Recommendation System for Qualitative Data Analysis

MASTER THESIS

Dominik Schöpf

Submitted on 31 August 2023

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

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

Declaration of Originality

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

Erlangen, 31 August 2023

License

Erlangen, 31 August 2023

Abstract

QDAcity is a cloud-based application for performing qualitative data analysis. The analysis takes place within the coding editor enabling users to assign codes to various text segments across multiple documents. These text segments are called codings. Besides its task-solving capabilities, collaboration plays a central role in QDAcity. Teams can be organized through different project types, with role-based permissions. It does offer technical support for data synchronization via its real-time service and a service for concurrent text editing is currently in development. These services, however, primarily target the technical aspects of collaborating on shared data. Currently, there is little support for communicating change proposals regarding the content of the data.

The goal of this thesis is to design and implement a recommendation system for QDAcity. As an integral part of QDAcity, this system enables users to create and review codesystem recommendations. The system focuses on codes while placing an emphasise on extensibility to include other entities such as codings or documents in the future. As a result, users are now able to communicate potential changes within QDAcity without leaving the platform. This improves the collaborative working capabilities of QDAcity and, consequently, improves the overall user experience.

Contents

1 Introduction **3**
1.1 Thesis Structure . . . 3
1.2 QDAcity . . . 4
1.3 Problem Statement . . . 6
1.4 Objective and Approach . . . 7

2 Related Work **9**
2.1 Literature Review . . . 9
2.2 Applications Supporting Recommendations . . . 9
2.2.1 Collaborative Editing . . . 10
2.2.2 Collaborative Reviewing . . . 12
2.2.3 Discussion . . . 15

3 Requirements **17**
3.1 Constraints . . . 17
3.2 Functional Requirements . . . 19
3.2.1 General Functionality . . . 19
3.2.2 Code-related Functionality . . . 20
3.3 Quality Requirements . . . 21

4 Architecture **27**
4.1 Background . . . 27
4.2 Refined Layered Architecture . . . 29
4.3 Recommendation Controller Design . . . 32
4.4 Recommendation Life Cycle . . . 32

5 Implementation **35**
5.1 Backend . . . 35
5.1.1 General Recommendation Classes . . . 35
5.1.2 Action Classes . . . 38

5.2 Frontend 39
5.2.1 Creation Flow 40
5.2.2 Review Flow 41
5.2.3 Review Editor and Modals 43
5.2.4 Conflict Handling 44
5.2.5 Notifications 45
5.3 RTCS 46

6 Evaluation **49**
6.1 Usability Test 49
6.1.1 Design 49
6.1.2 Results 52
6.2 Functional Requirements 57
6.2.1 General Functionality 57
6.2.2 Code-related Functionality 58
6.3 Quality Requirements 59

7 Discussion **65**
7.1 Findings and Limitations 65
7.2 Future Work 67

8 Conclusion **71**

Appendices **73**
A Usability Test Notes 75
A.1 Expert 1 75
A.2 Expert 2 76
A.3 Expert 3 77
B Questionnaire Results 78
B.1 Custom 79
B.2 The System Usability Scale (SUS) 80

References **81**

List of Figures

1.1 QDacity's coding editor 5
1.2 Real-time Collaborative Service (RTCS) Overview 6

2.1 Simple comment in Google Docs 10
2.2 Review panel in Google Docs 11
2.3 Text segment with suggestions in Google Docs 11
2.4 "Review suggested edits" component in Google Docs 12
2.5 Merge Request (MR) entry page in GitLab 13
2.6 "Activity" list on the "Overview" tab of an MR in GitLab . . 13
2.7 Threaded comments within the "Activity" list of an MR in GitLab 14
2.8 Commented change of an MR in GitLab 14

3.1 FunctionalMASTeR template 19
3.2 Quality characteristics of ISO/IEC 25010 21
3.3 PropertyMASTeR template 22

4.1 Refined Layered Architecture 30
4.2 Recommendation controller design 32
4.3 Flowchart of the recommendation life cycle 33

5.1 UML class diagram of the `Recommendation` class 36
5.2 UML class diagram of the recommendation controller 37
5.3 UML class diagram of `RecommendationAction` class 38
5.4 UML class diagram of the Action controllers 39
5.5 Recommendation mode button 40
5.6 User Interface (UI) of the creation modal 41
5.7 UI of the review component 42
5.8 UI of a deleted comment 42
5.9 UI of the recommendation list. 43
5.10 UI of the review editor tabs. 44

5.11 UI of a recommendation with a terminal invalid action. 44
5.12 Red dot on code (left) and recommendation level (right). . . . 45
5.13 Data flow of a RTCS request. 46
5.14 Relationships between recommendation and code handlers. . . 47

6.1 Global notification feature 55
6.2 Project settings modal 57
6.3 Unittest coverage of the recommendation system. 63

List of Tables

3.1 Permissions matrix for actions and project roles. 24
5.1 Extended RTCS messages (left) and events (right). 47
6.1 Added permissions to the `ProjectPermissions` enum class. . 61

Acronyms

API	Application Programming Interface
CES	Collaborative Editing Service
CD	Continuous Delivery
CI	Continuous Integration
DAO	Data Access Objects
JDO	Java Data Objects
LoC	Lines of Code
MR	Merge Request
PM	PersistenceManager
PR	Pull Request
QDA	Qualitative Data Analysis
RTCS	Real-time Collaborative Service
SUS	System Usability Scale
UI	User Interface

1 Introduction

Qualitative Data Analysis (QDA) is a process consisting of multiple activities. In general, QDA methods emphasize extracting pertinent information from qualitative data, analyzing its content, and deriving abstract interpretations. As collaboration in research increases, so does the coordination overhead. While having multiple researchers brings diverse perspectives, it may also result in varied interpretations. QDA Software (QDAS) aids by providing platforms that allow concurrent analysis of shared datasets, streamlining communication and ensuring consistent interpretation among researchers. Nevertheless, collaborating on shared data often involves iterative data adjustments before consensus is reached. Recommendations ensure that data changes occur once consensus is achieved. In the context of this thesis, the notion of a recommendation is restricted to *user-to-user* recommendations. In contrast, the more commonly used interpretation encompasses *system-to-user* recommendations. In this case, the system utilizes some type of algorithm to provide personalized recommendations based on user data. While there is some overlap in requirements, the focus of this thesis is on the user-to-user case. The terms recommendation and suggestion are used interchangeably throughout the thesis.

1.1 Thesis Structure

Chapter 1 starts with a brief overview of QDAcity[1]. and introduces related QDA concepts. The chapter continues with the objectives and the general approach of this project. In chapter 2, related research and systems that offer similar functionalities are examined. Inspired by the related work, chapter 3 provides an outline of the requirements for the recommendation system. The chapter is divided into sections covering constraints (3.1), functional requirements (3.2), and quality requirements (3.3). Chapter 4 is dedicated to the

[1]https://qdacity.com/

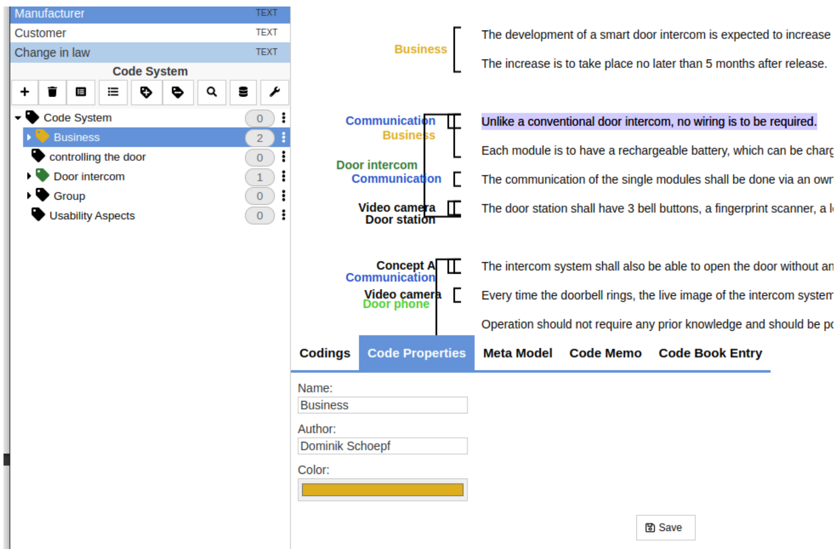
architectural design of the system. Building upon this, chapter 5 delves into the system's implementation. The first section of chapter 6 examines the usability test conducted as an integral part of this thesis. The chapter then proceeds with an evaluation of the requirements defined in chapter 3. In chapter 7, a discussion of the findings from the previous chapters and their implications is provided, addressing any limitations or challenges encountered during the project. This chapter also briefly presents prospects for future work. Finally, the thesis is concluded with a summary of the results in chapter 8.

1.2 QDAcity

QDAcity is a cloud-based web application for QDA. In addition to conventional QDA methods, it is specifically designed to support QDAcity-RE, a domain analysis method for requirements engineering (Kaufmann & Riehle, 2018). The primary artifact of QDAcity is the codesystem, which can be developed through the QDAcity-RE method or other QDA methods. The codesystem forms a hierarchical structure of codes, representing a model that encapsulates concepts, categories, their properties, and interactions. Therefore, a code represents a single concept. The code system emerges as a consequence of an iterative coding process. According to (Corbin & Strauss, 1990), the coding process of a single document is divided into three phases. The initial phase is to annotate relevant text segments (open coding), followed by a restructuring of the codesystem (axial coding). Finally, the last phase entails arranging and relating codes to the core category, resulting in a refined focus (selective coding) Every code represents a concept rooted in the collected data, encompassing attributes such as the label, a definition, and instructions for its intended usage.

The coding editor is the central feature of QDAcity. It supports various QDA-related workflows, which are beyond the scope of this thesis. Figure 1.1 shows the coding editor in action. The editor is divided into three main sections. The left section contains the codesystem, which is represented as a tree structure. The middle section displays the document to be coded. The bottom section presents details about the currently selected code, including its associated codings and properties.

QDAcity allows teams to work on projects through various organizational forms. Access to projects can be controlled on an individual user level through default permission roles: *Owner*, *Organizer*, *Editor*, and *Viewer*. When creating a brand-new project, it is of type *Project*. Alongside the



The development of a smart door intercom is expected to increase

The increase is to take place no later than 5 months after release.

Unlike a conventional door intercom, no wiring is to be required.

Each module is to have a rechargeable battery, which can be charg

The communication of the single modules shall be done via an owr

The door station shall have 3 bell buttons, a fingerprint scanner, a l

The intercom system shall also be able to open the door without an

Every time the doorbell rings, the live image of the intercom systerr

Operation should not require any prior knowledge and should be pc

Name: Business

Author: Dominik Schoepf

Color: 🟨

Figure 1.1: QDacity's coding editor

project itself, this encompasses a fresh codesystem. Initially, no documents or codings are included in the project. The remaining project types stem from the original project. A *ProjectRevision* can be regarded as a project snapshot. It replicates the entire codesystem, all codings, and documents. Separated from the original data, the revision project serves as a reference for recodings. For this practice, researchers recode the documents based on an existing codesystem. A *ValidationProject* enables this by duplicating all documents, but excluding any codings. It uses the same codesystem as the ProjectRevision, while not allowing any changes to it. The *ExerciseProject* shares the same characteristics as the *ValidationProject*, with the addition of duplicating all codings.

As a cloud-based platform, QDAcity places significant emphasis on collaborative work. The RTCS is the central feature that enables a collaborative workflow. Its responsibility is to synchronize various entity types among collaborators, such as codes, codings, documents, and users. Figure 1.2 shows a high-level overview of the RTCS.

For better coordination, the coding editor showcases active collaborators and the current document for each user. Additionally, to further enhance these capabilities, a dedicated Collaborative Editing Service (CES) is currently under development.

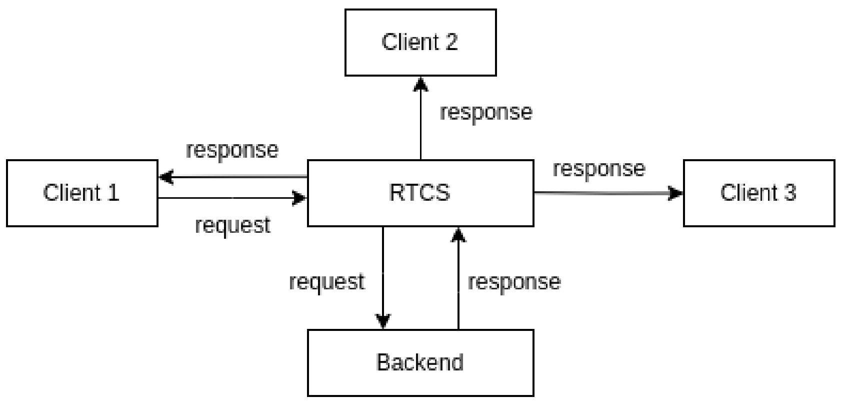


Figure 1.2: RTCS Overview

1.3 Problem Statement

As described in the previous section, QDAcity already offers various features to facilitate collaborative work. The RTCS enables multiple users to collaborate on a codesystem while maintaining an up-to-date[2] view of its current state. However, this mechanism solely addresses changes where the user has already made the decision and then proceeded to enact it. Where QDAcity lacks support is in a process that enables teams to coordinate what changes need to be made in the first place. This absence of a coordination mechanism can lead to several negative consequences for the codesystem's state. This might lead to needless changes, which require extra revision efforts. In such cases, it also introduces additional noise to evaluation methods reliant on tracking codesystem changes. The worst-case scenario is that changes might go unnoticed entirely or receive insufficient attention, resulting in an inconsistent model. Even when a specific change is conceptually agreed upon, errors can still occur without mechanisms in place to visualize and apply it automatically. While the primary focus of this project is on codesystem changes, this problem is also relevant to document changes (or any other form of shared data).

The topic of permission structures is always an important aspect when addressing change management of shared data. A closer examination of the permission roles *Editor* and *Viewer* reveals an interesting dichotomy. While the former allows users to make changes to the codesystem without any gatekeeping mechanism, the latter completely prohibits any form of interaction.

[2]Up-to-date here means that the RTCS works as expected with a delay of a few seconds at most.

A recommendation feature can introduce new policies by which users can actively contribute to the project without having the permission to make changes directly.

1.4 Objective and Approach

Derived from the problem statement, the primary objective of this thesis is defined as follows:

Design and implement a user-to-user recommendation system for QDAcity that allows users to create and review recommendations for potential changes.

In terms of project management, an agile approach was employed. This approach extended beyond the development workflow, shaping the emergence of the thesis through an iterative process. Due to the high-level nature of the functional requirements, a majority of them became evident after a few iterations. On the contrary, a significant portion of the quality requirements were implicitly established during the review process at QDAcity. They were later formalized as concrete requirements. This was primarily done for documentation purposes. Typically, an iteration encompassed sections that targeted related requirement definitions, their implementation, and their evaluation.

Tightly connected to the project management process are the software development practices that accompany it. However, it is difficult to point to a single practice constantly followed across QDAcity. Due to their oversight of multiple independent projects, the practices they adopt often vary among projects. Another factor is that time and frequency are key differentiators among the various strategies. Unlike teams primarily consisting of full-time developers, most members of QDAcity simultaneously engage in research activities. Consequently, even short development and integration cycles can span days, often with minimal divergence to the mainline. The following terminology is based on the works of Fowler (2020). Depending on the project, elements of traditional feature branching, trunk-based development, as well as Continuous Integration (CI) can be identified. As these branches are frequently integrated prior to being fully developed features, this approach would typically be considered trunk-based development or CI. Furthermore, not only are these branches integrated frequently, but they are also deployed to production, which constitutes a key aspect of Continuous Delivery (CD). At the same time, certain feature branches have lifespans extending beyond a month. Generally, the team advocates for small, short-lived feature branches.

The approach adopted in this project evolved over time. In the beginning, features were already divided into smaller chunks of work. Nevertheless, the review process extended over weeks due to the scale of code changes. As time progressed, the integration units decreased in size, leading to development-review-integration cycles lasting only a few days. Another contributing factor was the growing usage of follow-up integrations, rather than refining a single integration request.

2 Related Work

Section 2.1 presents a review of related literature on the topic. Then, section 2.2 examines two examples of applications that offer recommendation features.

2.1 Literature Review

The field of system-to-user recommendation systems is actively researched (Ko et al., 2022; Roy & Dutta, 2022). While much of the research on recommendation systems centers on algorithmic accuracy, Knijnenburg et al. (2012) adopts a user-centric approach to evaluating such systems. Nonetheless, this remains limited to system-to-user recommendations. Recommendations are also studied exclusively from a UI perspective (Harley, 2018a, 2018b). Yet, these research outcomes largely relate to system-to-user recommendations, given their focus on content and product suggestions. However, despite extensive exploration, no specific research on user-to-user recommendations was identified. This can largely be attributed to the fact that user-to-user recommendations are highly application-dependent. Even so, given the UI's central role in the recommendation feature, general UI principles are still relevant. Most notably, the usability heuristics by Nielsen (1994), serve as general principles for interaction design.

