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

NAZEEH AMMARI
MASTER THESIS

A TEACHING PLATFORM FOR QDA

Submitted on April 30, 2018

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

Versicherung

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

Nuremberg, April 30, 2018

License

Nuremberg, April 30, 2018

Abstract

Qualitative Data Analysis (QDA) requires a set of competences which can best be aquired through direct experience. Transferring these competences through practical exercises to students in a classroom setting is a challenging task, due to the difficulties which arise with scaling high-touch teaching methods to support large numbers of students.

QDAcity is a cloud based web application which supports collaborative QDA, and has been successfully employed in a teaching environment with 40-60 students, however without native support for courses and exercises.

This thesis builds upon three previously defined use cases for QDAcity which address teaching QDA in the classroom. The requirements of the thesis are derived from these use cases, with the purpose of extending QDAcity's feature set with features common to teaching platforms. After the implementation of these requirements, QDAcity will enable instructors to offer courses and exercises to students on the cloud platform and evaluate their solutions using different evaluation methods.

Contents

1 Introduction

Qualitative Data Analysis software has been evolving at a fast pace in the past two decades. The QDA software market is dominated by desktop applications which provide a wide feature set. However, there has not been a lot of support for reliable web based QDA tools (Boston University, 2018).

QDAcity is a cloud based web application which runs on the Google App Engine. It provides researchers with the ability to work on qualitative data analysis projects directly from the web browser. Researchers can create their own projects, upload text documents and apply codes to the text in the documents. QDAcity also provides the option to work on projects collaboratively.

Researchers can also invite other people to work on clones of these projects and then evaluate their work through intercoder agreement. This feature of QDAcity is currently being utilized to teach qualitative research methods at the Open Source Software Chair of the University of Erlangen-Nuremberg. In this case, the researcher is an instructor who is responsible for setting up the projects and code systems, upon which the students apply the research methods they learned during the courses offered by the chair.

The instructors of the Open Source Software chair have analyzed three use cases which can be implemented in the classroom with the purpose of extending the feature set of QDACity to support more teaching scenarios.

The purpose of this thesis is to extend QDACity accordingly in order to support these three use cases.

The thesis is organized as follows:

- Chapter 2 (Demo Script) describes the mentioned use cases in detail.
- Chapter 3 (Existing Architecture of QDAcity) provides an overview of the current architecture of QDACity.
- Chapter 4 (Requirements) lays out the requirements of the thesis, which are derived from the use cases of the demo script.

- Chapter 5 (Implementation) describes the implementation of the features which fulfill the requirements of the thesis.
- Chapter 6 (Evaluation) provides an evaluation of whether the requirements have been fulfilled successfully and the extent of their fulfillment.
- Chapter 7 (Conclusion) summarizes the outcomes of the thesis and how they can be utilized to extend the features of QDAcity further in the future.

2 Demo Script

This chapter describes the demo script which forms the basis of the requirements of the thesis. There are three use cases in the script, which address three types of exercises and their evaluation. These types are:

- Codebook Coding.
- Open Coding.
- Axial Coding.

The next three sections describe the use case for each type, along with its evaluation method.

2.1 Codebook Coding

Codebook coding, also known as Selective coding in literature , is a coding method based on applying a code from a set of codes which have already been defined, in order to categorize the data contained in a resource (Pandit, N. R., 1996, p. 11).

In the context of this thesis, Codebook coding refers to the method of applying codes to a text document or a set of text documents, where the instructor provides a set of codes, namely a code system, which students use as a guide to apply codes from to the content of the exercise.

Currently, QDAcity supports applying codes to text which is contained in the assets of a project that is created by the instructor.

In this use case, the instructor evaluates the performance of students through intercoder agreement. The sequence of steps of the use case is defined as follows:

1. The instructor creates a new course instance.
2. The instructor invites students to the join the course instance.
3. The instructor creates an exercise.

- The instructor sets the deadline of the exercise.
- The instructor sets the type of the exercise to Codebook Coding.
- The instructor selects a project, which includes the data upon which the exercise will be based.

4. The student accepts the invitation to the course instance.
5. The student chooses the newly created exercise to start working on.
6. The student applies codes to the text from the code system defined by the instructor.
7. The instructor views a list of student projects.
8. The instructor creates an evaluation report for all students based on inter-coder agreement.
9. The students view the evaluation of their solution.

2.2 Open Coding

Open coding is defined as the interpretive process by which data are broken down analytically. In this type of coding, the researcher is labelling the data without being restricted to a predefined set of codes (M. Corbin & Strauss, 1990, p. 12).

In the context of this thesis, Open coding refers to the method of applying codes to a text document or a set of text documents, where the students create the codes without an existing code system created by the instructor. QDACity currently supports this feature. In this use case, the evaluation method is based on the direct feedback of the instructor, who has the ability to write comments to the codes applied by the students.

The sequence of steps of the use case is defined as follows:

1. The instructor creates a new course instance.
2. The instructor invites students to the join the course instance.
3. The instructor creates an exercise.
    - The instructor sets the deadline of the exercise.
    - The instructor sets the type of the exercise to Open Coding.
    - The instructor selects a project, which includes the data upon which the exercise will be based.

4. The student accepts the invitation to the course instance.
5. The student chooses the newly created exercise to start working on.
6. The student creates open codes and applies them to the text documents of the exercise.
7. The instructor views a list of student projects.
8. The instructor provides individual feedback for each student by writing comments. These comments can address:
    - The original material of the exercise.
    - The codes applied by the student.
9. The students view the comments written by the instructor.

2.3 Axial Coding

Axial coding is defined by Strauss and Corbin as "a set of procedures whereby data are put back together in new ways after open coding, by making connections between categories. This is done by using a coding paradigm involving conditions, context, action/interactional strategies, and consequence" (Strauss & Corbin, 1990, p. 96).

In the context of this thesis, Axial coding refers to the method of applying open codes to a text document or a set of text documents, which other students comment on for the purpose of providing a peer based feedback.

Hence, the evaluation method in this exercise type involves the participation of more than one student. The number of students (peers) can be determined by the instructor. For simplicity, the use case will incorporate two students as an example. They are referred to as Student A and Student B. The sequence of steps of the use case is defined as follows:

1. The instructor creates a new course instance.
2. The instructor invites students to the join the course instance.
3. The instructor creates an exercise.
    - The instructor sets the deadline of the exercise.
    - The instructor sets the type of the exercise to Axial Coding.
    - The instructor selects a project, which includes the data upon which the exercise will be based.

- Sets the number of peers to provide feedback.

4. Both Students A and B accept the invitation to the course instance.
5. Student A chooses the newly created exercise to start working on.
6. Student A creates open codes and applies them to the text documents of the exercise.
7. Student B joins the course instance and opens the project of Student A.
8. Student B writes comments on the codes applied by Student A.
9. Student B writes a summary commentary for Student A's work.
10. Student B applies his own open codes to the text documents of the exercise.
11. Student A provides feedback for Student B's work, the same way Student B did.
12. The students view the comments written by their peers.
13. The instructor views the projects of students individually, while showing all the feedback from their peers.

3 Existing Architecture of QDAcity

3.1 Architecture Overview

The architecture of QDAcity consists of two main parts, the frontend and backend. The backend is built using Java 8 as the language of programming (previously Java 7 but it has been migrated to Java 8 during the timeline of this thesis).

QDAcity uses Google App Engine[1] for hosting its backend, which is one of the products of the Google Cloud Platform. The frontend is built on Html5 and Javascript. For development, the project utilizes the following tools and frameworks:

- React: a Javascript library for building user interfaces.
- Gulp: a build system for front end web development.
- Webpack: a bundler for javascript files.
- Maven: used as a tool for dependency management for Java libraries.

