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

MARTIN BUCHALIK

BACHELOR THESIS

RICH VISUALIZATIONS FOR INNER SOURCE PATCH-FLOW

Submitted on 13 May 2019

Supervisor: Prof. Dr. Dirk Riehle, M.B.A.; Maximilian Capraro, M.Sc.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

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

Erlangen, 13 May 2019

License

Erlangen, 13 May 2019

Abstract

"Inner source" is the application of open source methods within a company's boundaries (Capraro & Riehle, 2017; Riehle, Capraro, Kips & Horn, 2016). Every employee is able to access and contribute to any software component developed within that company. This creates an open source-like environment inside the organization (Capraro & Riehle, 2017) and makes the interaction between the departments much more intertwined. To analyze these interactions, a software application called "CMSuite" has been developed by the Professorship of Open Source Software at Friedrich-Alexander University Erlangen-Nürnberg.

Currently, CMSuite allows the user to explore the aggregated data sets using a tabular view. However, with larger data sets, the user loses the ability to quickly understand the relationships between the involved parties.

To solve this problem, this thesis describes the implementation of multiple types of graphical visualizations into the Angular frontend of CMSuite. After a comparison of various visualization libraries, the integration of the best-suited one into the existing structure of CMSuite is explained.

As a result of this thesis, CMSuite users are now able to analyze large data sets with much less effort. Also, the generated visualizations can, for instance, be shown in presentations to non-technical users without the need to manually create simplified diagrams of the usually hard to interpret data sets.

Contents

1 Introduction

"Inner source" (IS) is the application of open source (OS) techniques within a company's boundaries (Capraro & Riehle, 2017; Riehle et al., 2016).

The source code of the software components developed within that company is made accessible for every employee, not only for the organizational unit responsible for its maintenance. Members of other units are able to contribute to these components, similarly to how it is done in open source software (OSS) projects: After a modification has been suggested, a committer (i.e. an employee responsible for the particular project) may review the proposed changes and decide whether they should be applied or not. Typically, the individuals involved are also able to openly communicate about the work. (Capraro, Dorner & Riehle, 2018; Stol, Avgeriou, Babar, Lucas & Fitzgerald, 2014) This creates an open source-like environment within the organization (Capraro & Riehle, 2017).

It is important to note that the source code is only made accessible internally (Dinkelacker, Garg, Miller & Nelson, 2002), which is the key difference to OSS.

Since IS leads to a much more intertwined interaction between all departments, it becomes hard to understand which parties actually collaborate with each other. To solve this problem, it is necessary to measure the "patch-flow". Patch-flow is "the flow of code contributions across organizational boundaries such as project, organizational unit, or profit center boundaries" (Capraro et al., 2018). To analyze the patch-flow, a software application called "CMSuite" has been developed by the Professorship of Open Source Software at Friedrich-Alexander University Erlangen-Nürnberg.

Prior to this thesis, CMSuite only included a tabular overview of the aggregated patch-flow[1] data sets. For larger amounts of data, it is hard for the user to understand the relationships between the involved parties. Also, to present the results to other individuals (e.g. managers), diagrams need to be created "manually" with the help of third-party software.

[1]CMSuite is able to display various kinds of economic events. At the time of writing this thesis, the focus of CMSuite relied on the analysis of patch-flow data, which is one example of economic event data.

The overall goal of this thesis is to provide rich visualizations of the patch-flow data. The added functionality should enable the user to explore the patch-flow with the help of diagrams that are generated in the browser. These diagrams are designed in a way that even non-technical users should be able to interpret large data sets without much effort.

This thesis is structured as follows. Chapter 2 explains the most important terms needed to understand the rest of this thesis. Chapter 3 lays out the functional and non-functional requirements the code written for this thesis is supposed to fulfill. In chapter 4, a rough overview of the most important structural decisions is given. Chapter 5 covers interesting implementation details. Finally, chapter 6 evaluates whether and in what degree the requirements defined in chapter 3 were met in the final product.