2.2 Applications Supporting Recommendations

When examining relevant applications, the first category of interest is QDAcity's competitors. At the time of the review, none of its competitors provides a feature of this nature.[1] There are, however, promising examples outside the space of specialized QDA software. By focusing on the general concept of

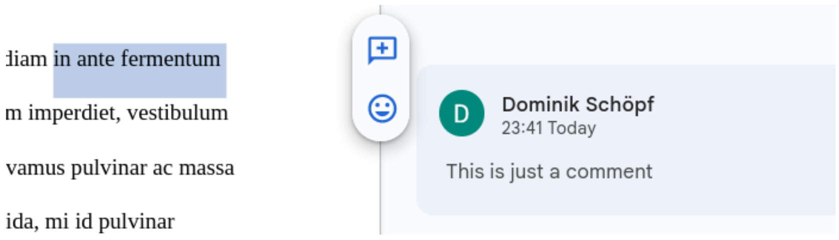
[1]MAXQDA, NVivo, ATLAS.ti, WebQDA, Taguette, Dedoose; as of 2023-08-30

recommendations, two discernible categories can be identified: collaborative editing and collaborative reviewing.

2.2.1 Collaborative Editing

Two prominent products in the text editing space are *Google Docs*'[2] *Suggestion Mode* and *Microsoft Word*'s[3] *Track Changes* feature. To ensure consistency, the following analysis will be confined to Google Docs. However, within the scope of this thesis, Microsoft Word's Track Changes offers similar capabilities.

Google Docs supports three distinct modes: *Editing*, *Viewing*, and *Suggesting*. The meaning and functionality of the first two modes are self-evident. In Suggestion Mode, every text change is interpreted as a suggestion. Hence, the actual content of the document is not modified. Before exploring the Suggestion Mode, we examine the comment feature. Comments can be used to communicate change requests or to get feedback on specific segments of the document. Additionally, they are an integral component of suggestions. A comment can contain plain text or links to other users, which allow to assign the comment to a particular user. Comments can be marked as resolved, which hides them from the regular view. However, they can still be seen in the comment history. Figure 2.1 shows a simple example of a comment.



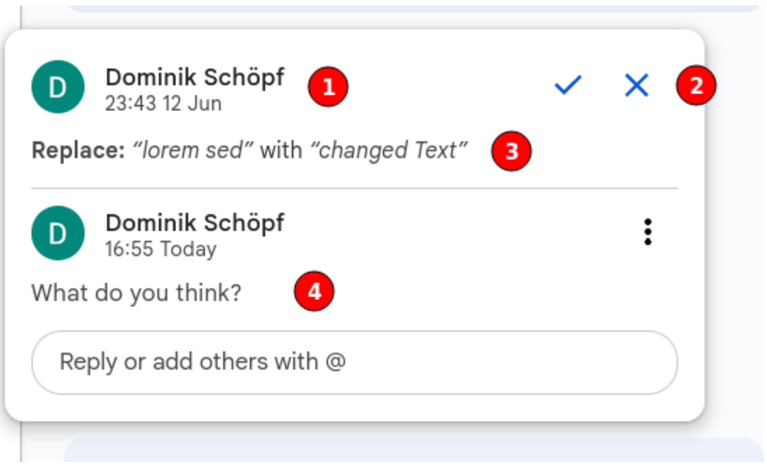
diam in ante fermentum
m imperdiet, vestibulum
vamus pulvinar ac massa
ida, mi id pulvinar

Dominik Schöpf
23:41 Today
This is just a comment

Figure 2.1: Simple comment in Google Docs

Interestingly, Google Docs largely provides multiple ways to achieve the same task. In total, there exist four ways to enter the Suggestion Mode. Either through the top toolbars or by using a context menu near the target. To further highlight a mode change, an extra notification-like pop-up briefly appears in the lower left corner. On a surface level, the UI appears unchanged apart from the selected mode. However, upon closer examination, it becomes apparent that extensions are disabled. This indicates, that all remaining actions are regarded as suggestions.

[2]https://docs.google.com
[3]https://www.microsoft.com/en-ww/microsoft-365/word



Dominik Schöpf
23:43 12 Jun

Replace: *"lorem sed"* with *"changed Text"*

Dominik Schöpf
16:55 Today

What do you think?

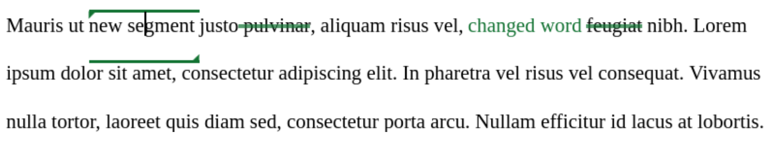
Reply or add others with @

Figure 2.2: Review panel in Google Docs

After creating the suggestion, the review component appears on the right side of the document. Figure 2.2 illustrates the review component, which can be further divided into sub-components.

1. The author and the creation date of the suggestion
2. Controls to resolve the suggestion (accept or reject)
3. A textual representation of the change
4. A comment section

Considering the context of text editing, the mode supports several actions, most notably: add, delete, and replace. Relocating text segments is achieved through a combination of delete and add actions. Apart from text changes, it offers numerous types of formatting changes. In addition to the textual representation, suggestions are displayed inline within the document. When selecting a suggestion, the corresponding text segment is further highlighted. Figure 2.3 showcases a text segment featuring several suggestions.



Mauris ut new segment justo ~~pulvinar~~, aliquam risus vel, changed word ~~feugiat~~ nibh. Lorem ipsum dolor sit amet, consectetur adipiscing elit. In pharetra vel risus vel consequat. Vivamus nulla tortor, laoreet quis diam sed, consectetur porta arcu. Nullam efficitur id lacus at lobortis.

Figure 2.3: Text segment with suggestions in Google Docs

Once a suggestion is accepted, the corresponding change is automatically applied, and the suggestion disappears.

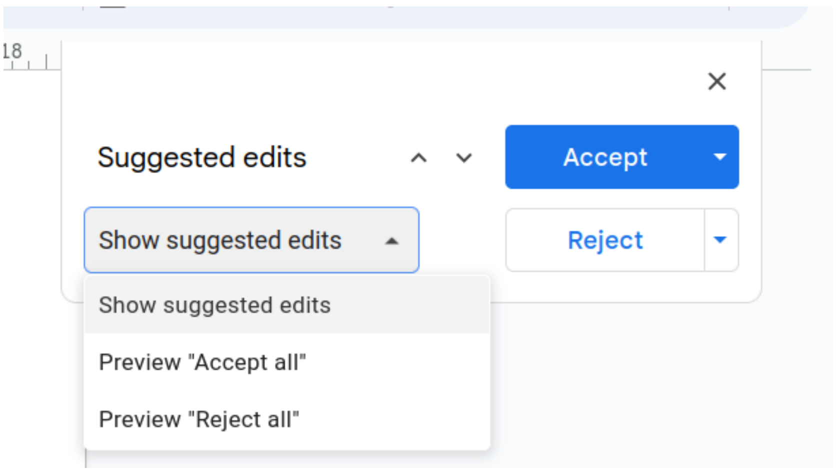


Figure 2.4: "Review suggested edits" component in Google Docs

Apart from manually selecting and reviewing individual suggestions, Google Docs offers a collective review titled "Review suggested edits". This feature allows users to browse through their suggestions sequentially and provides options to preview and resolve all suggestions at once. Figure 2.4 shows the corresponding element, which is located in the top toolbar.

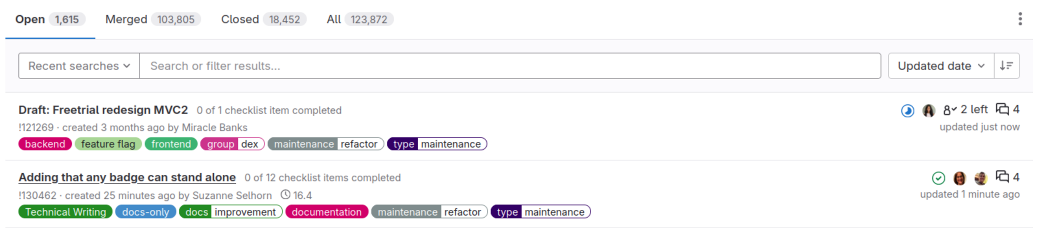
2.2.2 Collaborative Reviewing

The process of code review is especially interesting for the scope of this thesis. In *GitLab*[4] they are called MR while in *GitHub*[5] they are called Pull Request (PR). Considering the context of this thesis, they offer equivalent features. For simplicity, only UI elements from GitLab are shown in the following figures. The screenshots used in this section are taken from the official GitLab repository.[6]

GitLab offers a dedicated page for managing MRs. When a new MR is created, it is listed under the "Open" tab. Additionally, the pages include "Merged", "Closed", and "All" tabs. For every MR, an overview is displayed, which is divided into two sections: left and right. The left side provides details such as the name, ID, author, labels, and the creation date. The right section displays information like the last update, assigned users, comment count, and the current pipeline status. Except for the content of the MR, including comments, this encompasses all the vital information about the

[4]https://gitlab.com
[5]https://github.com
[6]https://gitlab.com/gitlab-org/gitlab

MR. To better manage multiple MRs, the entry page features a search bar that allows users to search and filter through existing MRs. All attributes of an MR, such as the author or labels, are integrated into the search, enhancing user-friendliness.



Open 1,615 Merged 103,805 Closed 18,452 All 123,872

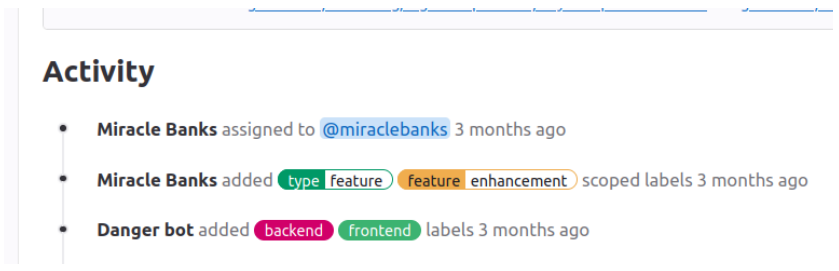
Recent searches | Search or filter results... | Updated date

Draft: Freetrial redesign MVC2 0 of 1 checklist item completed
!121269 · created 3 months ago by Miracle Banks
backend | feature flag | frontend | group dex | maintenance refactor | type maintenance
2 left 4
updated just now

Adding that any badge can stand alone 0 of 12 checklist items completed
!130462 · created 25 minutes ago by Suzanne Selhorn 16.4
Technical Writing | docs-only | docs improvement | documentation | maintenance refactor | type maintenance
4
updated 1 minute ago

Figure 2.5: MR entry page in GitLab

Following the same layout, the review page for an individual MR features the following tabs: "Overview", "Commits", "Pipelines", and "Changes". The "Overview" and the "Changes" tabs are particularly interesting for our purposes. The "Overview" tab provides a textual description of the MR, typically explaining the rationale behind the request and offering additional context information. In software projects, traceability is a prevalent requirement. Thus, the "Overview" tab features an "Activity" list, which showcases the initiator, the corresponding action, and the date of each MR update. Figure 2.6 displays parts of the "Activity" list of an MR.

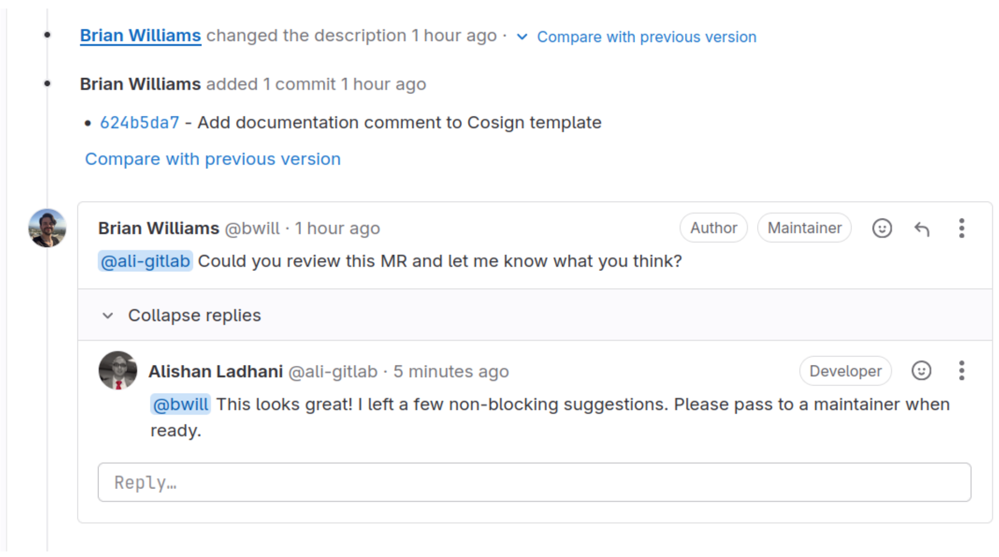


Activity

- Miracle Banks assigned to @miraclebanks 3 months ago
- Miracle Banks added type feature feature enhancement scoped labels 3 months ago
- Danger bot added backend frontend labels 3 months ago

Figure 2.6: "Activity" list on the "Overview" tab of an MR in GitLab

Given the complexity of MRs, GitLab provides a comprehensive commenting feature. Beyond supporting markdown text, it enables user tagging, the attachment of files, and emoji reactions. Users can reply to individual comments up to one level deep. Moreover, comments within MRs support threaded discussions that can be resolved. The thread status allows distinct issues to be addressed within an MR. Figure 2.7 shows a threaded comment within an MR.



Brian Williams changed the description 1 hour ago · Compare with previous version

Brian Williams added 1 commit 1 hour ago

- 624b5da7 - Add documentation comment to Cosign template

Compare with previous version

Brian Williams @bwill · 1 hour ago Author Maintainer

@ali-gitlab Could you review this MR and let me know what you think?

Collapse replies

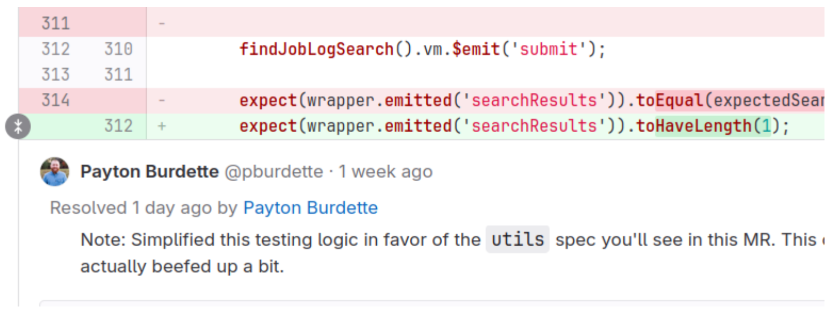
Alishan Ladhani @ali-gitlab · 5 minutes ago Developer

@bwill This looks great! I left a few non-blocking suggestions. Please pass to a maintainer when ready.

Reply…

Figure 2.7: Threaded comments within the "Activity" list of an MR in GitLab

The "Changes" tab presents a list of all proposed changes. Since code is primarily structured by lines rather than words, modifications are displayed as a line-by-line delta. However, the precise alterations within those lines are also highlighted for clarity. While the comment feature is intrinsically valuable, reviewers frequently want to discuss specific sections of code. To accomplish this, comments can be anchored directly to a designated range of lines. Figure 2.8 showcases a commented change within an MR.



```
-
        findJobLogSearch().vm.$emit('submit');

-       expect(wrapper.emitted('searchResults')).toEqual(expectedSear
+       expect(wrapper.emitted('searchResults')).toHaveLength(1);
```

Payton Burdette @pburdette · 1 week ago

Resolved 1 day ago by Payton Burdette

Note: Simplified this testing logic in favor of the `utils` spec you'll see in this MR. This
actually beefed up a bit.

Figure 2.8: Commented change of an MR in GitLab

MRs can be approved by users possessing the appropriate permissions. Notably, this does initiate the merging process but rather conveys the approval status of the MR. The act of merging remains a separate action.

2.2.3 Discussion

Google Docs and GitLab both provide features for suggesting and reviewing changes. Yet, their design and extent of functionality greatly differ due to their different users needs. By comparing these two applications and identifying similarities we can get insights into the design of a recommendation feature for QDAcity.

The most prominent difference is the scale of changes they manage. Collaborative text editing, typically, centers around individual documents, often involving modifications of small text segments. In contrast, code repositories deal with entire projects where changes can span over hundreds of files. In our context, the extent of changes would be more similar to the text editing scenario. The codes within codesystem possess only a handful of attributes that can be changed. The similarity becomes even more evident when considering coding recommendations, which essentially are annotated text segments. Thus, in-line changes, similar to those in Google Docs, are appropriate for coding recommendations. When designing a UI for code recommendations, it is important to factor in the existing UI of the coding editor. The codesystem is presented as a tree, showcasing only the name and a few icons. The properties of a code are accessed by opening a separate panel. Therefore, any UI designed for code recommendation must integrate into the codeystem component and the properties panel. While recommendations for updating code attributes could be shown in-line, the limited space for each attribute might lead to a cluttered UI. Displaying the proposed change inside a separate modal seems more suitable since QDAciy already relies on modals for various user interactions. A UI similar to how GitLab presents changes seems suitable for displaying the proposed changes of a codesystem code. The codesystem element should indicate existing recommendations. However, given the limited space, only visual cues are practical. For consistency, it is logical to reuse the modal for the remaining actions (e.g. relocate, remove). Another consequence of the scale of changes is the expected communication needed for individual changes. This is reflected in how sophisticated the comment features are. Given the anticipated communication for individual recommendations is relatively small, a comment feature similar to Google Docs should suffice.

