- *getModel()* - This method returns enum as string.
- *getModel(LanguageModel)* - This method implicitly calls *getModel()* and returns enum as string.
- *getInputStreamModel(LanguageModel)* - This method loads the pre-trained model for the input language from resources and returns it as *InputStream*.
- *isLanguageSupported(String)* - This method validates whether the input language is supported.

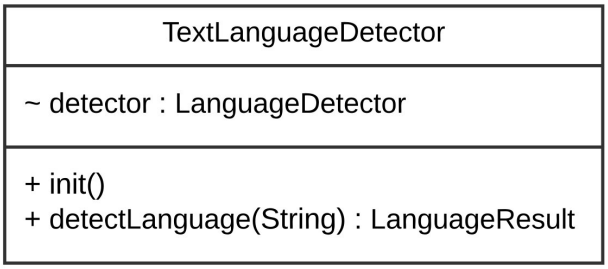


Figure 5.2: UML class diagram of *TextLanguageDetector*

To determine the language of the text, we use the Apache Tika library. The library helps to determine the language of the text, which then helps to load the

appropriate pre-trained model for sentence detection using the OpenNLP library. The UML class diagram for *TextLanguageDetector* is shown in Figure 5.2. This class uses the *LanguageDetector#detect()* method from the library to identify the language.

When *TextLanguageDetector* class is instantiated, the constructor calls the *init()* method which instantiates the class *detector* attribute and loads the language profile models based on the specified languages in the *LanguageModel*. The language of the text can be determined by passing text as a *String* to the *detectLanguage()* method. The method identifies the language and returns *LanguageResult*, which contains *language* as 'en' or 'de', *confidence* as LOW, MEDIUM or HIGH and *rawScore* between 0 and 1.

If the text is blank or only contains special characters, the language detector returns *LanguageResult* language as 'None'. In such cases, English is considered as the default language.

5.1.2 Text Document

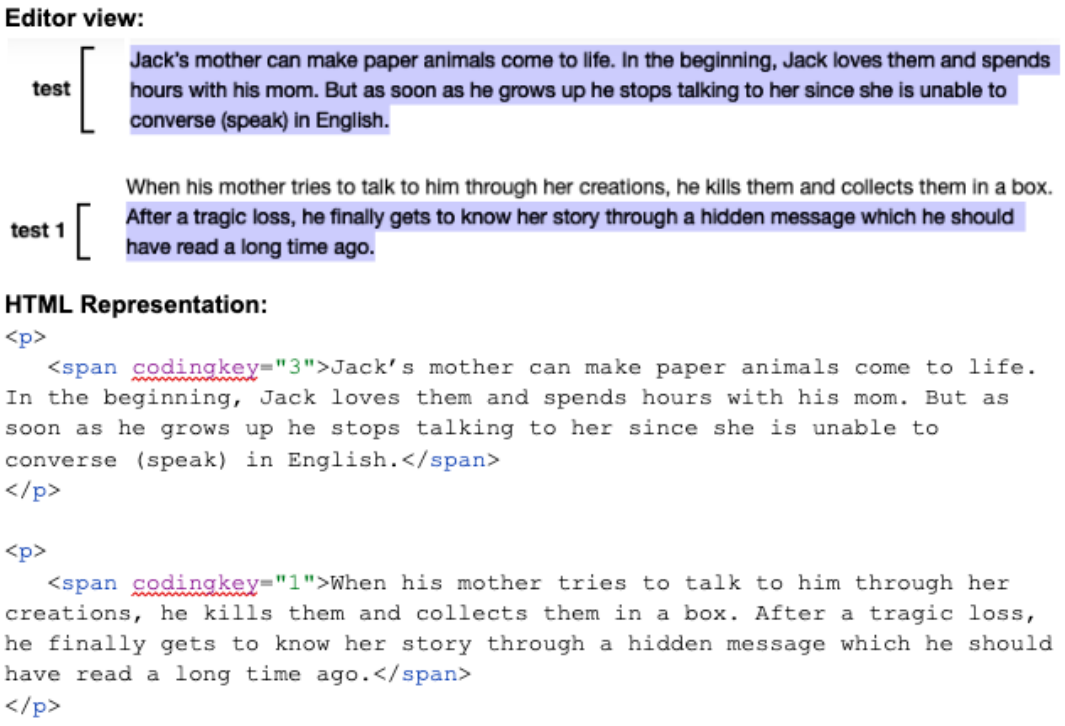


Figure 5.3: Coded paragraphs and its HTML representation

The sentence detection is only implemented for *TextDocument*. Figure 5.3 shows the coded paragraphs in a text document and its HTML representation. Each

paragraph has a span which contains the text and a unique key called *codingKey*. The highlighted region in the editor view represents the coded part.

The *Coding* data structure has four attributes that represents start and end of a coding. These attributes are critical in determining the applied codings for each sentence.

- *anchorkey* - represents the *codingKey* in which the coding starts
- *anchorOffset* - represents the starting position of the coded text in a paragraph
- *focusOffset* - represents the ending position of the coded text in a paragraph
- *focusKey* - represents the *codingKey* in which the coding ends

5.1.3 Codings Per Sentence

Figure 5.4 shows the class diagram *CodingsPerParagraph* and *CodingsPerSentence* implementing *CodingsPerUnit*. The existing implementation of *CodingsPerParagraph* is modified by adding two new attributes to the class.

- *codingKeys* - This attribute contains the list of paragraph *codingKey*.
- *codingKeyParagraphTextMap* - This attribute contains the paragraph *codingKey* and the corresponding paragraph text as a key value pair.

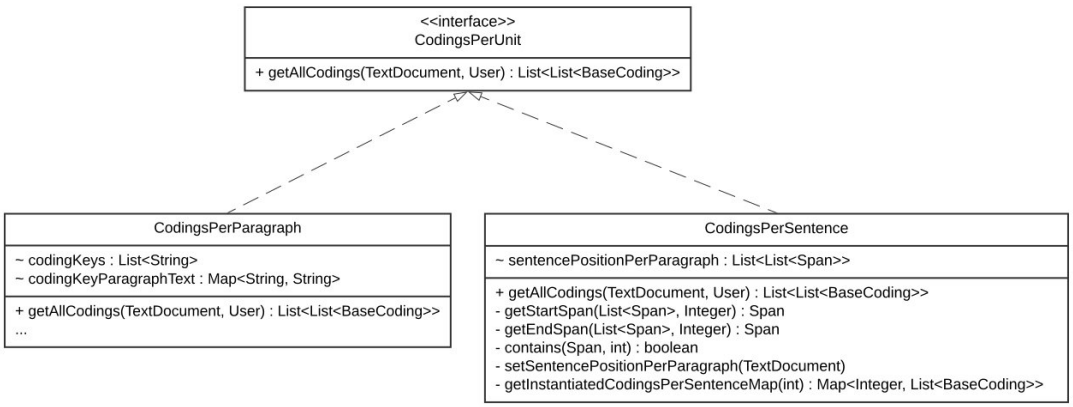


Figure 5.4: UML class diagram of Codings Per Unit

The *CodingsPerSentence* contains only one attribute *sentencePositionPerParagraph* which contains the List<List<Span>>. The *getAllCodings(TextDocument)* is the main implementation method for finding codings in a sentence and it returns List<List<BaseCoding>>. The List<List<BaseCoding>> represents the list of sentences containing list of applied codings.

The Apache OpenNLP library is used in the *CodingsPerSentence* class to identify the sentences. The library has *SentenceDetectorME* class which detects the sentences using *sentPosDetect(String)* method. The *sentPosDetect(String)* method gives the position of the sentences for a given string. Consider the following paragraph for sentence detection, which is passed to the method as a *String*.

Jack's mother can make paper animals come to life. In the beginning, Jack loves them and spends hours with his mom. But as soon as he grows up he stops talking to her since she is unable to converse (speak) in English.

The *SentenceDetectorME#sentPosDetect(String)* method detects three sentences in the paragraph and returns a list of span as [[0..50), [51..115), [116..218)]. The span represent the start and end position of a sentence in a given text. The returned list contains the span of each sentence in a paragraph. The term list of span or List<Span> mean the sentences or position of the sentences in a paragraph. The span representation of sentence is similar to the data structure of the *Coding* class. The *anchorOffset* and *focusOffset* of the *Coding* also represents the starting and ending position of the applied coding in a paragraph.

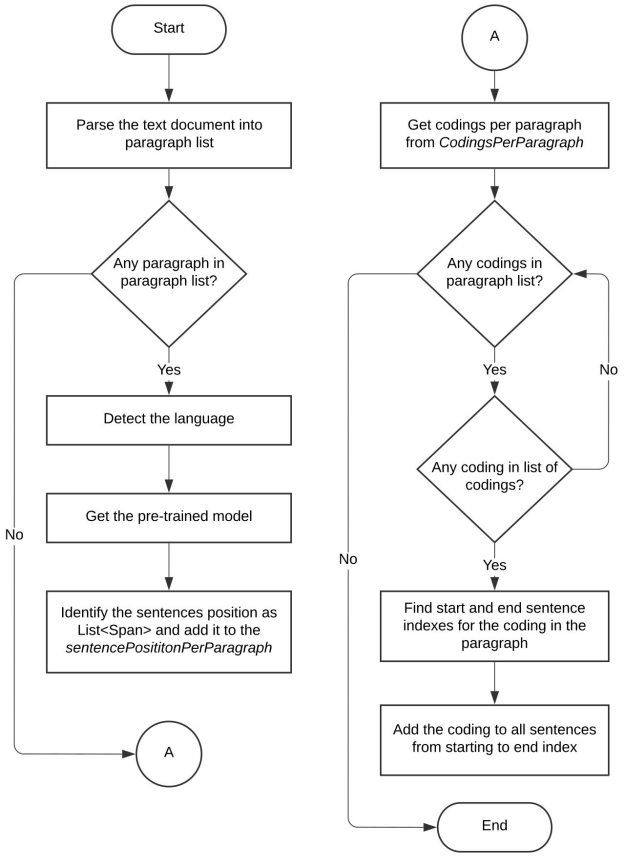


Figure 5.5: Codings per sentence detection flowchart

Figure 5.5 shows the flowchart of finding codings in a sentence for text document. We must first detect the sentences in a text document in order to find the codings. The procedure for determining the position of the sentences per paragraph is explained on the left side of the flowchart. The text document is parsed using JSOUP[2] library to obtain the paragraphs.

The *sentencePositionPerParagraph* attribute of the class is used to contain the position of sentences for each paragraph. For each paragraph, the language is detected to obtain the pre-trained model required by the *SentenceDetectorME* class, and the paragraph is passed to the *sentPosDetect(String)*, which returns a list of spans and is added to the *sentencePositionPerParagraph*.

The procedure to find codings in a sentence is explained on the right side of the flowchart. The paragraph is made up of sentences and if the first and last sentences contain coding, then all of the sentences in between will definitely contain that coding. Therefore, for each coding in a paragraph, we find the start and end index of a sentence in a paragraph containing the coding. *Coding* data structure is used to identify which codings belong to which sentence.

```
if (coding.getAnchorKey().equals(paragraphCodingKey)) {
        Span startSpan = positionOfSentencesInParagraph.stream().
           filter(sentenceSpans -> contains(sentenceSpans, coding.
           getAnchorOffset()))
    .collect(Collectors.toList()).stream().findFirst().orElseGet(()
         -> getStartSpan(positionOfSentencesInParagraph, coding.
        getAnchorOffset()));
        start = positionOfSentencesInParagraph.indexOf(startSpan);
}
```

Code 5.1: Sentence index finding of coding

Following the identification of the position of sentences per paragraph, the codings in each paragraph are obtained from the *CodingsPerParagraph* class. Code 5.1 demonstrates how starting index of the sentence containing the coding is determined.