2 Fundamentals

In the following chapter, the most important terms and structural elements used in this thesis are explained.

2.1 Short overview of CMSuite

CMSuite is divided into two functional parts: A **server** and a **client**.

The server is able to extract contribution data from multiple repositories using its “patch-flow crawler”. The relevant pieces of data are stored in a PostgreSQL database. To allow the client to access the data sets, the server provides a JSON API that has been written in Java, utilizing the Jersey framework.

The client, created in Angular 6 and TypeScript, is the main focus of this thesis, as the server already provides the necessary functionality needed to visualize the contribution data. The responsibility of the client is, amongst others, to display the data in a way that allows users with no programming knowledge to understand the patch-flow.

2.2 Chart types

In this thesis, two different types of charts are used to visualize patch-flow: Sankey Diagrams and Graphs.

2.2.1 Sankey Diagrams

Sankey Diagrams visually represent the flow of data (or similar concepts) from sources to targets.

There are different types of Sankey Diagrams. Figure 2.1 shows an example of one of the most common variants that is also used in this thesis. It can be described as follows.

Such a diagram consists of two essential pieces: nodes and links. Nodes (in figure 2.1 shown as dark thin rectangles) represent sources or targets of data flow. Links (in figure 2.1 shown as light-colored wide polygons) "connect" two nodes each. If a link exists between two nodes A and B, it can be interpreted in a way that some data, originating in A, has flown into B (or vice versa).

It is common to use different link and node heights in order to visualize proportions between the different amounts of the measured data.

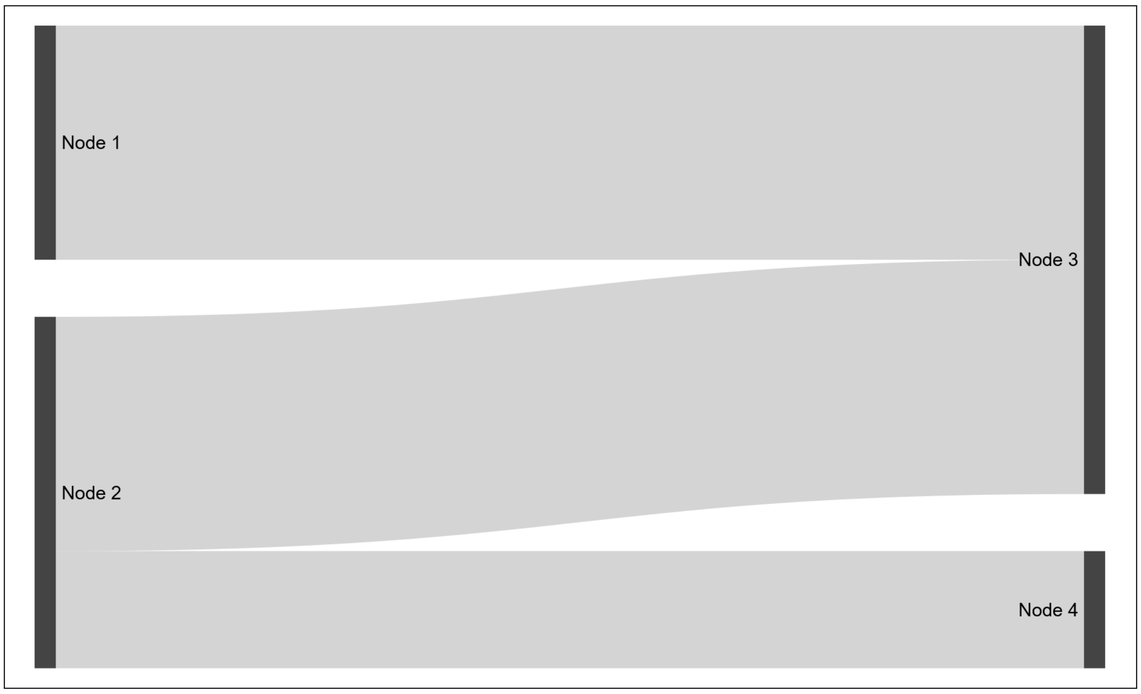


Figure 2.1: An example of a Sankey Diagram as used in this thesis.