3 Requirements

The objective of this project is to design and implement a recommendation system for QDAcity. This chapter outlines the requirements that are essential for achieving this goal. Requirements are divided into functional and quality (or non-functional) requirements. To reduce language ambiguities and minimize efforts in determining, communicating, and documenting requirements, a template-based approach is adopted (Rupp & SOPHISTen, 2020). It is important to note, however, that the authors' recommendation is to use the following templates for projects where the requirements are known beforehand. Commonly, a waterfall approach is used in these types of projects. Despite this, due to their simplicity and the fact that the requirements target relatively high-level functions, it was deemed appropriate in this context.

3.1 Constraints

The recommendation system shall be integrated into QDAcity's core technology stack.[1] Unlike a micro-service, which typically comes with a higher degree of freedom in terms of technology choices [2], the technologies used in this project are entirely determined by the surrounding system. The following is an overview of the current stack at QDAcity.

Backend

- Programming language: Java 8[3]
- Cloud infrasructure: Google Appengine[4]

[1]Separate services (e.g. collaborative-editing) are not considered part of the core.
[2]Independent technology stacks are actually one of the main benefits of micro-services.
[3]https://openjdk.org/projects/jdk8/
[4]https://cloud.google.com/appengine

- Database: (Google) Datastore[5]
- Data Access APIs:
  - Java Data Objects (JDO)[6],
  - Objectify[7],
  - App Engine API[8],
- Testing framework: JUnit 5[9]

Frontend

- Programming language: JavaScript[10] (Node v16)[11]
- Framework: React (v18)[12]
- Styling library: Styled-components[13]

The CI/CD approach, described in section 1.4, puts additional constraints on the project. While CI primarily impacts the development process, CD has additional implications regarding data consistency. Given that real users have access to intermediate versions of the recommendation system, new releases may have to be accompanied by data migration tasks.

During the duration of this project, the development team decided to migrate from JDO to Objectify. To speed up the process it is desirable to extend the backend architecture described in chapter 4 to other existing classes.

QDAcity uses the frontend framework React. Due to an ongoing migration within the maintenance of the overall QDAcity codebase, the frontend consisted of a mix of functional and class-based components. A requirement for this thesis was to write new react components as functional components. Section 3.3 details these migration tasks from the standpoint of compatibility and maintainability.

[5]https://cloud.google.com/datastore
[6]https://db.apache.org/jdo/
[7]https://github.com/objectify/objectify
[8]https://cloud.google.com/appengine/docs/legacy/standard/java/datastore/
[9]https://junit.org/junit5/
[10]https://developer.mozilla.org/en-US/docs/Web/JavaScript
[11]https://nodejs.org/en
[12]https://react.dev/
[13]https://styled-components.com/

3.2 Functional Requirements

The functional requirements defined in this section adhere to the Functional-MASTeR template (Rupp & SOPHISTen, 2020) depicted in figure 3.1.

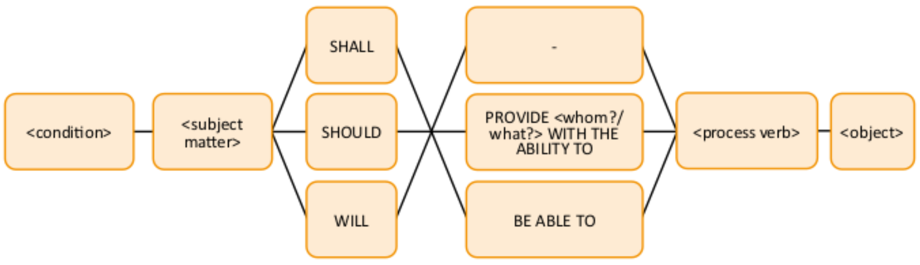


Figure 3.1: FunctionalMASTeR template

The auxiliary verbs *shall*, *should*, and *will* are used to express a degree of importance. In practice, they are associated with legal bindings. Mandatory requirements are expressed with the keyword *shall*. The keyword *should* is used for requirements that are desirable but not mandatory. The *will* keyword allows making statements about the future, thereby facilitating development efforts in anticipation of future requirements.
The following conventions are used to further improve readability. The `<subject matter>` is the *system* and always refers to the recommendation system that is developed in this project. Instead of `PROVIDE <whom?/what?> WITH THE ABILITY TO`, the phrase *allow users to* is used. The authors' recommendation is to use more specific terms than *users* because it is too generic. However, in this particular scenario for the development of a recommendation system, alternatives like *"QDA practitioners"* or *"coding editor users"* do not add much value either.

The requirements defined in this section assume that the user is authorized to perform the respective operation. Details on the permission structure associated with each action are discussed in section 3.3.

3.2.1 General Functionality

The following requirements are related to the general functions of the recommendation system. They are independent of the supported entities or actions.

FR1	The system shall allow users to enable/disable recommendations on a project level throughout the duration of the project.

FR2	The system shall allow users to access a recommendation mode to create new recommendations.

FR3	The system shall allow users to access to all recommendations for a given project through a dedicated editor.

FR4	The system shall allow users to accept and reject recommendations.

FR5	The system shall allow users to vote on recommendations.

FR6	The system shall allow users to discuss recommendations via a comments section.

FR7	The system shall allow users to identify unseen changes in recommendations.

3.2.2 Code-related Functionality

While there are general requirements that apply to all entities, the primary focus of this project is on code-related recommendations.

FR8	The system shall allow users to create a recommendation to add a new code.

FR8.1	The system shall allow users to work with a partially functional code draft.

FR8.2	The system shall be able to remove the code draft after the recommendation is rejected or deleted.

A code draft refers to a code generated as a recommendation. While it is integrated into the codesystem, it is marked as a draft until the recommendation is accepted.

FR9	The system shall allow users to create a recommendation to relocate a code.

FR10	The system shall allow users to create a recommendation to update a code.

FR11	The system shall allow users to create a recommendation to remove a code.

FR12	The system shall be able to apply the action of a recommendation after the recommendatoin is accepted.

For the rest of this thesis, if not specified otherwise, the terms *AddCode*, *UpdateCode*, *RelocateCode*, *RemoveCode*, refer to the **recommended** code action. Note that this is different from the **actual** code action. It is also different from the responsible controller classes which would be denoted as `AddCode` and so on.

3.3 Quality Requirements

The requirements outlined in the previous section are essential for evaluating the outcome of this project. However, the quality of a software system cannot be properly assessed solely through functional requirements.

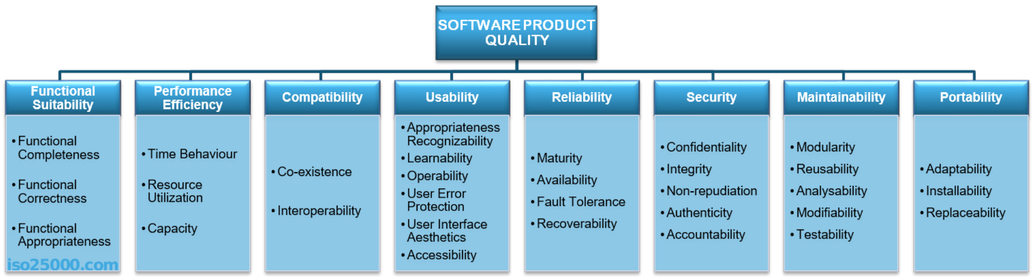


Figure 3.2: Quality characteristics of ISO/IEC 25010

According to ISO25010[14], the quality of a system is determined by how well it meets the needs of its stakeholders (e.g. usability, security, maintainability). For this project, the two most important stakeholder groups are the users and the developers at QDAcity. The quality model categorizes product quality into characteristics and sub-characteristics. Figure 3.2 shows the eight different quality characteristics.

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

The requirements defined in this section follow the PropertyMASTeR template (Rupp & SOPHISTen, 2020), shown in figure 3.3.

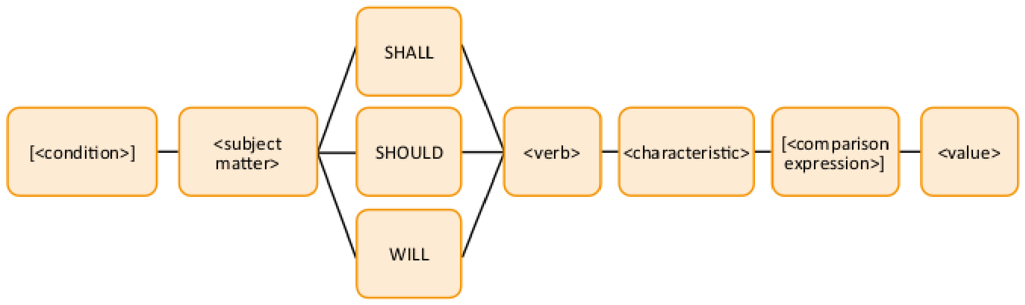


Figure 3.3: PropertyMASTeR template

Functional Suitability

QR1	The system shall not break any existing functionality of QDAcity.

Because of the CI/CD process, QR1 is applicable not only to the final outcome but also to every integration.

QR2	The system shall support a fully functional end-to-end recommendation workflow.

Due to the agile nature of the project, the set of supported entities and actions was always open for reevaluation. However, end-to-end functionality means that regardless of how the scope of this set of entities and actions is defined, the workflow from creation, review, to resolution shall be fully covered.

Performance Efficiency

QR3	The system shall provide the same synchronization capabilities as the rest of the code system.

QR4	The system shall only load open recommendations by default. Resolved or deleted recommendations shall be loaded on demand.

Compatibility

QR5	The system shall be integrated into the existing QDAcity technology stack.

QR6	The system should be integrated into the technology stack according to the roadmap.

See section 3.1 for details on the technology stack.

Usability

QR7	The system shall provide users a localized (German) version for all user-facing text.

QR8	The system shall provide a UI that is consistent with the rest of the QDAcity design.

Security

The scope of security is limited to extending the respective modules. It is assumed that the existing checks work as expected.

QR9	Each endpoint shall check the authentication of a request. This shall be the first course of action.

QR10	Each endpoint shall check the authorization of a request.

QR10.1	The check shall be as early as possible. No data shall be persisted before the check.

QR10.2	The check shall implement the permission structure defined in Table 3.1.

Maintainability

Quality requirements for self-contained projects are typically confined by the boundaries of the project. Projects that consist of a feature for an ongoing project can adopt a wider view. This can include the degree to which the maintainability of the project as a whole was affected. To fulfill such a requirement the concept of a refactoring radius was utilized. It involves

Table 3.1: Permissions matrix for actions and project roles.

Type of action	Owner	Organizer	Editor	Viewer
Enable/disable service (FR1)	Yes	Yes	No	No
Accept and reject recommendations (FR4,12)	Yes	Yes	No	No
Enable/disable recommendation mode (FR2)	Yes	Yes	Yes	No
Create, update, delete recommendations (FR5,6,8-11)	Yes	Yes	Yes	No
View recommendations	Yes	Yes	Yes	No

refactoring code that is in close proximity to the code that is currently being developed. Proximity is determined by a cost-benefit analysis on a case-by-case basis. Even though this will not be defined as a formal requirement, it was always an important aspect of the development process. There will be references to the refactoring efforts throughout this thesis.

The recommendation system is supposed the handle various entity types. Thus, modularity, reusability, and modifiability are particularly important for this project.

QR11	The system shall be open to extension for additional entities and actions. The extension shall be possible without modifying any code related to the general functionality.

The quality of a test suite of a system has a direct impact on multiple quality characteristics. In most applications, the functional correctness (functional suitability) of a system can only be assessed by an automated test suite. In general, the more test coverage a system has, the easier it is to maintain and modify. Additionally, test suites can serve as a form of documentation.

QR12	The system shall include an endpoint unit-test suite.

QR12.1	The suite shall have at least an 85% Lines of Code (LoC) coverage.

QR12.2	The suite shall have a 100% function coverage (of public endpoints methods.

QR12.3 The suite shall include authorization and authentication tests for all public endpoints methods.

QR13 The system should include acceptance tests that cover the features defined in 3.2.

4 Architecture

This chapter presents the backend architecture of the recommendation system. First, section 4.1 elaborates on several aspects of the backend codebase that guided the architecture design. Following this, section 4.2 outlines the overarching architecture that aims to address these concerns. Equipped with the core concepts, section 4.3 delves into the architecture of the recommendation system. Shifting the focus to the user, the last section (4.4) is dedicated to the modeling of the recommendation life cycle.

4.1 Background

The fact that the recommendation feature is embedded in a larger system has implications for the architecture design. The pattern not only has to be appropriate for the problem, but it also has to fit into the overall system. Otherwise, it becomes challenging to maintain and encourages developers to add additional patterns that may not align with the overall system. The goal of the exploration phase of the backend was to better understand the different patterns that were being used throughout the codebase. This involved consultation with the core team to assess whether certain patterns are in line with the overall strategy. During this process, it became clear that the pattern, which eventually emerged, serves a dual purpose. It is not only appropriate for the recommendation feature but also has the potential to improve the overall system when applied. The list below presents the key observations derived from the exploration.

Google Cloud Endpoints Framework

Endpoint classes and methods make use of Google's Cloud Endpoints Framework annotations for endpoint configuration. The key consideration for us is that these methods have the requesting user as the last parameter.

Anemic Domain Model

The backend predominantly adheres to an Anemic Domain Model, implying that most (domain) objects are simple data classes containing minimal logic. As a result, all business logic and persistence access must be located elsewhere. Primarily, this responsibility is managed by the endpoint methods. Consequently, all processing requires the user as a parameter. This binds permissions more to business logic than to individual requests. Although the testing strategy is intentionally centered around endpoint testing, this approach inherently leaves no alternatives.

The question of whether this constitutes good or bad practice is not within the scope of this thesis (Fowler, 2003). As previously mentioned, the goal is to devise a pattern that fits into the current system.

Authentication and Authorization

Given the absence of an explicit authentication concept, an invalid user parameter would lead to authorization checks failing. Typically, each endpoint employs authorization check(s) via the `Authorization.check` method. Importantly, this approach already separates this concern. However, because endpoints frequently invoke other endpoints, authorization is checked multiple times. This not only lacks clarity but is also inefficient.

Peristence Layer

Due to historical factors, the backend employs multiple persistence Application Programming Interface (API)s. The formerly used JDO library, which is now deprecated, is still present in the code base. The Objectify library is adopted for new features and is intended to replace JDO in the future. Nonetheless, this migration will require a substantial amount of time. The App Engine Datastore API is a low-level API, designed specifically for the datastore. This stands in contrast to JDO, a standardized API that makes no assumptions about the underlying technology. The low-level API is used in cases where the inheritance structure permits datastore access via JDO. The `DatastoreFacade` is a simple wrapper for the low-level API, which can be seen as an initial attempt to wrap vendor APIs. Additionally, the `Cache` class provides a cache layer to increase performance. However, it is not used consistently across the code base. Data access in JDO works distinctly from all other APIs. Whereas the other APIs provide functionality via static methods, JDO requires the caller to manage a PersistenceManager (PM) object manually. This led to extensive nested PM handling across the codebase.

Typically, endpoints combine direct data access through JDO with other functions, which manage their own PM instance.

This amalgamation of APIs is a result of multiple factors. Primarily, the vendor APIs lack custom interface wrappers. Additionally, QDAcity's organizational structure (see section 1.4), makes it difficult to perform large-scale restructurings in a short timeframe.

As a result, the development of high-level features often requires low-level knowledge that is not directly related to the task at hand.

4.2 Refined Layered Architecture

Following the explanations of Richards (2015, Chapter 1), the Layered Architecture Pattern is the most common architecture pattern, especially for Java applications. In a layered architecture, components are organized into horizontal layers. Each layer is concerned with a specific concern within the application. While the pattern itself does not specifiy the number or type of layers, the majority of applications have four layers: presentation, business, persistence, and database.

The separation of concerns is also one of the main advantages of this pattern. Layers can be categorized into two groups: open and closed. A closed layer strictly requires that higher layers only interact with that layer. In contrast, open layers permit higher layers to bypass them. This is an important feature to avoid inefficiencies and unnecessary complexity solely based on a rigid architecture definition.

While QDAcity already implements a layered architecture with its foundational business and persistence layers, addressing the issues outlined in the previous section requires a more refined approach. Figure 4.1 shows a high-level overview of the refined layered architecture.

Endpoints

Endpoint methods are primarily tasked with handling authorization and authentication of requests. Additionally, in certain cases, they handle unpacking request objects or managing response objects. They should be devoid of any business logic and definitely not interact with any of the persistence layer interfaces.