For each coding in a paragraph's codings, the *codingKey* of the paragraph is retrieved from the *CodingsPerParagraph#codingKeys* attribute and set to *paragraphCodingKey* and the coding *anchorKey* is compared with the *paragraphCodingKey*; if it matches, the span containing the coding *anchorOffset* is determined using the *contains()* method which checks if the coding *anchorOffset* is in between the span. The index of the span will give the starting index of the sentence in a paragraph.

[2]https://jsoup.org/

Although the coding is present in the paragraph since we obtain the codings for each paragraph and iterate over each one, the beginning of the coding, i.e. *anchorKey*, may not correspond to the current *paragraphCodingKey*, as coding might have begun in any of the paragraphs before that. In this scenario, the first sentence's index, which is zero, is assumed to represent the start as the *anchorOffset* does not belong to any of the sentences. The Code 5.1 demonstrates how the starting index of the sentence containing the coding is determined.

Similarly, the *focusKey* and *focusOffset* are used to find the index of the sentence where coding ends. If a coding is present in the paragraph but the *focusKey* does not match the *paragraphCodingKey*, the index of the last sentence is considered to be the end.

There is a possibility that the list of spans does not contain the *anchorOffset* or *focusOffset* in a paragraph even if the paragraph *codingKey* matches *anchorKey* or *focusKey*. Consider the following coded paragraph with white spaces as shown in Figure 5.6.

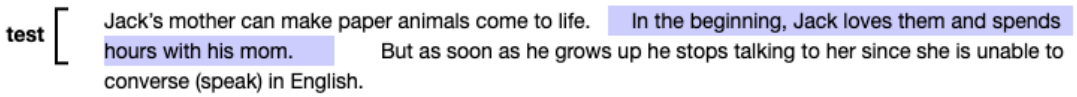


Figure 5.6: Coding white space in a paragraph

The *anchorKey* and *focusKey* of the coding are same and the value of *anchorOffset* is 53 and *focusOffset* is 128. When detecting sentences, the sentence detector disregards white spaces and returns the List<Span> as [[0..50), [57..121), [136..238)] and no span contains the *anchorOffset* or *focusOffset* of the coding.

To identify the sentences in such a scenario, the *CodingsPerSentence* contains *getStartSpan()* and *getEndSpan()* methods, which help to identify the start and end spans of the coding respectively. Code 5.2 and Code 5.3 demonstrate method implementation for determining the start and end span for coding.

```
Span startSpan = null;

for (Span span : spanList) {
if(anchorOffset<span.getStart())
   {
       startSpan = span;
       break;
    }
}
```

Code 5.2: startSpan of coding

```
Span endSpan = null;
Collections.reverse(spanList);
for (Span span : spanList) {
if (focusOffset > span.getEnd())
    {
       endSpan = span;
       break;
   }
}
```

Code 5.3: endSpan of coding

5.2 ICA by Code

The ICA by code has two aspects. One is to allow the user to view the agreement per code for F-Measure report and the other one is to allow the user to create ICA for a subset of the codesystem.

To view the agreement per code requires changes in both backend and frontend. The selection of a subset of the codesystem to create ICA requires only frontend refactoring. In the following the implementation changes are explained in two sub sections Backend and Frontend.

5.2.1 Backend

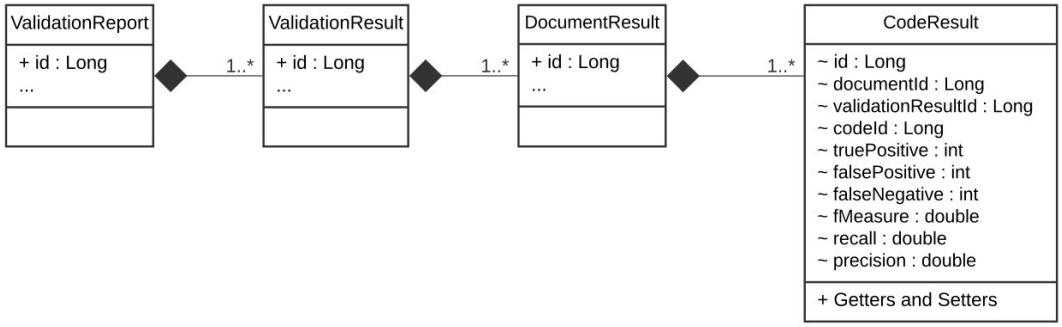


Figure 5.7: Simplefied UML class diagrams of *Report* data structure

Figure 5.7 shows the simplified UML class diagram of *Report* classes. The *ValidationReport* and *ValidationResult* class stores the information of the existing ICA metrics report. Each *ValidationResult* corresponds to one coder who is involved in the ICA report. The *DocumentResult* is used to store the information to provide the functionality of agreement per document for F-Measure report. The *DocumentResult* is used only during the F-Measure report generation and each *DocumentResult* corresponds to a document which is selected during report creation. These functionalities are already supported in the QDAcity.

To provide the functionality of agreement per code for F-Measure report, we need a class to store the necessary information for each code. Thereby we implemented a *CodeResult* class which stores the information per code per document. The F-Measure is a measure of intercoder accuracy which is calculated using precision and recall. Each term is explained as follows

- **True Positive (TP)** - It indicates both coders have coded in the given unit.

- **False Positive (FP)** - It indicates that baseline coder has not coded but compared coder has coded in the given unit.
- **False Negative (FN)** - It indicates that baseline coder has coded but compared coder has not coded in the given unit.
- **Precision**

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

- **Recall**

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

- **F-Measure**

$$\text{F-Measure} = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

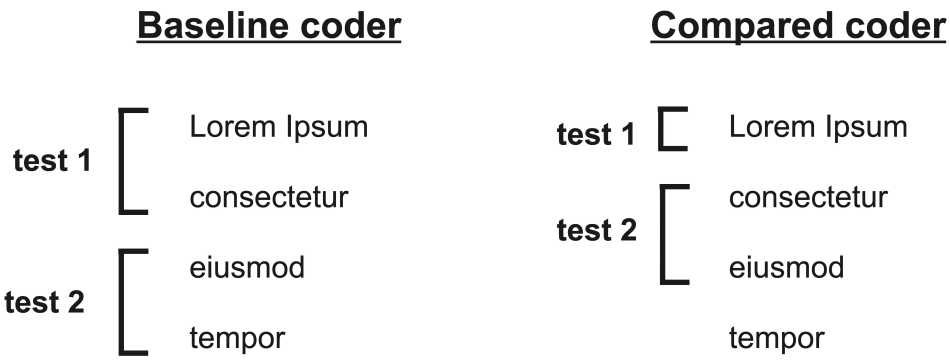


Figure 5.8: Codings of two coders

test 1

Paragraph	TP	FP	FN
Lorem Ipsum	1	0	0
consectetur	0	0	1
eiusmod	0	0	0
tempor	0	0	0
Total	**1**	**0**	**1**

test 2

Paragraph	TP	FP	FN
Lorem Ipsum	0	0	0
consectetur	0	1	0
eiusmod	1	0	0
tempor	0	0	1
Total	**1**	**1**	**1**

Code	Precision	Recall	F-Measure
test 1	1	0.5	0.666
test 2	0.5	0.5	0.5

Table 5.1: ICA per code

The calculation of ICA per code is explained using an example. Figure 5.8 shows the codings of baseline and compared coder who has coded the codes ‘test 1’ and ‘test 2’. Consider each line corresponding to a paragraph and evaluation unit as paragraph. The Table 5.1 demonstrates how the ICA F-Measure calculation is done for each code.

5.2.2 Frontend

The codesystem is a hierarchical structure of codes which is used to annotate the data. During the ICA creation, instead of generating the report for the whole codesystem, the option to provide the selection of subset of codesystem to generate a report gives user the flexibility to analyse ICA only for the required codes.

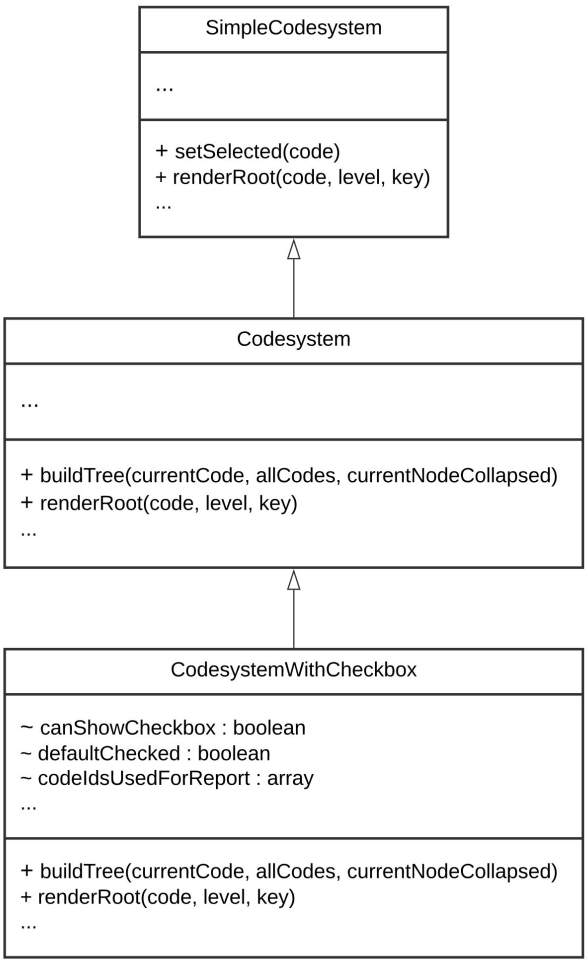


Figure 5.9: Simplified UML class diagram of *SimpleCodesystem*, *Codesystem* and *CodesystemWithCheckbox* components

Figure 5.9 layouts the UML class diagram of the codesystem class components. The class component *CodeSystem* extends *SimpleCodeSystem* and these components were already implemented in the QDAcity to render codesystem in CodingEditor. To provide the codesystem with checkbox, we implemented *CodeSystemWithCheckbox* class component which extends *CodeSystem*.

The *CodeSystemWithCheckbox* is used to select codes during creation, display selected codes and view results for selected codes during the analysis. To provide such functionality, the props were added to customize the checkbox rendering for each code. The props are

- *canShowCheckbox* - boolean value to decide if a code checkbox is required
- *codeIdsUsedForReport* - Array of long values identifying the code Ids that are selected during creation
- *defaultChecked* - boolean value to initially select or deselect all codes

The *buildTree()* method is used to create the codes structure hierarchy. Each code attributes is defined in this method and is defined based on the above mentioned props to decide if a code requires a checkbox or is it selected by default. The *renderRoot()* method is used to render every code in the codesystem using *SimpleCode* component.

