LEIPZIG UNIVERSITY

Faculty of Mathematics and Computer Science
Institute of Computer Science

Visual Editor Based on SHACL Shapes

BACHELOR THESIS

Author:
Lucas LANGE

Degree Course:
Computer Science

Supervisors:
Dr. Michael MARTIN
Leipzig University
Dr. Sebastian TRAMP
eccenca GmbH

Leipzig, January 12, 2022

LEIPZIG UNIVERSITY

Abstract

Faculty of Mathematics and Computer Science
Institute of Computer Science

Computer Science

Visual Editor Based on SHACL Shapes

by Lucas LANGE

A reasonable visualization of the attributes and relations in a dataset fosters the comprehension of the underlying semantics and concepts, especially in the context of graph datasets.

eccenca Corporate Memory is an enterprise application suite for working with semantic models and building knowledge graphs. The main components of this software can be used for managing and integrating datasets, exploring knowledge graphs as well as curating and authoring resources. Most of these functionalities can be adjusted to specific requirements by using the Shapes Constraint Language (SHACL).

The main goal of this thesis is the specification and implementation of a visual editor for linked data knowledge graphs. Furthermore, this editor should be adjustable to specific domains by utilizing SHACL shape descriptions. Thus, users will be able to explore and edit their data in a graphical representation.

List of Figures

List of Tables

List of Prefixes and Abbreviations

dct:	`http://dublincore.org/2012/06/14/dcterms#`
ex:	`http://example.com/ns#`
foaf:	`http://xmlns.com/foaf/0.1/`
ld:	`http://leipzig-data.de/Data/Model/`
ldot:	`http://leipzig-data.de/Data/Ortsteil/`
ldp:	`http://leipzig-data.de/Data/Person/`
org:	`https://www.w3.org/TR/vocab-org/#org:`
owl:	`http://www.w3.org/2002/07/owl#`
rdf:	`http://www.w3.org/1999/02/22-rdf-syntax-ns#`
rdfs:	`http://www.w3.org/2000/01/rdf-schema#`
sh:	`http://www.w3.org/ns/shacl#`
skos:	`http://www.w3.org/2004/02/skos/core#`

IRI	**I**nternationalized **R**esource **I**dentifier
JS	**J**ava**S**cript
KOS	**K**nowledge **O**rganisation **S**ystem
RDF	**R**esource **D**escription **F**ramework
RxJS	**R**eactive E**x**tensions for **J**ava**S**cript
Sass	**S**yntactically **A**wesome **S**tyle **S**heets
SHACL	**Sha**pes **C**onstraint **L**anguage
SKOS	**S**imple **K**nowledge **O**rganisation **S**ystem
SPARQL	**S**PARQL **P**rotocol **A**nd **R**DF **Q**uery **L**anguage
URI	**U**niform **R**esource **I**dentifier

Contents

Chapter 1

Introduction, Motivation and Structure of the Thesis

eccenca Corporate Memory by eccenca GmbH is an enterprise application suite for working with semantic models. It allows the management, integration, exploration, and manual curation of knowledge graphs. As part of the DataManager component, the form-based front-end user interface is created utilizing Shapes Constraint Language (SHACL) descriptions. The descriptions are a set of shapes, deposited in a shape graph[1]. These shapes describe constraints on Resource Description Framework (RDF)[2] data in data graphs[3] [25]. Exploiting these definitions, SHACL "can [then] be used for documentation, user interface generation, or validation..." [19].

With this thesis, an expansion to the suite, in the form of a visual editor is developed, providing an alternative interface for exploring and managing data graphs. This is done by applying a similar method of user interface generation from SHACL shapes. The shapes are used to configure the editor, which then visualizes the data graph and allows exploration as well as management. Supporting such graphical interaction is a feature considered valuable by eccenca GmbH. Also, a reasonable visualization of the facts in a graph dataset fosters comprehension of the underlying semantics and concepts [22, 28].

The main goal of this thesis is the specification and implementation of a visual editor for linked data knowledge graphs, which should be adjustable to specific domains by utilizing SHACL shape descriptions. Thereby, enabling users to explore and edit their data in a graphical representation.

This thesis is structured as follows. After introducing the initial situation, basic concepts, and the motivation in Chapter 1, I will elaborate on the requirements engineering process in Chapter 2. Next, a comparing look at the state of the art is given in Chapter 3. Chapter 4 then presents the further specification of the editor. Subsequently, the key aspects of the implementation are discussed in Chapter 5. Ultimately leading to Chapter 6—the conclusion, split into the evaluation and an outlook into future work.

[1] "[Conditions] provided as shapes and other constructs ... in the form of an RDF graph" [25].

[2] "A framework for representing information in the Web," and "expressing information about resources" [12, 32].

[3] "RDF graphs that are validated against a shapes graph are called 'data graphs'" [25].

Chapter 2

Requirements

This chapter presents the conducted requirements engineering process and its results.

Opening section 2.1 introduces the use cases that set the focus and scope of the project. Achieving a workable state for these cases is the minimum need and they, therefore, constitute the major requirements.

The next two sections (2.2, 2.3) list the requirements catalog developed. In that, requirements represented as issues, are divided into either functional or non-functional, a classification defined by Sommerville et al. as follows:

> *1. Functional requirements: These are statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do.*
>
> *2. Non-functional requirements: These are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process, and constraints imposed by standards. Non-functional requirements often apply to the system as a whole, rather than individual system features or services. [35]*

Issues are couched in user stories, which assist in understanding the scope and outcome [4]. The art of writing said stories is simplified and unified by using a template. Instead of going with the popular role-feature-reason template, also known as Connextra format, I decided in favor of the format used by eccenca GmbH that differs in wording and order [3]. It is specified as: "In order to XXX AND YYY As a AAA I want to BBB". Also, issues carry names defined by a summary of their corresponding story. They are organized into appropriate categories, which are recorded and explained in Table 2.1, and given a priority rating. In addition to the priority rating, story points[1]—ranging from one to ten[2]—are assigned to measure the relative complexity of a task and by that the time needed to finish it [10]. Further information on an issue is given in the description. Since issues only are a representation, I interchangeably refer to them as requirements.

In the final section (2.4), I review the information from the previous two sections. To continuously focus on essential parts of the editor, priority was determined by use

[1] "Story points are a relative measure of the size of a user story" [11].

[2] One being the smallest, five the medium, and ten the biggest option.

Category	Description
Functional	A functional requirement
Non-functional	A non-functional requirement
backend	Issues regarding database and authorization
convention	Required standards e.g. for coding
feature	Implement new functionalities not directly integrated in the user interface
test	Necessary for testing the use cases
ui	User interface design e.g. styling
ux	User experience design for interacting with the user

TABLE 2.1: Requirement categories with descriptions.

cases and in consultation with eccenca GmbH. Altogether, I will give a concluding overview of the requirements specification.

2.1 Use Cases

Utilizing SHACL shapes enables an agnostic approach for the visual editor. Since any RDF graph can be validated, shapes are customizable to validate each ontology in such data graphs [25]. Hence, assuming that adjustment, every ontology can be visualized and edited in the editor. I proof this concept by defining different use cases, which also show editor functionality.

A use case is achieved when a workable state is reached. For that, SHACL shapes have to be adjusted for the vocabulary, a reasonable amount of resources can be visualized, and finally, creating, editing, and linking resources should be possible.

The defined use cases are objectives representing the goal of this thesis. Consequently, they play a key role in the Evaluation in Chapter 6.

2.1.1 SKOS Taxonomy Management

In a W3C Note editors Summers and Isaac state, that "the Simple Knowledge Organization System (SKOS) is an RDF vocabulary for representing semi-formal knowledge organization systems (KOSs)[3]..." [36]. In this use case, the KOS in focus is the taxonomy. Looking at different groups of KOSs, taxonomies are classifications and categorizations and defined as "divisions of items into ordered groups or categories based on particular characteristics" [39]. This is accomplished by the use of a single relationship between resources: broader/narrower. Subjects may be described further, but relationships are restricted—resulting in a hierarchical structure [18].

[3] "The term knowledge organization systems is intended to encompass all types of schemes for organizing information and promoting knowledge management" [23].

Now, SKOS offers some other options but breaks down to the broader/narrower relation, offering great potential for taxonomy management. The core part of the vocabulary is the concept[4], given by the class `skos:Concept`. To better explain the idea, the following example showcases a simple SKOS taxonomy and is copied from the *SKOS Simple Knowledge Organization System Primer* [36]:

```
1 ex:animals rdf:type skos:Concept;
2   skos:prefLabel "animals"@en;
3   skos:narrower ex:mammals.
4 ex:mammals rdf:type skos:Concept;
5   skos:prefLabel "mammals"@en;
6   skos:broader ex:animals.
```

In the first line the `skos:Concept` for animals is defined, while in the next line the property `skos:prefLabel` is introduced, assigning a character string as the favorite label for the resource—here in English as stated by the language tag @en. The third line describes a narrower relationship—which is `skos:narrower` in SKOS vocabulary—between `ex:animals` and `ex:mammals`.[5] It expresses that mammals is linked to animals and hierarchically mammals is under animals—i.e., mammals is narrower than animals, translating to: mammals are animals. The counterpart to `skos:narrower`, as expected, is the `skos:broader` property which can be seen in line 6 of the example. When there is a `skos:narrower` relation linking two resources, the other resource will have the appropriate `skos:broader` relation. SKOS features not in the example are the class `skos:ConceptScheme` and the property `skos:hasTop Concept`, for which Figure 2.1 provides an example. `skos:Concepts` can be part of `skos:ConceptSchemes`, a relationship that grants an additional hierarchical layering option by using `skos:hasTopConcept`. With that property, a concept scheme can be linked to its more general concepts, which can then be connected to lower concepts through the broader/narrower relation. Ultimately, a concept scheme is "a set of concepts, optionally including statements about semantic relationships between those concepts" [26].

These simple examples illustrated the fundamental use of the vocabulary but real-life applications will see significantly larger instances. Thus, for testing the workable state of the use case, example data is necessary. I decided to go with two different graphs fulfilling the purposes of a) experimenting with the key features of the editor and reaching a workable state, and b) exploring the limits of the visualization. For a), I use a graph named catering industry, which links different restaurants and bars as well as their servings for food and drinks, using the SKOS vocabulary. All the necessary points for the workable state can be tested using this graph. Regarding b) on the other hand, it is too small with only 52 resources. So, a

[4] "Concepts are the units of thought—ideas, meanings, or (categories of) objects and events—which underlie many knowledge organization systems" [36].

[5] The prefix ex: for animals and mammals only shows the exemplary character of the resources, its namespace is located at: `http://www.example.com/` .

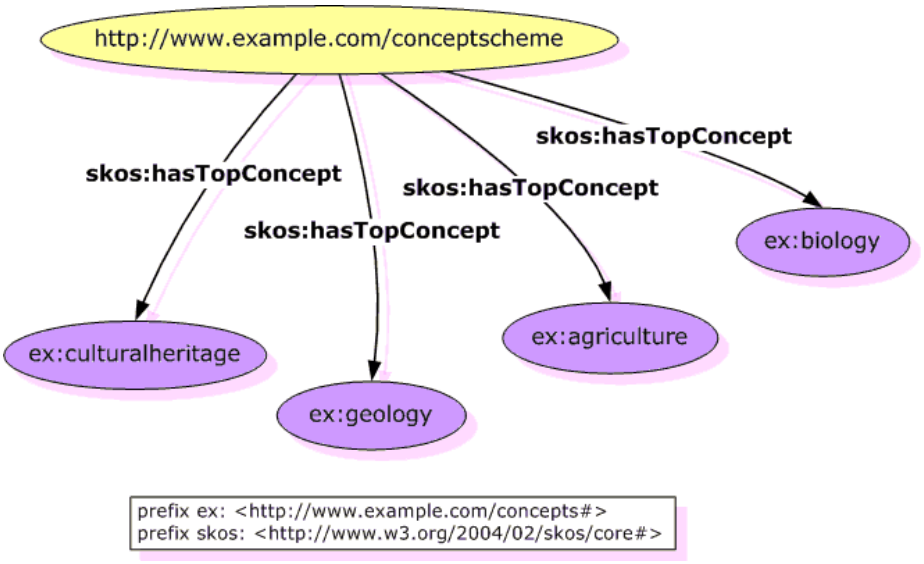

FIGURE 2.1: Exemplary illustration of a concept scheme from the *SKOS Core Guide* [26].

second, larger graph is needed. I went with the EuroVoc[6] graph because it is free to access, utilizes SKOS, is a real-world example, and should be large enough—209,287 resources—to stress the limitations of visualization in the editor.

The SKOS taxonomy management is a superior use case because the taxonomic structure of SKOS can be combined with ontological ones. This, for example, is utilized in the second use case's ontology, which is discussed in the next section.

2.1.2 Organizational Structure

"[The Organization Ontology] is designed to enable publication of information on organizations and organizational structures including governmental organizations" [31]. Regarding KOSs ontologies are in the group of relationship models and are defined by Zeng as "specific concept models representing complex relationships between objects, including the rules and axioms that are missing in semantic networks" [39]. Compared to taxonomies the main difference is that the vocabulary can be described "at will, overcoming the vocabulary limitations" [18]. By that, the main point of distinction is that, in general, ontologies provide a wider range of information concerning relationships.