The endpoint layer is by default a closed layer because the presentation layer

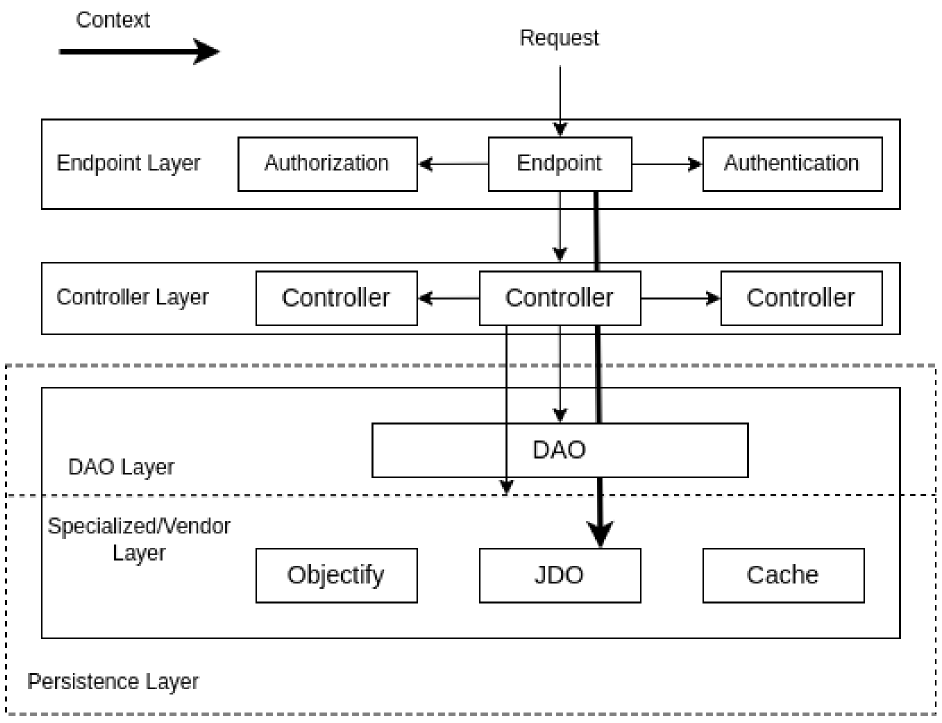


Figure 4.1: Refined Layered Architecture

is located on the client side.

Controllers

Controllers play the central role in this design approach. Within the system, they have multiple responsibilities. Primarily, every endpoint must have an associated controller function that encompasses the required logic to process the request. Often, these functions mirror the endpoints, but without the dependency on a user. Simultaneously, controllers act as an API for other controllers. This approach also prevents multiple authentication checks, delegating this concern to the calling endpoint. Owning to the Anemic Domain Model, they also provide public interfaces to interact with semantically related objects. This implies that not every object requires its own controller class. Controllers encapsulate all interactions with the persistence layer, enabling a smoother migration of persistence APIs without modifying any endpoints or calling functions. Such decoupling, also has advantages for testing, allowing the testing of functionality irrespective of authorization and authentication.

The controller layer functions as a closed layer, as endpoints are prohibited to access the persistence layer directly.

Data Access Objects (DAOs)

The Data Access Objects (DAO) pattern is a structural design pattern that enables the separation of the business layer from the persistence layer. While this concept is applicable to most environments that deal with persistence, it is predominantly used in Java applications and relational databases. Originally, the pattern features a relatively intricate class structure that involves abstract factories. However, the primary objective at this stage was to embed the logic within controllers and delegate datastore access to the DAOs. Instead of defining a fixed interface to encapsulate several interfaces upfront, the intention was to let the interfaces converge from slightly varying signatures. The majority of the DAOs employ either JDO or Objectify to perform datastore access. Merging these two into a unified interface remains significantly simpler than replacing the direct usage of the public APIs. Additionally, DAOs are responsible for simple caching mechanisms.

The DAO layer acts as an open layer, allowing controllers to directly interact with the different persistence APIs for operations like custom caching or complex queries.

Context Object

One observation described in section 4.1 indicates that two predominant dependencies: the user and the PM object. The purpose of the context object is to encapsulate these entities while making them accessible to all controllers. The context object manages the PM (i.e. opening and closing) and provides the methods (`execute(func)`, `executeWith(user, func)`), which allow the execution of arbitrary code without the necessity to manage PM directly. Additionally, it initializes the `UserSerivce` which provides access to the currently authenticated user. In the case of a non-authenticated user, it throws an exception. The creation of the context object is handled by the endpoint layer and is supposed to be the first action of every endpoint function. Subsequently, this object is passed to the controller constructors. Given that the context object encapsulates all shared data for a request, the construction of controllers is less likely to change. This stability allows for the development of controller functions without requiring specific knowledge about their usage.

The context object enables the passing of data from the endpoint layer to the DAO layer. While this is generally regarded as an anti-pattern, it is a necessary evil that expedites the migration process.

4.3 Recommendation Controller Design

The previous section introduced the key components of the refined layered architecture. While it later evolved into a broader strategy, its initial design was tailored for the recommendation system. In alignment with the layered pattern, the recommendation system comprises a `RecommendationEndpoint`, `RecommendationController`, and a `RecommendationDAO` class. Notably, the recommendation controller exclusively interacts with action controllers. A crucial element of this design is that the recommendation controller remains unaware of the specific types of recommendations supported. Instead, it only interacts with an action controller interface. These action controllers in turn interact with controllers responsible for the respective entity types. The separation of controllers and endpoints, allows us to define different permission requirements when accessed through the recommendation workflow. Figure 4.2 shows the component view of the system.

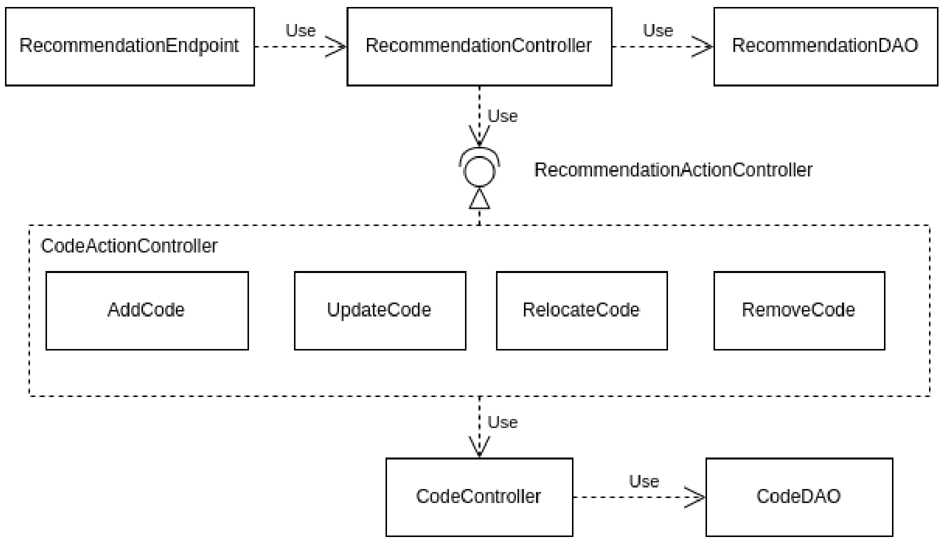


Figure 4.2: Recommendation controller design

4.4 Recommendation Life Cycle

Section 3.2.1 has already outlined the various features that a single recommendation must encompass. However, solely based on the requirements, it remains unclear how these individual features fit into the end-to-end recommendation workflow. There are multiple benefits to explicitly modeling the recommendation life cycle. Foremost, it enables us to verify our understanding of the problem. Consequently, it aids in determining the correctness and completeness of the requirements. Moreover, a model serves as documentation and a point of reference for future developments. Figure 4.3 depicts the flowchart of the recommendation life cycle.

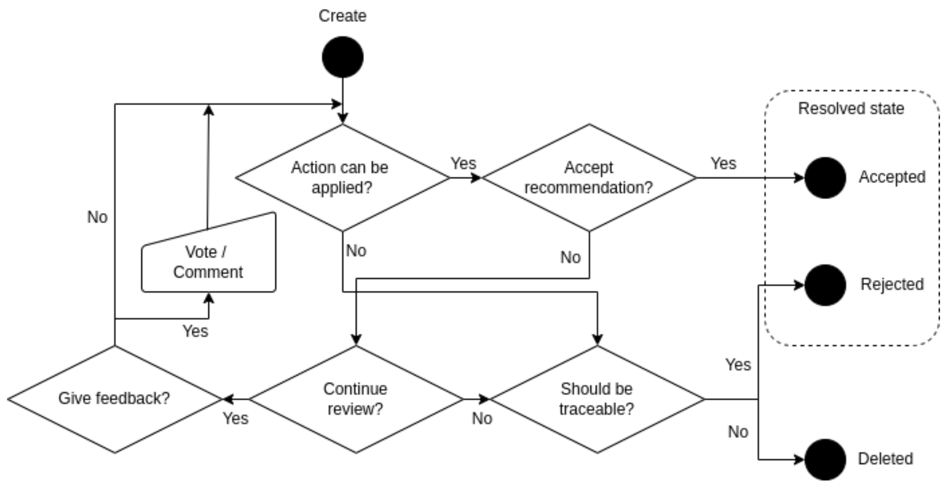


Figure 4.3: Flowchart of the recommendation life cycle

The recommendation's life cycle starts with its creation. In principle, it is possible to model the flow of the creation process too. However, this depends on the type of recommendation, and the life cycle model should remain independent of the specific type. After the creation of the recommendation, it transitions into the review phase, which essentially involves a loop of voting and commenting. This continues until a decision is reached regarding the resolution of the recommendation. However, a recommendation can only be accepted if its action can be applied. Whether an action can be applied or not depends on the codesystem state. The action of a recommendation can be seen as a deferred change, which implies that the state of the data may have changed between the creation and the review of the recommendation. When human decisions are involved, this can take days or even weeks. This makes outdated reference values a very common scenario. However, this does not necessarily mean that the action cannot be applied anymore. For example, the recommendation to change the name of a code is still valid, even if the name has already been changed. On the other hand, the recommendation to change the name of a code, that has already been deleted, is not valid anymore. In case the user disapproves (or the action cannot be applied), there are two possible ways to dismiss a recommendation. Depending on whether the recommendation should still be traceable, the user can reject or delete it.

5 Implementation

Chapter 5 describes the implementation of the most important classes and components of the recommendation system. The chapter is divided into three sections. The first section (5.1) focuses on the core backend classes involved in the recommendation workflow. Following that, section 5.2 discusses the frontend implementation and UI elements. In the last section (5.3), the RTCS extension is explained, which is responsible for the synchronization.

5.1 Backend

To ensure a clean separation of concerns, the functionality of the recommendation system is divided into two distinct groups of classes. Subsection 5.1.1, focuses on classes concerned with general functionality. Then, subsection 5.1.2 describes the classes that handle code-related functionality.

5.1.1 General Recommendation Classes

As outlined in the previous chapter the `RecommendationController` (Figure 5.2) acts as the central class, encompassing the entirety of the recommendation system's functionality. This section omits the `RecommendationEndpoint` class because its sole purpose is to check authorization and then delegate the task to the controller. Figure 5.1 shows the class diagram of the `Recommendation` class, while Figure 5.2 depicts the class diagram of the `RecommendationController` class.

The majority of the controller functions follow a two-step control flow. The initial step varies based on the specific function, primarily focusing on updating the recommendation's target fields The subsequent step is a universal update procedure, invoking `setLastUpdate()` and `updateSeenByUsers()`. The former simply modifies the `lastUpdate` field with the current user and timestamp. Meanwhile, the latter pertains to the notification feature. To

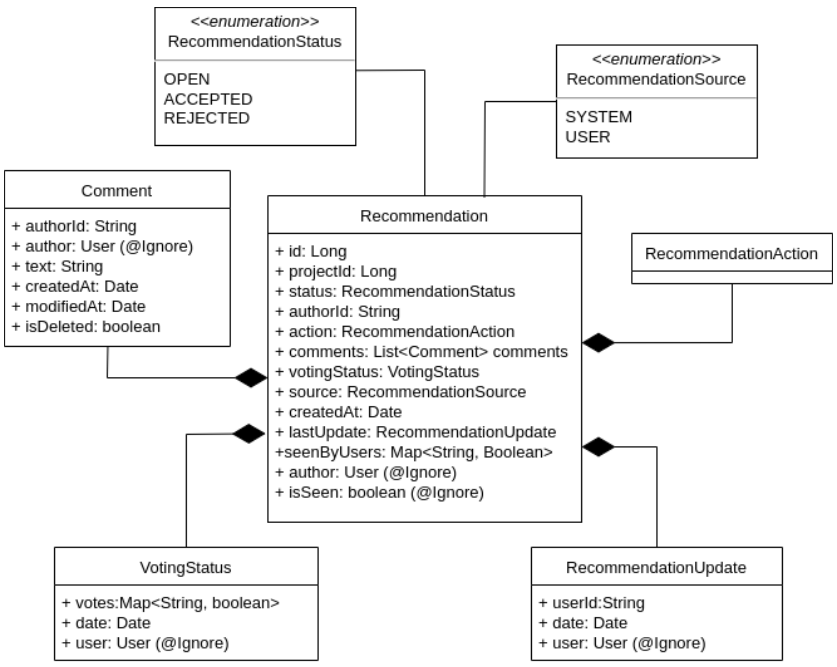


Figure 5.1: UML class diagram of the `Recommendation` class

implement the notification feature, it is essential to track whether a user has seen a recommendation. since its most recent update. The private attribute `seenByUsers` serves this purpose. Upon entering the coding editor, recommendations for the respective project are loaded. During this operation, the `isSeen` attribute is set dynamically according to the `seenByUsers` entry corresponding to the requesting user. For requests, that update the recommendation, all users are removed and only the user that initiated the request is set. The controller function `updateSeenByUsers()` manages this action. As soon as a user opens the review (see subsection 5.2.2), of an unseen recommendation, the `setSeen()` endpoint is invoked, updating the `seenByUsers` attribute.

The `VotingStatus` class is employed for storing user votes and acts as a wrapper around a `Map<String, Boolean>`. In this structure, the user ID serves as the key, while the vote itself is stored as a boolean value. Naturally, `true` represents an upvote, and `false` a downvote. The controller functions `vote()` and `retractVote()` are used to modify the voting status for the current user.

Due to the adoption of a flat comment structure, a basic `List<Comment>` suffices. As a result, it is only possible to append a new comment but not

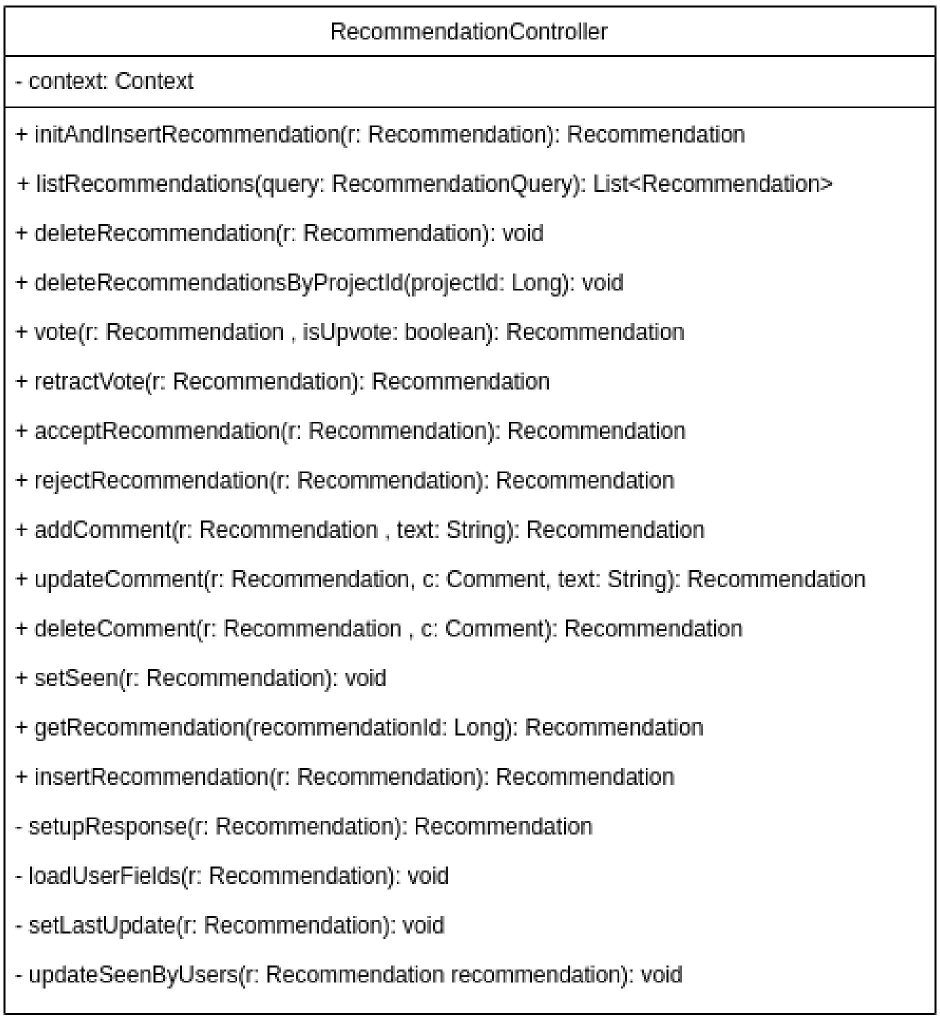

RecommendationController
- context: Context
+ initAndInsertRecommendation(r: Recommendation): Recommendation
+ listRecommendations(query: RecommendationQuery): List<Recommendation>
+ deleteRecommendation(r: Recommendation): void
+ deleteRecommendationsByProjectId(projectId: Long): void
+ vote(r: Recommendation , isUpvote: boolean): Recommendation
+ retractVote(r: Recommendation): Recommendation
+ acceptRecommendation(r: Recommendation): Recommendation
+ rejectRecommendation(r: Recommendation): Recommendation
+ addComment(r: Recommendation , text: String): Recommendation
+ updateComment(r: Recommendation, c: Comment, text: String): Recommendation
+ deleteComment(r: Recommendation , c: Comment): Recommendation
+ setSeen(r: Recommendation): void
+ getRecommendation(recommendationId: Long): Recommendation
+ insertRecommendation(r: Recommendation): Recommendation
- setupResponse(r: Recommendation): Recommendation
- loadUserFields(r: Recommendation): void
- setLastUpdate(r: Recommendation): void
- updateSeenByUsers(r: Recommendation recommendation): void

Figure 5.2: UML class diagram of the recommendation controller

to reply to an existing comment. Because a recommendation is always associated with a single code, the set of changes stays small. Consequently, the comment structure is not expected to grow very complex. Hence, this limitation is acceptable. The comment feature is fully supported with the following API endpoints and their corresponding controllers functions: `addComment()`, `updateComment()`, and `deleteComment()`. Naturally, only the author of the comment is authorized to update or delete it. The `Comment` class is a simple data class that contains the following attributes: `authorId`, `author`, `text`, `createdAt`, `modifiedAt`, `isDeleted`. As indicated by the `isDeleted` attribute, comments are not actually removed from the database. Subsection 5.2.2 explains the reasoning behind this decision.