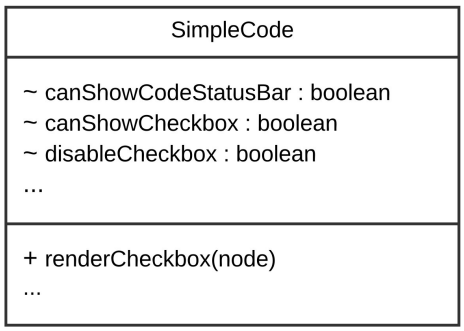


Figure 5.10: Simplified UML class diagram of *SimpleCode*

The *SimpleCode* class component is used to render a code and its properties. Figure 5.10 shows the UML class diagram of *SimpleCode* class. The *renderCheckbox()* method is added to render checkbox along with the code and default props are introduced to customize the code rendering. The default props are

- *canShowCodeStatusBar* - boolean value to decide if a status bar is required
- *canShowCheckbox* - boolean value to decide if a checkbox is required
- *disableCheckbox* - boolean value to enable or disable the checkbox selection

5.3 Agreement Queries

The agreement queries metric allows user to define the criteria and do comparison of multiple coders project with single coder or main project to analyse the ICA in the means of agreement, disagreement and agreement percentage value. The user can generate agreements for different agreement types such as code occurrence, code frequency and code intersection percentage and analyse agreements at coder, document and code level. The implementation is explained in three following sections Data Structure, General Process and Process Implementation.

5.3.1 Data Structure

Figure 5.11 outlines Agreement Queries data structure. The *AgreementQuery* class stores the overview and baseline coding details and has ten persistent and three non persistent properties. The *AgreementQuery* holds a value of type *AgreementType* to refer to the type of agreement created. *CODE_OCCURRENCE*, *CODE_FREQUENCY* and *CODE_AGREEMENT_BY_PERCENTAGE_INTERSECTION* are the three different types of agreements. The properties of *AgreementQuery* are

- *id* - unique identifier
- *agreementType* - refers to *AgreementType* enum
- *creatorId* - reference to user Id of the agreement query initiator
- *creatorName** - name of the creator who initiated agreement query
- *isReady* - boolean value to check if agreement query is completed or not. This attribute is provided because GAE lacks a good mechanism for synchronizing results from multiple asynchronous tasks in the task queue.
- *revisionId* - reference to the revision
- *projectId* - reference to the project
- *baselineProjectId* - reference to the baseline project
- *baselineProjectName** - name of the baseline project
- *baselineProjectType* - type of the baseline project
- *coderAgreements** - list of *CoderAgreement* class
- *considerTrueNegatives* - boolean value to check if true negatives are considered when *agreementType* is *CODE_OCCURRENCE*
- *minimumIntersectionPercentage* - minimum value for agreement when *agreementType* is *CODE_AGREEMENT_BY_PERCENTAGE_INTERSECTION*

Note: The starred(*) properties mean that they are non persistent which means they are not stored in the datastore.

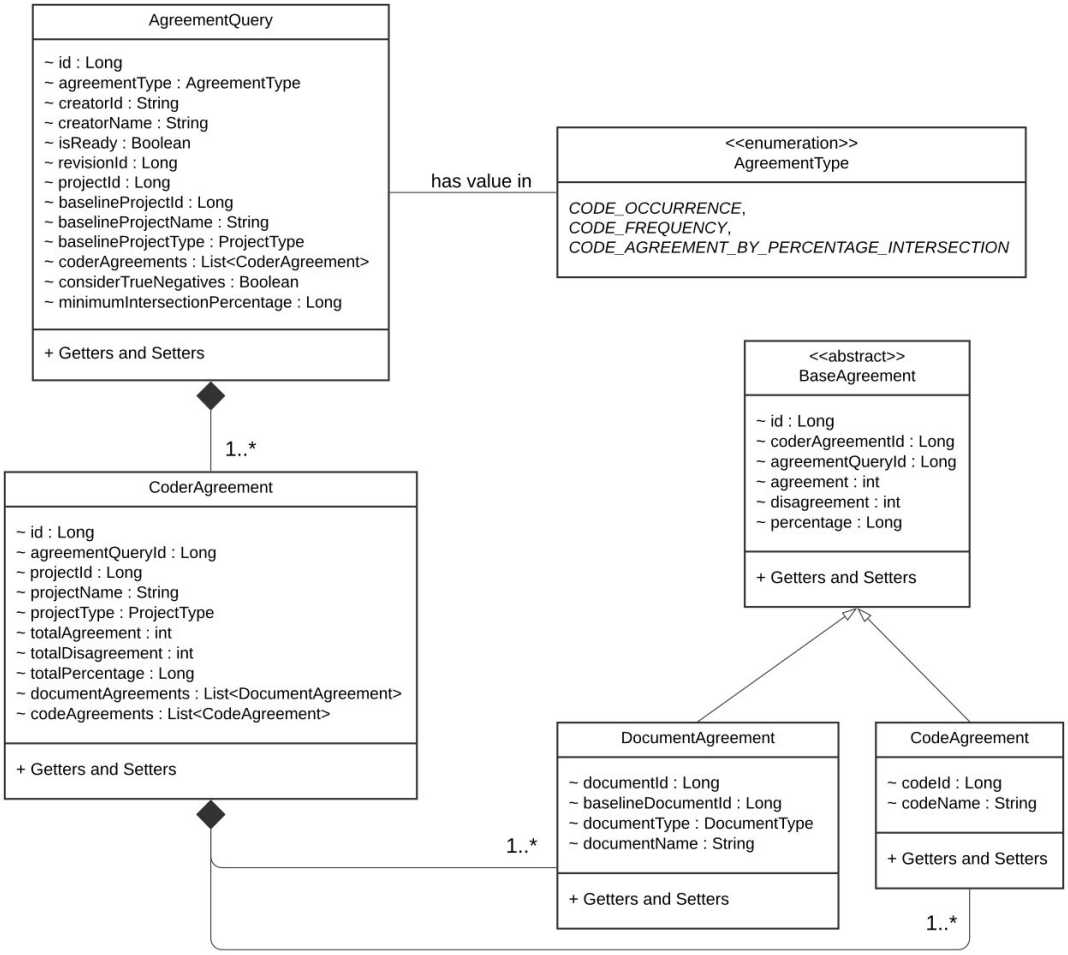


Figure 5.11: UML class diagram of agreement queries data structure

The *CoderAgreement* class is used to store the information of compared coders project. There can be several instances of *CoderAgreement* that belong to *AgreementQuery* and the number of instances is equal to the number of coders project selected for comparison. The *CoderAgreement* has the following properties

- *id* - unique identifier
- *agreementQueryId* - reference to the *AgreementQuery*
- *projectId* - reference to the project selected for comparison
- *projectName** - name of the project

- *projectType* - type of the project
- *totalAgreement* - total agreed value of documents or codes agreement
- *totalDisagreement* - total disagreed value of documents or codes agreement
- *totalPercentage* - ratio of *totalAgreement* to the sum of *totalAgreement* and *totalDisagreement*
- *documentAgreements** - list of *DocumentAgreement* class
- *codeAgreements** - list of *CodeAgreement* class

The *DocumentAgreement* and *CodeAgreement* class stores the information of document and code per coder respectively. The *DocumentAgreement* has document specific properties like *documentId*, *baselineDocumentId*, *documentType*, *documentName* whereas the *CodeAgreement* has code specific properties like *codeId*, *codeName*. Both these classes extend the abstract *BaseAgreement* class which has following properties.

- *id* - unique identifier
- *coderAgreementId* - reference to the *CoderAgreement*
- *agreementQueryId* - reference to the *AgreementQuery*
- *agreement* - calculated agreed value for the defined criteria
- *disagreement* - calculated disagreed value for the defined criteria
- *percentage* - ratio of *agreement* to the sum of *agreement* and *disagreement*

The *DocumentAgreement* has the values calculated per document for a coder while the *CodeAgreement* has the values calculated per code from all the documents for a coder. The number of instances of *DocumentAgreement* and *CodeAgreement* that belong to *CoderAgreement* depends on the number of selected documents and codes during creation respectively.

5.3.2 General Process

Figure 5.12 shows the flowchart of agreement query creation. The creation of an agreement query request is handled in the *AgreementQueriesEndpoint*. The endpoint first validates the input parameter for following criteria and on a failed validation throws the corresponding error code.

- User authorization - throws HTTP 401 Unauthorized

The following validation throws HTTP 400 Bad Request and the corresponding error code. The codes are in sync with frontend to throw specific error.

- No project selected for comparison - *EMPTY_COMPARED_PROJECTS*

- No selection of documents - *EMPTY_DOCUMENTS*
- No codes selected - *EMPTY_CODES*

The following validations are for agreement criteria value defined for agreement type *CODE_AGREEMENT_BY_PERCENTAGE_INTERSECTION*

- Value is empty - *EMPTY_MINIMUM_PERCENTAGE*
- Value less than 20 - *LESS_MINIMUM_PERCENTAGE*
- Value more than 100 - *MORE_MINIMUM_PERCENTAGE*

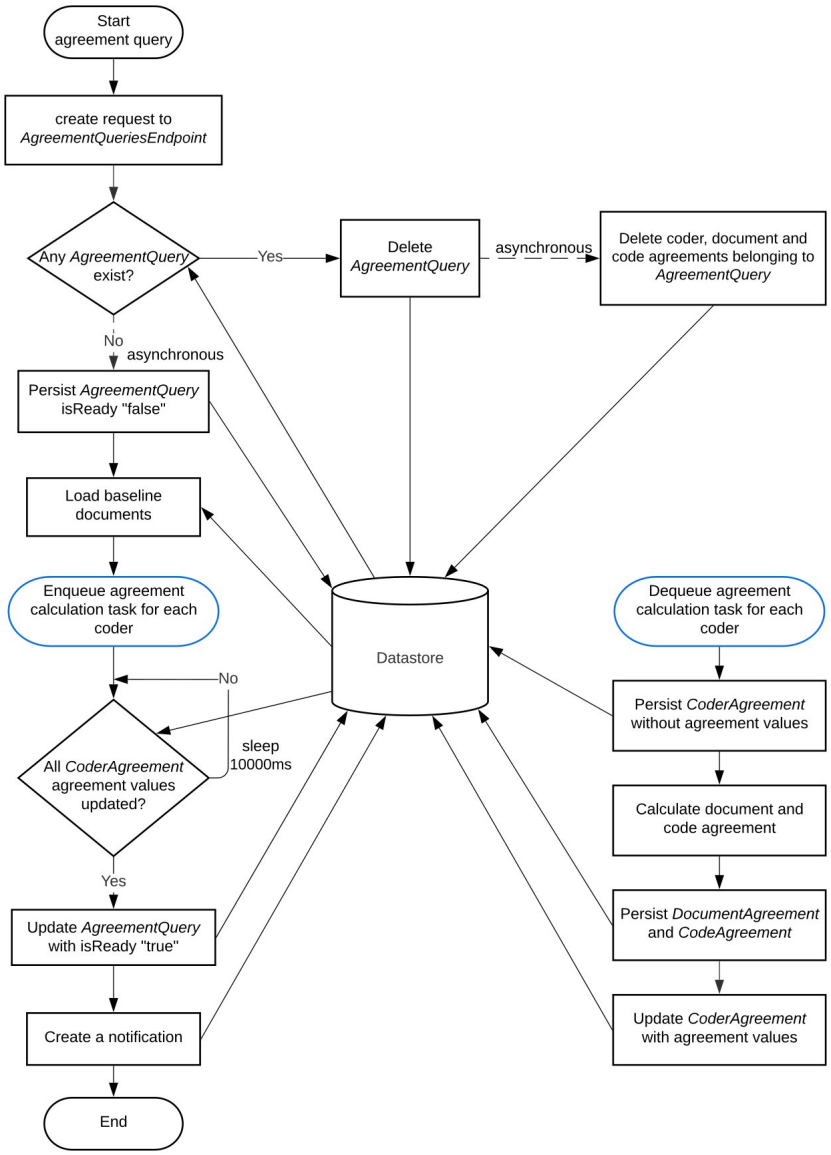


Figure 5.12: Flowchart of agreement query creation

After validation, the deletion of agreement query is done if any agreement query for the revision already exists in the datastore. The deletion of corresponding coder, document and code agreements are done asynchronously in a *DeferredCoderAgreementDeletion* class. The creation task is then added to deferred task *DeferredAgreementQuery* which runs asynchronously and the endpoint returns HTTP 204 No content response.