2.2.2 Graphs

Graphs, as used in computer science and mathematics, visually represent the interconnection between multiple entities. An example of a simple Graph can be found in figure 2.2.

In general, Graphs are made of two types of elements: nodes and edges. Nodes (in figure 2.2 shown as circles) represent entities. In this thesis, an entity is one potential contributor or receiver of patch-flow.

The nodes may be connected with edges (in figure 2.2 shown as straight lines).

A Graph can be undirected or directed. For two nodes A and B, an undirected Graph provides the user with the information that some kind of connection exists between these two nodes if an edge has been drawn between A and B. A directed Graph additionally shows the “direction” of the data flow, in many cases indicated by an arrow at the end of the edge pointing towards the target of the data flow.

Often, to represent the “weight” of different edges (e.g. the relative amount of data flow), the width of the edges is varied: The higher the weight, the larger the width and vice versa.

The position of a node may be selected randomly. However, for the user to better understand the relations between individual nodes, it is often useful to place nodes that share many edges closer to another, while nodes not sharing edges are drawn further apart.

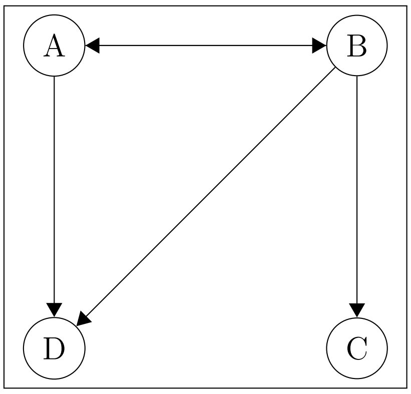


Figure 2.2: An example of a directed Graph as used in this thesis.

2.3 Angular

In CMSuite, the frontend is built based on the Angular[1] framework. This framework has been developed by Google and is licensed under the MIT License.

Angular allows the developer to build web applications using TypeScript[2], HTML and CSS, provides a rich set of ready-to-use TypeScript libraries and includes support for the creation of unit and integration tests.

[1]https://angular.io, accessed on March 27, 2019
[2]https://www.typescriptlang.org, accessed on March 27, 2019

2.3.1 Modules

An Angular project is divided into multiple Modules that are created by the programmer. A Module “groups” pieces of code that are responsible for similar tasks. Each Module consists, among others, of Components and Services.

2.3.2 Components

Components are used to display content to the user. This content is created and styled using HTML and CSS. A Component may embed other Components and exchange data with them. Also, Components may use Services to fetch and process data.

2.3.3 Services

Services are used when HTTP calls, complex calculations or other tasks that are not directly related to the visual representation within a Component need to be performed. This shows that Angular provides a clear separation of concerns: Components are responsible for the visual parts of the application. When, for instance, a network request needs to be made, these Components use Services that provide the necessary data.

3 Requirements

3.1 Purpose and description

CMSuite currently only includes a tabular journal view to display the patch-flow data (or other types of economic events). All contributions related to an organizational unit or project are listed and can be filtered using a menu. The goal of this thesis is to create a visual representation of IS patch-flow.

3.1.1 Desired functionality

Besides the already existing journal view, it should be possible to explore the data using Sankey Diagrams and Graph representations. The visualizations should be "rich", not "static", by allowing basic interactivity. For instance, hovering a node with the cursor should reveal additional information or functionality. Also, users should be able to use filtering mechanisms that are similar to the journal view. By using different colors and sizes of nodes and edges, it should be possible for the user to easily find the parts of the data that are most important for him.

Since most of the server-side infrastructure already exists, the main focus relies on the client side.