5.1.2 Action Classes

The `RecommendationAction` class is a data class that contains all data necessary to perform a code action. The class diagram is shown in Figure 5.3.

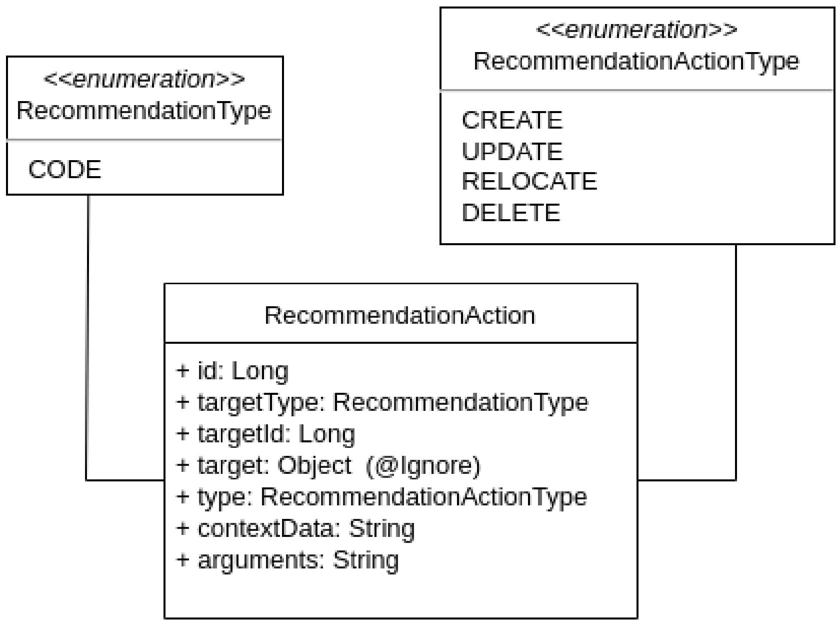


Figure 5.3: UML class diagram of `RecommendationAction` class

The `targetType` specifies the entity type. The `RecommendationType` enum currently only contains the `CODE` enum. In the case of code recommendations, the `targetId` refers to the code ID. The `Long` data type is sufficient since all potential entity candidates have IDs of type `Long`. The `type` attribute is, again, an enum of `RecommendationActionType` with values `CREATE`, `UPDATE`, `RELOCATE`, and `DELETE` that mirror the regular code actions. The actual parameters of the operation are stored in the JSON-formatted `String` attribute `arguments`. The `contextData` field is used to store additional information, such as the values of the target object at the time of the creation of the recommendation. Following the described controller approach the actual logic is implemented in action controllers.

Controller classes

For every valid pair of `RecommendationType` and `RecommendationActionType`, one action controller has to be implemented. This controller has to implement at least the top-level interface `RecommendationActionController`. In this case, only the `apply(RecommendationAction)` method has to be overridden.

Based on the values of `RecommendationType` and `RecommendationActionType` the factory method `createInstance` returns the responsible action controller. Additionally, it contains a default implementation for `hasDependencies` which simply returns `false`. The concrete action controllers `UpdateCode`, `RelocateCode`, and `RemoveCode` implement this interface directly because they reference already existing codes. However, for `AddCode` recommendations the code draft needs to be created and inserted into the actual code system during the creation process. More generally, actions that have some sort of dependency associated with them implement the `DependentActionController` interface. While `setupDependecies()` is called inside of `initAndInsertRecommendation()`, `removeDependecies()` is called inside `rejectRecommendation()` and `deleteRecommendation()`. Figure 5.4. shows the class diagram of the action controllers.

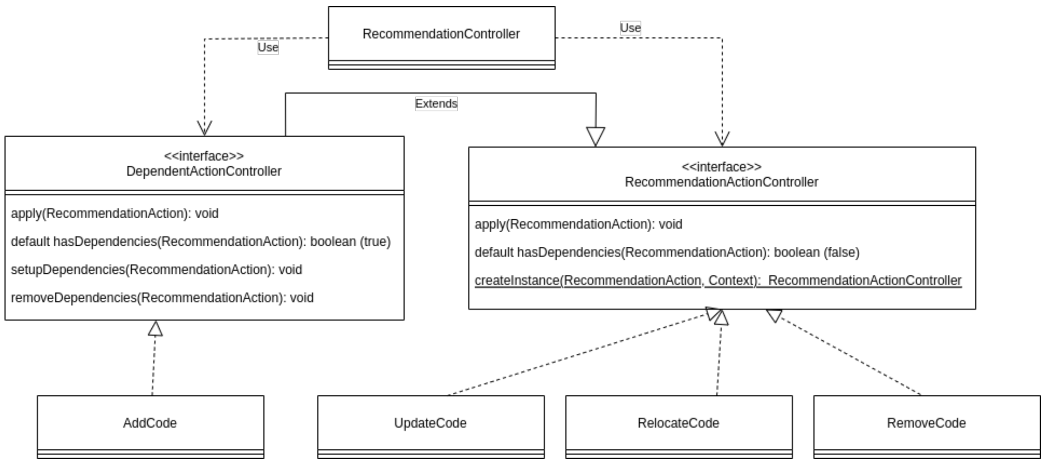

RecommendationController
Use
Use
Extends
<<interface>>
DependentActionController
apply(RecommendationAction): void
default hasDependencies(RecommendationAction): boolean (true)
setupDependencies(RecommendationAction): void
removeDependencies(RecommendationAction): void
<<interface>>
RecommendationActionController
apply(RecommendationAction): void
default hasDependencies(RecommendationAction): boolean (false)
createInstance(RecommendationAction, Context): RecommendationActionController
AddCode
UpdateCode
RelocateCode
RemoveCode

Figure 5.4: UML class diagram of the Action controllers

As described in 4.4, values in `arguments` may be outdated. Therefore, one would expect the interface to have a method like `checkForConflicts()`. However, the controllers can just assume that the data is still valid. The conflict handling is done on the frontend, which is explained in subsection 5.2.4.

5.2 Frontend

The first two subsections (5.2.1 and 5.2.2) address the fundamental components of the recommendation life cycle. Subsection 5.2.3 discusses the similarities and differences between the editor and the modals. The question of whether an action can be applied or not is explored in 5.2.4 *Conflict Handling*. To make this feature practical for collaborative work, subsection 5.2.5 explains the implementation of the notification feature.

5.2.1 Creation Flow

The codesystem supports four high-level code operations: insert, update, relocate, and remove. The recommendation system has counterparts for all four. While having a consistent overall UI for recommendations, an important goal of the creation process was to align the different operations with their counterparts. There are essentially two different approaches when designing the creation process for a feature of this type. The first one is to simply extend the existing UI with all the necessary components, resulting in a new element for each action. Once the recommendation system supports more entities and actions this would result in a very cluttered UI. The second approach is to introduce a new mode in which the familiar actions are executed as recommendations. We took the second approach.

So, to create any kind of recommendation the user has to first activate the recommendation mode. Figure 5.5 shows the button to toggle the recommendation mode, which is located at the top right corner of the coding editor.

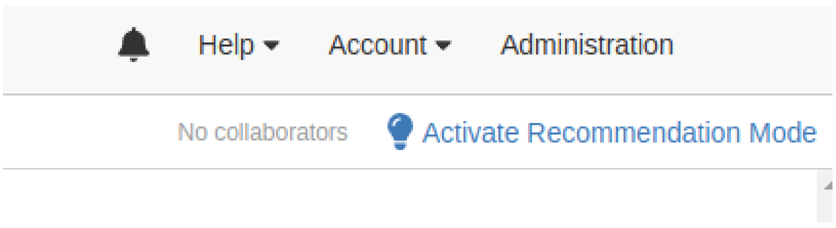
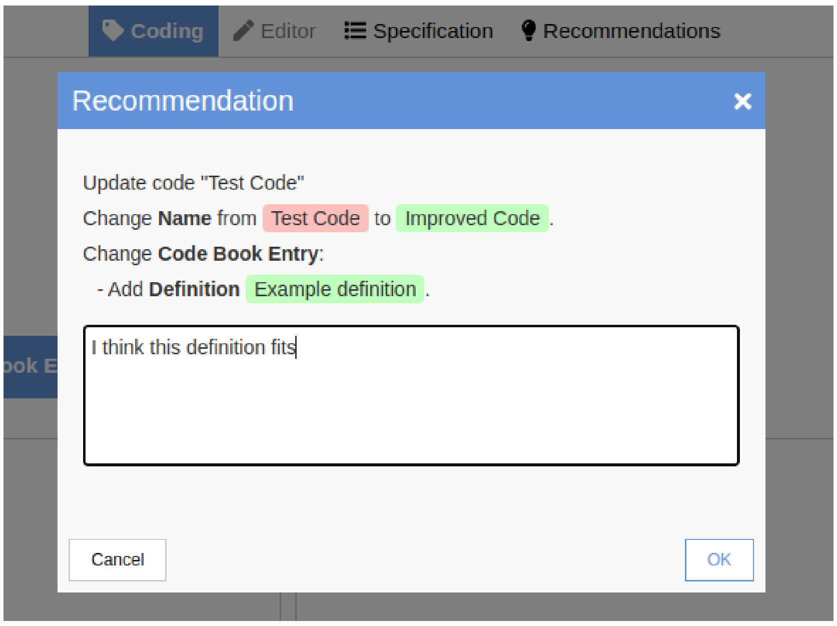


Figure 5.5: Recommendation mode button

When the recommendation mode is activated, the UI is slightly altered, hiding components that are not relevant to the recommendation workflow. An example of this is the documents' toolbar.

The initial step mirrors the procedure of its counterpart. The *AddCode*, *UpdateCode*, and *RemoveCode* recommendations are crafted using the (recommendation mode) codesystem toolbar. Meanwhile, a *RelocateCode* recommendation is created via drag and drop. In normal mode, the process ends at this point and the action is executed. In recommendation mode, the second step opens a modal that displays the resulting action and a comment field to add a rationale. Figure 5.6 shows the final modal for an *UpdateCode* recommendation.



Recommendation

Update code "Test Code"
Change **Name** from Test Code to Improved Code .
Change **Code Book Entry**:
- Add **Definition** Example definition .

I think this definition fits

Figure 5.6: UI of the creation modal

5.2.2 Review Flow

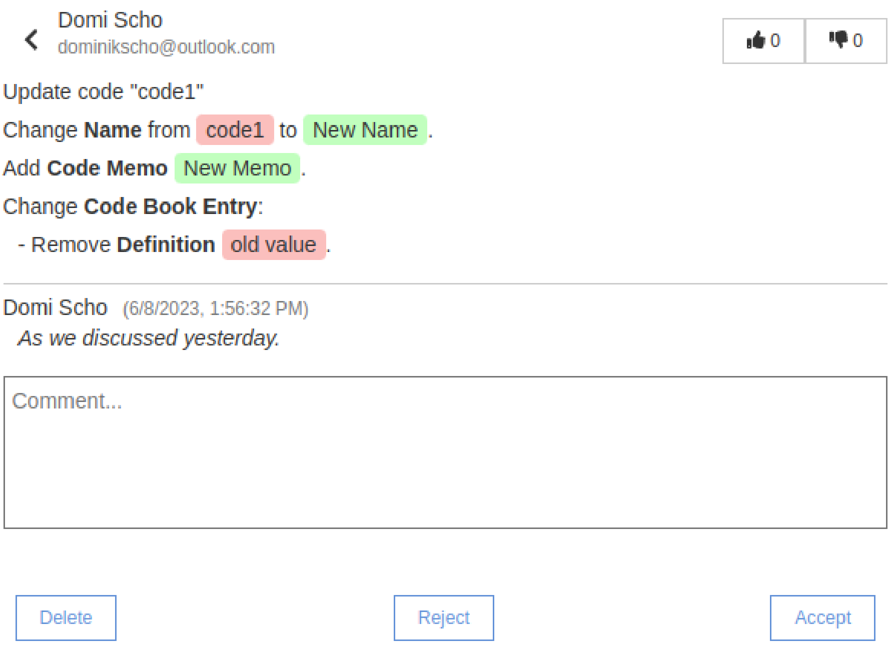
The review of a single recommendation is the cornerstone of the entire workflow. It has to provide all relevant information and encompass every interaction associated with it. This is encapsulated within the `RecommendationReview` component, which is showcased in Figure 5.7.

Voting mechansim

The `VotingController` component displays existing votes and enables users to vote. When a user votes, the associated button is highlighted and the number of votes is increased by one. In adherence to the UI principle of providing undo operations, a subsequent click retracts the vote (Nielsen, 1995). Currently, both buttons only display the vote count. An option for future enhancement could involve displaying the individual users who have voted. However, because of its low priority, this has not been implemented yet.

Comments Section

The comments section consists of two main components: `CommentInput` and `Comment`. In addition to presenting the content, the `Comment` component provides various related features. To mitigate cognitive load, icons are only


Domi Scho
dominikscho@outlook.com

Update code "code1"
Change **Name** from code1 to New Name.
Add **Code Memo** New Memo.
Change **Code Book Entry**:
- Remove **Definition** old value.

Domi Scho (6/8/2023, 1:56:32 PM)
As we discussed yesterday.

Figure 5.7: UI of the review component

displayed when the user hovers over the comment area. If there is no difference between the original comment and the input, saving does not trigger an `updateComment` call, which avoids unnecessary API requests.

There are two approaches to deleting comments: either the comment is entirely removed, or it is marked as deleted. The latter approach is implemented to prevent misunderstandings, especially when comments refer to other comments that have been deleted. Figure 5.8 shows a deleted comment.

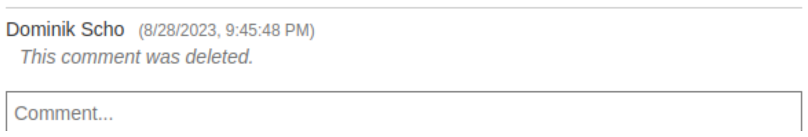

Dominik Scho (8/28/2023, 9:45:48 PM)
This comment was deleted.

Figure 5.8: UI of a deleted comment

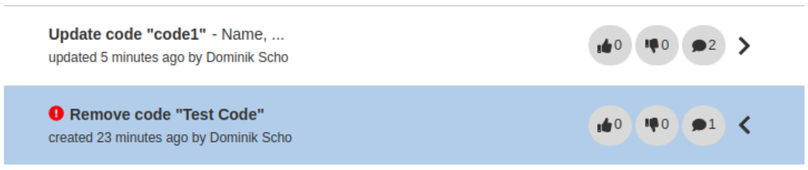
Recommendation Action

The `RecommendationAction` serves as the central component of any recommendation, defining the actual suggested action. Unlike the preceding elements, this aspect varies based on the action type.

The first line (`ActionStatement`) articulates the operation to be executed. For *RelocateCode* and *RemoveCode* actions, this sufficiently represents the action. In the case of *UpdateCode* operations, the section below is utilized to give a detailed overview of the specific attributes that are being changed. To accurately assess the value of the proposed changes, users need to be able to identify both the proposed and current values. This is visually highlighted in the form of a red background for the current values and a green background for the new values.

5.2.3 Review Editor and Modals

The previous subsection discussed the review of a single recommendation. However, typically, there are multiple recommendations to consider, necessitating the use of a list component. Figure 5.9 displays two items (`RecommendationOverview`) from the `RecommendationList` component. The left side presents details about the action, author, and creation date. For UpdateCode recommendations, only the first attribute is shown. Moreover, it indicates the count of upvotes, downvotes, and comments.


Update code "code1" - Name, ...
updated 5 minutes ago by Dominik Scho
0 0 2

Remove code "Test Code"
created 23 minutes ago by Dominik Scho
0 0 1

Figure 5.9: UI of the recommendation list.

By combining both components, the review flow is almost complete. In fact, from the user's perspective, the modal version consists primarily of these two components. To ensure consistent modal behavior, these components are incorporated into generic modal components.

Unlike the modal version, the editor functions as a central hub for all types of recommendations. Furthermore, it offers distinct tabs for open, accepted, and reject recommendations (Figure 5.10). The `RecommendationReview` component for resolved recommendations closely resembles the one described in the previous subsection. However, votes are disabled and the toolbar is concealed, leaving only the comment section functional.