The mentioned parts constitute the majority of QDAcity's existing architecture. In this thesis, there have been no fundamental architectural changes, since the use cases of the demo script build upon the existing functionality. The next sections elaborate on these parts in detail.

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

3.2 Google App Engine

Google App Engine is a web framework and cloud computing platform for developing and hosting web applications. QDAcity runs its backend on the app engine, which provides persistent cloud storage (Google Cloud Datastore[2]). The backend of QDAcity interfaces with the datastore through its API endpoint methods.

The Java development environment of QDAcity uses The App Engine Java SDK in conjunction with the Java Data Objects (JDO) interface in order to use the App Engine Datastore as a database for QDAcity. The JDO is a standard interface for accessing databases in Java which provides a mapping between Java classes and database tables.

Although Google provides its own low level API (The Datastore API) which can be used instead of the JDO, QDAcity uses the JDO to reduce the extent of dependency on the app engine, mainly because the JDO can still be used in the future if the App Engine is discontinued or goes through changes which make it unsuitable for QDAcity.

3.3 React

React is a JavaScript library for building user interfaces. It uses JavaScript XML (JSX) as an extension to the JavaScript language syntax. Some of its most prominent features are:

- One way data flow: the properties of UI elements are passed from parent to child components in the Domain Object Model (DOM) tree.
- Virtual DOM: React keeps a virtual representation of the UI in memory and matches it with the real DOM (Facebook Inc, 2018b).
- Composition of Components: React supports building components which can be used independently. Components can also be built from other existing components (Facebook Inc, 2018a).
- Supports declative syntax: due to its ability to provide the VDOM representation, React provides the option to use declarative syntax.

These features provide a good framework for the development of the frontend of QDAcity. Especially in the context of this thesis, since many of the new features can make use of existing components and extend them with great flexibility.

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

3.4 Gulp

Gulp is a task runner that uses Node.js as a platform. It enables the automation of frontend build tasks. In the context of QDAcity, the most important purposes of using gulp are:

- Compilation of JSX files to JavaScript.
- Running the local development server.
- Minifying frontend code.
- Watching the source files and recompiling them automatically whenever changes are made.
- Generation of language files for the purposes of localization.

3.5 Webpack

Webpack is a package bundler which, in the context of QDAcity, is used to concentrate all source files and create a browser executable ECMAScript artifact. In QDAcity's frontend setup, this is done in conjunction with gulp as an automated task that is configured through a webpack configuration file.

3.6 Maven

Maven is a project management and comprehension tool that provides developers a complete build lifecycle framework. In QDAcity, Maven is used as a dependency management tool for the backend Java project.

4 Requirements

The requirements of the thesis are based on the Demo Script described in chapter 2. In order to cover the use cases mentioned in the script, the requirements have been extracted from the demo script.

They are divided into functional and non-functional requirements. The first section of this chapter presents an overview of stakeholders. The second and third sections lay out the functional and non-functional requirements of this thesis.

4.1 An Overview of Stakeholders

This section provides a brief description of the stakeholders of the project and their roles in relation to the requirements.

4.1.1 Instructor

Instructors who are interested in teaching QDA are the target audience of the use cases in the demo script. In the context of the requirements, any reference to an instructor refers to instructors who are experienced in Qualitative Data Analysis research and have an interest in teaching the related competences to their students using QDAcity.

4.1.2 Student

Students are the second target group of the use cases of the demo script. Any reference to a student in the requirements refers to university students who are interested in research and in improving their research skills before pursuing research on a higher educational level.

4.1.3 User

The user is a person who can represent any of the stakeholders. Any reference to users in the requirements refers to users of QDAcity in general.

4.1.4 Administrator

An administrator is a user who has access to all API methods. In a real world scenario, administrators are users who have an interest in coordinating courses and making changes to course and exercise structures without having to be the original owners of the course.

An example use case can be when a professor wants to design the high level structure of a course in order for other instructors to use the course for teaching QDA to a specific class.

4.2 Functional Requirements

The functional requirements are described using the template FunctionalMASTER by Rupp (2014, pp. 215 sqq.) as shown in figure 4.1 below. The dotted boxes are an optional part of the requirement specification and the italic words are required.

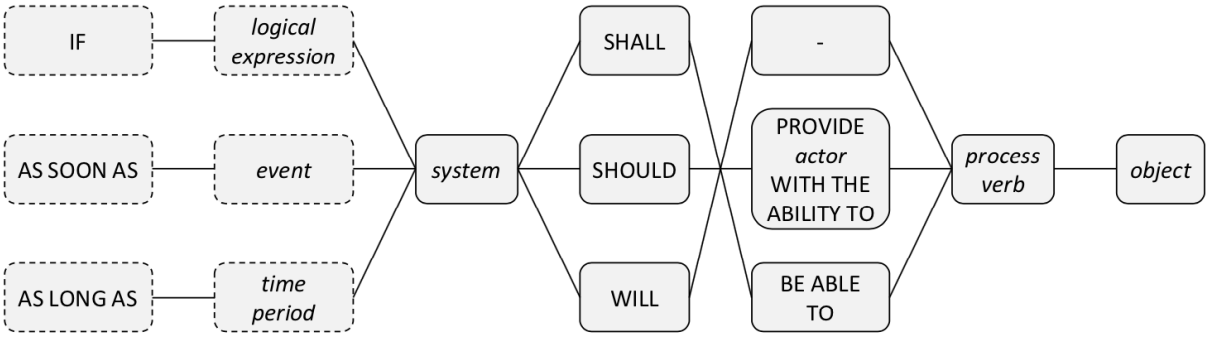

Figure 4.1: FunctionalMASTER requirement specification template

The semantic definition of the keywords shall,should and will as used in this thesis can be found in Table 4.1.

Keyword	Semantic Definition
Shall	A requirement that has to be fulfilled for the project to succeed.
Should	A requirement that is important but not necessary for the software to work correctly.
Will	A requirement that is not necessary but desired.

Table 4.1: Definition of the requirement keywords

In order to address the scenarios mentioned in the demo script, the concepts of courses, instances of courses in a specific term and exercises shall be added to QDAcity. In the frame of the following requirements, an instance of a course will be referred to as a term course.

The requirements will be divided into different sections based on their relationship with these concepts. In addition to a section that addresses authentication and authorization related requirements. Accordingly, the requirements are divided into the following sections:

- Section 4.2.1 (Course Management) defines requirements related to the management of new courses and term courses, in addition to providing requirements related to enrolled students. The actor responsible for them is the instructor.
- Section 4.2.2 (Course Access) defines requirements related to the accessibility of courses, term courses and corresponding exercises. The actor affected by them is the student.
- Section 4.2.3 (Coding Editor) defines requirements related to the existing Coding Editor.
- Section 4.2.4 (Exercise Management) defines requirements related to the management of exercises, The actor responsible for them is the instructor.
- Section 4.2.5 (Exercise Evaluation and Feedback) defines requirements related to the evaluation of the performance of students in exercises and the feedback methods mentioned in the demo script.
- Section 4.2.6 (Authentication and Authorization) defines requirements related to the authentication of users and their authorization to execute the api methods defined in the endpoints of the backend.

These functional requirements are described in detail next. For visual clarity,

requirements are surrounded by a black box, and subrequirements are surrounded by a grey box.

4.2.1 Course Management

In order to cover the use cases mentioned in the demo script, the instructor should be able to add and remove courses. In addition to the possibility of creating multiple term courses under each course. The requirements related to course management are mentioned next:

FR–1: QDAcity shall provide the instructor with the ability to create courses.

FR–1.1: A button on the personal dashboard should trigger a creation dialog for a new course.

FR–1.2: The creation dialog shall allow the user to define a name and description for the course as well as the option for an instantiation of a term course.