The asynchronous task runs and persist *AgreementQuery* in the datastore with *isReady* as false. The persistence creates an unique identifier which is required for reference in *CoderAgreement*. All the baseline project documents are loaded from datastore and based on the agreement type the *DeferredAgreement* task is created and added to the queue for each compared coder.

The enqueued tasks are dequeued by the handler and processed at a rate of 5/s. The handler runs the task which persist the *CoderAgreement* without agreement values to get the unique identifier required for reference in *DocumentAgreement* and *CodeAgreement*. The agreement and disagreement values of document and code are calculated using algorithm defined for each agreement type and are persisted in *DocumentAgreement* and *CodeAgreement* respectively.

The agreement values of the *CoderAgreement* is then updated using document agreement values. The agreement values of *CoderAgreement* can also be updated using code agreement values as the total agreement and total disagreement values of code and document is same for a coder.

After enqueuing tasks, the *DeferredAgreementQuery* class checks to see if all of the agreement values of the *CoderAgreement* have been updated; if not, it checks again after 10 seconds; if yes, it updates the *AgreementQuery#isReady* to true and creates a notification for the completion of the agreement query.

5.3.3 Process Implementation

In this subsection, we discuss the implementation of agreement query in detail. Figure 5.13 shows the UML class diagram of deferred task classes. The *DeferredAgreementQuery* implements the *DeferredTask* as it is run asynchronously. The class has four properties and nine methods. The properties are

- *evaluateAgreement* - reference to the input *EvaluateAgreement* class
- *user* - reference to the user who initiated agreement query
- *agreementQuery* - reference to the *AgreementQuery* class
- *baselineDocuments* - collection of document of the baseline project

When queues are dequeued, the *run()* is called which creates the task based on the agreement type with *DeferredAgreement* as reference and adds it to the

queue for each coder or project selected for comparison. After creating the task, the *waitForAllCoderAgreementsUpdateToFinish()* methods keeps the task waiting until all the values of all the *CoderAgreement* are updated. Once all the updates are done, the agreement query is updated as completed and a notification is created for the initiated user through the *createNotification()* method.

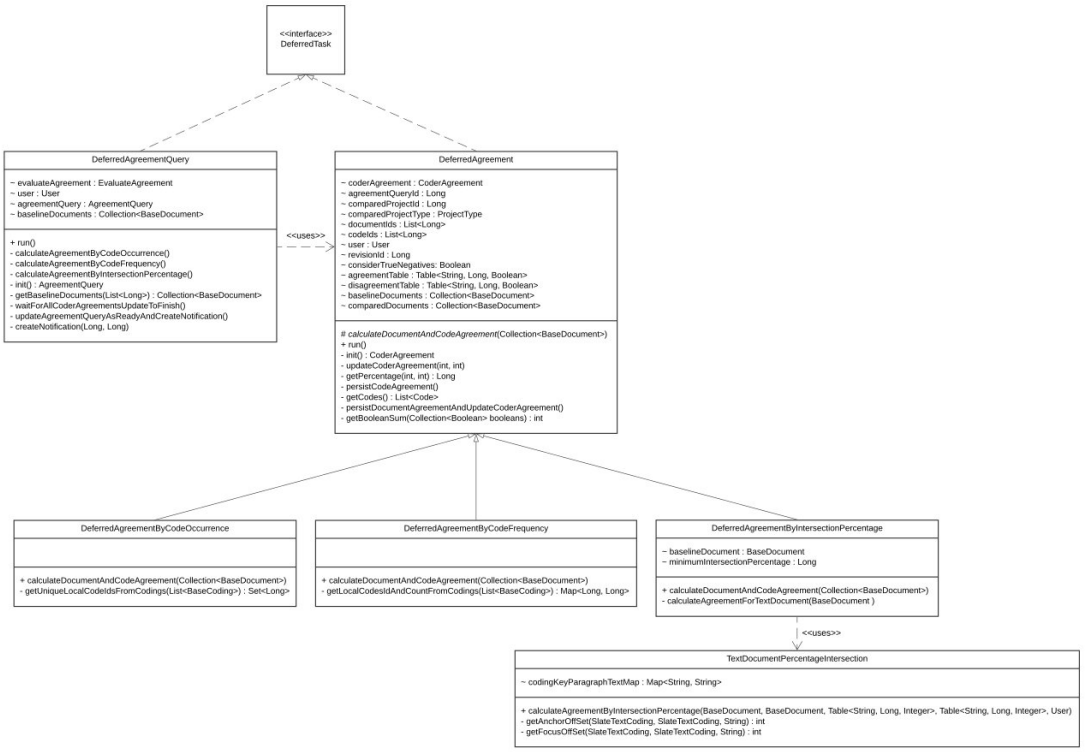


Figure 5.13: UML class diagram of agreement queries deferred tasks

Currently, we have implemented three agreement types code occurrence, code frequency and code intersection percentage. For each of these three types, we have *DeferredAgreementByCodeOccurrence*, *DeferredAgreementByCodeFrequency* and *DeferredAgreementByIntersectionPercentage* class with implementation logic. These classes extend the abstract *DeferredAgreement* class which implements *DeferredTask*. Only the calculation logic of agreement and disagreement for documents and codes based on agreement type is performed in the child class, the persistence of the data is handled generically in the parent class.

The implementation follows a behavioural pattern known as template design pattern. The abstract method *calculateDocumentAndCodeAgreement()* in the parent class is implemented in each child class. When the *DeferredAgreement* task is dequeued, the *run()* method of the class is invoked, which calls the *calculateDocumentAndCodeAgreement()* method. *DeferredAgreement* properties are

- *coderAgreement* - reference to the *CoderAgreement* class
- *agreementQueryId* - reference to the *AgreementQuery*
- *comparedProjectId* - reference to the project
- *comparedProjectType* - type of the project
- *documentIds* - list of document Ids selected for comparison
- *codeIds* - list of code Ids selected for comparison
- *user* - reference to the user who initiated agreement query
- *revisionId* - reference to the revision
- *considerTrueNegatives* - boolean value to decide true negatives as agreement or disagreement when agreement type is *CODE_ OCCURRENCE*
- *agreementTable* - *HashBasedTable* based on multi maps to hold the agreement values of document and code in a single table. The example Table 5.2 depicts typical *agreementTable*. It contains boolean value 'true' if there is an agreement and sum of the boolean values across document and code gives the total agreement for the document and code respectively.

	Code 1	**Code 2**	agreement over document
Document 1	true		1
Document 2	true	true	2
Document 3		true	1
agreement over code	2	2	Total = 4

Table 5.2: Example usage of *HashBasedTable*

- *disagreementTable* - *HashBasedTable* similar to *agreementTable* except it contains boolean value 'true' if there is disagreement
- *baselineDocuments* - collection of documents of the baseline project
- *comparedDocuments* - collection of documents of the compared project

The implementation logic of all the three agreement types are explained as follows:

Agreement by Code Occurrence

The agreement is based on the code occurrence in the document and is implemented in *DeferredAgreementByCodeOccurrence* class for all the document types. The *getUniqueLocalCodeIdsFromCodings()* methods gets all the *codeIds* in the document for baseline and compared coder and then checks if both the coders

have coded the code or not. Table 5.3 depicts the agreement and disagreement of a code for coders.

	Coder A	**Coder B**	Agreement
Code 1	x	x	yes
Code 2	x		No
Code 3			depends on selection

Table 5.3: Agreement table for code occurrence

The 'Code 3' is true negative because neither coder coded the code. The true negative is considered as agreement if *considerTrueNegatives* is selected else it is neither agreed nor disagreed. From the Table 5.3, the agreement percentage is $1/2 = 50\%$ and with *considerTrueNegatives* selected it will be $2/3 = 67\%$.

Agreement by Code Frequency

The agreement is based on number of times the code occurs in the document and is implemented in *DeferredAgreementByCodeFrequency* class for all the document types. The single code can be applied more than once in a document, this agreement type helps to find if both coders has coded a code and the code assignment frequency of both coders are same. The *getLocalCodesIdAndCountFromCodings()* gets the *codeId* and its frequency count as a map in the document for baseline and compared coder. If both coders have coded a code equal number of times then it is considered as agreed else it is disagreed. The true negatives are neither agreed nor disagreed as it is not coded by both coders and the frequency will always be zero.

Agreement by Intersection Percentage

The complex of all the three is to find the percentage of intersection of the coded region of two coders. The agreement is based on how much both coders have coded the same coding segment. The *DeferredAgreementByIntersectionPercentage* class has the implementation and supports only text document. It uses *TextDocumentPercentageIntersection* class to find the percentage intersection of coded segments in a text document.

To find the coded segments intersection in a text document, we count the number of characters in a coded text. The agreement is calculated on how much the compared coding intersect with the baseline coding. Consider the baseline and compared coder coding in a text document as shown in the Figure 5.14. The *TextDocumentPercentageIntersection* calculates the agreement per document and uses two *HashBasedTable* to contain the characters count. One table contains the number of coded characters per paragraph by baseline coder and another table

contains the number of agreed characters of compared coder with baseline coder per paragraph.

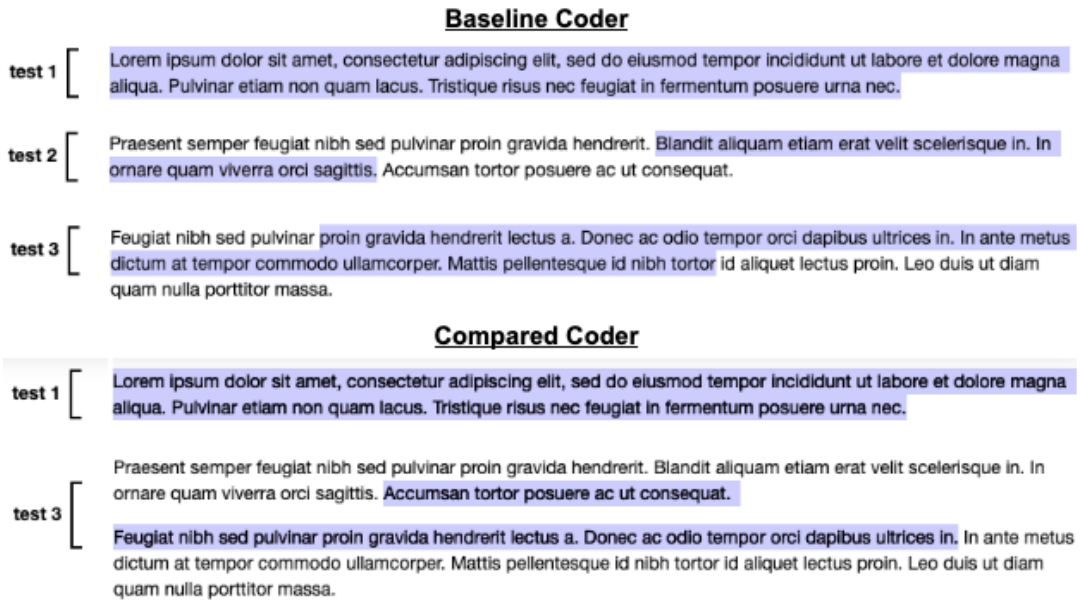


Figure 5.14: Baseline and compared coders coding in a text document

HashBasedTable has *codingKey* which is unique key to represent a paragraph in a text document, *codeId* of the code and the number of characters count. Table 5.4 shows the generated *HashBasedTable* having baseline and agreed characters for the example Figure 5.14. *HashBasedTable* Table 5.4 is shown with paragraph number and code name instead of *codingKey* and *codeId* for the better understanding. Consider that test 1, test 2 and test 3 are three codes, while paragraph 1, paragraph 2 and paragraph 3 represent three paragraphs in order.