Figure 2.2 gives an overview of the Organization Ontology. It contains the main classes and relationships, also indicating other vocabulary used—e.g. `skos:Concept`. The core of the ontology and focus of this use case is the organizational structure. Here the class `org:Organization` is essential. Hierarchical relations between organizations are implemented with `org:subOrganizationOf` and `org:hasSubOrganization`, while also providing specialization through the class `org:OrganizationalUnit`, which is meant to represent a part of a organization in its

[6] Project website: `http://eurovoc.europa.eu/drupal/?q=abouteurovoc&cl=en` .

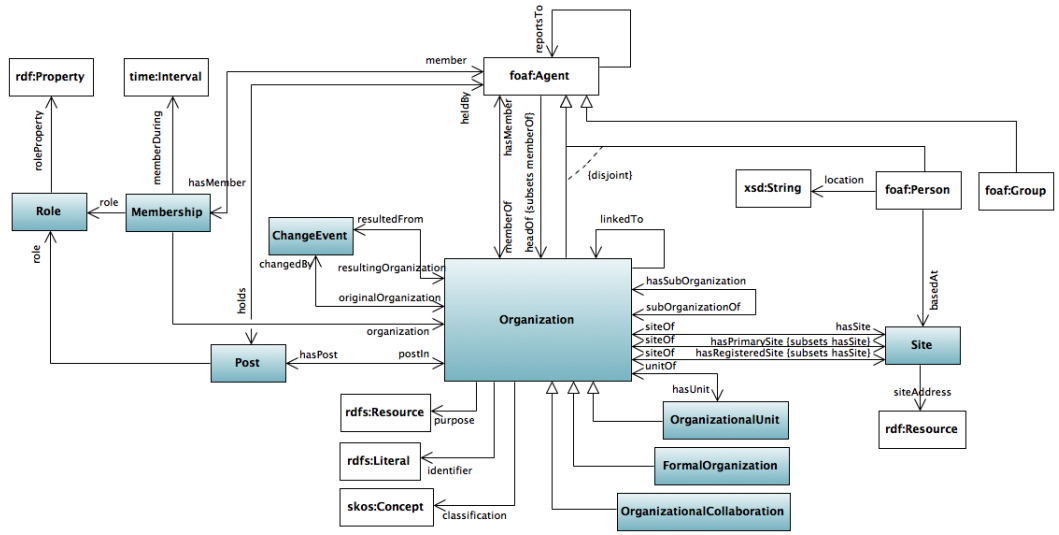

FIGURE 2.2: A pictorial illustration of the main classes and relationships in ORG from *The Organization Ontology* by Reynolds [31].

context—e.g. departments. Apart from the specialization, focus once more is the hierarchical layout but as shown in Figure 2.2 there is more vocabulary the editor has to process.

This excerpt of the representation of the UK Cabinet Office's structure using the ontology was derived from Reynolds's *The Organization Ontology* [31]:

```
<http://reference.data.gov.uk/id/department/co>
    rdf:type org:Organization , central-
       government:Department;
    skos:prefLabel "Cabinet Office" ;
    org:hasUnit <http://reference.data.gov.uk/id/
       department/co/unit/cabinet-office-communications> .

<http://reference.data.gov.uk/id/department/co/unit/
   cabinet-office-communications>
    rdf:type org:OrganizationalUnit ;
    skos:prefLabel "Cabinet Office Communications" ;
    org:unitOf  <http://reference.data.gov.uk/id/
       department/co> ;
    org:hasPost <http://reference.data.gov.uk/id/
       department/co/post/246> .

<http://reference.data.gov.uk/id/department/co/post/246>
    skos:prefLabel "Deputy Director, Deputy Prime Minister
       's Spokesperson/Head of Communications" .
    org:postIn <http://reference.data.gov.uk/id/department
       /co/unit/cabinet-office-communications> ;
```

```
15      org:heldBy <#person161> .
```

I first want to point out the use of `skos:prefLabel`, again supporting the statements concerning the connection with the first use case. In the lines 1–4 and 6–10 an `org:Organization` and an `org:OrganizationalUnit` are introduced. The relationship between the two resources is defined reflexive as `org:hasUnit` and `org:unitOf` in lines 4 and 9, respectively. Other than that, the description and assignment of a position in the organizational unit is demonstrated in lines 10 and 12–15. First, in line 10, the position is added to the unit using `org:hasPost`, whereupon it is detailed in lines 12–14. Finally, in the last line, the relation `org:heldBy` refers to the person holding the position.

Again this only is a small instance and therefore parts[7] from the project Leipzig Data[8] are employed as a data graph for testing the editor. In addition to the Organization Ontology and its organizational structure, other namespaces are used frequently—especially their own[9]. The main justification for this second use case is showing and proving the agnostic approach. Further, the Organization Ontology provides more and other relationships between resources than the SKOS—which is also used—and thereby delivers a different problem for visualization and work with the editor.

2.2 Functional Requirements

Requirement **PM-358 — Create new resources**
Categories: **Functional, ux**
Priority: **Major**; Story Points: **5**
User Story:

In order to create new resources based on existing templates

As an editor

I want to create resources on the board by using stencils based on `sh:NodeShapes`

Description:

- NodeShapes provide common attributes in relation to other resources
- In addition to that they provide validation information
- Resource is then displayed in the editing area

Requirement **PM-359 — Load existing resources**
Categories: **Functional, ux**
Priority: **Major**; Story Points: **5**
User Story:

[7] The parts are Personen and Buergervereine. Information on the parts: `http://leipzig-data.de/ontology/akteure/` .

[8] Project website: `http://leipzig-data.de/` .

[9] Public data available here: `https://github.com/LeipzigData/RDFData/` .

In order to load existing resources based on `sh:NodeShapes`
As an editor
I want to load resources on the board by selecting them from a search result
Description:

- The user selects the corresponding `sh:NodeShape`
- Then a search assists in finding the correct resource
- Resource is then displayed in the editing area

Requirement **PM-360 — Start from scratch**
Categories: **Functional, feature**
Priority: **Major**; Story Points: **5**
User Story:
In order to start a graph from scratch
As an editor
I want to create new resources in the graph based on available `sh:NodeShapes`
Description:

- Stencils based on available `sh:NodeShapes` enable resource creation
- At the start an empty editing area is displayed
- Created resources are saved in the graph

Requirement **PM-361 — Edit resources**
Categories: **Functional, ux**
Priority: **Major**; Story Points: **5**
User Story:
In order to edit resources displayed in the editing area
As an editor
I want to edit resources using stencils based on `sh:NodeShapes`
Description:

- New information can be given or existing information can be change
- Changes are saved in the graph
- Updates the resource in the editing area
- Can also be used to view all the information for a resource
- Users are able to save their changes or cancel

Requirement **PM-362 — Specific resource as starting point**
Categories: **Functional, feature**
Priority: **Major**; Story Points: **6**
User Story:

In order to start with a specific resource as starting point
As an editor
I want to load an exisiting resource and have it displayed from the start

Description:

- A resource can be selected to view in the editor
- It will then be loaded and displayed in the editing area from the start
- It will enable a user to directly jump from a resource in the database view to viewing it in the visual editor

Requirement **PM-363 — List of specific resources as starting point**
Categories: **Functional, feature**
Priority: **Major**; Story Points: **6**
User Story:

In order to start with a list of specific resources as starting point
As an editor
I want to load a list of exisiting resources and have them displayed from the start

Description:

- A list of resources can be selected to view in the editor
- They will then be loaded and displayed in the editing area from the start
- It will enable a user to directly jump from a list of several resources in the database view to viewing them together in the visual editor

Requirement **PM-364 — Sidebar with stencils**
Categories: **Functional, ux**
Priority: **Major**; Story Points: **5**
User Story:

In order to use stencils based on available `sh:NodeShapes`
As an editor
I want to have a sidebar listing stencils for available `sh:NodeShapes`

Description:

- Sidebar contains als stencils necessary to work with the underlying graph
- Stencils listed can be dragged

- Dropping one in the editing area will let the user create or load resources using existing templates
- There is a warning if no `sh:NodeShapes` exist

Requirement **PM-365 — Adjust to specific domains by utilizing sh:NodeShapes**
Categories: **Functional, feature**
Priority: **Major**; Story Points: **7**
User Story:

In order to work with different graphs containing `sh:NodeShapes`
As an editor
I want to have the editor adjust to specific domains utilizing definitions given in the `sh:NodeShapes`

Description:

- sh:Nodeshapes contain all information needed to work with the specific domains different graphs provide
- Resources can be created, loaded and edited based on the underlying `sh:NodeShapes` given in the domains
- No further adjustment is needed as long as the domain uses `sh:NodeShapes`

Requirement **PM-366 — Toolbar with Buttons**
Categories: **Functional, ux**
Priority: **Major**; Story Points: **3**
User Story:

In order easily access features
As an editor
I want to have corresponding buttons in a toolbar

Description:

- Offers buttons for different features, e.g. zoom (out/reset/in), editing, adding relations, layouting, extending edges by adding levels(, counting) ...
- Relevant information should be displayed, e.g. zoom level(, count) ...
- The user has all features at hand at all times
- Buttons added along new features

Requirement **PM-370 — Zoom**
Categories: **Functional, ux**
Priority: **Major**; Story Points: **4**
User Story:

In order to better view the visualized graph in the editing area
As an editor
I want to zoom my view in and out
Description:

- Zoom level can be increased, decreased or reset
- Enbales the user to work with the visualization of huge or small graphs
- Parts of graphs can be viewed in detail by zooming in
- In the same manner an overall view can be achieved by zooming out

Requirement **PM-371 — Panning**
Categories: **Functional, ux**
Priority: **Major**; Story Points: **4**
User Story:
In order to adjust the view in the editing area
As an editor
I want to use panning
Description:

- While holding down the left mouse button while hovering an empty space in the editing area panning is activated
- This enbales the user to move the view around and explore the visualization
- Releasing the button deactivates panning

Requirement **PM-372 — Move displayed resources**
Categories: **Functional, ux**
Priority: **Major**; Story Points: **2**
User Story:
In order to move displayed resource around the editing area
As an editor
I want to use my mouse to move displayed resources
Description:

- Holding down the left mouse button while hovering a displayed resource enables the user to move this resource around the editing area
- Moving them to the edge of the area activates edge panning which allows for adjustment of the view
- Releasing the button deactivates movement

Requirement **PM-373 — Load graph**
Categories: **Functional, feature**
Priority: **Major**; Story Points: **3**
User Story:

In order to work with my data
As an editor
I want to load data from a specific graph to be used in the editor

Description:

- The graph is given to the component
- `sh:NodeShapes` are loaded from the graph and enable the editor to work
- Data utilizing `sh:NodeShapes` is available for work

Requirement **PM-374 — Divide sidebar for active sh:NodeShapes**
Categories: **Functional, ux**
Priority: **Minor**; Story Points: **6**
User Story:

In order to work with the `sh:NodeShape` templates from the sidebar
As an editor
I want to distinguish between active and available `sh:NodeShapes`

Description:

- Active `sh:NodeShapes` are those for which there are existing resources
- More might be defined
- Active shapes are shown on top
- Also other available shapes which are not active are visible separately
- The user gains fast access to `sh:NodeShapes` with existing resources without missing out on other available shapes
- Collapsibles/Accordions to organize these categories in the sidebar

Requirement **PM-375 — Fit specific view**
Categories: **Functional, ux**
Priority: **Minor**; Story Points: **8**
User Story:

In order to view a specific part of the graph in the editing area
As an editor
I want to be able to make use of automatic zoom and view scaling

Description:

- Zoom to optimal factor fitting the wanted part of the graph (or the whole graph)
- Move the view in the editing area to the part
- Automatically fit the view to the whole graph or a specific resource

Requirement **PM-376 — Layout graph**
Categories: **Functional, ux**
Priority: **Major**; Story Points: **9**
User Story:
In order to better view the graph in the editing area
As an editor
I want to be able to run automatic layouting
Description:

- Applies a specific layout to the graph in the editing area
- View should be fitted automatically as well
- Give the user different layout options to choose from

Requirement **PM-377 — Search for displayed resource**
Categories: **Functional, ux**
Priority: **Minor**; Story Points: **7**
User Story:
In order to view a specific resource in the editing area
As an editor
I want to be able to search the editing area and get to the resource by selecting them from a search result
Description:

- Small search bar on top of the sidebar
- Selecting a search result selects the resource in the graph and fits the view in the editing area to the selected resource
- Enables easier working with large graphs

Requirement **PM-378 — Add relation**
Categories: **Functional, ux**
Priority: **Major**; Story Points: **7**
User Story:

In order to add a relation between resources
As an editor
I want to add relations for resources using a button
Description:

- Button in the toolbar
- The selected resource has possible added relations defined through `sh:NodeShapes`
- `rdfs:seeAlso` relation is always available
- The User selects a relation for his selected resource
- Then the other resource which shall be related is selected by the user
- Selectable relations and resources are validated through the underlying `sh:NodeShapes`

Requirement **PM-379 — Load relation**
Categories: **Functional, ux**
Priority: **Major**; Story Points: **10**
User Story:
In order to view relations between resources in the editing area
As an editor
I want to have edges displaying the relations between displayed resources
Description:

- Edges connect two resources through a directed arrow
- An edge has a label stating the same as the relation's label
- The label is located in the center of the arrow
- When adding resources to the graph, edges to already displayed resources are drawn automatically

Requirement **PM-380 — Update editing area**
Categories: **Functional, feature**
Priority: **Major**; Story Points: **5**
User Story:
In order to get changes to the data displayed in the editing area
As a developer
I want to update and validate the editing area's content whenever necessary
Description:

- Changes to resources are updated

- Changes to relations are updated
- The editing area must be kept updated

Requirement **PM-381 — Add edge level**
Categories: **Functional, ux**
Priority: **Major**; Story Points: **5**
User Story:
In order to explore the data related to displayed resources
As an editor
I want to load all relations for a resource
Description:

- Possible for a single selected resource or all displayed resources at once
- The option to load for all resources may cause heavy load times
- Enables the user to load the relations of a resource

Requirement **PM-382 — Remove edge level**
Categories: **Functional, ux**
Priority: **Major**; Story Points: **7**
User Story:
In order to explore the data related to displayed resources
As an editor
I want to remove all relations for a resource from the view
Description:

- Possible for a single selected resource
- Enables the user to remove the displayed relations of a resource from the view
- Might be possible for all displayed resources at once

Requirement **PM-383 — Outline**
Categories: **Functional, ux**
Priority: **Major**; Story Points: **8**
User Story:
In order to have an overview of the editing area **AND** navigate using it
As an editor
I want to be presented an outline of the editing area with which I can navigate the view
Description:

- The outline is located in the bottom of the left sidebar space

- A small preview of the editing area is shown
- Using a rectangle in the outline the current view of the editing area is shown
- View can be navigated through the moving and resizing rectangle
- The view is being moved or zoomed accordingly

Requirement **PM-387 — Keyboard shortcuts**
Categories: **Functional, ux**
Priority: **Minor**; Story Points: **4**
User Story:
In order to work faster using simple commands
As an editor
I want to have keyboard shortcuts for different functionalities
Description:

- Many possible shortcuts, e.g. zoom, edit a resource, change the edge level of a resource and almost every other feature
- The user doesn't have to move his mouse cursor out of the editing area to use features from the toolbar

Requirement **PM-388 — Gesture and touch commands**
Categories: **Functional, ux**
Priority: **Minor**; Story Points: **6**
User Story:
In order to work faster and with more devices
As an editor
I want to use gesture and touch commands
Description:

- Touchpads, e.g. pinch-to-zoom, two-finger-scrolling ...
- Touch devices, e.g. touch gestures for zoom, scrolling ...
- Easier use of the editor with more devices

Requirement **PM-389 — Tooltips**
Categories: **Functional, ux**
Priority: **Minor**; Story Points: **6**
User Story:
In order to understand the functionality **AND** get more information
As an editor
I want to see helpful tooltips

Description:

- Tooltips to explain the general functionality of the editor and its interface
- When hovering buttons explains their function to the user
- Better readability when zoomed out or with many resources in the editing area
- Tooltip when hovering a resource to see its label and possibly more information
- Tooltip when hovering a relation to see its label, the resources it relates and possibly more information
- Tooltips should have a delay before displaying and should contain short descriptions only

Requirement **PM-390 — Force refresh**
Categories: **Functional, ux**
Priority: **Minor**; Story Points: **4**
User Story:
In order to make sure the information in the editing area is updated properly
As an editor
I want to force a refresh of the editing area
Description:

- Editing area is updated
- Enables the user to remove outdated information from the editing area
- Editing area should updated itself but some failure could occur
- Because of the specific and rare usage this is a minor issue

Requirement **PM-391 — Delete**
Categories: **Functional, ux**
Priority: **Minor**; Story Points: **6**
User Story:
In order to delete resources
As an editor
I want to have the option to delete a resource
Description:

- Embedded as a button in editing mode
- User is asked to confirm
- Deletes the resource, meaning removing it from the database and editing area (view)

Requirement **PM-392 — Remove**
Categories: **Functional, ux**
Priority: **Minor**; Story Points: **2**
User Story:
In order to remove resources from the editing area
As an editor
I want to have the option to remove a resource from the view
Description:

- Embedded as a button
- Removes the resource, meaning removing it from the editing area (view)
- It is not deleted and therefore not removed from the database

Requirement **PM-393 — Clear**
Categories: **Functional, ux**
Priority: **Major**; Story Points: **1**
User Story:
In order to create an empty editing area
As an editor
I want to remove all resources from the editing area
Description:

- Embedded as a button
- User could be asked to confirm
- Removes the resources, meaning removing them from the editing area (view)
- It is not deleted and therefore not removed from the database

2.3 Non-Functional Requirements

Requirement **PM-367 — Directory and component structure**
Categories: **Non-functional, convention**
Priority: **Minor**; Story Points: **4**
User Story:
In order to conform to existing structuring rules
As a developer
I want to keep the directory and component structure according to the rules
Description:

- eccenca GmbH defined rules for their GitLab directories and React.js component structuring

- Conforming to these rules enables other developers to better navigate and comprehend the component
- It is also needed to integrate the component with the existing architecture

Requirement **PM-368 — Linter conformity**
Categories: **Non-functional, convention**
Priority: **Minor**; Story Points: **4**
User Story:

In order to conform to set coding standards
As a developer
I want to keep the code conform to enabled linter rules

Description:

- eccenca GmbH defined rules for their codebase using linters
- Conforming to these rules enables other developers to better navigate and comprehend the code
- It is also needed to better integrate with the existing architecture and codebase

Requirement **PM-369 — Example data**
Categories: **Non-functional, test**
Priority: **Major**; Story Points: **6**
User Story:

In order to test the editor
As a developer
I want to create and load different example datasets to be used while testing the editor

Description:

- Creating a new example dataset allows to use optimal data for different test cases
- Creating, editing or deleting resource in this dataset has no unwanted effects and therefore is safe
- Loading existing datasets as examples (e.g. the eurovoc graph) is an opportunity to test the editor in real case studies
- Use case specific test data can be loaded (skos vs org)

Requirement **PM-384 — Triple-Store backend**
Categories: **Non-functional, backend**
Priority: **Major**; Story Points: **4**
User Story:

In order to store and use data
As a developer
I want to run a triple-store backend
Description:

- Database backend to store semantic data
- Enables querying of stored data

Requirement **PM-385 — OAuth 2 support**
Categories: **Non-functional, backend**
Priority: **Major**; Story Points: **3**
User Story:
In order to connect to eccenca backend services **AND** APIs
As a developer
I want to use OAuth 2 for authorization
Description:

- The OAuth 2 protocol is used to authorize the client for service use
- Authorization ensures security within the service environment

Requirement **PM-386 — Database connection**
Categories: **Non-functional, backend**
Priority: **Major**; Story Points: **2**
User Story:
In order to store, edit and retrieve data from the database
As a developer
I want to establish a databse connection
Description:

- Connecting to an exisiting database is necessary for the editor to work
- While using the editor data will be stored, edited and retrieved
- eccenca has a test database to connect to for this project

Requirement **PM-394 — UI Styling**
Categories: **Non-functional, ui**
Priority: **Trivial**; Story Points: **3**
User Story:
In order to make the editor more appealing and usable
As a developer
I want to style the different elements of the editor accordingly

Description:

- Elements are styled using SCSS
- A uniform style should be achieved
- Conformity with eccenca standards
- Use existing style templates by eccenca where possible

2.4 Prioritization and Overview

For the start, I refer to Figure 2.3, in which two charts illustrate the number of requirements in each category. The left chart tracks every category and whether they are associated with functional or non-functional, whereas the smaller chart to the right only displays the latter two. Functional and non-functional are universal in that every requirement is either the one or the other. This is shown in the right chart as its numbers add up to 37, the total amount of items in the catalog. The same is true when summing up the fractions in the bigger chart, indicating a requirement may only also have one of the other six categories and hence always has exactly two categories overall. More examination of the pie chart exhibits a vast majority of functional requirements at above 80%. This mainly stems from the fact that just slightly less than two-thirds of requirements are in the ux category, a functional one. It is by far the biggest part and represented in the right portion of the doughnut. The size is no surprise, since the main goal of the thesis is to allow user interaction in the visual editor. Giving a short glance at the story points in Table 2.2, one can confirm, that the number of requirements correlates with the sum of story points, with test being the only exception by ranking above ui. Feature is the other functional category from the catalog and ranks second. The remaining four, viz., backend, convention, ui, test, are non-functional and yield the remaining fifth of requirements. The distributions in the charts clearly demonstrate the greater concern with functional issues in the development of the editor.

As stated by Alkandari and Al-Shammeri, “Requirement prioritization is a main phase in requirement engineering” [2]. Therefore, keeping the findings from Figure 2.3 in mind, focus now shifts towards Table 2.2. After having studied the spread of requirements over categories, this listing adds an extra division into priorities. Shown are the priority levels—major, minor, trivial—and their counts in general and for every single category. The table also displays the total story points for every category. As stated before, discoveries from Figure 2.3 hold true, comparing the numbers to story points. According to the first line of the table, the 37 requirements in general break down into 23 major, 13 minor, and 1 trivial. A major priority indicates an issue of critical importance for the editor, where not implementing translates to a major loss of function. Analogous to this definition, minor resembles

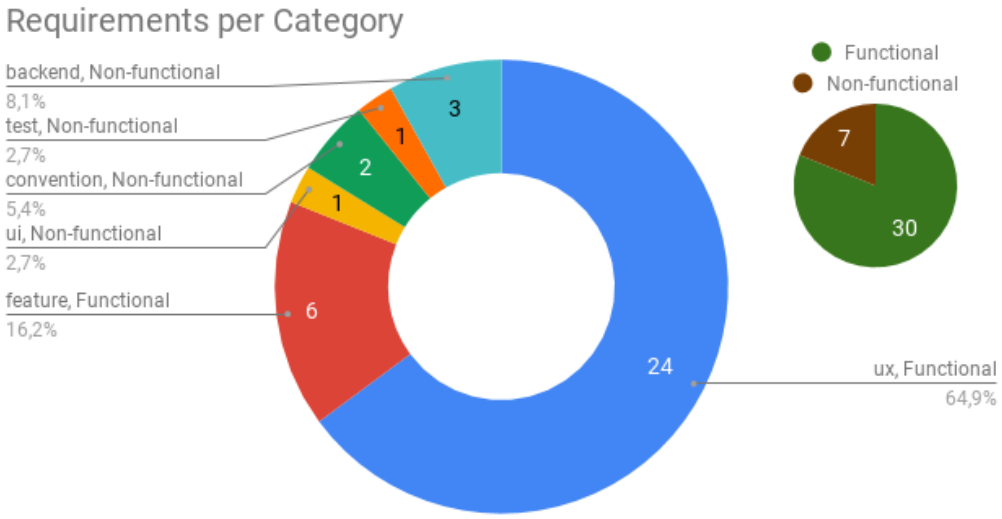

FIGURE 2.3: Requirements counted per category and associated with functional or non-functional.

problems not crucial for key functionality. To give an example, PM-387 is minor because it describes an optional function, whereas PM-365 is a fundamental concept. This priority assessment was undertaken in collaboration with eccenca GmbH to determine the correct needs. Trivial is the lowest level and mainly stands for cosmetic problems. The only trivial issue—PM-394—is focused on such nonessentials and is the only piece labeled ui. Further examination of the table, gives away that backend and test are major categories, while convention contains minor issues alone. A phenomenon easily explained by the facts that backend centers around database use and test contains the issue that provides needed experimental data for development, altogether forming the foundation for every other feature working with data. Convention, on the other hand, revolves around conformity to coding standards and directory structure, desirable but not major points. Functional and non-functional both hold about twice as many major as minor ones and the same remains true for feature and ux. As the name implies, feature embodies functions, some belonging to core functionality, also forming a basis for the solution of many ux issues. Besides these, there are two minor requirements involved, PM-391 and PM-392, which introduce requested features but have been put behind for more important ones. Being the largest class, ux accommodates many major requirements, which as said before is based on the fact that the user experience is a key part of the main goal. Thus, the fulfillment of these major requirements is needed to enable the expected functionality set for the editor. There is also a fair share of minor ux requirements, for the most part describing functions, which—while being optional for the basic development of the editor—deliver valuable enhancements (e.g. PM-377).

As a conclusion from the past analysis, the ux requirements stand out as the

Category	Priority			Story Points
	Major	Minor	Trivial	
—	23	13	1	187
Functional	19	11	0	161
Non-functional	4	2	1	26
backend	3	0	0	9
convention	0	2	0	8
feature	4	2	0	32
test	1	0	0	6
ui	0	0	1	3
ux	15	9	0	129

TABLE 2.2: Requirement categories in relation to priority ratings and story points.

biggest part of the editor. Nonetheless, core functionality from feature, backend, and test are required as well. Minor issues are suggested functions and associated with every aspect of the editor. Resulting from that knowledge, development must concentrate on the major part and after that progressing in the minor category is beneficial. Trivial issues should only be tackled if time is spare. It is important to note, that because of the agile process, prioritization and requirements themselves may change [24].

The requirements catalog resulting from this chapter is the groundwork for the following chapters. After evaluating the state of the art in Chapter 3, the requirements engineering process will culminate in Chapter 4, delivering the specification.

Chapter 3

State of the Art

In this chapter, I will list the current knowledge relevant to visual editors for linked data knowledge graphs. This is done by analyzing and evaluating projects that offer features comparable to this thesis's goals.