FR–2: QDAcity shall provide the instructor with the ability to search for a course by name.

FR–3: QDAcity shall provide the instructor with the ability to create term courses which belong to previously created courses.

FR–4: QDAcity should provide the instructor with the ability to delete courses.

FR–4.1: QDAcity shall show a confirmation dialog after the user clicks the course delete button.

FR–4.2: When the user deletes a course, all of its instances (term courses) should be deleted automatically.

FR–5: QDAcity should provide the instructor with the ability to delete term courses.

FR–5.1: QDAcity shall show a confirmation dialog after the user clicks the term course delete button.

FR–5.2: When the user deletes a term course, all of its exercises should be deleted automatically.

FR–5.3: QDAcity shall show a confirmation dialog after the user clicks the term course delete button.

FR–6: QDAcity should provide the instructor with the ability to view participants of a term course.

FR–6.1: QDAcity shall provide the instructor with the ability to view the participants in a list and order them by first name.

FR–7: QDAcity will provide the instructor with the ability to remove students from a course.

FR–8: QDAcity will provide the instructor with the ability to set the status of a term course to open or closed for joining.

FR–9: QDAcity should provide the instructor with the ability to invite students to a term course through email.

4.2.2 Course Access

Students should be able to access the courses and their corresponding term courses. These requirements can be summed up as follows:

FR–10: QDAcity shall provide the student with the ability to join term courses related to a course.

FR–10.1: A user can join a term course only if the instructor makes it open for joining

FR–11: QDAcity shall provide the student with the ability to leave term courses they have previously joined.

FR–12: QDAcity shall provide the student with the ability to view a list of exercises which belong to a term course they have joined.

FR–13:	QDAcity shall provide the student with the ability to view the deadline of an exercise.

FR–14:	QDAcity shall provide the student with the ability to start working on an exercise which belongs to a term course they have previously joined.

4.2.3 Coding Editor

In order to review finished exercises, the option of running the Coding Editor in read-only mode is desired. Hence the requirement:

FR–15:	The coding editor should provide the user with the ability to open an exercise in read-only mode after a deadline has passed, with the exception of the case when the exercise extends an exercise group.

4.2.4 Exercise Management

An instructor should be able to manage exercises which belong to a term course. Including the exercise type, deadline and the project it is related to. These requirements can be summarized as follows:

FR–16:	QDAcity shall provide the instructor with the ability to create exercises which belong to a previously created term course.

FR–16.1:	QDAcity shall provide the instructor with the ability to specify an exercise deadline upon the creation of an exercise.

FR–16.2:	QDAcity shall provide the instructor with the ability to specify the type of the exercise upon its creation from three types: Open Coding, Axial Coding, Codebook Coding.

FR–16.3:	QDAcity shall provide the instructor with the ability to select an evaluation method for this exercise.

FR–16.4: Valid evaluation methods are "Intercoder Agreement", "Instructor Feedback", "Peer Feedback"

FR–16.5: QDAcity shall provide the instructor with the ability to create exercises which are an extension of previous exercises, without affecting their accessibility.

FR–16.6: QDAcity shall provide the instructor with the ability to specify an existing project and its revision upon the creation of an exercise, which the exercise will be based on, in order to use a specific revision's data (TextDocuments and CodeSystem).

FR–16.6.1: When an exercise is created based on existing data, and the exercise type is codebook coding, then the applied codings are removed from the text documents during the copy process.

FR–17: QDAcity shall provide the instructor with the ability to delete exercises.

FR–17.1: When the user deletes an exercise, all of its related data should be deleted automatically. This includes the exercise projects and the text documents related to them.

FR–18: QDAcity shall provide the instructor with the ability to view the exercise projects of other students.

4.2.5 Exercise Evaluation and Feedback

Upon the expiration of an exercise deadline, the instructor should be able to evaluate the performance of students without affecting their ability to still view their work. The requirements below address the situation:

FR–19: QDAcity shall provide the instructor with the ability to evaluate the coding of students according to the inter-coder agreement.

FR–20: As soon as the deadline of an exercise has passed, QDAcity shall collect all the data of student ExerciseProjects automatically without a dedicated submission workflow.

FR–21: QDAcity shall provide the instructor with the ability to view the progress of a student by opening their exercise projects with the Coding Editor.

FR–22: QDAcity should provide the instructor with the ability to add comments to the codes applied by a student as a method of instructor feedback.

FR–23: QDAcity should provide the student with the ability to view the evaluation of their solution of an exercise.

FR–23.1: QDAcity should provide the student with the ability to view the intercoder agreement evaluation results of an exercise, when the exercise is of the type Codebook Coding.

FR–23.2: QDAcity should provide the student with the ability to view the comments of the instructor on their codings, when the exercise is of the type Open Coding.

FR–23.3: QDAcity should provide the student with the ability to view the comments of their peers on their codings, when the exercise is of the type Axial Coding.

FR–24: QDAcity should provide the student with the ability to add comments and summary to the codes applied by other students as a method of peer feedback.

FR–25: QDAcity should provide the instructor with the ability to set the number of peers who will provide feedback to other students for an exercise.

4.2.6 Authentication and Authorization

The current version of QDAcity includes authentication and authorization checks for all its endpoint methods. Therefore, it is important that any methods created as a part of this thesis do these checks as well. This can be summarized in the following requirements:

FR–26:	QDAcity shall only allow authenticated instructors to make changes to a course, term course or exercise which they are an owner of.

FR–27:	QDAcity shall only allow authenticated students to join term courses.

FR–28:	QDAcity shall only allow authenticated students to participate in exercises of a term course they previously joined as a student.

FR–29:	Administrators shall have access to all API methods.

4.3 Non-functional Requirements

The formulation of the non-functional requirements is based on the PropertyMASTeR template by Rupp (2014, pp. 234 sqq.) as shown in Figure 4.2. As in Figure 4.1, the parts in dashed boxes are optional and the italic words are required.

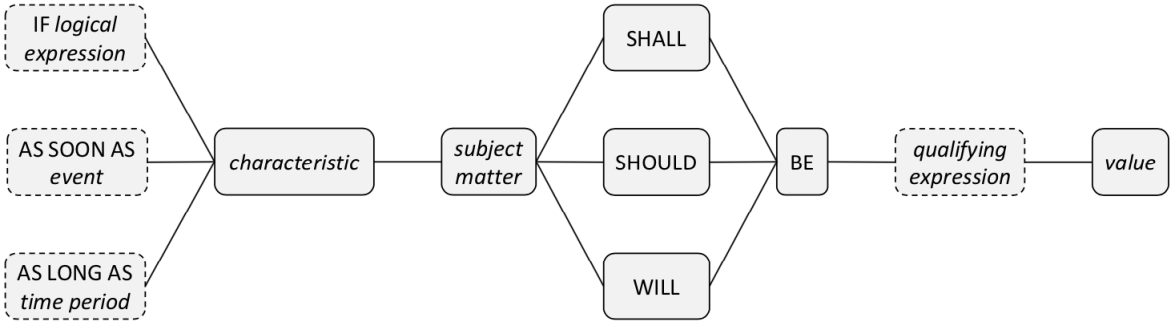

Figure 4.2: PropertyMASTeR template for non-functional requirements

The non-functional requirements of this thesis are contained in the following part:

- Quality Metrics: the code written during the thesis is measured against the characteristics mentioned by ISO-25010 (ISO/IEC, 2010) to verify that the implementation fulfills reasonable quality metrics of software.

The section addresses the quality metrics and defines the requirements which are specific to QDAcity according to the characteristic which they affect.

4.3.1 Quality Metrics

According to the standard ISO-25010, a high quality software should adhere to the following characteristics shown in figure 4.3 below. These characteristics are divided into eight categories and their corresponding subcategories.