Baseline characters table

	test 1	**test 2**	**test 3**
paragraph 1	213		
paragraph 2		86	
paragraph 3			167

Agreed characters table

	test 1	**test 3**
paragraph 1	213	
paragraph 3		80

Table 5.4: Characters count table

In paragraph 1, both coders used code test 1 to code the identical segment with characters equal to 213. The baseline coder coded a segment comprising 86 characters with test 2 in paragraph 2, but the compared coder did not code the test 2, hence the agreed characters table does not include test 2. In paragraph

3, the baseline coder coded test 3 comprising 167 characters and the compared coder coded a portion of paragraph 2 and a portion of paragraph 3 with test 3. Only the highlighted segment in the Figure 5.15 represents the characters in agreement for the baseline and compared coders in paragraph 3, which is equal to 80 characters.

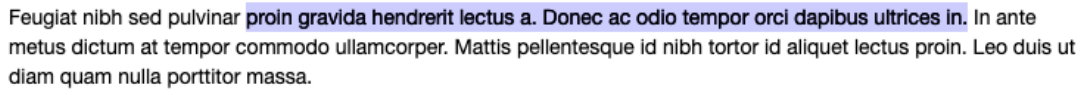


Figure 5.15: Agreed characters in paragraph 3

Table 5.5 shows the agreement and disagreement table. The ratio of agreed characters to baseline characters gives the agreement percentage. If "minimum intersection percentage" is taken as 80%, then for test1: 213/213 = 100% which is more than 80% so agreed. For test 2: 0/80 = 0% so disagreement, test 3: 80/167 = 48% which is less than 80% so disagreed.

Agreement table

	test 1
text document	true

Disagreement table

	test 2	**test 3**
text document	true	true

Table 5.5: Agreement and Disagreement tables

5.4 User Interface (UI)

The ICA has two sections, one is Report where reports can be generated for the existing metrics and another section is Agreement Queries which is a new metric to analyse the agreements for different agreement types. The UI modifications are discussed in two subsections Report and Agreement Queries.

5.4.1 Report

Figure 5.16 shows the initial view on navigating to the ICA which contains the list of generated reports. The contents of the page are explained as numbered in the screenshot.

1. Sidebar: It contains the heading 'Intercoder Agreement' and two menus 'Report' and 'Agreement Queries'.
2. ICA content header: It contains the name of the revision project and revision number on the right and back button on the left to navigate back to project dashboard.

3. Report list header: It contains the report headers and a button to create report.
4. Report hover: On hovering over the report, the background turns grey and provides an option to select report to view and delete button to delete the report.

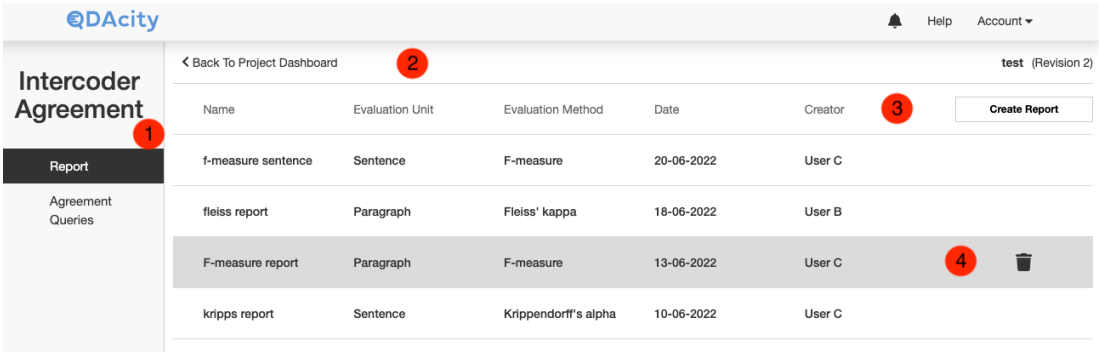


Figure 5.16: ICA screen with list of generated reports

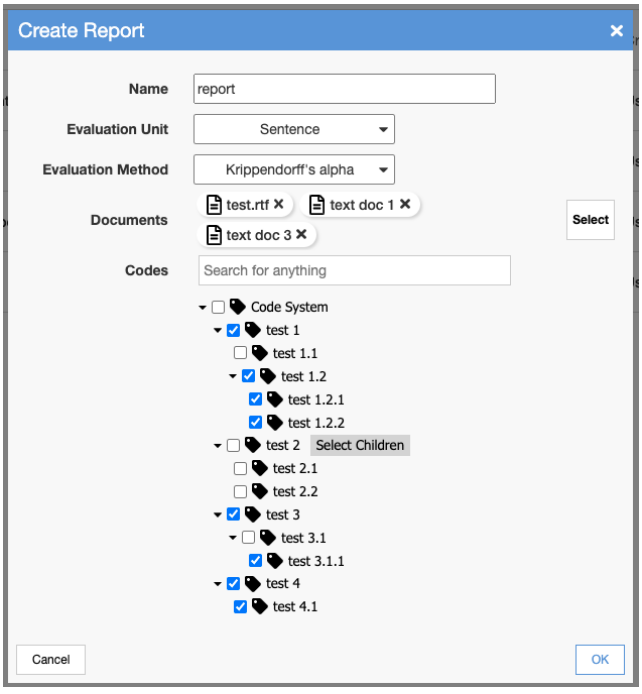


Figure 5.17: Create report modal dialog

Figure 5.17 shows the refactored create report modal. The report creation form includes a input field for name as well as three dropdown menus for selecting the evaluation unit, evaluation method and documents. The codes section is used

to select the codesystem's subset. The codesystem with checkbox is the usage of *CodesystemWithCheckbox* component discussed in subsection 5.2.2. When a user hovers over the code, if it has children, an option to select or deselect the children will appear next to the code, as shown for code 'test 2'. The blue tick mark indicates that the code is selected, while the empty white box indicates that it is not.

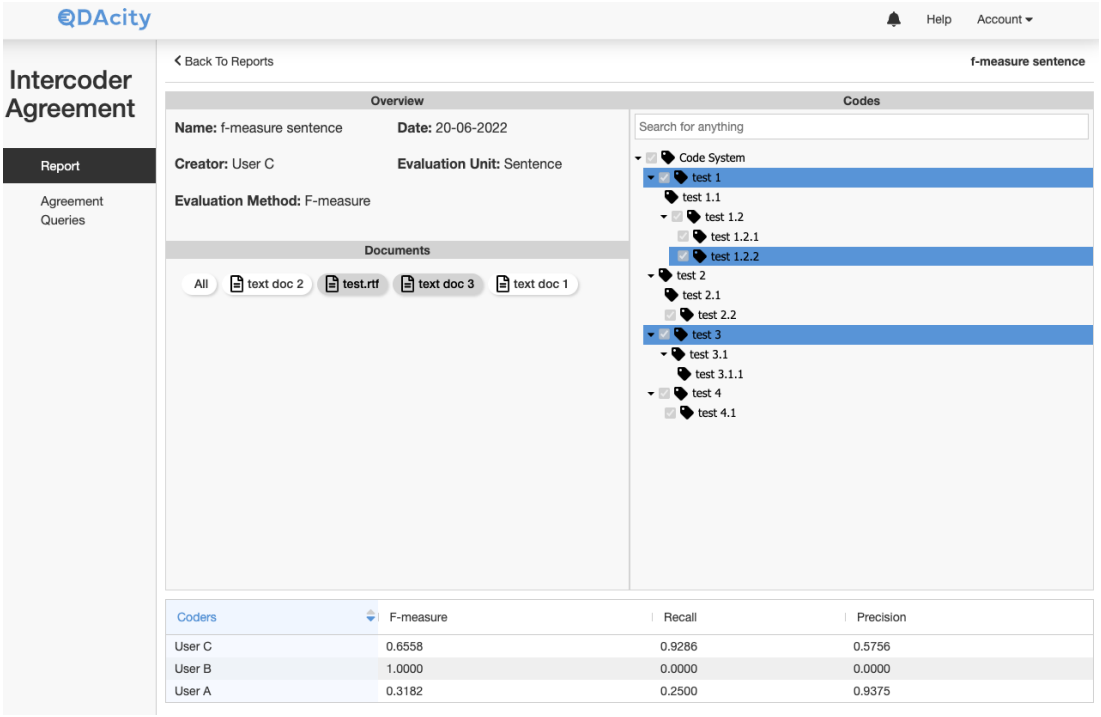


Figure 5.18: F-Measure report

Figure 5.18 shows the F-Measure report. The overview, documents and codes sections, as well as the result table, are all on a single page in the report. The result is displayed for all coders and the results for evaluated documents and codes can be viewed by selecting them.

The document selection is highlighted in grey, while the code selection is highlighted in blue. Another usage of *CodesystemWithCheckbox* component is the codes interface in F-measure report. The codes with checkboxes represent that the codes are used for report creation and can be selected to view the result for that code, while the codes without checkboxes are not used for report creation. The result can be viewed for any combination of documents and codes selection.

The report view for Krippendorf's Alpha and Fleiss' Kappa are identical. Figure 5.19 depicts the report generated for Fleiss' kappa. The report includes an overview section and a result table. In a result table, evaluated documents are

displayed in the first row with their average at the row end, while evaluated codes are displayed in the first column with their average at the column end.

The codes are displayed in a hierarchical structure that is analogous to the codesytem tree structure. With the sorting button provided in the column header, the table can be sorted in ascending or descending order based on column values. The 'Export Table' button allows users to save the result table as a CSV document.