In "Towards a uniform user interface for editing data shapes" by De Meester et al., the unSHACLed editor[1] is presented, offering an interface for data graph and shape editing. The editor is focused less on visualization but editing. This can be concluded from the stated features, e.g. allowing different constraint languages—other than SHACL—or editing multiple graphs at the same time. Two features comparable to this thesis are the similar agnostic approach regarding the ontologies and the editing possibility for shapes and data shapes in one editor. Relevant for the specification is the design of the prototype interface, which is illustrated in Figure 3.1. The idea is to have a separation between a sidebar (a) and an editing area (c). Templates from the sidebar are moved onto the editing area using drag-and-drop, allowing the creation of new resources. Adding an action toolbar (b) to the composition allows easy access to several features. The visualization (d) should not be implemented in the same way, since the always visible properties for every resource hinder the comprehension of larger graphs. Further, the editing for the rectangles and its visible properties, as implied in the figure (e), is not needed in the same way. These differences in workflow and requirements stem from the integration into the existing eccenca GmbH environment. [13]

The LodLive project[2] is focused on browsing RDF resources. Figure 3.2 shows an example visualization taken from the paper "LodLive, Exploring the Web of Data" by Camarda, Mazzini, and Antonuccio. Visualized resources are circles with their label and their relations are connections through directed arrows, also with their label. A single resource is loaded and starting from that the exploration is done by adding related resources by selecting one of the smaller bubbles around the resource. An idea that should be evaluated after the prototypical implementation for this thesis, is whether a comparable selective approach to the visualization of relations could be beneficial to the user experience. The circles and arrows are arranged to not overlap. LodLive has no editing functionality to review but offers a

[1] Github: `https://github.com/dubious-developments/UnSHACLed` .
[2] Available at: `http://en.lodlive.it/` .

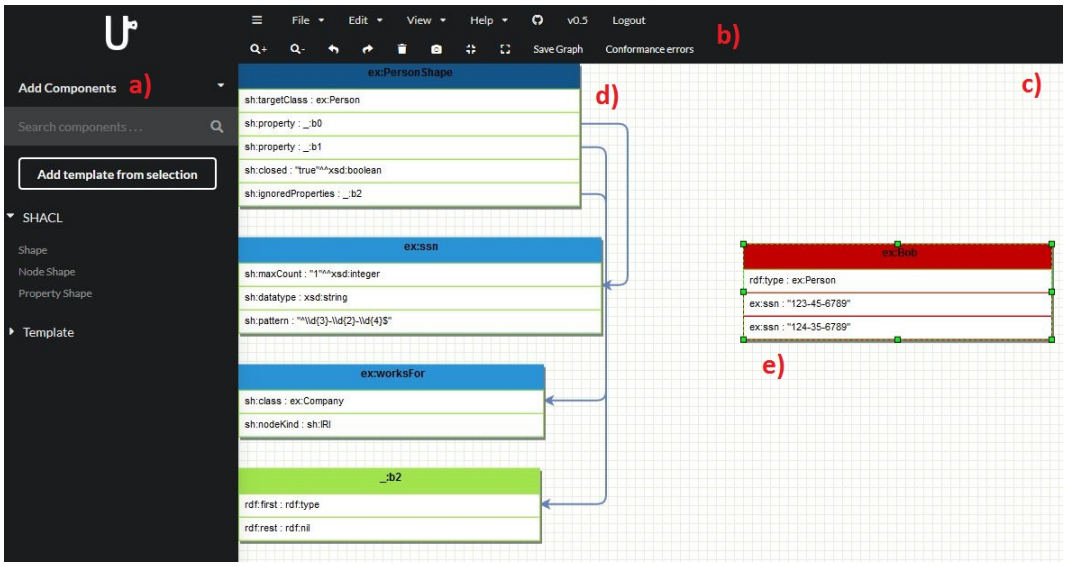

FIGURE 3.1: "The unSHACLED UI, consisting of an Overview Sidebar (left[, a]), an Action Toolbar (top[, b]), and an Editing Area (middle-right[, c])" [13].

better visualization than the unSHACLed editor. Problems might arise visualizing hierarchic relationships with LodLive's approach—e.g. with SKOS taxonomies. [8]

The visual editor should combine insights from both projects. Delivering a scalable visualization like LodLive, while also offering a user interface close to the unSHACLed prototype. After evaluating the implementation of the visual editor, future work might benefit from reviewing these projects again.

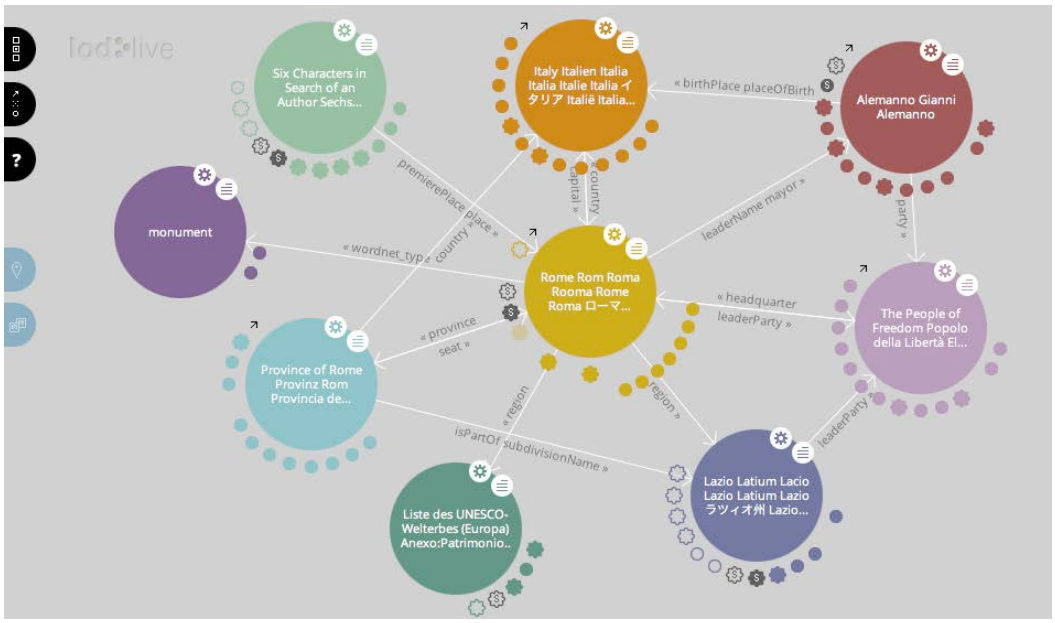

FIGURE 3.2: An example showing "some "exploded" Object Properties connecting different resources" in the LodLive project, from the paper "LodLive, Exploring the Web of Data" [8].

Chapter 4

Specification

The next sections provide the specifications for data management and user interface, derived from the requirements engineering process.

4.1 Data Management

In this section, I want to describe the process of data management that fuels the editor, first clarifying which data is needed. The graph given to the editor is the main source,—hopefully—containing a suitable data graph utilizing vocabulary with existing SHACL shape definitions. The knowledge is stored inside the graphs and has to be fetched using SPARQL Protocol And RDF Query Language (SPARQL)[1] queries. Shape information, in general, is provided by the overall shape catalog graph in the eccenca GmbH database but searching in the given graph itself could be a conceivable idea as well.

The semantic vocabulary for the editor to process the data is illustrated in Figure 4.1 and defined as Combined Visual Editor Ontology. Classes are pictured as rectangles with their names, also embedding their datatype properties[2] and a short description. All arrows symbolize relations between the classes but differ in meaning. These with a black arrowhead wear a label and are object properties[3] linking the classes, while those having a clear/white head represent the relationships, in which the class is a subclass of the one the arrow points at—`rdfs:subClassOf`. The classes stem from three different namespaces: `rdfs:`, `sh:`, and `owl:`. From the first, both classes are located at the top right part of the figure. `rdfs:Resource` is, as stated in its description, everything—i.e. every class is a resource. This is depicted by the fact, that every other class is directly or indirectly connected through an arrow indicating a subclass relationship. Therefore, every node visualized in the editor can be handled as a `rdfs:Resource`. In regard to properties, a resource has two datatype and one object variant. The datatype ones are `rdfs:label` and `dct:description`, which are necessary to show the names of and give further information on resources in the editor. `rdfs:seeAlso` refers from one resource to another and thus could be between any two classes. It is a universal relation offered in the editor, just as

[1] "A query language and protocol for RDF" [9].
[2] These are an `owl:DatatypeProperty`, which contain literal data [6].
[3] They are an `owl:ObjectProperty`, meaning their value is another resource [6].

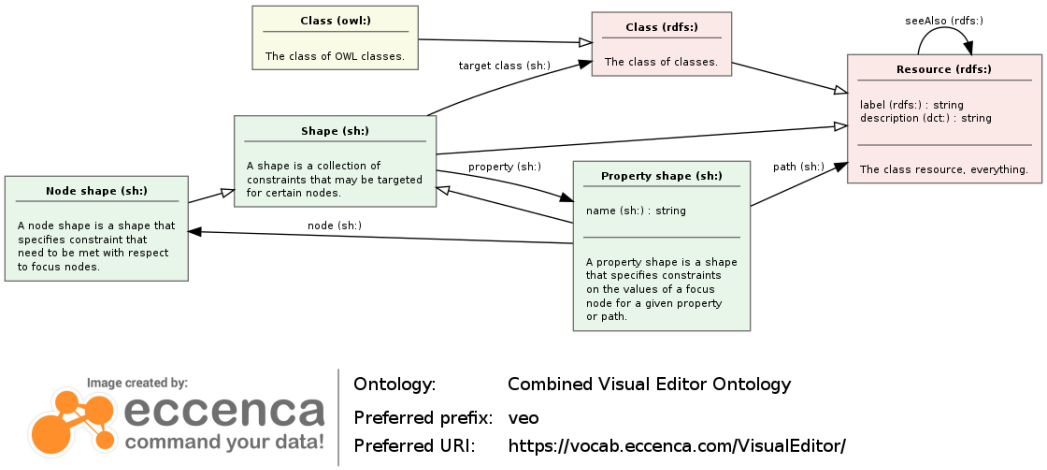

FIGURE 4.1: A representation of the Combined Visual Editor Ontology used.

stated in PM-378. Switching over to the `sh:` namespace, the class `sh:Shape` and its two subclasses `sh:NodeShape` and `sh:PropertyShape` are relevant. The shape class itself is not needed and is replaced with a combination of its subclasses. The object property `sh:targetClass` specifies which classes of resources should be validated by the referencing shape, delivering important information. These validated classes are represented by the `rdfs:Class` itself, which poses for every other class. For the sake of completeness, its subclass `owl:Class` is added as well, because it finds use in many ontologies. Furthering on the replacement of `sh:Shape`, `sh:NodeShape` and `sh:PropertyShape` are combined in that a node shape specifies the target class and then encapsulates any number of property shapes the target should be validated against. This encapsulation is done using `sh:property` which defines a property shape and in the editor should be used to fetch information to display relations between resources. Taking a closer look at property shapes, the datatype property `sh:name` is wanted to get the names of shapes themselves and also of referred properties. The object property `sh:node` is used to refer to a node shape, whose defined restrictions will also be used for the property shape—i.e. a union of the node shape's and the property shape's restrictions [25]. It should be used to restrain from loading sub node shapes. Looking at the last object property `sh:path`, it determines the focus of the property shape. Other than giving a concrete property[4], `sh:path` also allows "a path expression, which would allow you to constrain values that may be several "hops" away from the starting point" [37]. There is always exactly one `sh:path` for a given property shape, defining its target [25]. To sum up the recent statements, node shapes should be used to extract the classes relevant to the graph, while querying property shapes offers access to the relevant properties of resources.

After examining the vocabulary for the editor, I now want to explain how SHACL shapes can form the basis of the editor. For that, I want to start by investigating

[4]Specified as an Internationalized Resource Identifier (IRI) [25].

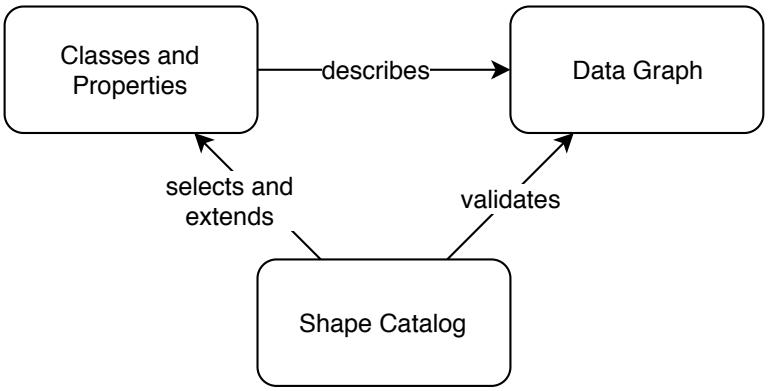

FIGURE 4.2: Ontology, instance and shape data relationship.

the connection between ontology, instance, and shape data. Figure 4.2 suggests an illustration, where the input graphs for the editor are broken down in their individual parts. The top left rectangle stands for the data from an ontology—not being limited to one single ontology—, which are classes and properties, while to the right instance data referred to as a data graph is portrayed. The third rectangle represents the shape catalog with its node and property shape data. Arrows are used to characterize the relationship between these three types. Looking at the top part, excluding the shape catalog, it is expressed, that ontology data describes the data graph. This is done by the instance data from the graph being defined utilizing the vocabulary—i.e. classes and properties—from the ontology. Now, including the shape catalog, other relations come into play. For the ontology data, the node and property shapes from the catalog first select the wanted classes and properties, while simultaneously offering the possibility of extending the given information—`sh:name` being a simple example. On the other side, this data is then used to validate the data graph. The combination of the selected data from the ontology with its extension and added restrictions through the shape, results in a template, the instance data from the data graph has to conform to. This template is the main reason for basing the editor on these SHACL shapes. The selection and extension process can be done for all ontologies describing the data graph. The shapes can be seen as a converging lens refracting the light, which then bunches up and meets in one point. Transferring this metaphor, the different ontology data is refracted—and filtered—through the shapes to then bunch up as one catalog of restrictions and information. The editor is making use of that by directly fetching data from the shape catalog, effectively being independent of the ontology and instance data. Since the data graph has to conform to the template, created from ontology data and shapes, the editor can easily adopt the template for visualization. Hence, the editor is based on SHACL shapes.