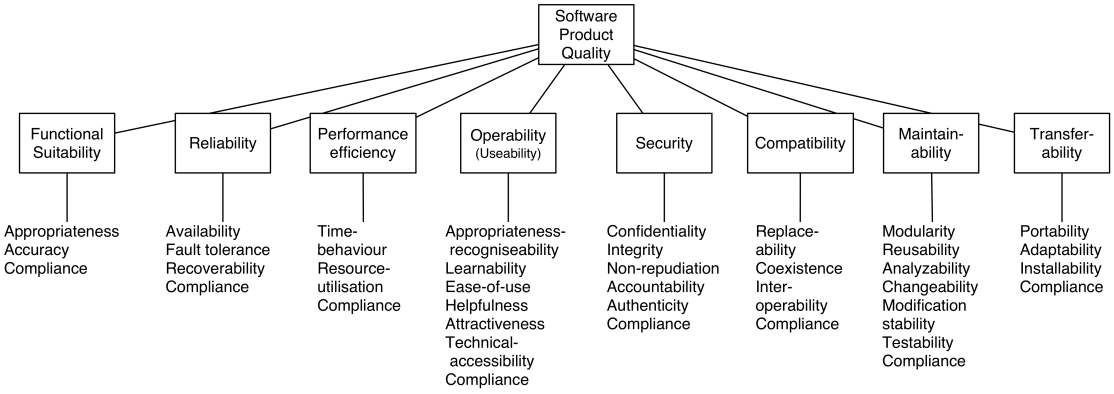

Figure 4.3: ISO-25010 Categories and their subcategories

These characteristics are described in the following sections.

4.3.1.1 Functional Suitability

This characteristic implies that the newly implemented features during this thesis should be fully functional. Additionally, all stated and implied needs of the stakeholder should be met by these features.

4.3.1.2 Reliability

This characteristic implies that QDAcity should be available to operate under normal conditions all the time. In case errors occur, it should be able to handle them and continue operating without interruption.

4.3.1.3 Performance Efficiency

This characteristic implies that the new features implemented by this thesis should maintain the same level of performance which is supported by the existing QDAcity web application.

4.3.1.4 Usability

In the context of QDAcity, this characteristic is embedded in the intuitive user experience that the existing QDAcity offers. Therefore, the new frontend components and pages implemented in this thesis should continue to provide a similar quality and ease of user experience. In order to fulfill this quality metric, the following requirements have been specified:

NFR–1:	All strings in the elements of QDAcity web application shall be defined using the existing localization framework.

NFR–2:	All strings in the elements of QDAcity web application shall be translated into the german language.

NFR–3:	All user interface elements of QDAcity web application shall use theme colors which are defined in the front end code.

NFR–4:	For any common components (e.g buttons, panels, lists) the existing styled components should be used.

4.3.1.5 Security

In the context of QDAcity, this characteristic implies that only authorized users are allowed to read or modify the datastore.

4.3.1.6 Compatibility

In the context of this thesis, this characteristic is concerned with the ability to make the new features compatible with the existing QDAcity architecture and build environment.

4.3.1.7 Maintainability

This characteristic addresses the extensibility and future maintainability of the modules and classes which are created during the implementation phase of this thesis. It can be verified by making sure that the classes are modular and loosely coupled. This can be ensured by writing unit tests to cover the newly implemented functionality, specifically:

NFR–5: All API methods in the new endpoint classes shall be covered with unit tests.

NFR–5.1: The unit testing target coverage for new classes should be 90% or more.

NFR–6: An acceptance test should test the added features in every build pipeline.

In addition to unit tests, any new classes should be extensible and new features should reuse existing code whenever possible. One important instance of this is the extension of the intercoder agreement evaluation algorithm. This can be described by the following requirement:

NFR–7: The code used for evaluating the intercoder agreement should be shared between the exercises and the existing validation of a project used for investigator triangulation.

4.3.1.8 Transferability (Portability)

In the context of QDAcity, this characteristic addresses the web application's ability to run on different browsers running different operating systems while maintaining the same user experience. Any new features or components should adhere to these quality measures.

5 Implementation

This chapter describes the implementation details which were embedded in the code base of QDACity in order to fulfill the requirements of the thesis. It is divided into the following parts:

- Section 5.1 (Courses and Term Courses) describes the data model of the new entities Course and Term Course. These entities are required to fulfill all use cases of the demo script.
- Section 5.2 (Exercises) describes the implementation details of features related to the exercise and the entities which are affected by it.
- Section 5.3 (Exercise Data Collection) discusses the implementation details which are embedded in collecting the data of an exercise after its deadline has passed. Specifically by creating the snapshots of the progress of students for exercises, namely their ExerciseProjects and other entities related to it.
- Section 5.4 (Evaluation Algorithm) discusses the implementation of extending the evaluation algorithm which is based on the intercoder agreement, in order to work for the new exercises while reusing as much existing code as possible.
- Section 5.5 (Exercise Groups) discusses the implementation of the new entity ExerciseGroup, whose purpose is to support extending existing exercises (mentioned in requirement FR-16.5).
- Section 5.6 (QDACity Frontend and React Components) discusses the implementation details which are related to the front end of QDACity and the components which were introduced in order to fulfill the requirements.

5.1 Courses and Term Courses

The new entities Course and TermCourse were introduced to the existing data model of QDAcity. Each Course can have multiple TermCourses which belong

to it. This relationship is actualized in the data model by adding the property courseID to every TermCourse. Term Courses and Exercises share a similar relationship. When the instructor creates a new exercise, the exercise will belong to a specific term course. This relationship is shown in the UML class diagram in figure 5.1.

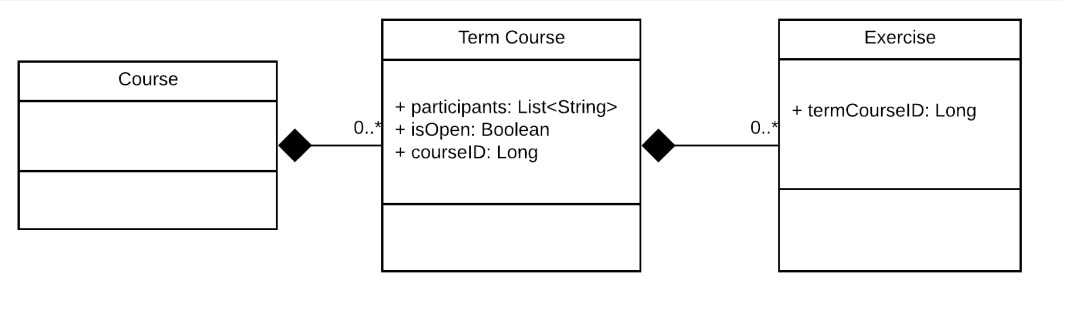

Figure 5.1: The relationship between Course, TermCourse and Exercise

The property `isOpen` of the `TermCourse` entity can be utilized to give the instructor control over the accessibility of the term course by students. The property `participants` is an array which contains the ids of students who have joined the term course (and consequently the course). The api method `listTermCourseParticipants` in `CourseEndpoint` is queried from the frontend in order for the instructor to view the list of participants of a term course.

5.2 Exercises

Multiple exercises can belong to the same term course. This relation is made by keeping the property termCourseID for the exercise entity, as shown in figure 5.1 previously. When the instructor creates a new exercise, they can specify an existing project revision from which the exercise data is cloned. This implementation fulfills the use case Codebook Coding in the demo script, which states that exercises can be based on existing project documents and code systems.