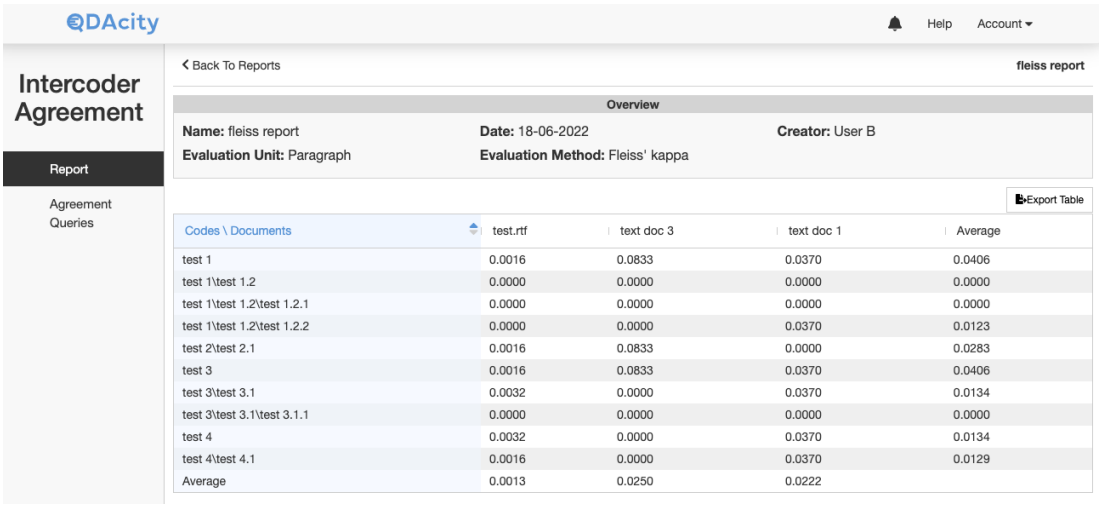


Figure 5.19: Krippendorf's Alpha and Fleiss' Kappa report

5.4.2 Agreement Queries

The agreement query creation page is depicted in the Figure 5.20. Dropdown menus are provided for selecting baseline coding, compared codings and documents. For baseline coding, only one coding can be chosen, whereas multiple codings can be chosen for compared coding. The 'more' option in the documents indicates that many documents have been selected but are not visible on the screen and clicking on 'more' will open a dropdown displaying all of the selected and deselected documents available for that revision. Codes are selected using a checkbox.

The selection of agreement type is provided using radio buttons. Hovering the mouse over the question mark of the agreement type will display a description of that type. When the agreement type "Code Occurrenc" is selected, "Consider True Negatives" with a checkbox appears, and when the agreement type "Code Agreement By Intersection Percentage" is selected, "minimum intersection percentage" with an input field to enter a number appears. After entering all of the required information, an agreement query can be initiated by clicking the 'create' button.

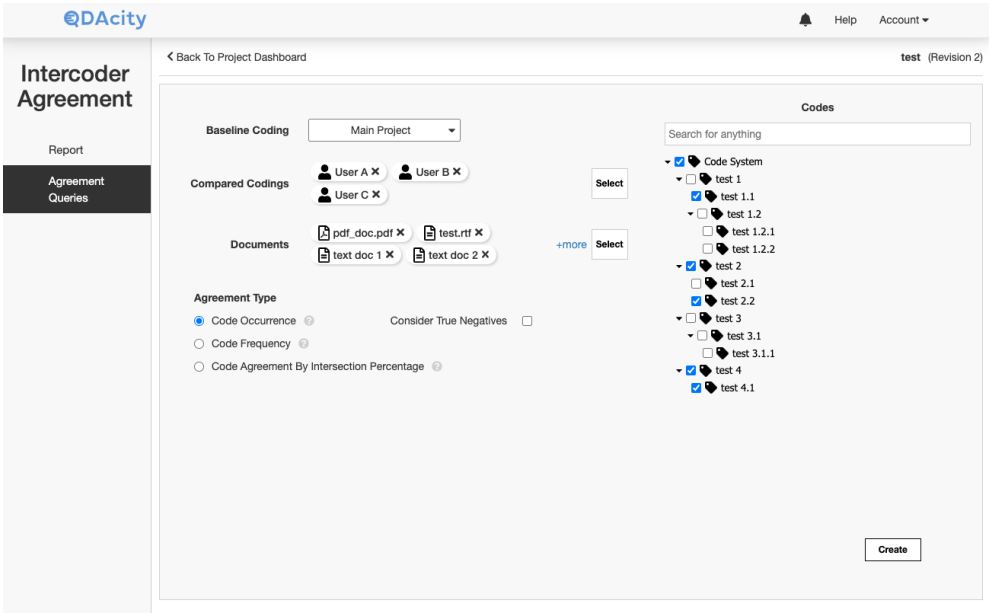


Figure 5.20: Agreement query creation

Figure 5.21 depicts the consolidated agreement query view. The top of the page includes overview with 'export' and 'create new' buttons. The overview has baseline coding as 'test' which is the name of the baseline project. The export function can be used to export the tables for all or specific coders, whereas the create new function can be used to create a new agreement query.

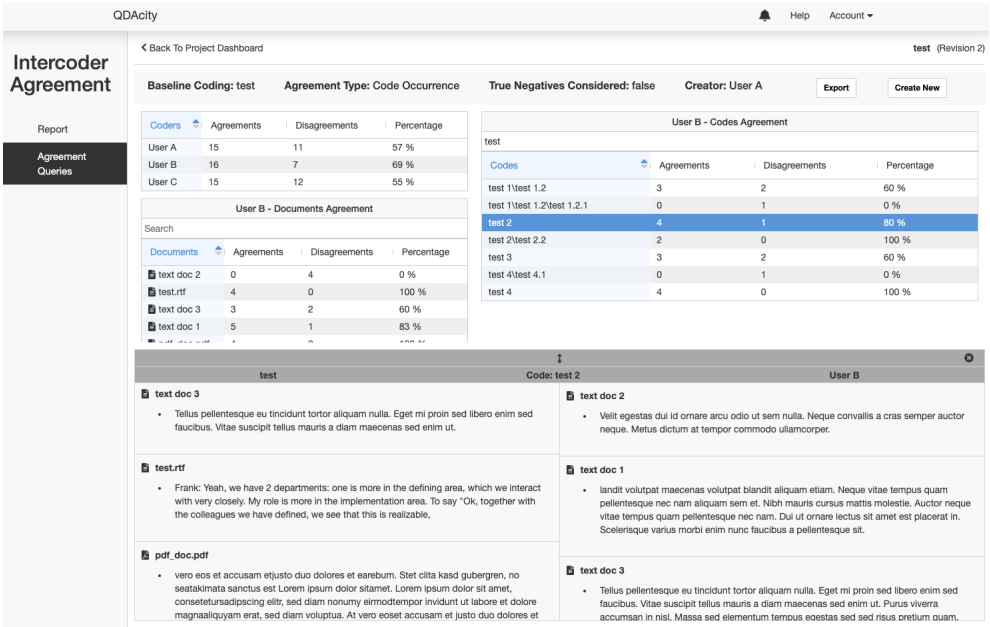


Figure 5.21: Agreement query view

Three agreement tables are shown, each with the agreement, disagreement and percentage values. The first table which shows the total agreements for each coder is displayed initially. On selecting a specific coder agreement, the documents and codes agreement tables for that coder are displayed. These tables have a header with the user name associated with the agreements, as well as a search bar for filtering the table. The document names are displayed with a document icon that corresponds to the type of the document.

The coder 'User B' agreement is selected in the Figure 5.21, and the documents and codes agreement for 'User B' are displayed. The codes agreement has search value 'test' and table is filtered with code names matching the search value 'test'. The column header of each table contains sort buttons that allow users to sort the table in ascending or descending order based on the column values.

A single code agreement can be selected from codes agreement table to view the codings applied for that code by the coders. When a code agreement is selected, a resizable section appears displaying the codings in each document for that code. The codings are shown in a two column comparison view, with the baseline codings on the left and the selected coder codings on the right. In the Figure 5.21, the code 'test 2' is selected and the codings applied in all the documents for baseline coder (test) and compared coder (User B) are displayed.

5.5 Agreement Queries vs MAXQDA ICA

The agreement queries in QDAcity is inspired by ICA in MAXQDA. The feature set is similar but there are some key differences:

- QDAcity allows comparison of multiple coders with a single coder. MAXQDA allows comparison of only two coders.
- For agreement type "Code Intersection Percentage", QDAcity supports only text document and agreement is calculated on how much the compared coding is intersecting with baseline coding. MAXQDA supports different document types and provide options to choose segment of coder 1 documents or coder 2 documents or segments of both the documents to find the intersection.
- QDAcity provides an agreement table of coders, documents and codes. MAXQDA provides agreement table of document and code as it allows comparison of only two coders.
- On selection of code agreement, QDAcity displays baseline and compared coder coded segments in different documents for the code in a two column comparison view. MAXQDA displays the coded segment in a plain view.

6 Challenges

6.1 Identification of Sentence for Coding

The sentence detection and finding the codings in a sentence was discussed in the chapter 5. There is a scenario where finding a sentence for the coding is ambiguous. Consider the Figure 6.1 where coding is applied to just white spaces in a paragraph. We could find the position of the sentences using library that provides a list of spans without taking white space into account. Because the coding only represents white space, it is difficult to determine which sentence the coding belongs to. The coding does not represent anything and assigning it to any of the sentences would vary the result. Therefore, such a scenario is ignored.

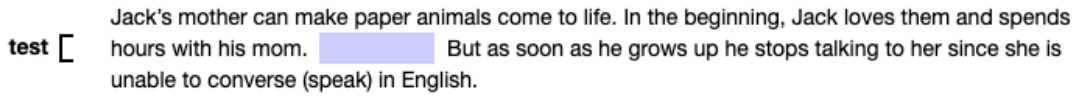


Figure 6.1: Coding of white space

6.2 Coding Intersection - PDF Document

The agreement type "code intersection percentage" in Agreement Queries is only implemented for text documents, while an attempt was made to incorporate it for PDF documents but this effort was not successful for a number of reasons. The PDF document uses two different forms of coding: text coding, which is used to code texts and area coding, which is often used to code images in a document. The challenges in finding the intersection of coding for each type are discussed in subsections.

6.2.1 Text Coding

The PDF document is rendered in the frontend using the PDF.js[1] library. Figure 6.2 shows the PDF document and its HTML representation. Unlike a text document which has one span per paragraph with a unique *codingKey*, the PDF has many spans with data-key in a paragraph. The data-key is a *codingKey* for the PDF document. Each span is not equal to a line as shown in the Figure 6.2 and structure of span depends on the parsing by PDF.js which varies for every PDF document.

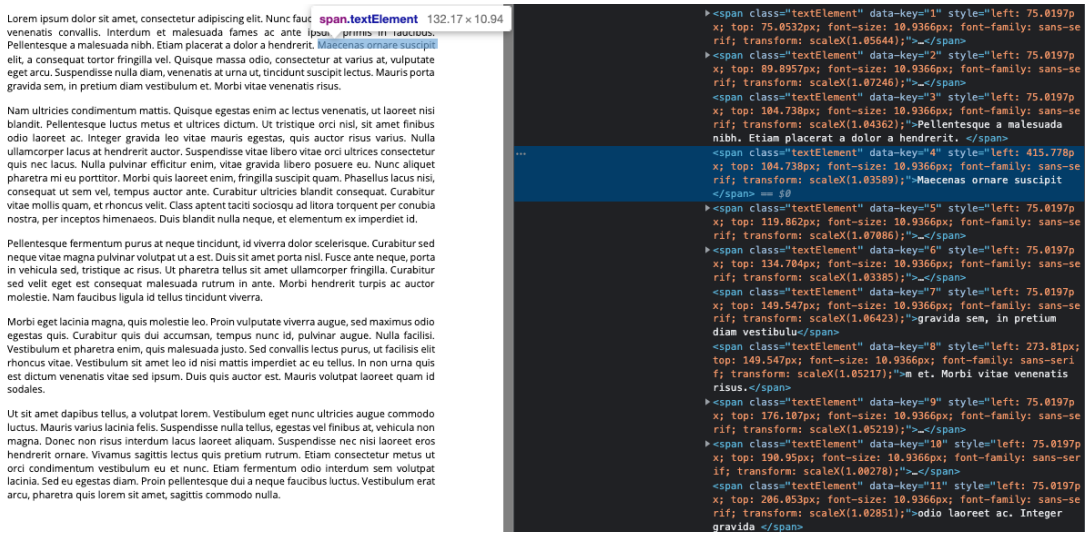


Figure 6.2: PDF document and its HTML representation

The *Coding* data structure provides the start of the coding in terms of *startPage*, *anchorKey* and *anchorOffSet* and end of the coding in terms of *endPage*, *focusKey* and *focusOffSet*. The *anchorKey* and *focusKey* holds one of the *codingKey* (data-key) in which the coding starts and ends. The *anchorOffSet* and *focusOffSet* is the start and end position of the text in a *codingKey*. Therefore, to find the coding intersection we need to know the text in each *codingKey* to get the characters count from *anchorKey anchorOffSet* to *focusKey focusOffSet*. So, in order to access the text of each *codingKey*, we must parse the PDF in the backend in the same way PDF.js renders in the frontend.