3.1.2 Motivation for added functionality

The journal view is a good starting point for users who want to see each and every action in detail. However, with larger data sets, it quickly becomes hard to "understand" the data, i.e. to quickly see which parties provided the most contributions, which parties did not receive many contributions from others, etc. This problem can be solved by providing a useful visual representation of the patch-flow.

3.2 List of requirements

Before starting the actual implementation, it is necessary to define the requirements that later need to be met by the final product. At the end of this thesis, an evaluation of these criteria will be made.

In order to allow a fine-grained evaluation of the requirements, a hierarchical naming/numbering scheme is used. If an element x of the list requires n additional tasks to be completed, these items will be listed as “children”, having the reference numbers x.1. to $x.n$.

Each element of this list consists of a short summary and an optional explanation.

The requirements are stated as follows.

1. **Make it possible for users to switch between the different presentation types.** At the moment, only a journal view exists, which shows the data as a table. Later, it should be possible to switch to the “Sankey” and “Graph” views.

   1.1. **Add the necessary menu(s).**

   1.2. **The navigation should be easy to use.** This can be evaluated by measuring the number of clicks that are required to switch to a different view. A user should not need more than two clicks to switch the view.

   1.3. **The user should easily (i.e. without having to perform an interaction with the UI) get a visual feedback on which type of view he is currently using.** This is not mandatory since the actual view will also be shown. However, a feature like this could make it possible for new users (who are not sure about the terminology) to navigate through the different elements with more confidence.

2. **Display the Sankey view.**

2.1. **Add the necessary menus to the view.** Users should be able to change/filter the displayed data similarly to how it is done in the journal view. It should be evaluated whether re-using the already existing menu components is possible.

2.2. **Find an open source library to display the Sankey Diagrams.** If no suitable library could be found, it needs to be implemented from scratch.

2.2.1. **The library should include all basic functions for data displaying and user interaction.** This includes basic styling of nodes and links (e.g. setting the background colors) and visualization of link weights by setting different heights. User interactions should be possible through click, right click and hover handlers.

2.2.2. **If features are missing in the library, they need to be implemented separately.**

2.2.3. **The library must have an open source license that is suitable to the rest of CMSuite.**

2.3. **Integrate the library into CMSuite.**

2.4. **Provide coloring/highlighting of nodes and links.**

2.5. **Visualize the weight of the links and their corresponding nodes.**

2.6. **Make it possible to "zoom" into nodes**, i.e. show the underlying organizational units that belong to the node the user selected.

2.7. **React to data changes.**

2.7.1 **When data changes (e.g. by a new user selection from the menu defined in 2.1.), update the diagram accordingly.**

2.7.2. **If possible, give visual feedback about the changes.** This can be done using animations and transitioning effects.

2.7.3. **Only change the position of those nodes that are affected by the data change.** Try to keep all other nodes at their position. In this way, it is easier for the user to understand which elements changed.

2.8. **It should be possible to export the currently seen diagram**, e.g. as image or PDF file.

2.9. **It should be possible to store the currently seen view so that it can later easily be recreated.**

3. **Display the Graph view.**

   3.1. **Add the necessary menus to the view.** Users should be able to change/filter the displayed data similarly to how it is done in the journal view. It should be evaluated whether re-using the already existing menu components is possible.

   3.2. **Find an open source library to display the Graphs.** If no suitable library could be found, it needs to be implemented from scratch. If possible, use the same library as in 2.2., since a more consistent code structure can be expected when using the same library for both tasks.

      3.2.1. **The library should include all basic functions for data displaying and user interaction.** This includes basic styling of nodes and edges (e.g. setting the background colors) and visualization of edge weights by setting different widths. User interactions should be possible through click, right click and hover handlers.