Once a student joins a term course, they will be able to view a list of exercises which belong to it. Upon joining the exercise, an entity called `ExerciseProject` is created for the student, which they can work on by coding the text documents. Since the students work on an `ExerciseProject` for each Exercise, there is a similar relationship between `ExerciseProject`, `ValidationProject` and the `ProjectRevision` which the exercise is created from. Therefore, `ExerciseProject` was created as a subclass of `ProjectRevision`.

The entity `ExerciseProject` is created once per student per exercise. An `ExerciseProject` is created when the student clicks the editor button for the first time for an exercise in the term course page. After that, the already existing `ExerciseProject` is opened in the Coding Editor. This behavior is shown in the sequence diagram in figure 5.2.

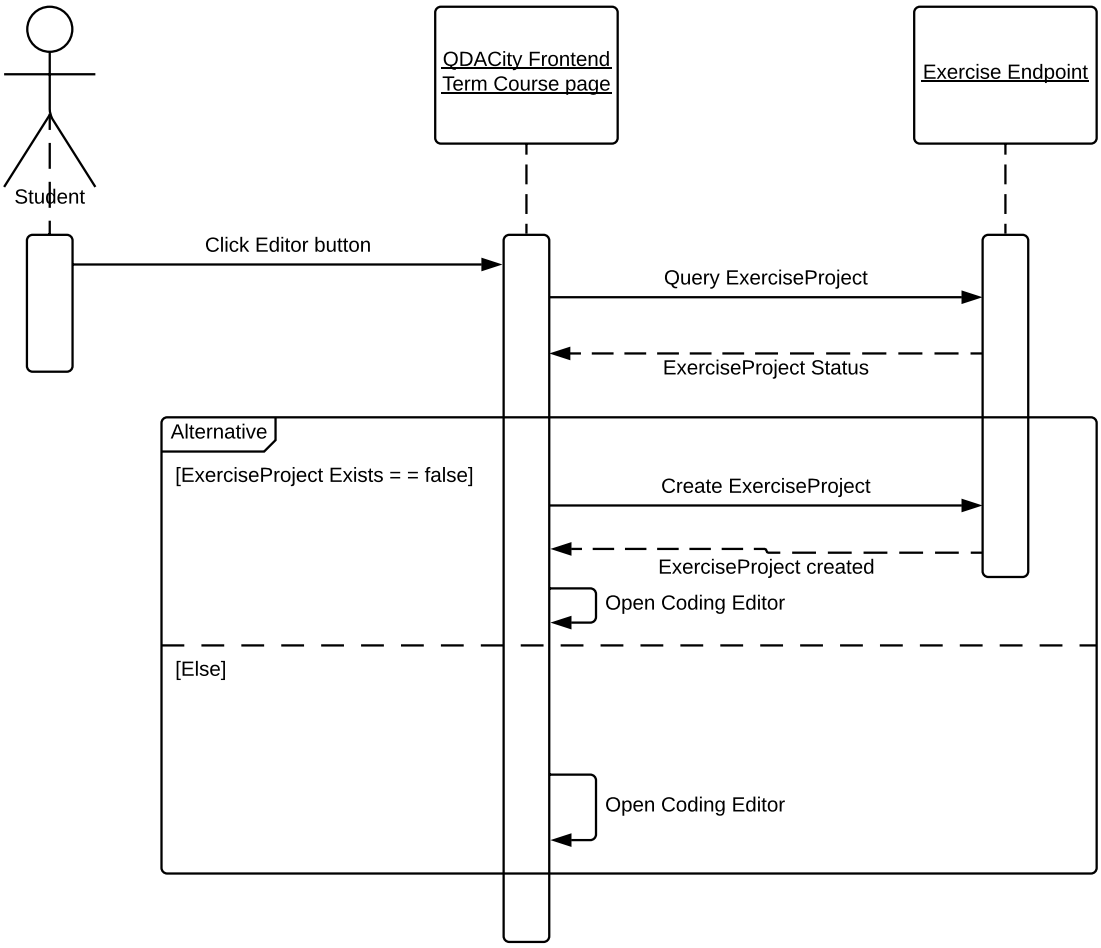

Figure 5.2: Exercise Coding Editor button sequence diagram

5.3 Exercise Data Collection

As mentioned in requirement FR-20, QDAcity should create snapshots of the work of students upon the expiration of the exercise deadline specified by the

instructor. In order to fulfill this requirement, a cronjob was designed to check for exercises whose deadline has passed. Once an exercise deadline expires, the servlet triggered by the cronjob does the following:

- Creates clones of the entities `ExerciseProject` and `TextDocument`: all exercise projects which belong to that exercise are cloned. Additionally, all text documents which belong to the cloned exercise projects are cloned with their codings.
- The cloned entities are flagged in the datastore by attributes intended for the purpose of identifying clones, which the instructor can later use for evaluation.
- The exercise is flagged in the datastore by the attribute `snapshotsAlreadyCreated` so that its `ExerciseProjects` and `TextDocument` are not cloned again when the cronjob runs in the next cycle.

Afterwards, these cloned entities can be used by the evaluation algorithm, while maintaining the ability of the students to access the exercises to review their work. This flow is shown in the sequence diagram in figure 5.3 below.

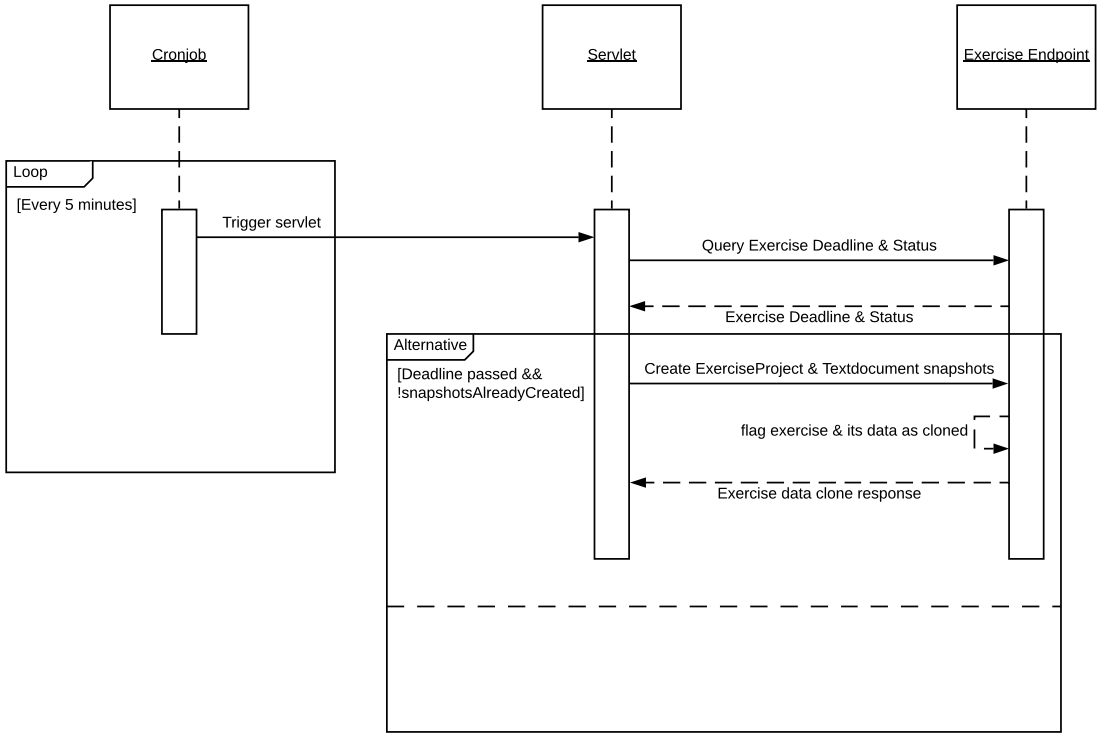

Figure 5.3: Sequence diagram of exercise data cloning cronjob

5.4 Evaluation Algorithm

The evaluation algorithm which was previously implemented in QDAcity in the class `DeferredEvaluation` supports the evaluation of validation projects which are subclassed from project revisions. This is done through a task that is performed asynchronously through the task queue. The evaluation of exercises of the type Codebook Coding shares a lot in common with evaluating validation projects. In that sense, the evaluation of a group of `ValidationProject` against the researcher's coding of a `ProjectRevision`, which is done to establish investigator triangulation, is very similar to that of a group of `ExerciseProject`. Specifically:

- Both `ExerciseProject` and `ValidationProject` are subclassed from the parent class `ProjectRevision` as described previously.
- The resulting artifacts of the evaluation in both cases are either of the same type, as in the case of `AgreementMap` and `DocumentResult` or have to implement the same interface, in the way that the result of evaluating the exercise projects will implement the same interfaces which `ValidationReport` and `ValidationResult` are currently implementing.
- The evaluation methods f-measure, krippendorffs-alpha,fleiss-kappa can be applied to both evaluations equally.
- Cloning `TextDocuments` related to `ValidationProject` and `ExerciseProject` works similarly. With the exception of some details which can be handled by parameterizing some api methods in `TextDocumentEndpoint`.

Based on these commonalities between the two classes, we conclude that the most efficient way to implement the evaluation algorithm for exercises is to create a common parent class `DeferredEvaluation` and create two child classes which implement the same interface. The parent class includes any methods which are common. It also includes the definition of abstract methods which shall be implemented differently in each of the subclasses `DeferredEvaluationExerciseReport` and `DeferredEvaluationValidationReport`.

This also requires refactoring the resulting classes `ValidationReport` and `ValidationResult`. Specifically by creating the common abstract classes Report and Result, and subclassing them into `ValidationReport`, `ExerciseReport` and `ValidationResult`, `ExerciseResult` respectively.

After this refactoring, the concrete classes `DeferredEvaluationExerciseReport` and `DeferredEvaluationValidationReport` can be constructed from the abstract parent class `DeferredEvaluation` and initialized as needed.

The new resulting structure of this refactoring is shown in the UML class diagram in figure 5.4.

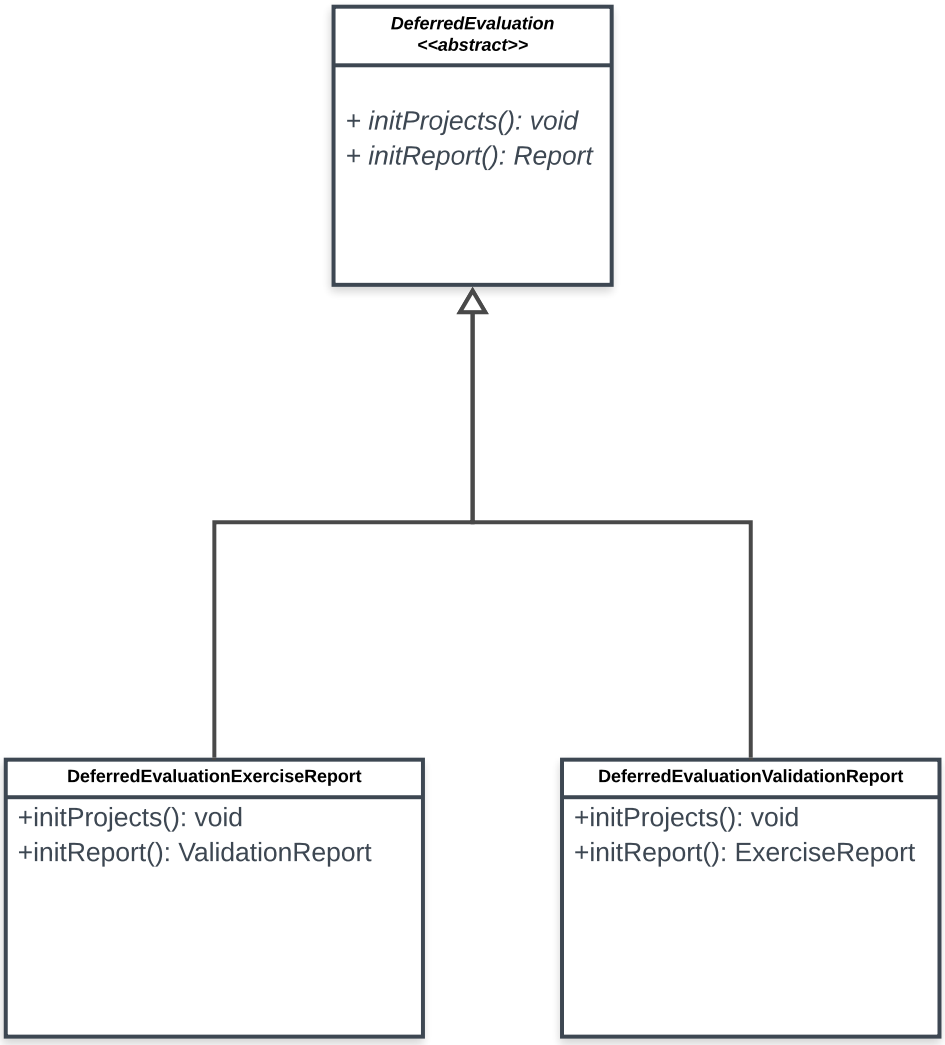

Figure 5.4: DeferredEvaluation abstract class and its subclasses

Figure 5.4 shows the class `DeferredEvaluation`. The two abstract methods `initProjects()` and `initReport()` are implemented in the child classes. The method `initProjects()` is responsible for fetching the projects which are to be evaluated, namely `ExerciseProject` or `ValidationProject`. While

the method `initReport()` is responsible for initializing the evaluation reports which will include the results of the evaluation, namely `ExerciseReport` and `ValidationReport`.

In a similar fashion, the abstract classes Result and Report are the parents of `ValidationResult`, `ExerciseResult` and `ValidationReport`, `ExerciseResult` respectively. This refactoring allows for the previously developed evaluation methods to be used to evaluate exercises. It also allows them to be extended to support other exercise types.

In addition to refactoring the evaluation algorithm, the new parameter ProjectType has been introduced to some api methods in the class `TextDocumentEndpoint`. This parameter provides the possibility to use the existing api methods on new entities, such as cloning the data of exercises whose deadline has passed and using existing project data for new exercises.

5.5 Exercise Groups

In order to fulfill the requirement FR-16.5, the entity ExerciseGroup was introduced. This entity contains a list of exercise ids, which construct the exercise group.

When the instructor wants to create an exercise which extends an existing exercise, they can choose this option by checking a box from the exercise creation dialog. If the box is checked, a list of exercises and exercise groups which belong to the selected project revision is shown. The dialog allows for the following use cases:

- Extending an existing exercise: choosing an exercise from the list will create a new exercise group and add the old and new exercise to it.
- Adding the exercise to an existing group: choosing an existing exercise group will add the new exercise to it.

5.6 QDAcity Front End and React Components

This part describes the implementation details related to the frontend of QDAcity. Namely, the web interface of QDAcity.

5.6.1 Term Course Page

The demo script mentions that there should be two views for the term course page:

- Participant View: this is the view which enables students to join or leave an exercise. It also includes the exercises which the student can start working on.
- Instructor View: this is the view which enables instructors to create exercises and view information related to student projects.

In the implementation of these requirements, the first idea was to create one web page and show one of these views depending on whether the user is a student or an instructor. However, due to major differences in the functionality and implementation of this webpage, the decision was made to create two separate pages to provide the required functionality.