There are many Java libraries which parses the PDF. The Apache PDFBox[2] library was tried with custom modifications and was successful to some extent as it could parse the PDF similar to PDF.js for some PDF documents but after testing on numerous different documents we found that the parsing is not consistent and requires in depth analysis of the library.

[1] https://mozilla.github.io/pdf.js/
[2] https://pdfbox.apache.org/

6.2.2 Area Coding

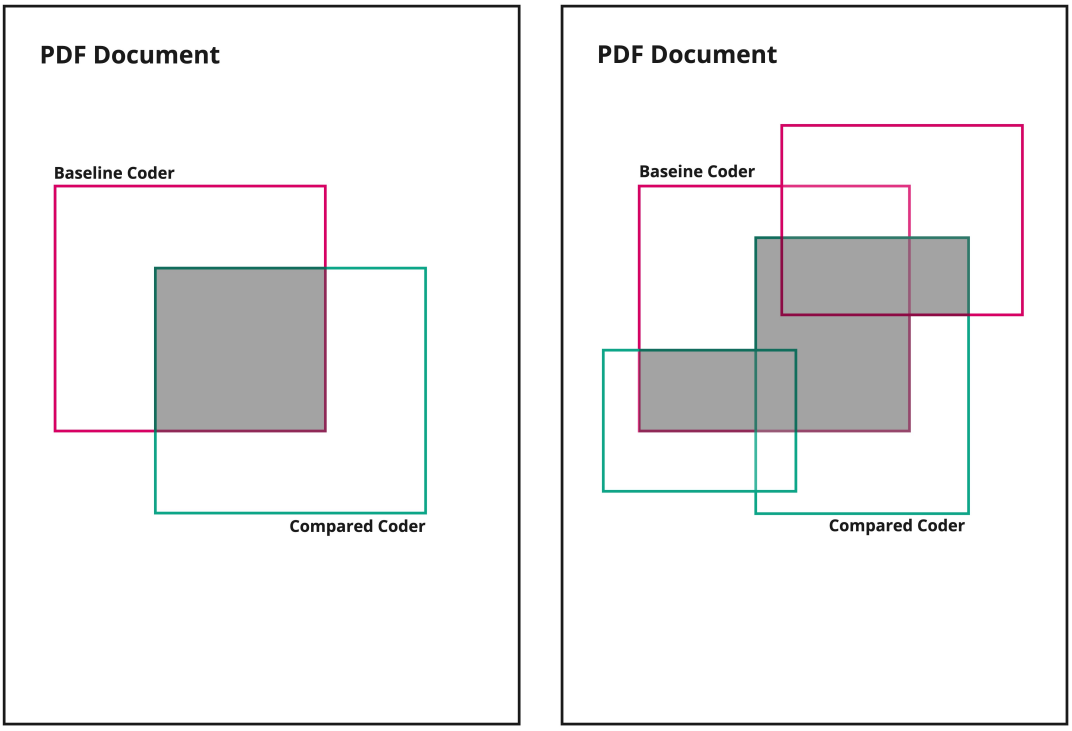


Figure 6.3: Area coding example in PDF document

Figure 6.3 shows the area coding example with coded region of two coders in PDF document. Consider the baseline coder coding in red and compared coder coding in green. The shaded region in black represent the intersection of the coders codings.

The area coding is rectangular in shape and the *Coding* data structure includes the x and y positions, as well as the height and width of the rectangle. We can calculate the area of any rectangle using these values. The PDF document on the left depicts a simple scenario in which each coder has only one coding for a code. Finding the intersection of two rectangles is simple because the end result is another rectangle and there is a ready formula.

The PDF document on the right depicts a complex scenario in which each coder has multiple codings for a code. Because the intersection of these codings does not represent a rectangle, determining the area of intersection of multiple codings is a difficult task requiring complex logic.

7 Evaluation

In this chapter, we evaluate the implementation provided in chapter 5 against the functional and non-functional requirements defined in the chapter 3.

7.1 Functional Requirements

1. **QDAcity shall provide new UI for ICA**

   This requirement is fulfilled.

   The validation projects section of each revision in the project dashboard includes the ICA button which satisfies the requirement 1.1. As discussed in the subsection 5.4.1 with Figure 5.16, the new UI for ICA has two sidebar menus 'Report' and 'Agreement Queries' and displays a list of generated reports and provides a button to create report on navigating to the ICA. As a result, requirements 1.2 and 1.3 are satisfied.

   The ICA results are displayed in a table, as explained in section 5.4, a common *Table* component is used for all of the results tables. As shown in Figure 5.19 and Figure 5.21, the table features sorting buttons in the column header that allow the user to sort the table in ascending or descending order based on the column values and the table containing the codes is hierarchically constructed, similar to the codesystem tree. Therefore, requirements 1.4 and 1.5 are satisfied.

   When creating a report, the report title is checked for emptiness, as are the documents and codes for at least one selection and appropriate errors are thrown as shown in Figure 7.1. The handling of errors for agreement queries by throwing an exception with error codes is discussed in subsection 5.3.2. The frontend constructs the error message in accordance with the error code and throws error as shown in Figure 7.2. As a result, requirement 1.6 is satisfied.

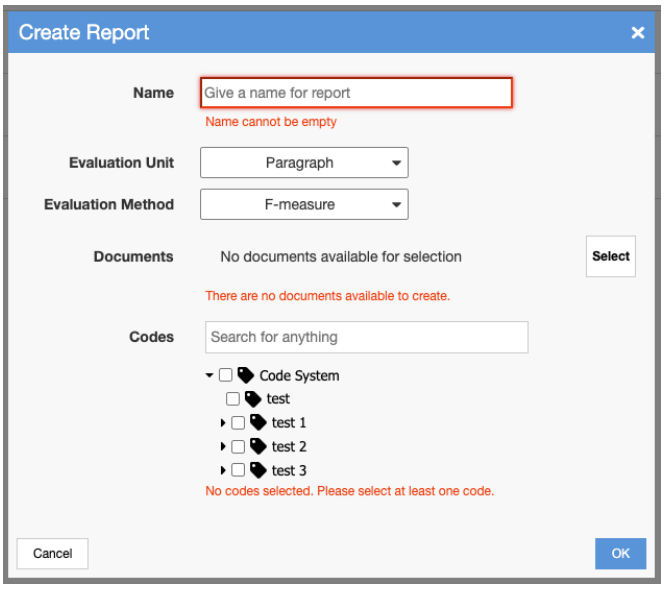


Figure 7.1: Error handling report creation

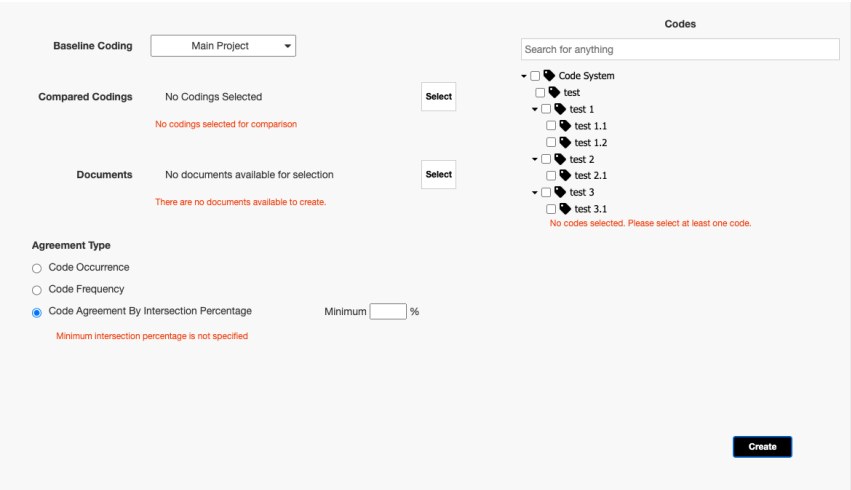


Figure 7.2: Error handling agreement queries creation

2. **QDAcity shall be able to allow only authorized users to create ICA from projects**

   This requirement is fulfilled.

   The API endpoint for creating reports *validation.createReport* and agreement queries *agreementQueries.createAgreementQuery* validates the user's authorization. If the user is not authorized, a HTTP 401 unauthorized error is thrown. In the frontend, the create button is made visible only to authorized users.

3. QDAcity shall be able to allow only authorized users to view ICA from projects and validation projects

This requirement is fulfilled.

The API endpoint to get report *validation.listReportsForRevision* and agreement queries *agreementQueries.getAgreementQuery* validates the user's authorization. If the user is part of the project then the data is retrieved and the user can view the generated ICA. If user's authorization fails, HTTP 401 unauthorized error is thrown.

4. Report

4.1. Report Creation

This requirement is fulfilled.

As discussed in the section 5.1, report generation using evaluation unit as sentence is implemented, satisfying requirement 4.1.1 The implementation details discussed in section 5.2 and Figure 5.17 show how a subset of the codesystem can be selected for report generation, satisfies requirement 4.1.2.

4.2. Report View

This requirement is fulfilled.

4.2.1. As discussed in subsection 5.4.1 with Figure 5.18 and Figure 5.19, the report includes an overview section containing report name, evaluation unit, evaluation method, creation date and creator name.

4.2.2. F-Measure Report

As discussed in Figure 5.18, the evaluated documents for the report are displayed on the screen as chips and the codesystem is displayed with evaluated codes having tick box next to them. The user can view the agreement for the document and code by selecting the document chip or the code with a tick box.

We also discussed the implementation changes required to allow users to view the agreement per code in section 5.2. As a result, the report enables the user to view the agreement for any document and code combination, thus meeting the requirements 4.2.2.1 and 4.2.2.2.

4.2.3. Krippendorff's Alpha and Fleiss' Kappa Report

As discussed in subsection 5.4.1 with Figure 5.19, the report includes a result table with the evaluated documents and codes that

meet the requirement 4.2.3.1. To satisfy requirement 4.2.3.2, the document and code averages are displayed at the row and column ends of the result table. The 'Export Table' button located above the result table allows the user to export the result table as a CSV document, fulfilling requirement 4.2.3.3.

5. Agreement Queries

5.1. Agreement Query Creation

This requirement is fulfilled.

As discussed in subsection 5.3.1, the agreement query data structure is designed to include one baseline and multiple compared coders information. The frontend enables selection of just one baseline and multiple compared coders coding as discussed in subsection 5.4.2 which meet the requirements 5.1.1 and 5.1.2.