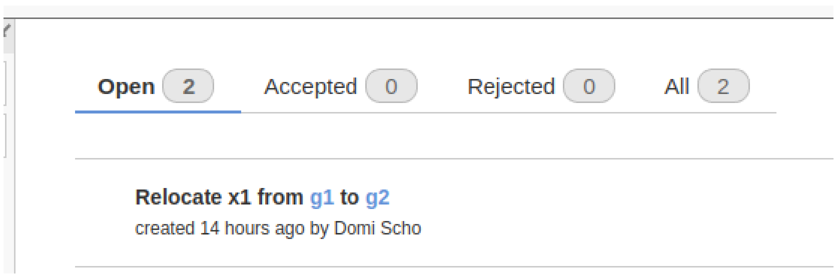


Figure 5.10: UI of the review editor tabs.

5.2.4 Conflict Handling

Modifying the codesystem state can result in varying impacts on the validity of an action. The `ActionStatus` constants are employed to encompass these different scenarios. A value of `VALID` signifies that none of the values related to the action changed. An action that remains applicable but includes outdated reference data is labeled `STALE_VALID`. When an action becomes irrecoverably invalid, it is assigned the status `TERMINAL_INVALID`. This typically arises when the target code has been removed or, in the case of a *RelocateCode* action, when the new parent code has been removed. Consequently, the only way to resolve such recommendations is to either reject or delete them. Figure 5.11 shows an example of a `TERMINAL_INVALID` action.

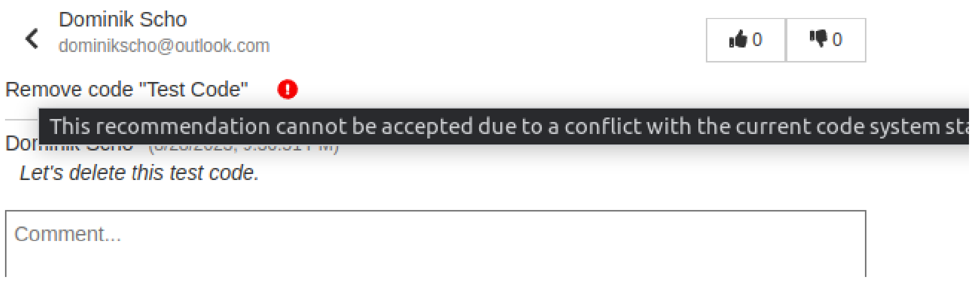
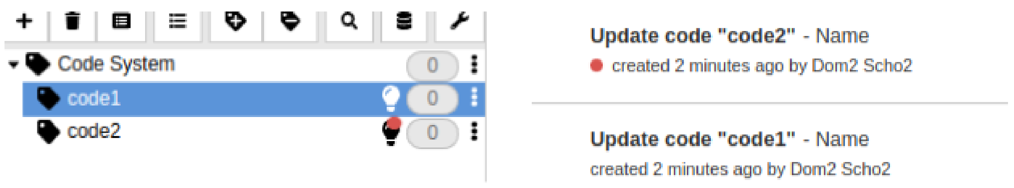

Dominik Scho
dominikscho@outlook.com
0
0
Remove code "Test Code"
This recommendation cannot be accepted due to a conflict with the current code system sta
Dor
Let's delete this test code.
Comment...

Figure 5.11: UI of a recommendation with a terminal invalid action.

The evaluation of the `ActionStatus` occurs entirely on the client side. The function `setActionStatus(recommendation, codes)` is used to set the status field of the action based on the state of the codesystem. This function is invoked within the state update function `setCodes(codes)` located in `CodingEditor`. Hence, whenever the codes are updated, the status of each action is reevaluated.

5.2.5 Notifications

The first instance of what can be considered a notification is the appearance of a light bulb when a recommendation is created. However, this is inadequate for two reasons. The light bulb solely indicates the presence of recommendations but does not specify whether they are new. Moreover, it fails to provide any indications of updates to recommendations, such as new votes or comments. In essence, the role of a notification feature is to track unseen changes. In this context, changes encompass both updates as well as new recommendations.



Update code "code2" - Name
● created 2 minutes ago by Dom2 Scho2

Update code "code1" - Name
created 2 minutes ago by Dom2 Scho2

Figure 5.12: Red dot on code (left) and recommendation level (right).

Visually, the notification takes the form of a red dot (Figure 5.12). The red dot can appear in two locations. On the code level (left), the dot indicates whether any of the recommendations associated with that specific code contain changes. The dot on the right is integrated into the recommendation overview and is associated with the recommendation itself. As anticipated for unseen changes, the red dot vanishes once the user accesses the corresponding review.

At a technical level, two scenarios have to be addressed the achieve this behavior. Firstly, when a user enters the coding editor, it is necessary to manage recommendations that have been updated since their last visit. The paragraph *Notifications* in subsection 5.1.1 describes how this is handled on the backend. Secondly, changes that occur while the user is actively using the coding editor must be handled. For this, the event handlers described in section 5.3 are utilized. Depending on whether it involves a new recommendation or an update to an existing one, either the `authorId` or `lastUpdate.userId` is used to determine whether the event was triggered by a different user, and thus needs to be marked as unseen.

5.3 RTCS

As already described in section 1.2, the RTCS manages the synchronization of various entities, but most importantly, codes and codings. To equip the recommendation system with similar functionality, the RTCS must be extended accordingly. This applies to all write requests that modify data observable by other users. In the context of the recommendation system, this encompasses all endpoints except `listRecommendations` and `setSeen`. While the former constitutes a read request, the latter solely modifies user-specific data.

Fortunately, the system is well designed in this aspect, which facilitated a seamless integration. At its core, the RTCS maps API calls to events and vice versa. Therefore, any extension requires defining new events and their associated handlers. The remaining section follows the data flow shown in Figure 5.13.

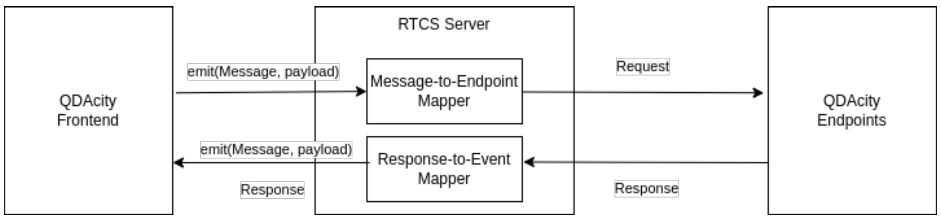


Figure 5.13: Data flow of a RTCS request.

As part of the `SyncService`, the main class of the RTCS client, the `RecommendationService` class exposes the new interface. The methods within this class simply dispatch the associated messages (along with their payload) to the RTCS server. The term *message* is used to refer to events sent from the client to the server, while events sent from the server to the client are simply referred to as *events*. There exists a direct correspondence between *messages* and endpoints, which has led to the inclusion of the newly added messages shown in Table 5.1 (left). On the server side, the task is to map incoming messages to endpoint requests and responses to events. To accomplish this, the events shown in Table 5.1 (right) were introduced.

Finally, the `SyncService` forwards the events to all connected users, prompting the execution of the associated handlers on the client. These handlers predominantly just update the state containing recommendations and codes. Updates that only affect the recommendation itself are handled by the `recommendationUpdated` handler. Since all of these updates are handled the same way, a single event (`updated`) suffices. This is different for the remaining events, since recommendations are never isolated entities and

Table 5.1: Extended RTCS messages (left) and events (right).

Messages	Events
`insert`	`inserted`
`accept`	`accepted`
`reject`	`rejected`
`delete`	`deleted`
`vote`	`updated`
`retractVote`	(`updated`)
`addComment`	(`updated`)
`updateComment`	(`updated`)
`deleteComment`	(`updated`)

are always associated with other objects, there is a close relationship to the handling of code-related events. This relationship is particularly apparent in the `recommendationAccepted` handler. For the *RelocateCode*, *UpdateCode*, and *RemoveCode* operations, the corresponding code-handler is invoked. The *AddCode* operation is an exception, as the code was already included during the creation process. In this case, `codeUpdated` is invoked with the `draft` attribute set to false. The fact that the recommended code is already part of the codesystem without being accepted has additional implications for handling `recommendationDeleted` and `recommendationRejected`. In both cases, `codeRemoved` is called to remove the code draft again. The relationships between the handlers are visualized in Figure 5.14.

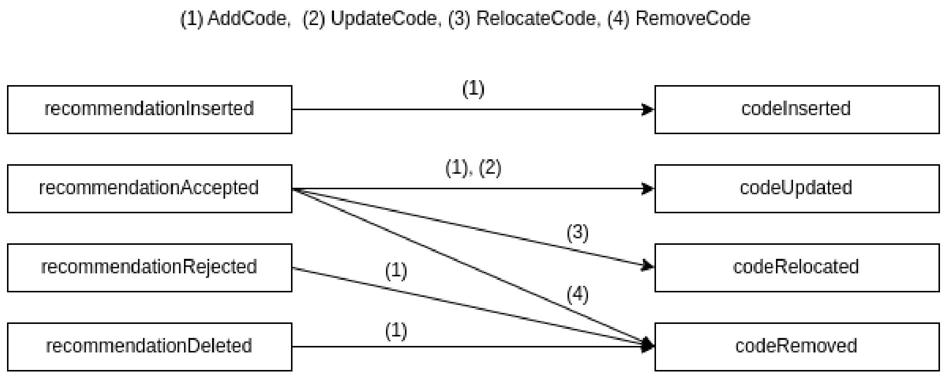


Figure 5.14: Relationships between recommendation and code handlers.

6 Evaluation

Chapter 6 offers a thorough evaluation of the recommendation system. The first section (6.1) presents the usability test conducted to evaluate the system's usability. Sections 6.2 and 6.3 then compare the final implementation state against the functional and quality requirements defined in chapter 3.

6.1 Usability Test

An essential component of evaluating the system's quality involves obtaining feedback from users. To accomplish this, a usability test was conducted. Although the primary focus of a usability test is the system's usability, it is also possible the gain insights into other quality aspects (e.g. functional suitability). The initial portion of this section explains the test design, while the subsequent portion presents the test results.

6.1.1 Design

The method used in this design is based on heuristic evaluation (Nielsen, 1995). Within the scope of this project, heuristic evaluation offers multiple advantages over traditional usability testing. Regarded as a "discount usability engineering" method, it proves to be more cost-effective than the conventional approach. Not only does it require a smaller number of evaluators, but usability experts are also usually very expensive. Nielsen recommends 3-5 evaluators for a heuristic evaluation. In this evaluation, we engage three evaluators, which meets the suggested minimal number. Since evaluators tend to uncover different usability problems, a relatively small number suffices to identify the most important issues. Furthermore, this method accommodates the presence of an observer who can assist during the test, making it well-suited for domain-specific applications. QDAcity undeniably falls into this category. Lastly, the evaluators are interviewed individually, which significantly reduces the coordination efforts.

Evaluators

An important aspect when recruiting evaluators is their area of expertise. The decision to concentrate on work-domain experts is based on several factors. Users who lack familiarity with a tool like the coding editor would find it challenging to concentrate exclusively on the facets of the recommendation feature. Additionally, Følstad (2007) suggests that work-domain experts tend to focus on issues that have a higher impact. Consequently, the criterion for selecting potential experts was as follows: they should possess substantial knowledge of QDA and some familiarity with QDA software. Beyond their commonalities, achieving a certain level of diversity was also important. Expert 1 is a frequent QDAcity user, with no further involvement in the project. Expert 2 exhibits some familiarity with QDAcity but primarily uses MAXQDA. Additionally, they have a strong software engineering background. Expert 3 serves as the team lead and a key developer at QDAcity. With these profiles in mind, the goal was to gather versatile feedback.

Procedure

As all evaluators were familiar with QDAcity, there was no need for a general introduction to the software. To ensure consistent results, each interview took place within a distinct but identical demo project. The project's problem domain was a door communication system, which was already set up as part of previous research. Besides taking the observer role, I served as a collaborator. This approach aimed to make the interview more realistic, as the recommendation system is inherently collaborative in nature. Moreover, it allowed for the testing of the notification feature.

Given that there are only two core workflows (creation and review), the objective was to cover all variations within those two. The evaluators were encouraged to provide commentary on the following questions throughout the test:

- What did you expect?
- Were you surprised?
- Is there anything missing?
- What would you improve?

At the beginning of the session, the evaluator was asked to enter the coding editor and gain an overview of all the codes, codings, and documents involved. During this phase, the recommendation service was disabled, making it no different from a standard project. Following a period of familiarizing themselves with the content, the evaluator was directed to re-enter the coding editor. However, this time, the recommendation service was enabled. Given that the test project was initially configured accordingly, the prepared recommendations became visible. As part of the initial task, the evaluator assumed the role of a reviewer, with the objective of assessing the newly introduced recommendations. For the subsequent task, the evaluator was tasked with creating their own recommendations. In cases where an evaluator only produced certain types (e.g. add or update), they were instructed to repeat the process for the remaining types.

After completing both tasks, the interview continues with two open questions. The first question pertains to whether the notification feature adequately addresses all the requirements of a collaborative workflow. To gain a deeper understanding of the different use cases of a recommendation feature, the second question explores the concept of batch recommendations.

The session proceeds with two concise questionnaires. The first consists of three questions, which aim to capture the impact of this feature on their collaborative workflow. The second questionnaire employs the System Usability Scale (SUS) (Brooke, 1995). a general-purpose usability assessment tool.

Finally, the evaluators are given the opportunity to share any additional thoughts. This serves to address any open questions and allows for feedback within a broader context.

Data Collection

The data collection process involved taking handwritten notes. Upon completion of the interviews, these notes were transferred to a basic text editor for additional cleaning and structuring. Furthermore, in some instances, the first evaluator utilized the comment section to provide feedback. Any cryptic or shorthand notes were augmented with contextual information while the details were still vivid in memory.

Data Analysis

After the collection phase, the interview notes were organized according to the individual evaluators. The next step was to restructure the notes based on topics and to eliminate redundant information.

6.1.2 Results

This section presents a comprehensive overview of the results. The complete set of notes and the questionnaires can be found in the appendix.

General Feedback

During the course of the interviews, several statements were made that do not fit into either the review or the creation flow. These statements mostly pertained to the feature as a whole.

First and foremost, it was noted that there should be a tutorial or help function to introduce this feature.

As part of the review flow, the evaluators experienced the process with different permission levels. Two evaluators perceived the permission structure as slightly counter-intuitive, questioning why editors cannot accept recommendations while they have the ability to make direct changes to the codesystem.

One evaluator proposed the idea of recommendations without specific actions (e.g. "Split this code"). Such a feature would resemble the general comment feature offered by Google Docs (2.2.1).

One idea, in a broader context, was aimed at the entire codesystem and not limited to recommendations alone. In general, users should be informed about codesystem changes, for example, "SomeUser changed 'group' code to 'Concepts'".

Review Flow

Several interesting observations could be drawn regarding the review workflow in general. Initially, all evaluators opted for the modals to review the existing recommendations. It was only later, and in two cases with additional guidance, that they noticed the dedicated editor. It was stated, that this was primarily attributed to the visual prominence of the red dots within the codesystem view. During the first encounter, the dots were interpreted as some kind of "to-do" action. Despite this, the editor was universally perceived as an essential feature. Indeed, one evaluator expressed confidence that a large portion of their review process would be performed using the editor. They continued by expressing a desire for an option to hide all recommendations from the codesystem view. They mentioned having distinct workflows and were concerned that the mere presence of recommendations could introduce bias into their coding process.

Since the review component is the central component automatically generated or manually typed. However, after reviewing several other recommendations, became convinced that it must be auto-generated. One significant criticism was that it is difficult to quickly identify the values being changed. Evaluating a change requires understanding the values that are being modified, which is nearly as important as the new values. To access these values, one must first search for the code and then open its properties. This issue was highlighted by all three evaluators. Two improvements were suggested to address this issue. First, the selection of a recommendation should simultaneously select the corresponding target code. This would eliminate the need for manually searching. Second, the recommendation should display the changes in the form of a diff or delta, similar to how changes are displayed in MRs on GitLab. The voting mechanism was deemed essential, as comments alone often do not make it clear what the current state of approval is. It was also understood that a vote serves as a form of communication and does not lead to any further actions (such as applying the action). One critique was that there was no way to retract a vote once cast. The comments section was generally considered sufficient for this context. One interesting idea was to add the capability to enable users to tag or link various entities, like users or codes (`"Hi @someUser, check out #someCode"`). The tagging of users could then be integrated into the notification feature to allow for targeted communication. Two further comments were provided regarding the styling of the accept and reject buttons. It was suggested that, in case the user has the permission, these buttons should be colored green and red respectively. In case the user lacks the permission, the tooltips appear too slow. This was followed by the suggestion to display such messages directly next to the button, rather than only on hover.

When exploring the editor, it was pointed out that the split screen is helpful as it eliminates the need to switch between views and having to reassess where to proceed next. One evaluator suggested several improvements for the editor, particularly if more entities are supported. Firstly, they proposed standard sorting and filtering options (author, date, type, etc.). Additionally, they put forward the idea of assigning labels (e.g. priorities) to individual recommendations.

Creation Flow

It appears that, without further explanation, the meaning of the recommendation mode depends on the familiarity with other tools, such as Google Docs. Two out of three evaluators immediately knew what to expect from this mode due to their prior experience with Google Docs' Suggestion Mode. That is, after activating the recommendation mode, all changes are interpreted as recommendations. Initially, the evaluator lacking this familiarity struggled to form a concrete idea of what to expect.

Continuing the exploration of the recommendation mode, it was unexpected for two evaluators to realize that document-related actions still functioned as direct changes to the project rather than as recommendations. Apart from the changed codesystem toolbar, there were no additional visual cues regarding whether a particular interactive element now operates as a recommendation or not. The expectation was that every visible element would be connected to the recommendation system. The lack of visual distinctiveness was also pointed out concerning the mode as a whole, indicating that it might not be immediately apparent in which mode the user is currently operating. The button text "Deactivate recommendation mode" was not perceived as visually prominent enough.