In the following example, I want to resolve the topic of adjusting SHACL shapes for different vocabularies.

```
1 <http://leipzig-data.de/Data/Buergerverein/
    Kolonnadenviertel>
```

```
2      ld:hasAddress <http://leipzig-data.de/Data/04109.
          Leipzig.Kolonnadenstrasse.11> ;
3      ld:hasCategories "Bürgervereine" ;
4      ld:hasOrtsteil ldot:Zentrum-West ;
5      ld:hasStadtId "50" ;
6      a ld:Buergerverein, ld:Verein, org:Organization ;
7      rdfs:label "B\ürgerverein Kolonnadenviertel e.V." ;
8      org:hasMember ldp:Schaetzl_Matthias,
          ldp:Vetterlein_Guenter ;
9      foaf:homepage <http://www.awo-leipzig-stadt.de> .
10
11 ldp:Moeckel_Christian
12     a foaf:Person ;
13     org:headOf <http://leipzig-data.de/Data/Buergerverein/
          Leutzsch> ;
14     foaf:name "Arne Ackermann" .
15
16 ldp:Knospe_Uta
17     a foaf:Person ;
18     org:memberOf <http://leipzig-data.de/Data/
          Buergerverein/Messemagistrale> ;
19     foaf:name "Uta Knospe" .
```

The above definitions are copied from the Leipzig Data project, along with any inconsistencies in them—e.g. lines 11 and 14 state different names for the same person. This example is based on the second use case and therefore is focused on the Organization Ontology. I now want to adjust SHACL shapes for every class and property I want in the visualization. These are the classes `org:Organization`, `foaf:Person` and the object properties `org:hasMember`, `org:headOf`, and `org:memberOf`. A possible shape description looks like this:

```
1  ex:OrganizationShape
2      a sh:NodeShape ;
3      sh:targetClass org:Organization ;
4      sh:name "Organization" ;
5      sh:property [
6          sh:path org:hasMember ;
7          sh:name "hasMember" ;
8          sh:class foaf:Person ;
9      ] ;
10     sh:property [
11         sh:path ld:hasAddress ;
12         sh:name "hasAddress" ;
13         sh:maxCount 1 ;
```

```
14      ] .
15
16  ex:PersonShape
17      a sh:NodeShape ;
18      sh:targetClass foaf:Person ;
19      sh:name "Person" ;
20      sh:property [
21          sh:path org:memberOf ;
22          sh:name "memberOf" ;
23          sh:class org:Organization ;
24      ] ;
25      sh:property [
26          sh:path org:headOf ;
27          sh:name "headOf" ;
28          sh:class org:Organization ;
29      ] .
```

The `ex:OrganizationShape` gets the `org:Organization` class as its target and defines two property shapes, the first aimed at the `org:hasMember` property, saying its value has to be a `foaf:Person` using `sh:class`, and the second at the `ld:hasAdress`, a datatype property—included to give an example for that as well—, limiting it with `sh:maxCount` to only always have one value. Similarly, the same is done in the `ex:PersonShape`. More restrictions could be made, but this is already enough for a simple adjustment. The targeted classes will be available and the object properties will be visualized. Creating property shapes for datatype properties may not be important for visualization, but is needed for creating the editing form for a resource. A shape adjustment, comparable to this example, will be required for both use cases, SKOS and Organization Ontology.

4.2 User Interface

Designing a user interface for the visual editor was an important task from the very beginning. It not only determines the whole feel of the user experience, but it is also a welcome tool for testing. Every feature for editing and exploring has to be accessible, while at the same time leaving enough room for the visualization, which also is an essential part needing specification. For that reason a ux design phase was put in, resulting in the mock-up[5] shown in Figure 4.3. The main layout was a personal idea, I then found approved by De Meester et al. in "Towards a uniform user interface for editing data shapes", where the unSHACLed prototype interface is described, see Figure 3.1 [13]. Because of the similar ideas, besides proving the concept, little inspiration could be gained. An example for such influence is the

[5]Created using the mock-up service Balsamiq Cloud: `https://balsamiq.com/` .

usage of accordions[6] in splitting the sidebar. Following I now want to give an overview of the editor's parts using Figure 4.3. After that, I will express further thoughts on the desired user experience.

Sidebar (Figure 4.3, a) The sidebar contains three segments, the first being the search bar. It is designed as an input field with suggestions showing up as a list. Selecting a result will center the view on the chosen resource in the editing area, as described in Requirement PM-377. The second segment holds the stencils used to load and create resources (PM-364). As mentioned earlier, comparable to the unSHACLed prototype, there are two accordions separating the stencils into groups, conforming to PM-374. These listings then consist of different stencils for shapes in the form of buttons to drag-and-drop onto the editing area. As the last segment, the sidebar accommodates the **Outline** (Figure 4.3, d), providing a smaller view of the editing area, one can use to navigate the actual editing area (PM-364). This idea was taken from draw.io[7], an online diagram software.

Toolbar (Figure 4.3, b) Explaining this part of the editor is straightforward. The toolbar is located above the sidebar and editing area, stretching over the whole space from left to right. Situated directly on top of the sidebar is the zoom control, enabling to zoom in, out, and revert to 100% by clicking the middle button, which shows the current zoom factor. From these three buttons onward, a sequence of buttons is placed, giving access to various features—as listed in PM-366.

Editing Area (Figure 4.3, c) The biggest rectangle represents the editing area. Navigating the area and drag-and-drop with a button from the sidebar allows the user to visualize and explore resources from the data graph. Principal part of the editing area is the **Visualization** (Figure 4.3, e). Resources are represented as rectangles with their label and references between them are sketched as arrows, also attached with the corresponding label. A resource can be selected by clicking it, making it highlighted—in the mock-up its border is colored blue.

Aesthetically important for the visualization is the layout. In "The Aesthetics of Graph Visualization" authors Bennett et al. give valuable insights regarding this topic: (i) nodes should be distributed evenly, should not overlap, and have a minimum distance from each other, while still keeping related ones closer together, and (ii) edges should have minimum crossings, bend in a uniform way, have minimum length and should be distributed at even angles around the same node [7]. The goal for the layout is to comply with as many of these rules as possible to facilitate comprehension. Since the structure of data graphs may vary completely, regarding requirement PM-376, it would be best to implement different layout options to choose from—e.g. a circular, hierarchical, or force directed layout. Thus, the user will be able to execute his preferred layout.

In general, the workflow in the editor is heavily dependent on the user experience regarding the editing area. A simple but profitable way of optimizing the flow is

[6] "An accordion allows users to toggle the display of sections of content" [38].
[7] Available at: `https://www.draw.io/` .

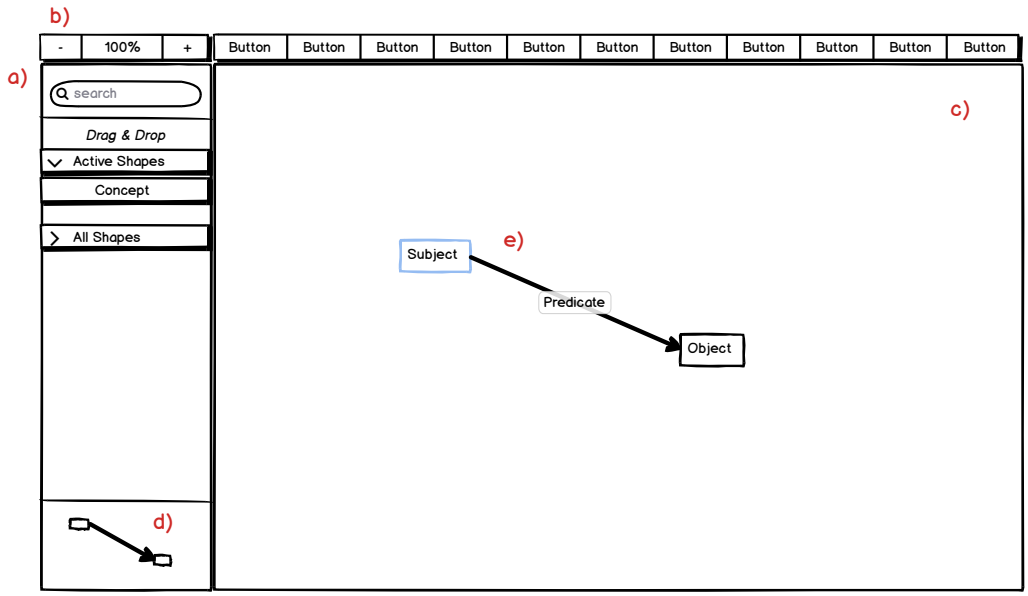

FIGURE 4.3: A user interface mock-up created in the design phase of development.

sticking to "the visual information seeking Mantra: [(i)] overview first, [(ii)] zoom and filter, [(iii)] then details on demand" [33]. This is translated to the editor in the following fashion: (i) for an overview only labels of resources are present in the visualization and the view shows the whole visualized graph with a layout applied, (ii) the user can explore by zooming or navigating the view, moving it to the desired parts of the graph to filter, while also having the option to filter using the search bar, (iii) finally, details for a specific resource can be retrieved by editing it or loading its relations into the editing area.

Chapter 5

Implementation

The implementation chapter sets forth the process and results of development based on the requirements and specification. Matters revolve around the technology stack used, the development process, and the created workflow.

5.1 Technology Stack

The technology stack or tech stack is a summarizing term for the "frameworks, languages, and software products" used to develop an application [27]. These offer fundamental functions and implementations to build the editor on. I, therefore, want to to give an overview of the stack, before laying out the realization of the editor.

The stack consists of the technology already in use by eccenca GmbH before starting this thesis and the needed additions. To start at the basis, the general programming language is JavaScript (JS)[1], in combination with the library Lodash[2]. Syntactically Awesome Style Sheets (Sass)[3] operates as the style sheet language. The stack is mainly based on different libraries, each providing essential functionality. One of the biggest roles in the user-interface-heavy development is reserved for the library ReactJS. It supplies features for building the interface itself, dividing it into multiple components, while also serving some data management abilities between them [16]. Speaking of data, the Reactive Extensions for JavaScript (RxJS) "is a library for reactive programming[4] using observables that makes it easier to compose asynchronous or callback-based code" [14]. Stated operations are used in communication with the database and in processing the yielded data. Up to this point, there was no addition to the existing eccenca Gmbh stack. What is missing, is a diagramming library to draw the visualization in the editing area. At first, the Rappid toolkit looked promising, but to avoid the license fee, only their open source core library JointJS could be applied. Unfortunately, many desperately wanted features are missing in that. After further research, I uncovered the mxGraph JS

[1] "JavaScript is an interpreted programming language with object-oriented capabilities" [17].

[2] "A modern JavaScript utility library delivering modularity, performance & extras" [34].

[3] "Sass is an extension of [Cascading Style Sheets] that adds power and elegance…" [21].

[4] "[A programming paradigm that] tackles issues posed by event-driven applications by providing abstractions to express programs as reactions to external events…" [5].

diagramming library, which constitutes the base of the already mentioned draw.io project [1]. mxGraph is open source and provides more features than its contender JointJS, while also offering draw.io as a working software example. The highly customizable nature of the library's functionality is a blessing, while also being a curse. A blessing regarding the fact everything can be adjusted for one's own needs and a curse because many features still have to be implemented.

Having practiced with the larger part of the stack, eccenca GmbH is able to deliver numerous ready-to-use solutions from the eccenca Corporate Memory. Regarding the back-end, this includes the needed triplestore database and its connection to the component through authorization and JxJS functionality. Moreover, many ReactJS components—e.g. small elements like a search bar or larger components like the complete form for editing a resource—and several other features, implemented in JS—, are available. A useful fountain to draw from, but still, the majority of the editor has to be realized, especially concerning the mxGraph library. Consequently, the next section concentrates on the actual implementation.

5.2 Development Process

The development process was carried out in an agile manner, also installing methods from Kanban[5] and Scrum[6]. Kanban delivered the Kanban board[7] as well as the more open nature regarding iterations—especially irregular lengths—and team size[8] [29]. While Scrum was applied regarding the backlog—e.g. prioritization and story points—and also provided the idea of regular meetings—i.e. weekly scrums [20]. By utilizing these methods and a general agile mindset, an iterative and incremental development approach was achieved. Ongoing planning and requirement engineering shaped a process, continuously changing and evolving the software. Advancement was made on a feature-by-feature basis, expanding the editor, while keeping a working state every increment.

The goal was to implement a software solution for the given use cases, resorting to the requirements catalog and the specification. To present the results, I first want to refer to Figure 5.1, which lays out the architecture. The design pattern is inherited from the component structure of ReactJS, in which parts of the actual user interface are separated as single components, which can then be displayed in an other one. The rectangles in the diagram are either such components or helpers, which are JavaScript files that don't use ReactJS features but rather provide functionality. The relationship between a component and helpers is labeled as Uses, showing which helpers a component utilizes. In the same way, the relationship between components

[5] "Kanban is a popular framework used to implement agile software development" [29].

[6] "Often thought of as an agile project management framework, Scrum describes a set of meetings, tools, and roles that work in concert to help teams structure and manage their work" [15].

[7] "A kanban board is an agile project management tool designed to help visualize work, limit work-in-progress, and maximize efficiency (or flow)" [30].

[8] Because in this case no whole team was involved.