The document and code selection options for the user, as well as the provision of three agreement types with a radio button to ensure only one type can be selected are discussed in subsection 5.4.2 with Figure 5.20. This satisfies requirements 5.1.3 and 5.1.4.

When the agreement type "Code Occurrence" is selected, the option "Consider True Negatives" is provided with a checkbox as shown in Figure 5.20 and is hidden if another agreement type is selected. When the "Code Agreement by Intersection Percentage" type is selected, an input field with the label "minimum intersection percentage" is displayed for the user to enter a minimum agreement criteria value between 20 and 100 and it is hidden when another agreement type is selected. The frontend handles the functionality of showing and hiding the agreement type fields while the backend handles the calculation based on these values as discussed in subsection 5.3.3, thus satisfying requirements 5.1.5 and 5.1.6.

5.2. Agreement Query view

This requirement is partially fulfilled as 5.2.5 is not met.

The creation process of an agreement query and implementation details of each agreement type is discussed in subsection 5.3.2 and subsection 5.3.3. The generated agreement query is shown in Figure 5.21 which includes an overview and coders agreement table on initial viewing and the documents and codes agreement tables are displayed for a coder upon selection of a coder agreement. As a result, the requirements 5.2.1 and 5.2.2 are met.

The documents and codes agreement table provides a search box to filter the table with document and code names as discussed in subsection 5.4.2 with an example, which meets the requirement 5.2.3.

When a code agreement is selected, the applied codings of the baseline and selected coder are displayed in a two-column view. However, when the document is selected, the codings for the combination of document and code are not displayed. As a result, requirement 5.2.4 is satisfied while requirement 5.2.5 is not.

The ability to export the tables to a CSV document via the 'Export' button, as well as the ability to create a new agreement query using the 'Create New' button, were also discussed in subsection 5.4.2 to meet requirements 5.2.6 and 5.2.7.

5.3. QDAcity shall be able to delete the previously created agreement query when initiating a new one

This requirement is fulfilled.

As discussed in subsection 5.3.2, when the new agreement query creation is initiated, the API endpoint *agreementQueries.createAgreementQuery* checks for existence of agreement query for the revision. If an agreement query exist for the revision, it is deleted and the deletion of coder, document and code agreements associated with it is added to a deferred task before initiating a new one.

6. QDAcity shall be able to notify the initiating user with the QDAcity notification feature upon successful creation of report or agreement query

This requirement is fulfilled.

As discussed in subsection 5.3.2, when the agreement query process is fully completed, that is, when all of the coder, document and code agreement values are persisted in the datastore, a notification is created for the initiated user and the user is notified.

7.2 Non-Functional Requirements

Functional Suitability

This requirement is fulfilled.

Manual testing of ICA reports and agreement queries was performed to ensure that all functionalities were operational. The previously and newly added ICA unit tests pass, confirming that everything is functioning properly.

Performance Efficiency

This requirement is not fulfilled.

The idea was to provide a dedicated service for metrics so that they could scale horizontally, in order to generate ICA reports and agreement queries in parallel. Due to time constraints, this could not be accomplished.

Compatibility

This requirement is fulfilled.

The backend and frontend class components added in this thesis are completely compatible with previously existing QDAcity components. Their compatibility can be asserted by their successful implementation and flawless functioning of the QDAcity system.

Usability

This requirement is fulfilled.

The requirements 5 and 6 are met as all of the user-facing strings in the UI are defined with a unique key using the react-intl format for localization and the newly added components adhere to the colours defined in the theme file. The user receives a feedback message on initiating the ICA report or agreement query which satisfies requirement 7.

Reliability

This requirement is fulfilled.

As shown in Figure 7.1 and Figure 7.2, the errors are handled. The logs are added where the exception are caught and the added test cases ensures that the system is working as expected.

Security

This requirement is fulfilled.

All API endpoints include a validation method that validates the user's authentication and authorization, ensuring the security characteristics required to meet this requirement.

Maintainability

This requirement is partially fulfilled.

Requirement 11 is met as discussed in subsection 4.1.3, a common interface *CodingsPerUnit* is defined. For further extension of the evaluation unit, the developers has to create a corresponding class implementing the interface to find the applied codings for that unit. The ICA reports can then be created for the

added evaluation unit as we have incorporated the strategy with factory design pattern to create and run the algorithm at the runtime based on evaluation unit.

As discussed in subsection 5.3.3, to support implementation of different agreement types, an abstract class *DeferredAgreement* is created. The implementation follows template design pattern which allows the developers to extend the abstract class and write the calculation logic for the added agreement type while rest of the functionalities is handled. Thus, requirement 12 is satisfied.

The total code coverage of 94.23% satisfies the requirement 13. The newly added API endpoints *createReport*, *getReport*, *listResults* and *listReportsForRevision* are included in the code coverage for *ValidationEndpoint* and the newly added classes *CodingsPerSentence*, *CodingsPerUnitFactory*, *SentenceDetectorModel*, *LanguageModel* and *TextLanguageDetector* are included in the util package's code coverage.

Package	**Lines missed**	**Lines covered**	**LOC coverage in %**
com.qdacity.project.metrics.CodeResult	12	31	61.29
com.qdacity.project.AgreementQueries	0	145	100
com.qdacity.project.data.util	15	126	88.09
com.qdacity.project.metrics.AgreementQueries.algorithms	4	64	93.75
com.qdacity.project.metrics.AgreementQueries.deferredTasks	14	294	85.23
com.qdacity.endpoint.ValidationEndpoint	0	38	100
com.qdacity.endpoint.AgreementQueriesEndpoint	0	82	100
Total	**45**	**780**	**94.23**

Table 7.1: LOC coverage for unit test

Due to time constraints, requirement 13 is only partially met because acceptance tests do not cover all of the scenarios included in ICA. A test has been added for creating and viewing the F-Measure report using the evaluation unit paragraph and another test has been added for creating and viewing the agreement query for the agreement type "code occurrence".

Portability

This requirement is fulfilled.

The newly added and modified ICA functionality was manually tested and found to work as expected in the most recent versions of Google Chrome and Mozilla Firefox.

8 Future Work

The ICA evaluation approaches has now been enhanced with a wide options but there is still room for improvement. The scope of future work in QDAcity ICA measurement is described in this section.

8.1 Report

Document types - Report generation is currently supported only for text documents, but it can be extended to all document types supported by QDAcity.

Evaluation unit - QDAcity now supports the evaluation units sentence and paragraph. This can be extended to larger units like pages or even smaller units like words or characters.

Evaluation method - There are many other methods for ICA measurement, such as Holsti's method (Holsti, 1969), Scott's pi (Scott, 1955) and Cohen's kappa (Cohen, 1960), which can be integrated into QDAcity.

8.2 Agreement Queries

PDF document support - The agreement query for agreement types code occurrence and code frequency is supported for all document types, but code intersection percentage is only implemented for text documents which could be extended to support PDF documents in the future.

Agreement types - The Agreement Queries have been implemented for three agreement types to generate and analyze ICA, which could be further extended to other agreement types.

Area coding - Since QDAcity does not save the preview of area coding for PDF document, the area coded region is displayed as "no preview available" in the comparison view. This area coding preview feature could be added in the future to show area coded region by the coders.

Export - The export functionality of ICA supports only the export of the result table to an CSV document, which could be extended to other document types such as XLS, PDF and so on. For agreement query, the user can specify which coders the agreements table should be exported to. This flexibility can be increased further by allowing the user to choose which documents and codes to export.

8.3 Visualization

The ICA can be visualised in a number of ways and we would like to discuss about some of the potential visualisation techniques that could be implemented in QDAcity.

Side by side view - This visualization compares the codings of two coders for a document. The coded region for codes in a document are shown in the CodingEditor with coding brackets and when the code is selected, the coded region is highlighted for that code. This feature could be expanded to show the two coders codings, with one coder coding brackets on the left and another coder coding brackets on the right. When a code is selected, the coding of both coders could be highlighted to show how closely the coding matches for that code.

Heat maps - The heat map can be created by comparing the consolidated agreement of all intercoders to the model solution. Each character in the document can be coloured based on the intercoders agreement's true positive, false positive, false negative and true negative. This provides an overall view of which text in the document is in high and low agreement.

Charts - To visualize the ICA, different charts such as bar, pie and column can be used. The chart, for example, can show the values generated for the agreement queries using a bar or pie chart for the top 5 codes or documents with the highest agreement. The column chart can be used to show different coders agreement for a code or document. Furthermore, options can be provided to chart type, coders, documents and codes to create a chart.

9 Conclusion

The objectives of this thesis were to provide a sentence as an evaluation unit, an option to calculate agreement for the codesystem subset and a new UI and metric for the ICA assessment.

In order to successfully implement these features in QDAcity, we analysed QDAcity state in chapter 2. We described the features of QDAcity as well as the methods used to generate ICA. The ICA limitations were stated and the requirements for ICA improvement were specified in chapter 3.

We explained the need for an NLP library for sentence detection and a library for language detection before discussing the architecture. We used a strategy with a factory design pattern to provide the architecture for finding codings in different evaluation units. Additionally, the implementation architecture of the agreement queries for three different agreement types were discussed in chapter 4.

In chapter 5, we started with implementation details on how an ICA report for an evaluation unit sentence can be generated, followed by implementation modifications to accommodate the ICA per code. The data structure of the agreement query, the creation procedure and implementation for each agreement type based on a template design pattern were all presented.

In chapter 6, we discussed the difficulties we found during implementation in finding the sentence for a coding in a specific scenario and determining the coding intersection in a PDF document. The requirements defined in chapter 3 were evaluated against provided implementation in chapter 7. We mentioned potential future works in ICA reports, agreement queries and visualisation in chapter 8.

To conclude, QDAcity now allows users to create an ICA report with an evaluation unit sentence and a codesystem subset. The user can view the agreement per code for the F-Measure report and can generate ICA agreement queries using three different agreement types, all of which are integrated into the new and intuitive ICA UI.

References

Cohen, J. (1960). A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, *20*(1), 37–46. https://doi.org/10.1177/001316446002000104

Google. (2022). *Using push queues in java* [[Online; accessed 2022-05-30]]. https://cloud.google.com/appengine/docs/standard/java/taskqueue/push/

Gwet, K. L. (2014). *Handbook of inter-rater reliability: The definitive guide to measuring the extent of agreement among raters*. Advanced Analytics, LLC.

Holsti, O. R. (1969). Content Analysis for the Social Sciences and Humanities. *Reading, MA: Addison-Wesley.*

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

O'Connor, C. & Joffe, H. (2020). Intercoder reliability in qualitative research: Debates and practical guidelines. *International Journal of Qualitative Methods*, *19*, 1609406919899220. https://doi.org/10.1177/1609406919899220

Rupp, C. (2014). Requirements templates — the blueprint of your requirement, 6. https://www.sophist.de/fileadmin/user_upload/Bilder_zu_Seiten/Publikationen/RE6/Webinhalte_Buchteil_3/Requirements_Templates_-_The_Blue_Print_of_your_Requirements_Rupp.pdf

Schweter, S. & Ahmed, S. (2019). Deep-eos: General-purpose neural networks for sentence boundary detection. *KONVENS*.

Scott, W. A. (1955). Reliability of Content Analysis:The Case of Nominal Scale Coding. *Public Opinion Quarterly*, *19*(3), 321–325. https://doi.org/10.1086/266577