After the initial exploration phase, the evaluators were tasked to create their own code recommendations. The alignment of the creation of code recommendations with the corresponding code actions was perceived as pleasing. Because of this, every creation process was self-explanatory. In the beginning, one evaluator, was uncertain if it was possible to add multiple attributes to an UpdateCode recommendation. Nonetheless, they proceeded with confidence, believing it to be feasible. Overall, the process did not reveal any issues.

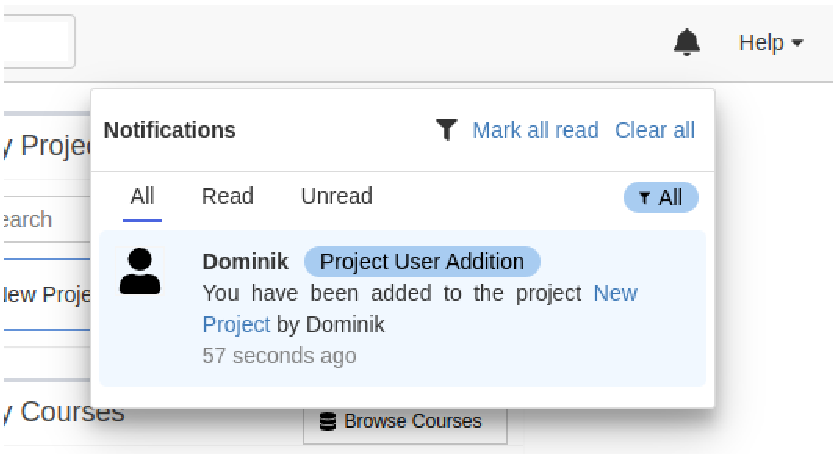
Regarding the creation modal, the action overview was viewed as helpful because it provides the user the opportunity to review the recommendation before its creation. One suggested improvement was to add an explanatory text to the comment field that encourages users to add their reasoning behind the recommendation. The concern was that users might frequently leave the comment field empty.

For all code recommendations except *AddCode*, the creation process is completed. The behavior of the newly created code draft matched the expectations of the evaluators.

Open Question 1 - Notification Feature

The notification feature was universally perceived as satisfactory for the coding editor. However, it was highlighted that the current version only aids in identifying updates of recommendations that are still open. Once a recommendation is either accepted or rejected, it simply disappears from the codesystem. There should be some form of notification to indicate that a recommendation has been resolved. Especially, if this results in a changed codesystem state.

Moving beyond the boundaries of the coding editor, the idea of integrating the recommendation-notification feature into QDAcity's global-notification mechanism (Figure 6.1) was described. As the number of projects a user is involved with increases, the signficance of global notifications becomes more prevalent.



Dominik Project User Addition
You have been added to the project New Project by Dominik
57 seconds ago

Figure 6.1: Global notification feature

Open Question 2 - Batch Recommendations

First, the concept of batch recommendations was described solely as the ability to bundle multiple recommendations into a set (batch). Consequently, the recommendations contained in a batch can only be accepted or rejected together. However, relying on this description, two evaluators required a simple example to comprehend the concept.

The value of such a feature was primarily recognized in more complex scenarios such as:

- Relocating multiple codes to a new parent
- Updating the same attributes of multiple codes
- Replacing a code by deleting an existing and adding new one
- Adding multiple codings at once (in case coding recommendations are supported)

  - for multiple codes for a new document
  - for single code for multiple documents

However, the frequency of such cases was estimated to be relatively low. Therefore, batch recommendations were not regarded as a high-priority feature.

Questionnaire 1 - Custom

The first questionnaire focuses on assessing the impact of the recommendation feature (B.1). It consists of three custom questions, which means that no reference values exist. Even though a statistical analysis is not applicable, the answers can still be used to gain insights. The scores suggest that the recommendation feature helps to reduce the communication overhead (Q1) and make the (collaborative) workflow more efficient (Q3). Based on the scores and the oral feedback for Q2, the current implementation is not capable of completely replacing the current communication channels. It was stated that a general-purpose communication channel is almost always needed. However, in cases where a certain change can be expressed via a recommendation, communication will shift to the recommendation feature.

Questionnaire 2 - SUS

The total (average) score of the SUS (B.2) is 79.2. When comparing the individual scores, the two evaluators who are not part of QDAcity average 83.75. In contrast, the internal evaluator gave a significantly lower score of 70. It is important to emphasize that, statically, an average of three participants, implies a high level of uncertainty. Especially, since there is already a large deviation between these three. Despite this limitation, it is important to understand what a score of 79.2 means. Bangor et al. (2009) conducted extensive research on the interpretation of individual SUS scores. As a result of their research, they propose three distinct scales: grade scale, acceptability ranges, and adjective ratings.

According to these scales, the value 79.2 can be categorized as follows:

- Grade scale: Assignment of *C*, but very close to *B*.
- Acceptability ranges: Assignment of *ACCEPTABLE*
- Adjective ratings: Assignment between *GOOD* and *EXCELLENT*

The authors delve into more detail regarding the differences among these methods and the question of which might be more suitable than the others.

However, such an in-depth discussion is beyond the scope of this thesis.

Based on the implementation (chapter 5) and the test results, the remaining chapter is concerned with the evaluation of the requirements.

6.2 Functional Requirements

The recommendation feature is very user-centric by its nature. It is, therefore, reasonable to evaluate the functional requirements mostly from the user's perspective.

6.2.1 General Functionality

The project settings now include an option to enable the recommendation feature (Figure 6.2). A section for services was added to separate it from the other self-contained editors. As with the rest of the project settings, it is possible to modify these settings throughout the project's life cycle. In the case of the recommendation service, this simply means that the codesystem is loaded without code drafts, and recommendations are not loaded at all. Thus, it is possible to disable the feature temporarily, without losing any of the data.

FR1 is satisfied.

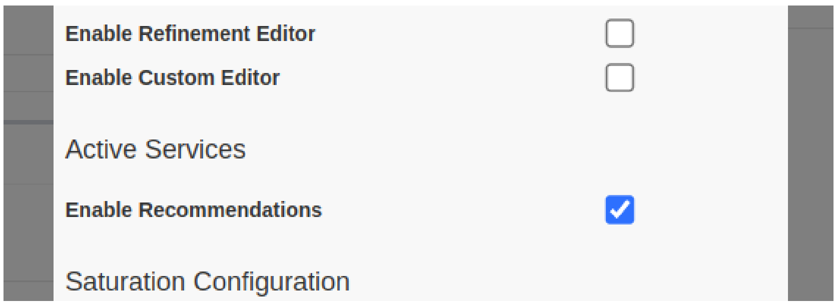

Enable Refinement Editor ☐
Enable Custom Editor ☐
Active Services
Enable Recommendations ✓
Saturation Configuration

Figure 6.2: Project settings modal

Nevertheless, the recommendation feature comes with a dedicated editor for reviews. The functionality, as well as the UI of the editor, were discussed in subsection 5.2.3.

FR3 is satisfied.

The review process is nearly identical, whether the user opts for the editor or the modal. Subsection 5.2.2 provided more details about the review process.

Figure 5.7, specifically, shows the review component, encompassing elements for resolving, voting, and commenting. The necessary backend functionality was discussed in subsection 5.1.1.

FR4 is satisfied.

FR5 is satisfied.

FR6 is satisfied.

The notification feature is responsible for informing the user about unseen changes. The elements comprising this feature were elaborated upon in subsections 5.2.5, 5.3, and 5.1.1.

FR7 is satisfied.

As highlighted in subsection 5.2.1, users are required to enter the recommendation mode to initiate the creation process. The corresponding button is depicted in figure 5.5.

FR2 is satisfied.

6.2.2 Code-related Functionality

In contrast to other code-related actions, *AddCode* requires special treatment, as it alters the code system before the recommendation is accepeted. The creation and deletion of the code draft is handled by the `setupDependencies()` and `removeDependencies()` methods. Both are enforced by the `DependentActionController` interface (subsection 5.1.2). From the backend perspective, the code draft is fully functional but is restricted by the frontend to prevent undesired actions.

FR8 (FR8.1, FR8.2) are satisfied.

The actions *UpdateCode*, *RelocateCode*, and *RemoveCode* are straightforward as they do not cause any modifications to the code system beforehand. Because of this, their corresponding action controllers only need to implement the `ActionController` interface, which contains a single method: `apply` (see next paragraph). Subsection 5.2.1 provided a detailed description of the creation process for each of them.

FR9 is satisfied.

FR10 is satisfied.

FR11 is satisfied.

Upon acceptance of a recommendation, the `apply` method of the responsible action controller is invoked. This delegates the actual work to the `CodeController` methods, which then apply the changes to the codesystem. Details regarding the synchronization are outlined in section 5.3.

FR12 is satisfied.

6.3 Quality Requirements

Given that the recommendation feature is part of a larger system, the quality requirements are evaluated with respect to the quality features of QDAcity.

Functional Suitability

Any change that would break other parts of the system is expected to be identified through existing (1) unit tests, (2) acceptance tests, or (3) manual tests. While each of these tests might not provide full coverage individually, their collective usage is sufficient to cover the core workflows. Even though the recommendation system did temporarily suffer from some bugs, the overall system remained unaffected.

QR1 is satisfied.

The evaluation of QR2 encompasses all the sub-features outlined in previous sections, and the usability tests did not indicate the absence of any significant features. Therefore, the workflow can be called end-to-end.

QR2 is satisfied.

The following example tries to illustrate the impact of QR2 on decision-making. During a later phase of the development process, the question emerged, of whether to allocate resources to coding recommendations or the notification feature. Given that the notification feature was deemed essential for an end-to-end workflow, it was prioritized over coding recommendations.

Performance Efficiency

Section 5.3 describes the integration of the recommendation system into the RTCS. By utilizing the same infrastructure, the recommendation system is expected to exhibit similar performance characteristics. Although no explicit measurements were conducted, there were no noticeable performance differences observed in terms of synchronization. However, it is worth noting that upon entering the coding editor, the initial loading time was slightly longer for recommendations than for codes. As a result, there were instances where code drafts appeared before the corresponding recommendations were loaded. Unfortunately, due to time constraints, this issue was not addressed. Consequently, QR3 is not fully satisfied.

QR3 is partially satisfied.

When entering the coding editor the `status` field of `RecommendationQuery` is set to `OPEN`, which loads all open recommendations for the current project. Only upon entering the review editor, the resolved recommendations are loaded. To ensure efficiency when dealing with a large number of recommendations, the implementation of a pagination mechanism would be necessary. In hindsight, the term "on-demand" lacks specificity, resulting in a conservative evaluation of QR4.

QR4 is partially satisfied.

Compatibility

Due to the direct integration of the recommendation system into both the frontend and backend, general compatibility is ensured. No additional services were introduced.

QR5 is satisfied.

Furthermore, neither any frontend nor backend code deviates from the roadmap. All react components have been implemented using functional components and hooks, and numerous related components have also been refactored accordingly. On the backend side, the project initially adopted a JDO-based class structure but was subsequently migrated to Objectify.

QR6 is satisfied.

Usability

At QDAcity, the localization strategy was ensured as part of their code reviews. Indeed, all newly added user-facing strings have corresponding German translations.

QR7 is satisfied.

Generally, design is much harder to evaluate than other qualities. In the case of QDAcity, it is even more challenging, given that the software is currently undergoing a redesign phase. Nonetheless, based on stakeholder feedback it is possible to make statements with some degree of certainty. From a user standpoint, none of the evaluators highlighted any inconsistencies with the existing design. Internally, most small adjustments were addressed in follow-ups as part of the iterative development cycle. By the end of the development phase, there were no significant change requests related to the design. Both of these factors point to a high level of consistency.

QR8 is satisfied.

Security

All ten endpoints are implemented following the context pattern described in section section 4.2. Consequently, authentication takes place during the initialization of the `UserService`. This implies that the actual request handling occurs thereafter.

QR9 is satisfied.

In light of the architecture refactoring a more explicit `UnauthenticatedException` was introduced. Prior to this, any authentication failure would throw an `UnauthorizedException` as part of the authorization process. The endpoint tests continue to be affected by this lack of differentiation. Numerous tests appear to be testing authorization when in reality, they are only testing authentication. A systematic solution to address this problem is described in the *Maintainability* paragraph.

Authorization, on the other hand, must be checked individually for each endpoint. Each of the ten endpoints evaluates user permissions according to the structure defined in Table 3.1. To accomplish this, the `ProjectPermissions` enum class was extended with the values shown in Table 6.1. Furthermore, the `Authorization.check()` method was overloaded to include the necessary types: `Recommendation`, and `RecommendationQuery`.

Table 6.1: Added permissions to the `ProjectPermissions` enum class.

RECOMMENDATION_SERVICE_TOGGLE
RECOMMENDATION_RESOLVE
RECOMMENDATION_CUD

The overall system does not differentiate between different project settings but does a general check against `ProjectPermissions.SETTINGS_UPDATE`. Therefore, `RECOMMENDATION_SERVICE_TOGGLE` is currently not used. The end result still adheres to the requirements. The `RESOLVE` permission covers accepting and rejecting recommendations, and `CUD` is an abbreviation for create, update, and delete.

QR10.1 is satisfied.

QR10.1 is satisfied.

⇒ QR10 is satisfied.

Maintainability

Despite the primary focus being on code-related recommendations, the system is designed to be extensible to other types of recommendations. The `RecommendationController` encapsulates all essential functionality irrespective of the type. Introducing a new entity type involves the following steps:

1. Add a new `RecommendationType` enum value.
2. If needed, add a new `RecommendationActionType` enum value.
3. Implement a `RecommendationActionController` for each new action-entity combination.

All of these entail extensions rather than modifications.

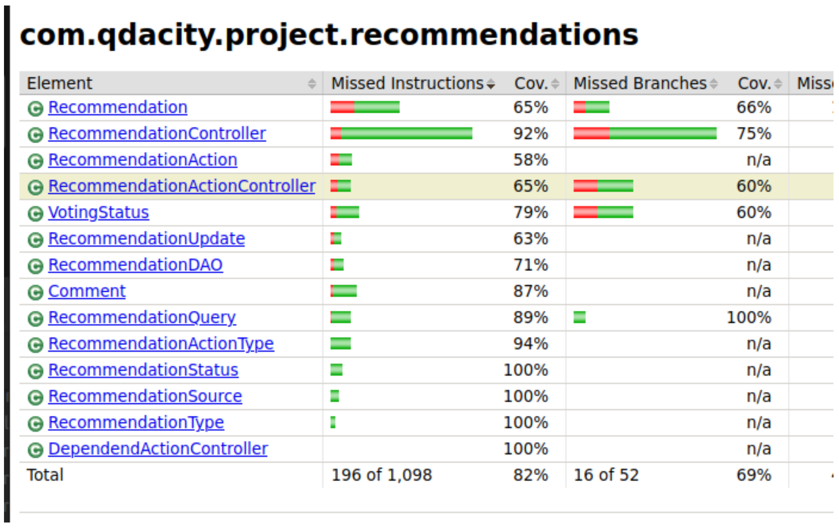
QR11 is satisfied.

To address the ambiguity issue discussed in the *Security* paragraph, a new class convention was introduced. Prior to this thesis, functional, authorization, and authentication tests were consolidated within a single test class, typically named `SomeEndpointTest`. The revised convention involves splitting the class into three separate classes:
`SomeEndpointTest`,
`SomeEndpointAuthorizationTest`,
`SomeEndpointAuthenticationTest`.
This approach offers two main benefits with regard to the distinct concerns: (1) simplifying the identification of missing or ambiguous tests, and (2) simplifying common test setup code.

According to QR12, the recommendation system should include an endpoint unit-test suite that fulfills three criteria.

The requirement (QR12.1) for the unit-test suite was to cover at least 85% of the LoC of the recommendation system. In total, the `com.qdacity.project.recommendations` package has an LoC test coverage of 82% (6.3).

QR12.1 is not satisfied.



com.qdacity.project.recommendations

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Miss
Recommendation		65%		66%	
RecommendationController		92%		75%	
RecommendationAction		58%		n/a	
RecommendationActionController		65%		60%	
VotingStatus		79%		60%	
RecommendationUpdate		63%		n/a	
RecommendationDAO		71%		n/a	
Comment		87%		n/a	
RecommendationQuery		89%		100%	
RecommendationActionType		94%		n/a	
RecommendationStatus		100%		n/a	
RecommendationSource		100%		n/a	
RecommendationType		100%		n/a	
DependendActionController		100%		n/a	
Total	196 of 1,098	82%	16 of 52	69%	

Figure 6.3: Unittest coverage of the recommendation system.

After the refactoring, the `RecommendationEndpointTest` is exclusively comprised of functional tests. Each endpoint method is tested at least once.

QR12.2 is satisfied.

Both the authentication and the authorization test classes contain ten tests, with one dedicated to each endpoint.

QR12.3 is satisfied.

⇒ QR12 is partially satisfied.

In addition to the endpoint tests, the recommendation system was intended to have acceptance tests. However, due to time constraints and other priorities, no such tests were added.

QR13 is not satisfied.

7 Discussion

Chapter 7 provides an in-depth discussion of the final outcome of the project. Section 7.1 analyzes the findings and limitations described in the previous chapters. Afterward, section 7.2 gives suggestions for future developments.