      3.2.2. **If features are missing in the library, they need to be implemented separately.**

      3.2.3. **The library must have an open source license that is suitable to the rest of CMSuite.**

   3.3. **Integrate the library into CMSuite.**

   3.4. **Provide coloring/highlighting of nodes and edges.**

   3.5. **Visualize the weight of the edges.**

   3.6. **Make it possible to "zoom" into nodes**, i.e. show the underlying organizational units that belong to the node the user selected.

   3.7. **React to data changes.**

      3.7.1 **When data changes (e.g. by a new user selection from the menu defined in 3.1.), update the diagram accordingly.**

      3.7.2. **If possible, give visual feedback about the changes.** This can be done using animations and transitioning effects.

      3.7.3. **Only change the position of those nodes that are affected by the data change.** Try to keep all other nodes at their position. In this way, it is easier for the user to understand which elements changed.

   3.8. **It should be possible to export the currently seen diagram**, e.g. as image or PDF file.

   3.9. **It should be possible to store the currently seen view so that it can later easily be recreated.**

4. **The code should have a high quality.**

   4.1. **The code should be formatted according to the guidelines of CMSuite.** This can be tested using the linter.

   4.2. **All important functions/methods should have a comment documenting their parameters and purpose.** Methods inherited from other classes (or similar) don't need to be documented.

   4.3. **For all components and services, there should be a test coverage of more than 90%.** Since the work done in this thesis extends an already existing product, it is sufficient only to create test cases for new components. Already existing files that have been extended during the development don't need to be tested.

   4.4. **Files should not exceed a length of 400 lines[1] to be better readable.** Like in 4.3., this only applies to newly created files. Also, files used for testing may be longer as "split" test files could cause confusion.

3.3 Evaluation scheme

Since a fine-grained hierarchical description of all requirements has been set, the evaluation should be done accordingly. For each item of the list, a score from 0 to 100% should be calculated. Here, 0 stands for a not met requirement and 100% a fully met requirement. Values between these numbers are linearly scaled and can be interpreted in the following way:

- 0: Requirement completely failed.
- 1% - 25%: Most parts of the requirement not fulfilled.
- 26% - 50%: Many parts of the requirement not fulfilled.
- 51% - 75%: Some parts of the requirement not fulfilled.
- 76% - 99%: Most parts of the requirement fulfilled.
- 100%: Requirement completely fulfilled.

[1]We use a limit of 400 lines per file as this number is a recommendation of the Angular Style Guide: "Consider limiting files to 400 lines of code.", see https://angular.io/guide/styleguide (accessed on April 24, 2019)

Each value v is rounded in the following way:

$$v_{new} = \frac{\lfloor v \cdot 100 \rfloor}{100}$$

A requirement that has no children can either be “fulfilled”, receiving a score of 100%, or “not fulfilled”, leading to a score of 0.

For a requirement r that has m children, the score s will be calculated as follows:

$$s = \frac{\lfloor \sum_{k=1}^{m} \frac{\text{score of } r.k}{m} \cdot 100 \rfloor}{100}$$

In the evaluation, only requirements with no children need to be directly analyzed. All other requirements will automatically receive a score based on the formula above.

3.4 Visualization of requirements

A visualization of the requirements defined in section 3.2 can be found in figure 3.1. The requirements for the Sankey view can be found in more detail in figure 3.2, while figure 3.3 shows the details of the requirements for the Graph view.