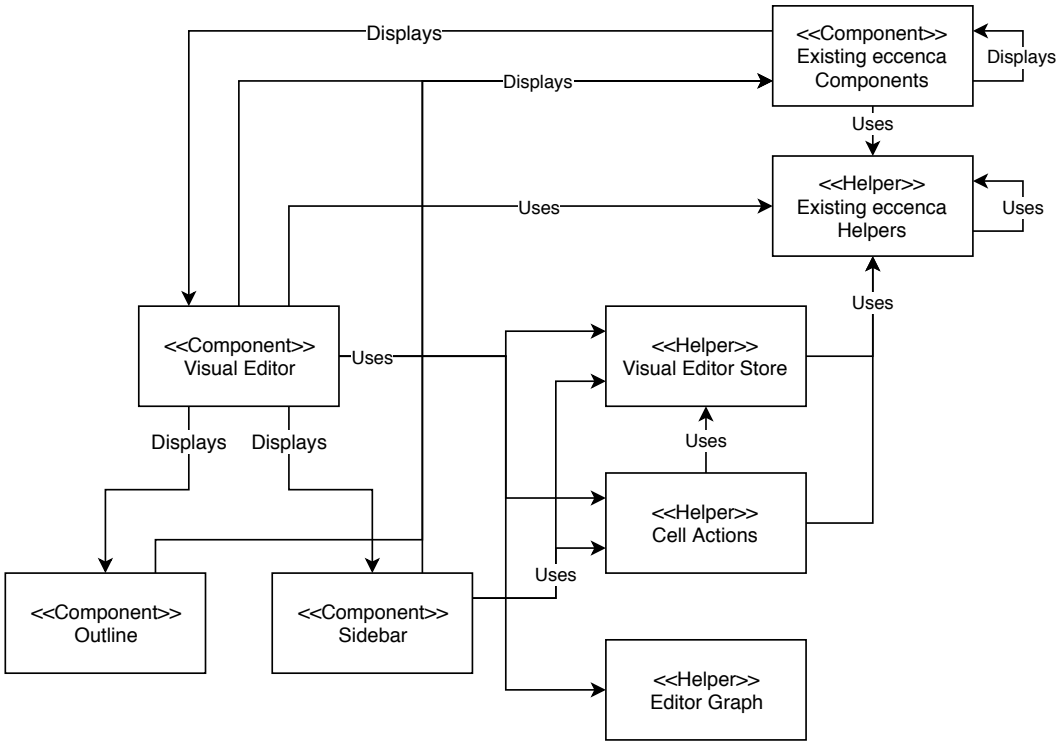

FIGURE 5.1: Basic diagram illustrating the architecture.

is Displays, saying a component is displayed by—i.e. is embedded inside—the other one. The displaying component is called parent and the displayed one its child. To better explain the different parts of the software, I will focus on one part at a time.

Existing eccenca Helpers This represents a set of helpers, containing functionalities already developed by eccenca GmbH. These are for querying the database and working with the requested data.

Existing eccenca Components Next to many smaller user interface elements, like different buttons, some larger existing components could be used. These are the editing form for a resource as well as the search bars and the resource creation form. Furthermore, the editor itself is embedded in a larger eccenca component.

Visual Editor Store In this helper, the channel for the database is defined and a connection is established. The created channel listens for requests, which can be made from other files by importing the channel from the helper.

Cell Actions Every resource visualized in the editor is called a cell in the internal representation. Cells are objects, holding the information of a resource. The helper provides different functions revolving around the problem of retrieving and updating this information, e.g. relations for resources. It also contains functions for adding the cells to the editing area. When a resource has to be added as a cell, a database request for its information is made. Together with another request, retrieving its relations, one cell object is created and then visualized. For updating a resource in the editing area, a similar process is happening, but the result is not a new object, instead, the data in the old object is overwritten and the visualization

of the resource is redone. Any changes to visualized relations are updated in the same way. For every change, the resources added or removed from the graph are returned as filter actions, to prevent adding duplicate resources to the editing area.

Editor Graph In this part, the mxGraph library is used to define the environment of the editing area. Only the functions requiring ReactJS features are left out and put in the main component. A mxEditor is a wrapper for several mxGraph classes. Therefore, a new mxEditor containing a mxGraph object[9] is created and several settings for it are adjusted. First cells are set to not interact in any other way than being moved around the view. Edges, which are the arrows representing relations, are made to not interact at all. For the editing area itself, the panning is activated, enabling the user to move the view. Styles for the cells and edges are defined and activated. Except for the colors and background colors, the default style is copied.

Visual Editor This is the main component of the editor. The visual editor component is displayed by an enclosing eccenca component and gets the Uniform Resource Identifier (URI) of the data graph in the props[10] object. In the same way, it displays two sub components, Sidebar and Outline. These two sub components, cover the functionalities regarding the stencils in the sidebar and the employment of the outline. The component also defines variables in its state object. The component will be re-rendered in the view when any of these variables changes. Also, they are available for use in any method of the component. When mounted, this means "React has finished the initial rendering of [the] component," the editor is configured through calling the statements from the Editor Graph helper and the `sh:NodeShapes` for the sidebar are retrieved [16].

For the first action, the mxEditor object is initialized, added to the state object, and two listeners are appended to it. One to start the editing using a double-click and a second one for starting the layout automatically, when a new resource is visualized. Two methods are used to layout the graph, with the first applying a mxCircleLayout on the mxGraph object and the second fitting the view in the editing area—by moving the view and changing the zoom—to show the whole visualized graph. Also, the zoom method for the graph is defined and the zoom value is added to the state object.

The node shapes are retrieved using a SPARQL query in combination with the methods from the Visual Editor Store helper. The query filters the returned node shapes and only adds those, that have a relevant `sh:targetClass` in the data graph. The results are put together as an array. Are there no node shapes for the data graph, a hint for the user is given in the sidebar, which then stays empty. Otherwise, the array is given to the Sidebar component to display them as stencils.

The editing of a resource visualized in the editing area is managed through a state variable. This variable is changed to display the editing popup when the user

[9] mxGraph is also the name of a class.

[10] "The props parameter is a JavaScript object passed from a parent element to a child element..." [16].

wants to start editing a resource. In the same fashion, it is changed to close the popup. The popup itself, contains an eccenca component, delivering a complete editing form for the given resource. The adding of a relation to a resource is done analogously to the editing, but only displays that part of the editing form.

If changes from editing occur, they are handled by utilizing the Cell Actions helper. The corresponding method updates the information for the resource and reloads its visualized relations. In the form of filter actions, changes are also given to the Sidebar component, where the resource filter is located.

Adding and removing an edge level for a resource or the whole editing area is done by calling methods from the Cell Actions helper. The visualized relations for the selected one or for a list of all currently visualized ones are added to or removed from the editing area. The resulting filter actions are again handled in the Sidebar component.

The button row is rendered as simple buttons, that are assigned their corresponding method as an on-click action. Furthermore, the Sidebar and Outline component are embedded in the rendering and given their needed props.

When the component updates and the data graph in the props changed, the editing area is cleared and the node shapes are retrieved anew to update the sidebar accordingly.

Sidebar In the props, the sidebar component gets the array of retrieved node shapes, the filter actions, the mxEditor object, and the URI of the data graph. When mounted, the array of node shapes is used to draw the stencils in the sidebar. For that, a prototype function used to create an object, as the representation of a resource, is defined. It is utilized as a template to create the cells and stores the label of the resource, the URI of the data graph, the URI of the resource, and the information regarding the resource's relations. Furthermore, for every element from the node shape array, a drop function is defined, that can be used with the mxGraph framework, to omit an object, when something is put in the editing area by drag-and-drop. This object contains the mxGraph object, the cell object template, the prototype function to set the templates values, the shape's URI, and data graph's URI. The omitted object will be stored in the Sidebar component's state object and is called drop object. Now, a stencil in the form of a button is created for every shape, then added to the sidebar and finally associated with the corresponding drop function for the shape.

Other than creating the stencils, the sidebar component also manages the different possible interactions with them. Therefore, when dragging a button from the sidebar onto the editing area and dropping it, a popup appears. This popup is handled with a state variable, in the same way as the editing popup from the Visual Editor component. Because of dropping the stencil, the state object now holds the omitted drop object for the shape. The popup has two available views, which can be changed at any time using a switch button. The first view allows the search and selection of an existing resource in the data graph, while the second enables the

creation of a new one. For the selection, a SPARQL query is created, returning the relevant resources from the data graph, while also filtering out resources already present in the editing area. This result is given to an eccenca component, allowing a search and selection out of this list. The creation of a new resource is also handled by an eccenca component, which offers the same editing possibilities as the editing form for editing a resource.

When either the selection or creation of a resource is done, the visualization is realized. For that, the drop object and the resource's URI as well as its label is given to the Cell Actions helper. Here, first of all, the relations of the resource are retrieved. The resource is then added to the graph as a cell, storing the information in the drop objects cell template. This object defines the information, the cell holds and is available in the functions of the mxGraph framework. Then, the edges for the relations are drawn, while also adding the related resources as cells. Finally, the layout is applied to the graph by sending a change event, triggering the function from the Visual Editor component. The ultimately returned filter actions, containing all added resources, are applied in the Sidebar component.

Since the selection of resources is done in this component, resource filtration is also located here. The resource filter is stored in the state object and used to keep track of the resources visualized in the editing area. It is represented as an array and manipulated by adding or removing elements. With the provided information, the resources can be hidden from the search, when adding existing ones from a stencil. The array could also be achieved by a built-in function of the mxGraph framework, that allows getting an array of all cells in the graph. The downsides regarding loading times would be the increase when many resources are visualized and that the whole array has to be completely redone every time. Furthermore, this function provides an overhead of unnecessary information for the cells, while the resource filter only stores the URIs.

When the component updates and the filter actions in the props changed, the resource filter is updated with the new information. If the array of node shapes changed, the sidebar itself is reloaded with the new stencils and the resource filter is cleared.

Outline The outline component takes the mxEditor object in the props. This is needed, to copy the graph from the mxEditor to the mxOutline object. After creating the initial outline from the provided graph object, it is appended to its desired location in the component—the lower left corner of the editor. Also, an update function is defined and a listener is added to the mxEditor object. The listener always updates the outline, when the view changes.

When mounted, the outline is created, as described above, and when the component updates and the mxEditor in the props changed, this process is redone.

5.3 Workflow

In this section, the editor will be described for a user. I will discuss the potential occurring situations and thereby expound the achieved workflow. The explanations are supported by several figures, showing the editor in different states. In these, the catering industry graph is used, showcasing a use case for the SKOS taxonomy management from section 2.1.

The first screenshot in Figure 5.2 presents the first look at the developed user interface. The editor is in its initial state, not visualizing any resources. The sidebar, on the other hand, is already filled with the required stencils. The user may now choose one of them, to drag and drop onto the editing area.

By doing so, a popup for the stencil is opened, as seen in Figure 5.3. It contains a search bar, from which the user is able to find existing resources. Selecting one and using the CONFIRM button then closes the popup and visualizes this resource. At any time before that, the CANCEL button can be used to close the popup without further ado. The switch button at the top left of the popup enables to change from searching existing resources to creating new ones. Activating the switch the popup displays the form seen in Figure 5.4. In this example, the user used the stencil for `skos:Concept` and may now put in the information for the new resource. Pressing the new button SAVE then closes the popup, adds the data to the data graph and visualizes the resource in the editing area.

A visualized resource is shown in Figure 5.5. The resource Bavarian beer was chosen by the user and is visualized with its broader relationship to Beer Bar. This affects the outline in the lower left corner of the user interface, where a miniature version of the editing area is given. The user may zoom or use his left mouse button, dragging the view in the editing area. This exploration can also be done using the outline. A resource can be moved around holding down the left mouse button. In the figure, the layout took effect, arranging the resources and changing the zoom to 200%. This can be achieved at any point, using the LAYOUT button from the toolbar. Other functionalities regarding the whole editing area are the COUNT and ADD EDGE LEVEL buttons. The first is mainly for development purposes in this version of the editor and gives the number of visualized resources, while the second button allows adding the next level of relations to all visualized resources. But there are also options for single resources. At this point, the user selected the resource Bavarian beer, which is indicated by its different border. The ADD/REMOVE EDGE LEVEL FOR SELECTION button enables the user to show or hide the relations for his selected resource. The ADD RELATION FOR SELECTION button opens a popup, in which the user may edit only the relations for his selected resource. The button to the right, EDIT SELECTION, opens the full editing popup for a resource, which is displayed in Figure 5.6. This popup can also be opened by double-clicking a resource in the editing area. Editing a resource, the user can change any information needed and confirm with the SAVE button. The visualization is then updated to

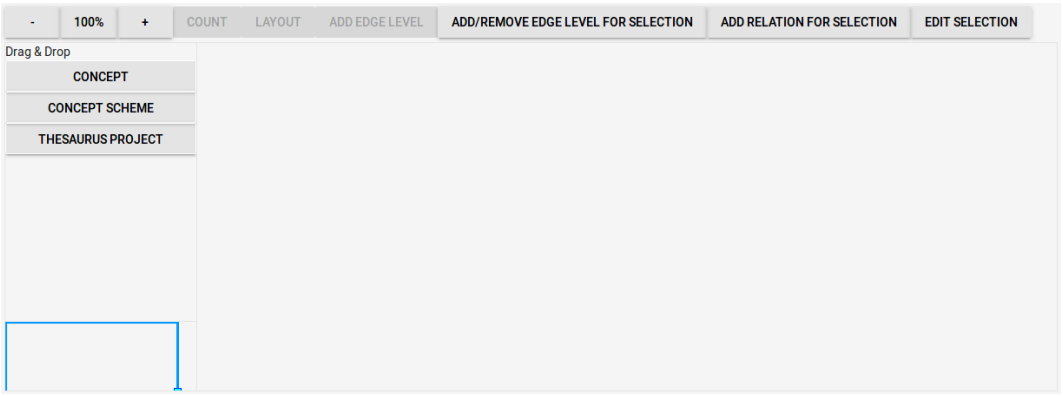

FIGURE 5.2: General view of the editor in initial state.