7.1 Findings and Limitations

Requirements

In retrospect, the requirements were formulated with excessive abstraction. Consequently, many were satisfied without having a production-ready feature In an agile methodology, requirements serve as continuous system documentation. However, their high level of abstraction offers limited insights into the concrete implementation. The ambiguity was not confined to the abstraction level, the terminology used was also not well defined. Instead of vaguely stating "...allow users to vote ...", requirements should specify the nature and purpose of a "vote".

Development Process

In the initial stages, the backend structure included an abstract `BaseRecommendation` class, which the `CodeRecommendation` class extended. This decision was driven by two factors. First, from a design perspective, recommendations without an identified target entity remain a mere abstract concept, thus justifying the need for specific classes for individual entity types. Second, during the early stages of familiarization with the code base, encountering foundational classes like `BaseProject` and `BaseCoding` was common. However, as the iterations progressed, the merits of adopting a composition-based model became apparent. Essentially, the act tied to a recommendation stands as its distinct notion. Although the advice to favor composition over inheritance is oft-repeated, its practical application was a valuable learning gleaned from this project.

The refined layered architecture pattern has already demonstrated its usefulness through multiple refactorings. Yet, without tangible extensions of the recommendation system, its extensibility remains speculative. While architectural designs often appear bulletproof on paper, they can falter when faced with the new requirements. To mitigate this risk, the current design makes almost no assumptions about the distinct entities' characteristics.

Usability Test

To better assess functional appropriateness and usability a heuristic evaluation was conducted. However, to make definitive statements the recommendation system needs to be in use for a longer period of time by actual users. This is especially true since the experts were aware of the fact that the usability test was part of a thesis. If it were facilitated by a company for a (paid) product, it is not clear if the test would yield the same results. Interestingly, the team at QDAcity is currently working on a user feedback system, which could be utilized to gather additional feedback.

Custom questionnaire

With only three questions, the custom questionnaire (B.1) offered valuable insights into the overall impact of the recommendation feature. However, upon reflection, Q2 was not optimally phrased. In section 2.2, we saw that Google Docs provides a general comment feature that is not tied to suggestions. In principle, such a general comment feature might be able to replace existing communication channels completely. Given that the current implementation only allows comments as part of a recommendation, it seems unrealistic to assume that it can be used for communication beyond change proposals. A more fitting question might have been: *Assuming that a specific change can be expressed through the recommendation feature, can this feature encapsulate all related communication?*

SUS

Section 6.1.2 highlighted the limitations of the SUS, stemming from high significant statistical uncertainty. Notably, this is not limited to the SUS. This generally applies to all forms of quantitative analysis that rely on a small number of data points. The primary intent behind using the SUS was to have an indicator for usability issues. Even with statistical uncertainty, a total sore below 50 would have signaled severe problems.

Improvements

The development phase and the feedback from the usability tests revealed several areas for improvement. The remaining section will discuss some elements that have been implemented. Afterward, section 7.2 presents a range of potential areas for future development.

While familiarizing themselves with the recommendation mode, two experts encountered difficulties in identifying which actions are considered recommendations. In response, both the document toolbar and the text editor are now hidden in recommendation mode. Given the ongoing redesign of the coding editor and the increasing number of task-specific editors, this issue is not limited to the recommendation mode.

Two experts highlighted that, when selecting a recommendation, the corresponding target code should be simultaneously selected. This becomes increasingly crucial, for codesystems with a complex hierarchy. Without this functionality, a target code located within a collapsed subtree remains entirely concealed. Although the current solution exclusively addresses codes, the concept of selecting the target object is equally important when incorporating new entities.

Previously, repeated up- or downvotes were simply ignored. Consequently, once a recommendation receives a vote, it can only be changed, but never be retracted completely. One of the experts characterized this as an unexpected behavior. Subsequently, this issue was addressed by adding the feature to a retract vote, which is initiated by clicking the already selected vote for a second time.

7.2 Future Work

Based on the experts' feedback, the recommendation of codes already provides a significant improvement to the workflow of collaborative QDA. Nevertheless, there are various ways in which the system can be further improved. The following section gives a short overview of ideas that emerged over the course of the project.

Coding recommendations

Aside from codes, codings are the second most important entities in a codesystem. Especially, because they are instances of applied codes. Once the recommendation system supports codings, the complete codesystem development process can be realized in the form of recommendations. Unfortunately, due to time constraints, it was not feasible to extend this feature to codings. During the interviews, two experts already mentioned that they would expect this feature to be available.

Text recommendations

An important step that is often overlooked is the process of *data cleaning*. This step comes after *data gathering* but before *data analysis*. This can include formatting documents so that they can be better analyzed or simply correcting mistakes that were made during the gathering phase. From a coordination perspective, this process is very similiar to the analysis. Multiple people work together on the same set of documents and any changes should be properly communicated. An example of such a feature was already discussed in subsection 2.2.1 in the form of Google Docs.

Batch recommendations

In the current implementation, each recommendation is accepted or rejected individually. Batch recommendations would allow users to accept or reject multiple recommendations at once. This allows the grouping of recommendations that are related to each other and should only be accepted or rejected together. Most of the review process would remain the same. However, the question remains whether the items of the batch should resemble child recommendations or a set of actions. The former would involve a distinct comment section for each item.

System-to-user recommendations

Chapter 1 introduced the distinction between user-to-user and system-to-user recommendations. Ultimately, the project was scoped to concentrate on the former. Initially, however, the recommendation system was supposed to facilitate system-to-user recommendations created by a machine learning algorithm. Early on during development, the priority moved away from an experimental application of machine learning towards a production-ready feature for user-to-user recommendations. As a result, a much higher emphasis was put on user interface aspects. There were two primary reasons that led to this decision. Firstly, it was estimated that the latter would provide greater business value for QDAcity in the short to mid-term. Secondly, assuming the developed models produce satisfying results they can only be integrated into the system if the necessary backend and frontend infrastructure is already in place. The requirements of such a feature are very similar to the ones for a user-to-user recommendation system. When looking at the set of features that are described in the previous chapters, it becomes clear that all of them are also applicable in the context of system-to-user recommendations. Especially, since the review process of such recommendations continues to occur within a collaborative environment.

8 Conclusion

The objective of this thesis was to design and implement a user-to-user recommendation system for QDAcity. Because the goal was to enhance existing workflows with new capabilities, rather than developing an independent feature, we provided an introduction to QDAcity and related QDA concepts. To better understand the problem and the solution space, we looked at related work in the area of (user-to-user) recommendation systems. In particular, Google Docs and GitLab were discussed in more detail. By looking at existing solutions, it was possible to refine our understanding of how a recommendation feature for QDAcity can look like. Building on top, we outlined a set of requirements. Because the extension to other entities and actions was always part of the design, the functional requirements were further categorized into general and code-related functionality. For the quality requirements, a subset of the characteristics and sub-characteristics from the quality model were chosen. A consequence of being an integrated feature is that most quality requirements were defined in reference to QDAcity's general quality features. In particular, the emphasis on maintainability impacted the architectural design of the recommendation system. Additionally, the observation that the existing backend did not have an overarching approach to request handling, further shaped the design of the architecture. Hence, the goal was not only to develop an architecture for the new feature but not to introduce yet another approach. The solution was a refined layered architecture pattern, which centers around procedural-based controllers to separate different concerns. Furthermore, a context pattern was introduced to accommodate for the dependencies on the deprecated JDO interface. Following the controller approach, while simultaneously making it extensible, the action controllers were implemented using object-oriented polymorphism. Even though architecture is an important part of any system, the user should always be the focus. Therefore, the modeling of the life cycle of an individual recommendation was an important step before diving into the implementation. Based on the described architecture, the backend is primarily concerned with controller and data classes. More concretely, a controller for general recommendation functionality and controllers for the different code actions were implemented. To provide a better feeling for the recommendation workflow the frontend section focused more on the user perspective than on a detailed implementation in the form of react components. Even though the recommendation system is made up of many different components, the central piece is the review component. It is responsible for displaying and handling all the different actions that are available for a single recommendation. To enable synchronization, the implementation was concluded with the integration of the recommendation system into the RTCS. Being a user-centric feature, a usability test was a crucial instrument in evaluating the new feature. So, we first went into detail about the test design and outlined the results that came out of it. We then compared the implementation to the requirements defined in chapter 3. In summary, all 14 functional (sub-)requirements were satisfied. Of the 18 quality (sub-)requirements, 13 were satisfied, three were partially satisfied, and two were not satisfied. Due to time constraints, it was not possible to implement the required acceptance tests. The project journey, from the definition of the requirements to the implementation, and the final evaluation was put into perspective in form of a comprehensive discussion. This included dedicated sections for findings, limitations, and suggestions for future work. Since QDAcity already has plans to continue the development of the recommendation system, these sections are a valuable resource for future reference.

In conclusion, QDAcity now provides a new collaborative feature which allows users to incorporate recommendations into their workflow. The support of code recommendations already brings immediate value to its users. By supporting more entities the value is expected to increase significantly further. The outcome of this thesis adds a valuable unique selling point to QDAcity and increases long-term maintainability.

Appendices

A Usability Test Notes

A.1 Expert 1

Notes:

- Initially I though signifies real changes
- No inital unerstanding of what the recommenation mode does
- After I applied the change, I expect QDAcity to inform the recommender
- CodeBook suggestion are important; Memo not such much
- Instead of the tooltip display the text directly
- The light bulb should be displayed on the parent (if collapsed)
- I expected to undo my vote but it is not possible
- Add text above the comment to motivate reasoning; otherwise it stays empty
- Text on comment to motivate reasoning
- The permissions are unclear; why can't an editor accept recommendations
- I want to see the diff of the code
- Modal isn't correctly displayed on large screens

Before you were notified via the red dot in case of recommendation updates. Would you say this is sufficient or do you think additional cues would be helpful.

- Open recommendation within the codesystem the red dot is sufficient
- Accepting/rejecting should also send a notification

Currently a recommendation is bound to a single code/coding. What do you think of the idea of batch recommendations?

- I would say it is not the priority.
- But I can see the value for more complex tasks

- But it isn't a use case I immediately thought of

A.2 Expert 2

Notes:

- The light blub is too small; didn't see it
- I want to see the diff of the code
- I expect the recommendation mode to be similar to google docs.
- ReviewEditor with search, sorting, labels, priorities
- Code selected on recommendation click / review open
- I would expect to remove my vote
- Styling of resolve buttons: accept (green) / reject (red)
- Header of the actions should be larger and bold
- Action should display concrete ops: not only update but add, change, remove
- Initial thought was that it is ML-based recommendation-feature
- Hide all recommendations to focus on one task
- Whats in the code system should be usable (drafts)
- Idea to only allow one recommendation per code but immediately saw the use case for multiple
- Highlight/Color for the recommendation mode
- The toggle button is visual not enough to quickly see the mode your are in
- What about recommendations without actions e.g. "Split this code"

Before you were notified via the red dot in case of recommendation updates. Would you say this is sufficient or do you think additional cues would be helpful.

- Can't think of anything missing

Currently a recommendation is bound to a single code/coding. What do you think of the idea of batch recommendations?

- Not sure depends on the use case

- E.g. when coding a whole doucment
- E.g. when coding doucments with single code
- E.g. merging multiple codes
- E.g. relocating mutiple codes

A.3 Expert 3

Notes:

- A deactivated mode probably means review mode
- The red dots look like TODOs
- A green code looks like it is a suggestion
- A cross-through is probably a recommendation to delete the code
- Since the red dot disappears it probably tracks seen/unseen updates
- Some form of Tutorial would be helpful to get a first introduction
- The light bulb should have a hover effect otherwise is does not stand out
- I can imagine a comments column on the right; like Google Docs
- To get to the target code of the recommendation, three clicks are necessary
- It should select the code together with the recommendation
- The comments are only single threaded but thread comments could be overkill
- The tooltip appears to slow in case of missing permissions
- The permission structure seems a bit off
- A simple vote count is enough; no need to see the individual users
- A history of changes/updates to the recommendation would be helpful
- The comment section could support user tagging and code tagging ("Hi @user, check out #code")
- There is some bug when closing the modal ESC

- The reommendation mode probably works like the Suggestion Mode (google docs)
- I don't see why only the author of a recommendation can delete it but not an owner
- It is useful that the tabs for resolved recommendation still allow comments
- Apparently the documents toolbar works like before
- all UI elements that are not part of the recommendation mode should be hidden

Before you were notified via the red dot in case of recommendation updates. Would you say this is sufficient or do you think additional cues would be helpful.

- sufficient for the coding editor
- maybe small pop notifications at the top right of the editor
- global notification could be useful across multiple projects; especially your own recommendations
- but can be to overwhelming

Currently a recommendation is bound to a single code/coding. What do you think of the idea of batch recommendations?

- I think it is a good idea
- E.g. one wants to delete and create only together
- E.g. one wants to update the same attribute for multiple codes
- E.g. one wants to add multiple codings for a single code
- E.g. one wants to delete multiple codes

B Questionnaire Results

The questionnaires (B.1 and B.2) use a likert-scale from 1 to 5 where 1 is strongly disagree and 5 is strongly agree. The evaluation, however, uses a scale from 0 to 4. Because the experts might hesitate to answer with 0 even though they strongly disagree, the scale is shifted by 1.

The calculation of the total score of the SUS (B.2), invovles a few more steps. First, the values of negative framed questions are inverted. This means that 0 is mapped to 4, 1 to 3, and so on. Now, for every question, a higher score is more desirable which makes the results easier to interpret. The scores of each question are then summed up and multiplied by 2.5 to get a final score between 0 and 100. It is important to note that the final score cannot be interpreted as percentage.

The results are shown in the tables below.

B.1 Custom

Question	E1	E2	E3
I think that this feature helps to reduce the communication overhead.	4	4	4
I think that this feature will replace our current tool for communication.	3	1	1
I think that this features will make our workflow more efficient.	4	3	3
Sum	11	8	8

B.2 The System Usability Scale (SUS)

Question	E1	E2	E3
I think that I would like to use this feature frequently	4	3	3
I found the feature unnecessarily complex.	4	3	4
I thought the feature was easy to use.	4	3	2
I think that I would need the support of an experienced user to be able to use this feature.	4	4	4
I found the various functions in this feature were well integrated into the regular workflow.	3	2	2
I thought there was too much inconsistency in this feature.	4	3	1
I would imagine that most people would learn to use this feature very quickly.	3	3	1
I found the feature very cumbersome to use.	4	4	4
I felt very confident using the feature.	3	3	4
I needed to learn a lot of things before I could get going with this feature.	3	3	3
Sum	36	31	28
$\times 2.5$	90	77.5	70
Total score (Average)	79.2		

References

Bangor, A., Kortum, P. & Miller, J. (2009). Determining what individual SUS scores mean: Adding an adjective rating scale. *J. Usability Studies*, *4*(3), 114–123.

Brooke, J. (1995). SUS: A quick and dirty usability scale. *Usability Eval. Ind.*, *189*.

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

Følstad, A. (2007). Work-domain experts as evaluators: Usability inspection of domain-specific work-support systems. *Int. J. Hum. Comput. Interaction*, *22*, 217–245. https://doi.org/10.1080/10447310709336963

Fowler, M. (2003). *Anemicdomainmodel*. Retrieved August 30, 2023, from https://www.martinfowler.com/bliki/AnemicDomainModel.html

Fowler, M. (2020). *Patterns for managing source code branches*. Retrieved August 30, 2023, from https://martinfowler.com/articles/branching-patterns.html

Harley, A. (2018a). *Individualized recommendations: Users' expectations & assumptions*. Retrieved August 30, 2023, from https://www.nngroup.com/articles/recommendation-expectations

Harley, A. (2018b). *UX guidelines for recommended content*. Retrieved August 30, 2023, from https://www.nngroup.com/articles/recommendation-guidelines

Kaufmann, A. & Riehle, D. (2018). The qdacity-re method for structural domain modeling using qualitative data analysis. In *Software engineering und software management 2018* (pp. 169–170). Bonn, Gesellschaft für Informatik.

Knijnenburg, B. P., Willemsen, M. C., Gantner, Z., Soncu, H. & Newell, C. (2012). Explaining the user experience of recommender systems. *User Modeling and User-Adapted Interaction*, *22*(4), 441–504. https://doi.org/10.1007/s11257-011-9118-4

Ko, H., Lee, S., Park, Y. & Choi, A. (2022). A survey of recommendation systems: Recommendation models, techniques, and application fields. *Electronics*, *11*(1). https://doi.org/10.3390/electronics11010141

Nielsen, J. (1994). *10 usability heuristics for user interface design*. Retrieved August 30, 2023, from https://www.nngroup.com/articles/ten-usability-heuristics/

Nielsen, J. (1995). *How to conduct a heuristic evaluation*. Retrieved June 12, 2023, from https://www.nngroup.com/articles/how-to-conduct-a-heuristic-evaluation

Richards, M. (2015). *Software architecture patterns: Understanding common architecture patterns and when to use them*. O'Reilly Media. https://books.google.de/books?id=ZLYtuwEACAAJ

Roy, D. & Dutta, M. (2022). A systematic review and research perspective on recommender systems. *Journal of Big Data*, *9*(1), 59. https://doi.org/10.1186/s40537-022-00592-5

Rupp, C. & SOPHISTen. (2020). *Requirements-engineering und -management: Das Handbuch für Anforderungen in jeder Situation* (7th ed.). Hanser.