5.6.2 VexModal Dialog Box

The user interface elements of QDAcity's frontend were based on the existing architecture. Since React framework allows for the extension of existing components, most new components were based on existing ones.

All new dialogs were based on the existing VexModal component. Which is a component that can be extended to support other components. For instance, the creation of a new exercise is done by interacting with a VexModal which includes components which allow setting the following parameters:

- Parent project.
- Project Revision: this, along with the previous dropdown list enable the instructor to select the project revision whose data will be cloned.
- Exercise Type: this parameter sets the type of exercise from the three available types "Codebook Coding", "Open Coding" and "Axial Coding".
- Extends existing exercise: this checkbox can be checked to make the exercise an extension of an existing exercise or exercise group.
- Exercise Deadline: the deadline after which all exercise data will be cloned. Any changes made by students to their projects after this date are not taken into account for evaluation.
- Exercise name: this name will appear in the exercise list viewed by students.

This VexModal dialog is shown in figure 5.5.

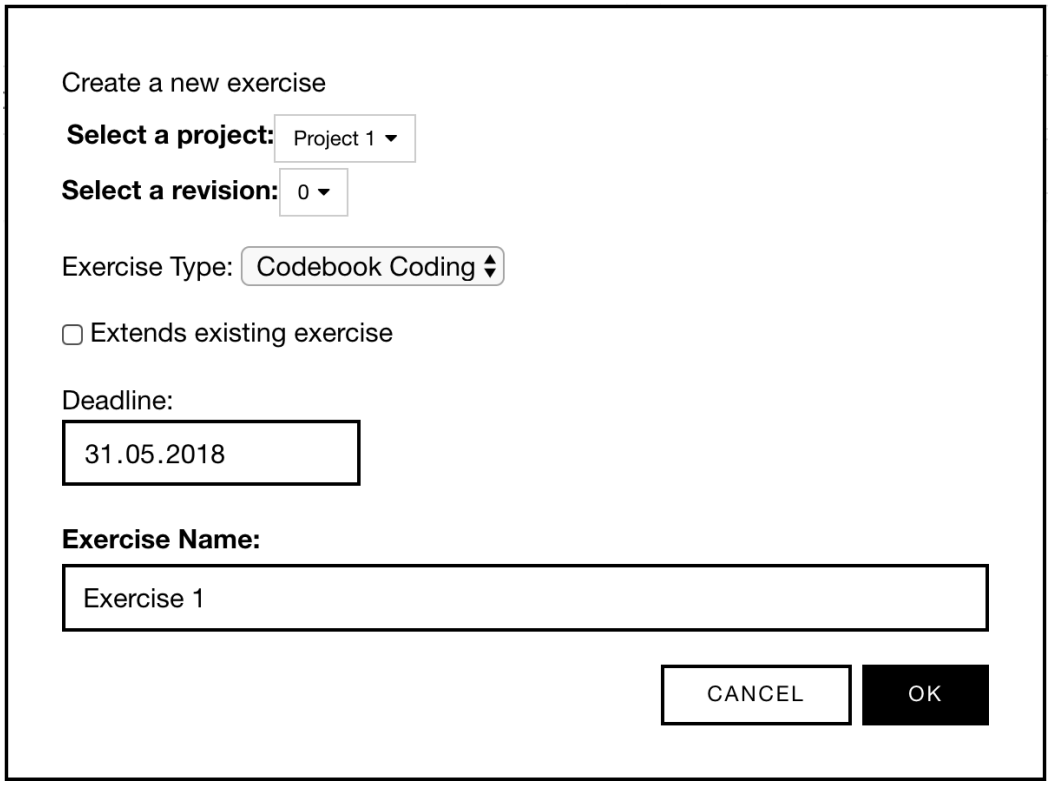

Figure 5.5: Exercise creation dialog box

6 Evaluation

This chapter reviews the requirements mentioned in chapter 4 and provides an assessment of their fulfilment. It is divided into three parts: functional, non-functional requirements and an evaluation of the requirements with regards to the demo script use cases.

6.1 Functional Requirements

6.1.1 Course Management

The requirements related to course management FR-1 to FR-6 and FR-9 have been successfully accomplished. The instructor can now create, remove and search for courses from the personal dashboard of QDAcity. They can create and remove term courses in the course dashboard page. In addition to the ability to create a term course upon the creation of a course in the course creation dialog.

The instructor can invite students to a term course and view a list of term course participants in the term course configuration page.

Requirements FR-7 and FR-8 have been partially accomplished. The api methods for these two requirements are written and covered by unit tests, but some minor additions are required in the frontend to fulfill these requirements completely. The requirement FR-7 can be accomplished by adding a delete button next to the list item in the participants list. The requirement FR-8 can be accomplished by adding a switch to the same page to set the status of the term course. However, these requirements are not prerequisites to any of the other requirements and they are not required for QDAcity to work as specified by the demo script.

6.1.2 Course Access

The requirements related to course access FR-10 to FR-14 have been successfully accomplished. Students can join and leave term courses by accessing the term course page and clicking the join/leave button. The term course page shows a list of exercises and their corresponding deadlines. The student can start working on an exercise by clicking the editor button as described in section 5.2 of this thesis. Afterwards, the student will be directed to the Coding Editor for that exercise.

6.1.3 Coding Editor

The requirement FR-15 has been accomplished successfully. The Coding Editor has a property readOnlyMode which can be set to true to open the editor in read only mode. However, since there is no use case for this requirement at this point, there is no call for the editor in read only in the current QDAcity frontend. But it can be utilitzed whenever this feature is needed.

6.1.4 Exercise Management

All requirements related to exercise management have been successfully accomplished. The following is a summary:

- FR-16 and its subrequirements: the instructor can create exercises from the term course configuration page. All the details mentioned in the subrequirement can be set from the exercise creation dialog shown in figure 5.5.
- The endpoint removes the applied codings during the copy process as FR 16.6.1 states.
- The instructor can extend the exercise by checking the box "Extends an existing exercise" mentioned in section 5.6.2. This does not affect the accessibility of the previous exercises and therefore fulfills the subrequirement FR-16.5.
- FR-17: the instructor can delete exercises from the term course configuration page.
- FR-18: the instructor can view a list of student exercise projects from the exercise configuration page.

6.1.5 Exercise Evaluation and Feedback

The requirement FR-19 has been successfully accomplished. The instructor can run the intercoder agreement algorithm (whose refactoring is described in section 5.4) from the exercise configuration page.

Regarding FR-20, the cronjob along with the servlet described in section 5.3 satisfy this requirement.

For requirement FR-21, the instructor can view the progress of a student by clicking the editor button in the list of exercise projects of the exercise configuration page.

Requirements FR-22, FR-24 and FR-25 have not been accomplished. The reason behind this is that the time constraints did not allow for the implementation of these requirements. Mainly because the previous requirements consumed more time than expected due to the refactoring efforts which were not accounted for in the demo script.

The requirement FR-23 has been partially accomplished. Specifically, its subrequirement FR-23.1 is fulfilled. The student can access the intercoder agreement evaluation by click the report button for an exercise in the term course page. The subrequirements FR-23.2 and FR-23.3 are dependent on FR-24 and FR-25 and therefore have not been accomplished.

6.1.6 Authentication and Authorization

The requirements FR-26 to FR-29 have been successfully accomplished. All endpoint methods in ExerciseEndpoint and CourseEndpoint make authentication and authorization checks on the required level before modifying or fetching the required data from the datastore. They also allow administrators to access them according to requirement FR-29.

6.2 Non Functional Requirements

6.2.1 Quality Metrics

The implementation of the features which were developed during this thesis is measured against the quality metrics mentioned in section 4.3.1,

The criteria to satisfy all of eight categories has been met. Table 6.1 on the next page provides the reasoning to support this evaluation.