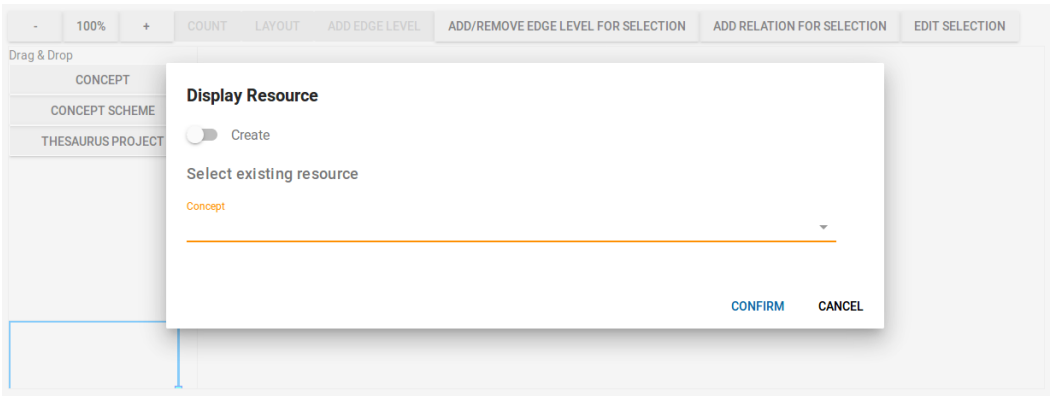

FIGURE 5.3: Selection of an existing resource for visualization.

conform to the changes.

Taking a look at Figure 5.7, the editor is shown after the user pressed the ADD EDGE LEVEL button. Before that, the state from Figure 5.5 was present. Five other resources have been added to the editing area, which are all related to the Beer Bar resource. The circular layout arranges them accordingly.

So, the workflow starts off with the selection or creation of resources, through the stencils from the sidebar, which are then visualized. From that the user is able to explore the data in a graphical way, by moving the view, zooming, or moving resources around the editing area. The user may expand the visualization with further resources, either by using the sidebar or loading relationships. Equally important, single resources in the visualization can be edited at will.

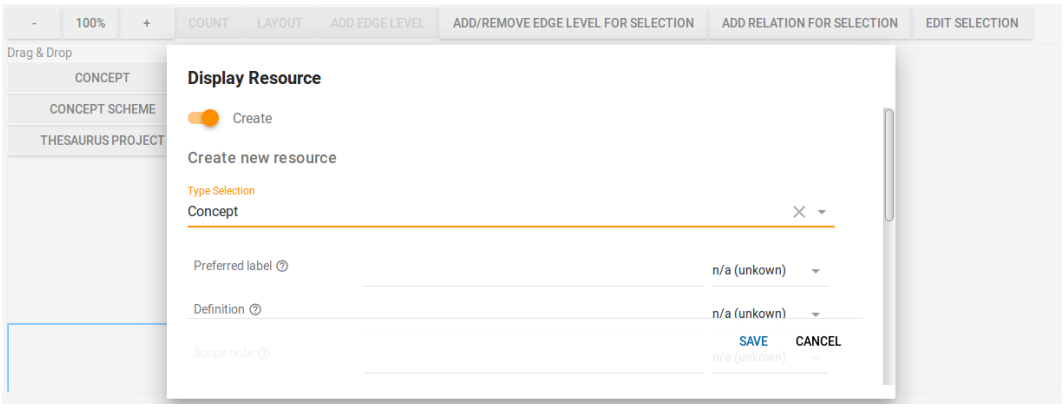

FIGURE 5.4: Creating a new resource instead of selecting an existing one.

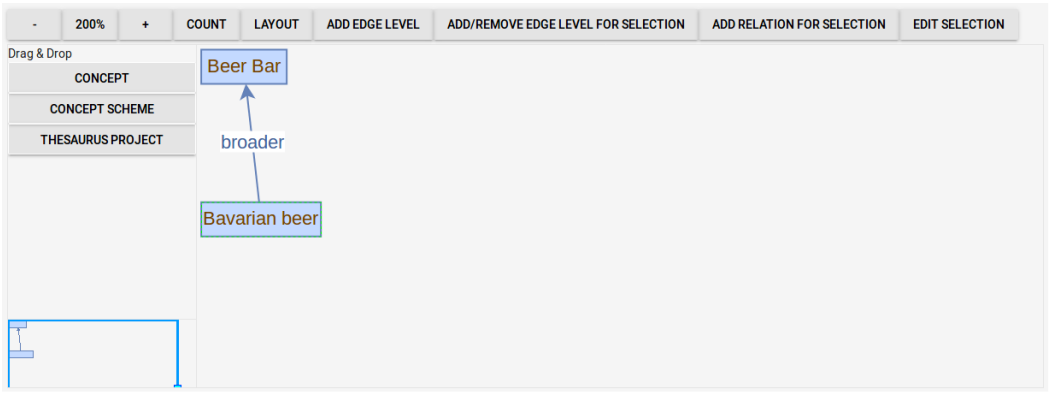

FIGURE 5.5: Visualization of two resources with a relation.

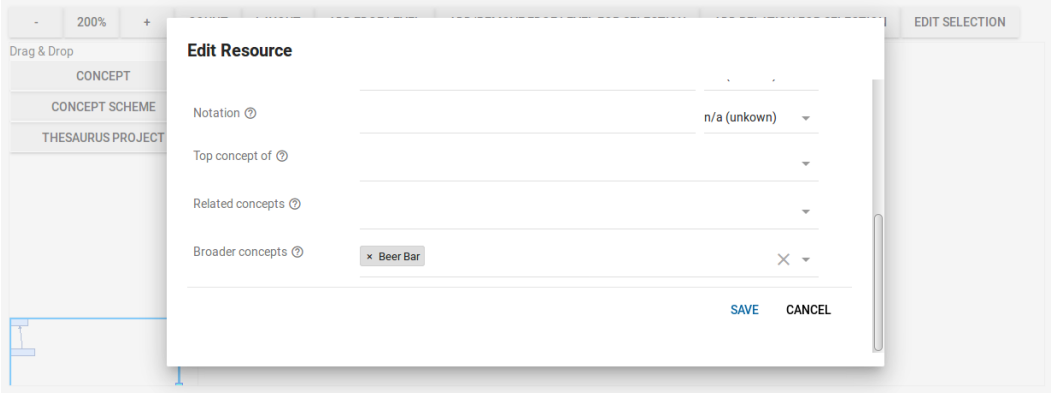

FIGURE 5.6: Popup used to edit a selected resource.

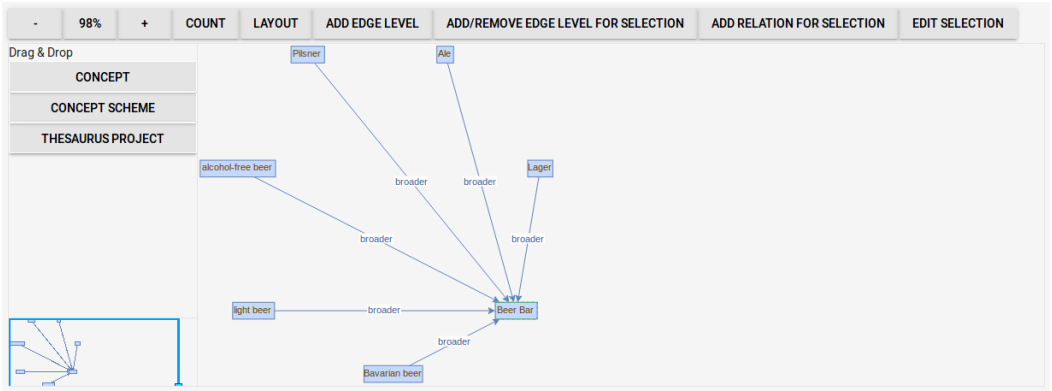

FIGURE 5.7: An edge level has been added, extending the visualization.

Chapter 6

Evaluation and Future Work

In this concluding part, two sections provide a final view on the thesis. I will start by presenting an evaluation of the results and findings given in the previous chapters. The second segment handles the possibilities for future work, focused on expanding the visual editor.

6.1 Evaluation

The main goal of this thesis was the specification and implementation of a visual editor for linked data knowledge graphs, which should be adjustable to specific domains by utilizing SHACL shape descriptions. The work will be rated based on the resulting implementation and how it handles the defined use cases.

For that, I first want to compare the requirements catalog to the actual implementation. In the development process, the issues were assigned a status, either Done or Approved. Approved means it has made it through the requirement engineering process and thus, is in the catalog. Done, on the other hand, says that a requirement is implemented and therefore finished. A full list of all requirements and their status at the end of this thesis can be found in Table A.1 of Appendix A. To get a general idea of how much of the catalog has been completed, I first want to look at Table 6.1. It presents the number of story points that have been accumulated in the two status categories. It thereby offers the insight, that close to two-thirds of all story points have been realized for the editor. This number can be materialized by comparing it to the statistics in Table 6.2, where requirements are divided by their priority and then connected to their status. Here the first entry exhibits, that slightly more than two-thirds of all requirements have been achieved, a finding almost identical to the one from Table 6.1. More data is available, exposing that 92% of major requirements have been met, which reveals that two requirements are missing from the implementation. However, two minor ones are marked as Done. This can be explained by taking a look at the four requirements in question. Since time was short, a decision had to be made, that either PM-362 and PM-363 or PM-367 and PM-368 should be implemented. The first two are the major ones and the editor works without them but loses critical functionality. But they have higher story points ratings and also, the other two requirements are important for

Status	Story Points	Proportion
—	187	100.0 %
Done	123	66.0 %
Approved	64	34.0 %

TABLE 6.1: Progress measured by story points.

Priority	Status		
	Done	Approved	Proportion
—	25	12	67.6 %
Major	23	2	92.0 %
Minor	2	9	18.2 %
Trivial	0	1	0.0 %

TABLE 6.2: Requirement priority levels and their progression.

the integration of the software in the eccenca GmbH environment and enable future work. For that reason, the two minor requirements were implemented before the major ones, which then were left out. Regarding the prototype, major requirements should have been implemented, with the minor ones following, if time is spare. Hence, looking at the minor and trivial issues, the catalog covers more requirements than necessary for the wanted implementation. This explains why almost no minor or trivial features are present in the editor.

A fundamental aspect of this evaluation are the use cases from section 2.1. The aim was to reach a workable state for both use cases. The shape adjustment for the vocabularies was shown—using the second use case—in the Data Management section (4.1) of the Specification chapter. Without this groundwork, no resource or relation would be visible in the editor. Thus, having them visualized proves the adjustment. For the SKOS taxonomy management, the catering service graph was used in section 5.3 concerning the workflow. The explanations and figures show how resources visualized in the editor can be created, edited and linked. The user is able to explore the visualization, confirming a workable state for the first use case. Regarding the Organizational Structure, Figure 6.1 displays a screenshot of the editor with the integrated data graph from the Leipzig Data project. The screenshot and previous explanations verify, that the second use case was achieved as well.

For the workable state, it has also been stated that “a reasonable amount of resources can be visualized”. To test and stress this issue the EuroVoc data graph was used as specified for the SKOS taxonomy management. Figure 6.2 shows a visualization of a part of the EuroVoc graph. In this case, the limitations of the editor were obvious. Loading times were long, moving the view or a resource in the editing area had a delay, and the general browsing experience was negatively affected. So,

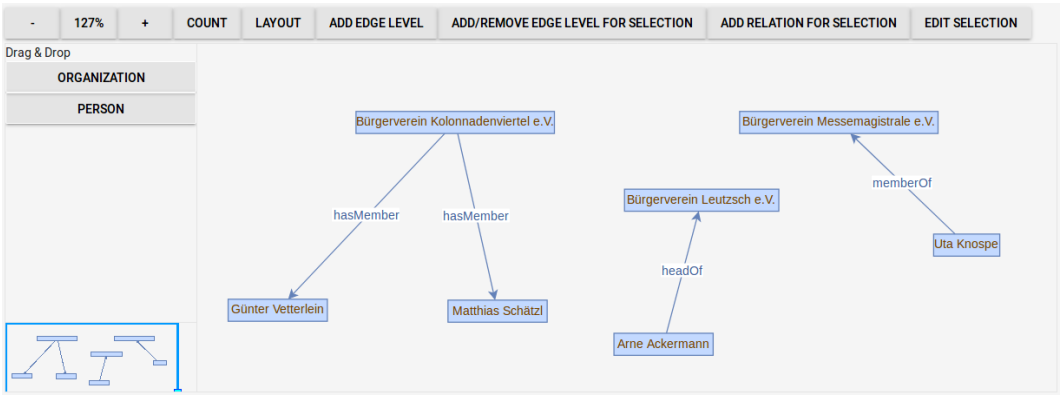

FIGURE 6.1: Organization Ontology data graph—parts from the Leipzig Data project—in the editor.

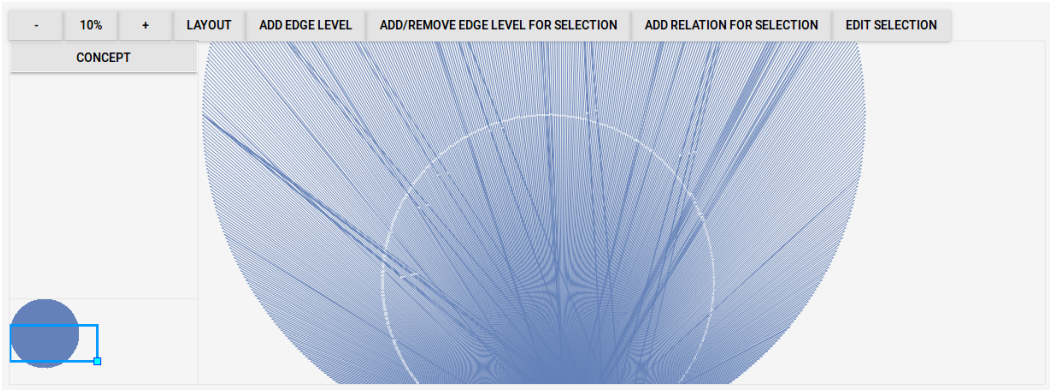

FIGURE 6.2: A screenshot showing the limitations in resource visualization—with EuroVoc data graph.

there are just too many resources in the editing area, which makes computation slow. Furthermore, the visualization is simply confusing—labels are tiny and the distance between resources enormous. With the features from PM-375 and PM-377, exploring and editing would be still possible—even with that many resources—, but the layout is still a hindering factor. Therefore, the layout should be improved and alternatives to the circular layout should be available. The problems with the layout could also be reduced by introducing a new feature, allowing selective visualization of related resources. Unnecessary relations could be eliminated from the visualization, leaving more room for the important parts.

It has been shown that users are able to explore and edit their data in a graphical representation, regarding the set use cases. Therefore, a prototype satisfying the workable state has been achieved. Furthermore, it proves the fact, that the requirements catalog and specification describe a functioning software solution for these tasks. But the evaluation has also shown potential for future work, which is discussed in the next section.

6.2 Future Work

As the evaluation has made clear, the requirements catalog was not fully implemented. Thus, future work should be focused on expanding the editor using this catalog. Shown in Table 6.3 are the requirements that were approved but not implemented in the prototype. The elements in the table are first ordered by the Need column, which divides them into important and less important. They are then ranked by their general influence measured by the provided utility of adding the feature to the editor. The column SP states the story points for each entry. With regard to the evaluation, the first two requirements in the table are the left out major ones. Since every other item has a lower priority—minor, except for the last—these two are at the top of the table. PM-391 and PM-374 follow them and are needed to allow full editing functionality. The last two important requirements—judging by the Need column—help with exploring the visualization in the editing area. Implementing both, permits the user to locate and view a specific resource or part of the visualized graph. PM-392 and PM-390 are less important items because workarounds are present, e.g. simply clearing the editing area. Still, PM-392 is more relevant since it delivers an easy way to keep the editing area clean from the visualization of unwanted resources. PM-389 and PM-387 don't provide extra core functionality and therefore are less important. PM-388 increases accessibility, but its story points are relatively high, making it a less appealing choice. The last entry is the only trivial requirement from the catalog, making it the least favorite improvement. Missing in the list but another finding from the evaluation is that more layout alternatives should be available. This can help with the visualization of different kinds of data graphs—e.g. strict hierarchies—, especially with varying sizes, and one such layout could be inspired by the one in the LodLive project (see 3.2). The selective visualization of related resources could also add to this aspect. This could be implemented as a list of available relations, which can be displayed for a selected resource. The user can then select from the list which relations to show or hide. In a similar way, further development of the editor may create new requirements, extending the catalog and as a result, the options for future work.

Key	Need	SP	Name
PM-362	+	6	Specific resource as starting point
PM-363	+	6	List of specific resources as starting point
PM-391	+	6	Delete
PM-374	+	6	Divide sidebar for active shacl:NodeShapes
PM-375	+	8	Fit specific view
PM-377	+	7	Search for displayed resource
PM-392	-	2	Remove
PM-388	-	6	Gesture and touch commands
PM-389	-	6	Tooltips
PM-387	-	4	Keyboard shortcuts
PM-390	-	4	Force refresh
PM-394	-	3	UI Styling

TABLE 6.3: Approved requirements for future work, ordered by added utility.

Bibliography

[1] Gaudenz Alder. *About*. [Online; accessed 12-February-2019]. URL: https://github.com/jgraph/drawio.

[2] Mohammad Alkandari and Asma Al-Shammeri. "Enhancing the Process of Requirements Prioritization in Agile Software Development-A Proposed Model." In: *JSW* 12.6 (2017), pp. 439–453.

[3] Agile Alliance. *GLOSSARY Role-feature-reason*. [Online; accessed 31-October-2018]. URL: https://www.agilealliance.org/glossary/role-feature/.

[4] Agile Alliance. *GLOSSARY User Stories*. [Online; accessed 31-October-2018]. URL: https://www.agilealliance.org/glossary/user-stories/.

[5] Engineer Bainomugisha et al. "A survey on reactive programming". In: *ACM Computing Surveys (CSUR)* 45.4 (2013), p. 52.

[6] Sean Bechhofer et al. *OWL Web Ontology Language Reference*. Ed. by Mike Dean and Guus Schreiber. World Wide Web Consortium, Recommendation REC-owl-ref-20040210. Feb. 2004. URL: http://www.w3.org/TR/2004/REC-owl-ref-20040210.

[7] Chris Bennett et al. "The Aesthetics of Graph Visualization". In: Jan. 2007, pp. 57–64. DOI: 10.2312/COMPAESTH/COMPAESTH07/057-064.

[8] Diego Valerio Camarda, Silvia Mazzini, and Alessandro Antonuccio. "LodLive, Exploring the Web of Data". In: *Proceedings of the 8th International Conference on Semantic Systems*. I-SEMANTICS '12. Graz, Austria: ACM, 2012, pp. 197–200. ISBN: 978-1-4503-1112-0. DOI: 10.1145/2362499.2362532. URL: http://doi.acm.org/10.1145/2362499.2362532.

[9] Kendall Clark, Elias Torres, and Lee Feigenbaum. *SPARQL Protocol for RDF*. W3C Recommendation. W3C, Jan. 2008. URL: http://www.w3.org/TR/2008/REC-rdf-sparql-protocol-20080115/.

[10] Evita Coelho and Anirban Basu. "Effort Estimation in Agile Software Development using Story Points". In: *development* 3.7 (2012).

[11] Mike Cohn. *Agile estimating and planning*. Pearson Education, 2005.

[12] Richard Cyganiak, David Wood, and Markus Lanthaler. *RDF 1.1 Concepts and Abstract Syntax*. W3C Recommendation. W3C, Feb. 2014. URL: http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

[13] Ben De Meester et al. “Towards a uniform user interface for editing data shapes”. In: *4th International Workshop on Visualization and Interaction for Ontologies and Linked Data*. Vol. 2187. 2018, pp. 13–24.

[14] RxJS Docs. *Introduction*. [Online; accessed 12-February-2019]. URL: https://rxjs-dev.firebaseapp.com/guide/overview.

[15] Claire Drumond. *Scrum*. [Online; accessed 14-February-2019]. URL: https://www.atlassian.com/agile/scrum.

[16] Artemij Fedosejev. *React. js Essentials*. Packt Publishing Ltd, 2015.

[17] David Flanagan. *JavaScript: the definitive guide*. " O'Reilly Media, Inc.", 2006.

[18] Lars Marius Garshol. “Metadata? Thesauri? Taxonomies? Topic maps! Making sense of it all”. In: *Journal of information science* 30.4 (2004), pp. 378–391.

[19] Jose Emilio Labra Gayo et al. “Validating RDF data”. In: *Synthesis Lectures on Semantic Web: Theory and Technology* 7.1 (2017), pp. 1–328.

[20] Luis Gonçalves. “Scrum”. English. In: *Controlling & Management Review* 62.4 (2018), pp. 40–42. ISSN: 2195-8262, 2195-8270. URL: https://doi.org/10.1007/s12176-018-0020-3.

[21] Chris Eppstein Hampton Catlin Natalie Weizenbaum and numerous contributors. *Sass (Syntactically Awesome StyleSheets)*. [Online; accessed 12-February-2019]. URL: https://sass-lang.com/documentation/file.SASS_REFERENCE.html.

[22] Ivan Herman, Guy Melançon, and M Scott Marshall. “Graph visualization and navigation in information visualization: A survey”. In: *IEEE Transactions on visualization and computer graphics* 6.1 (2000), pp. 24–43.

[23] Gail Hodge. *Systems of Knowledge Organization for Digital Libraries: Beyond Traditional Authority Files*. ERIC, 2000.

[24] R. Mazhar Iqbal and A. Waleed Abbasi. “Requirement Engineering Process in Agile Software Development: Review”. In: *Research Journal of Computer and Information Technology Sciences* 2.5 (Dec. 2014), 1–15.

[25] Dimitris Kontokostas and Holger Knublauch. *Shapes Constraint Language (SHACL)*. W3C Recommendation. W3C, July 2017. URL: https://www.w3.org/TR/2017/REC-shacl-20170720/.

[26] Alistair Miles and Dan Brickley. *SKOS Core Guide*. W3C Working Draft. W3C, Nov. 2005. URL: http://www.w3.org/TR/2005/WD-swbp-skos-core-guide-20051102/.

[27] Dave Nevogt. *What Is a Tech Stack and Why Do I Need One?* [Online; accessed 12-February-2019]. Aug. 2017. URL: https://blog.hubstaff.com/technology-stack/.

[28] Steven Pinker and R Feedle. “A theory of graph comprehension”. In: *Artificial Intelligence and the Future of Testing* (Jan. 1990), pp. 73–126.

[29] Dan Radigan. *Kanban*. [Online; accessed 14-February-2019]. URL: https://www.atlassian.com/agile/kanban.

[30] Max Rehkopf. *What is a kanban board?* [Online; accessed 14-February-2019]. URL: https://www.atlassian.com/agile/kanban/boards.

[31] Dave Reynolds. *The Organization Ontology*. W3C Recommendation. W3C, Jan. 2014. URL: http://www.w3.org/TR/2014/REC-vocab-org-20140116/.

[32] Guus Schreiber and Yves Raimond. *RDF 1.1 Primer*. W3C Note. W3C, June 2014. URL: http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/.

[33] B. Shneiderman. "The eyes have it: a task by data type taxonomy for information visualizations". In: *Proceedings 1996 IEEE Symposium on Visual Languages*. Sept. 1996, pp. 336–343. DOI: 10.1109/VL.1996.545307.

[34] Jean-Philippe Sirois and Zack Hall. *Lodash*. [Online; accessed 12-February-2019]. URL: https://lodash.com/.

[35] Ian Sommerville et al. *Software engineering*. Boston: Pearson, 2011.

[36] Ed Summers and Antoine Isaac. *SKOS Simple Knowledge Organization System Primer*. W3C Note. W3C, Aug. 2009. URL: http://www.w3.org/TR/2009/NOTE-skos-primer-20090818/.

[37] TopQuadrant. *SHACL TUTORIAL: GETTING STARTED*. [Online; accessed 11-February-2019]. URL: https://www.topquadrant.com/technology/shacl/tutorial/.

[38] Semantic UI. *UI Docs: Accordion*. [Online; accessed 06-February-2019]. URL: https://semantic-ui.com/modules/accordion.html\#/definition.

[39] Marcia Lei Zeng. "Knowledge organization systems (KOS)". In: *KO KNOWLEDGE ORGANIZATION* 35.2-3 (2008), pp. 160–182.

Appendix A

Requirement Status

Key	Done	SP	Name
PM-358	+	5	Create new resources
PM-359	+	5	Load existing resources
PM-360	+	5	Start from scratch
PM-361	+	5	Edit resources
PM-364	+	5	Sidebar with stencils
PM-365	+	7	Adjust to specific domains by utilizing sh:NodeShapes
PM-366	+	3	Toolbar with Buttons
PM-370	+	4	Zoom
PM-371	+	4	Panning
PM-372	+	2	Move displayed resources
PM-373	+	3	Load graph
PM-376	+	9	Layout graph
PM-378	+	7	Add relation
PM-379	+	10	Load relation
PM-380	+	5	Update editing area
PM-381	+	5	Add edge level
PM-382	+	7	Remove edge level
PM-383	+	8	Outline
PM-393	+	1	Clear
PM-367	+	4	Directory and component structure
PM-368	+	4	Linter conformity

Continued on next page

Table A.1 – continued from previous page

Key	Done	SP	Name
PM-369	+	6	Example data
PM-384	+	4	Triple-Store backend
PM-385	+	3	OAuth 2 support
PM-386	+	2	Database connection
PM-362	-	6	Specific resource as starting point
PM-363	-	6	List of specific resources as starting point
PM-374	-	6	Divide sidebar for active sh:NodeShapes
PM-375	-	8	Fit specific view
PM-377	-	7	Search for displayed resource
PM-387	-	4	Keyboard shortcuts
PM-388	-	6	Gesture and touch commands
PM-389	-	6	Tooltips
PM-390	-	4	Force refresh
PM-391	-	6	Delete
PM-392	-	2	Remove
PM-394	-	3	UI Styling

TABLE A.1: All requirements with their status and story points.

Declaration of Authorship

"Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann".

Leipzig, 2019 

Ort Datum Unterschrift