Category	Reasoning
Functional Suitability	The unit tests and manual tests performed verify that the criteria has been met.
Reliability	**1.** This is verified because QDAcity runs on Google App Engine, which provides a monthly uptime percentage of 99.95%(Google, 2018)
	2. All new endpoint methods provide accurate descriptions of exceptions that may occur and the frontend code is designed to continue running if one module fails.
Performance Efficiency	The new features and components are following the same design guidelines,which are used by the existing QDAcity.
Usability	The fullfillment of user interface guidelines described in the usability requirements section verifies the compliance with this criteria.
Security	**1.** Expected failures due to authorization are covered through unit tests.
	2. The authorization and authentication checks for all new api methods confirm that this criteria is met.
Compatibility	New features and components were based on the existing QDAcity codebase and extend its functionality and their compatibility is a prerequisite for their successful implementation.
Maintainability	**1.** The good test coverage ensures that any breaking changes will be identified immediately.
	2. The design for evaluation of exercises is easily extensible with other evaluation methods.
Transferability	The new features have been tested on different operating systems (OSX and Windows) and different browsers (Google Chrome, Firefox, Safari).

Table 6.1: The evaluation of non-functional quality metrics

Additionally, the Usability related non-functional requirments NFR-1 to NFR-4 which are mentioned in section 4.3.1.4 have all been met successfully. All strings use the localization framework which is used by QDAcity as requirement NFR-1 states. They are also translated into german according to requirement NFR-2. This requirement is verified by an existing test as well.

Requirement NFR-3 is also fulfilled since the new UI components are using only existing theme colors. All common components use the existing styled components as NFR-4 states.

The unit testing related requirement NFR-5 which is mentioned in section 4.3.1.7 have been accomplished. The new api methods in ExerciseEndpoint and CourseEndpoint have been covered with unit testing. The current code coverage for these classes is (as reported in gitlab's CI pipeline information):

- ExerciseEndpoint: 90%
- CourseEndpoint: 90%

The automated acceptance tests satisfy the requirement NFR-6. Regarding the refactoring of the evaluation algorithm mentioned in requirement NFR-7, the refactoring is a prerequisite to FR-19 and has been fulfilled accordingly.

6.3 Evaluation of Demo Script Use Cases

The following sections provide a detailed summary of the fulfilment of each of the use cases of the demo script with regards to the requirements which are derived from it.

Each section provides a table which shows the steps of the use case, the requirements which are derived from it, and whether each has been fulfilled in this thesis.

6.3.1 Codebook Coding

The use case "Codebook Coding" is fully covered. Table 6.2 shows a summary of its steps, related requirements and their fulfilment status.

Step	Short Description	Requirements	Status
1	Instructor creates term course	FR-1, FR-3	Fulfilled
2	Instructor invites students	FR-9	Fulfilled
3	Instructor creates exercise	FR-16	Fulfilled
4	Student accepts invitation	FR-10	Fulfilled
5	Student joins exercise	FR-14	Fulfilled
6	Student applies codebook codes	-	Existing Feature
7	Instructor views list of projects	FR-18	Fulfilled
8	Instructor creates evaluation report	FR-19	Fulfilled
9	Students view their evaluation	FR-23.1	Fulfilled

Table 6.2: Codebook Coding use case evaluation

6.3.2 Open Coding

The use case "Open Coding" is partially covered. Specifically, the functionality of writing comments by the instructor is not supported (mentioned in the requirement FR-22). However, the other steps in the use case are covered.

Table 6.3 shows a summary of its steps, related requirements and their fulfilment status. The main difference is between this use case and the previous one is that the last two steps here have not been fulfilled.

Step	Short Description	Requirements	Status
1	Instructor creates term course	FR-1, FR-3	Fulfilled
2	Instructor invites students	FR-9	Fulfilled
3	Instructor creates exercise	FR-16	Fulfilled
4	Student accepts invitation	FR-10	Fulfilled
5	Student joins exercise	FR-14	Fulfilled
6	Student applies open codes	-	Existing Feature
7	Instructor views list of projects	FR-18	Fulfilled
8	Instructor provides comments	FR-22	Not Fulfilled
9	Students view their evaluation	FR-23.2	Not Fulfilled

Table 6.3: Open Coding use case evaluation

6.3.3 Axial Coding

The use case "Axial Coding" has been partially covered. Specifically, the functionality of writing and viewing comments and project summaries by peers and setting this feedback method by the instructor is not supported. Similarly to the previous case, the rest of the steps are covered.

Table 6.4 shows a summary of its steps, related requirements and their fulfilment status.

Step	Short Description	Requirements	Status
1	Instructor creates term course	FR-1, FR-3	Fulfilled
2	Instructor invites students	FR-9	Fulfilled
3	Instructor creates exercise Instructor sets the number of peers	FR-16, FR-25	Partially Fulfilled
4	Students accept invitation	FR-10	Fulfilled
5	Students join exercise	FR-14	Fulfilled
6	Student A applies open codes	-	Existing Feature
7	Student B views the project of A	-	Existing Feature
8	Student B provides comments	FR-24	Not Fulfilled
9	Student B writes summary	FR-24	Not Fulfilled
10	Student B applies open codes	-	Existing Feature
11	Student A provides comments	FR-24	Not Fulfilled
12	Students view peer comments	FR-23.3	Not Fulfilled
13	Instructor view student projects	FR-21	Fulfilled

Table 6.4: Axial Coding use case evaluation

7 Conclusion

The goal of this thesis was to extend the feature set of QDAcity to further support teaching Qualitative Data Analysis through three types of exercises.

According to the previous detailed evaluation, the majority of the requirements have been accomplished. Specifically, the requirements FR-22, FR-24, FR-25 have not been accomplished, while the requirements FR-7, FR-8 and FR-23 have been partially accomplished. The rest of the requirements and their subrequirements have been successfully accomplished.

Accordingly, the outcome of the thesis has fully covered the first use case of the demo script "Codebook Coding" and partially covered the remaining two use cases "Open Coding" and "Axial Coding". Fulfilment of the remaining requirements FR-22, FR-23, FR-24 and FR-25 in the future will ensure the coverage of the two remaining use cases.

The refactored codebase, in addition to the new course and exercise structure, should serve as a solid foundation upon which more teaching use cases can be built in the future.

List of Figures

List of Tables

References

Boston University. (2018). Qualitative data analysis software comparison. Retrieved April 25, 2018, from https://www.bu.edu/tech/services/cccs/desktop/distribution/nvivo/comparison/

Facebook Inc. (2018a). Design principles. Retrieved April 25, 2018, from https://reactjs.org/docs/design-principles.html

Facebook Inc. (2018b). Virtual dom and internals. Retrieved April 25, 2018, from https://reactjs.org/docs/faq-internals.html

Google. (2018). App engine service level agreement (sla). Retrieved April 25, 2018, from https://cloud.google.com/appengine/sla

ISO/IEC. (2010). *Iso/iec 25010 system and software quality models*. ISO/IEC.

M. Corbin, J. & Strauss, A. [Anselm]. (1990). Grounded theory research: Procedures, canons and evaluative criteria. *13*.

Pandit, N. R. (1996). The creation of theory: A recent application of the grounded theory method. the qualitative report, 2(4), 1-15. Retrieved April 25, 2018, from http://nsuworks.nova.edu/tqr/vol2/iss4/3

Rupp, C. (2014). *Requirements-Engineering und -Management: Aus der Praxis von klassisch bis agil*. Munich: Carl Hanser Verlag.

Strauss, A. [A.] & Corbin, J. (1990). *Basics of qualitative research: Grounded theory procedures and techniques*. Sage Publications.