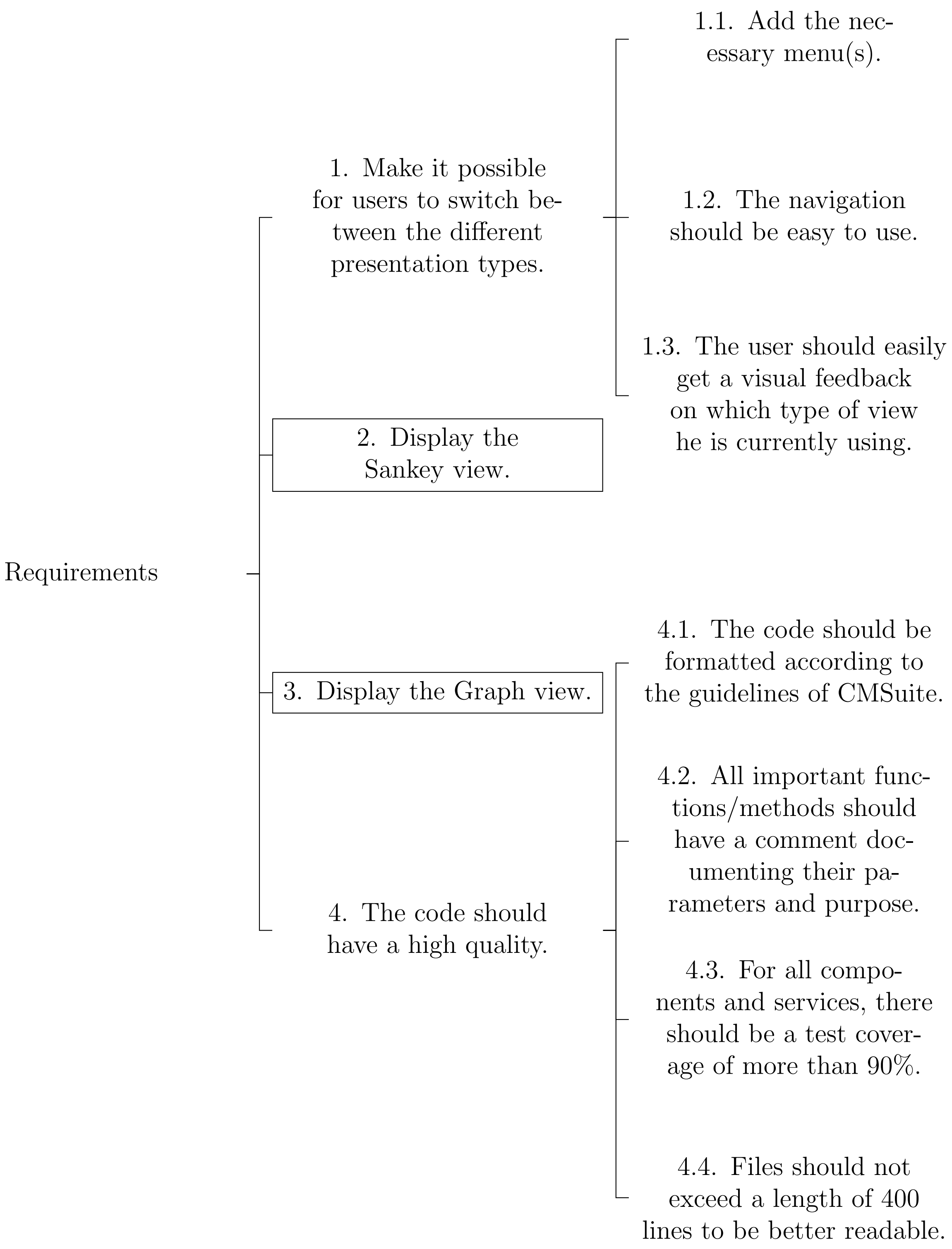

Requirements
1. Make it possible for users to switch between the different presentation types.
1.1. Add the necessary menu(s).
1.2. The navigation should be easy to use.
1.3. The user should easily get a visual feedback on which type of view he is currently using.
2. Display the Sankey view.
3. Display the Graph view.
4. The code should have a high quality.
4.1. The code should be formatted according to the guidelines of CMSuite.
4.2. All important functions/methods should have a comment documenting their parameters and purpose.
4.3. For all components and services, there should be a test coverage of more than 90%.
4.4. Files should not exceed a length of 400 lines to be better readable.

Figure 3.1: The requirements visualized using a tree representation. The requirements for the Sankey view can be found in figure 3.2 and the requirements for the Graph view are shown in figure 3.3.

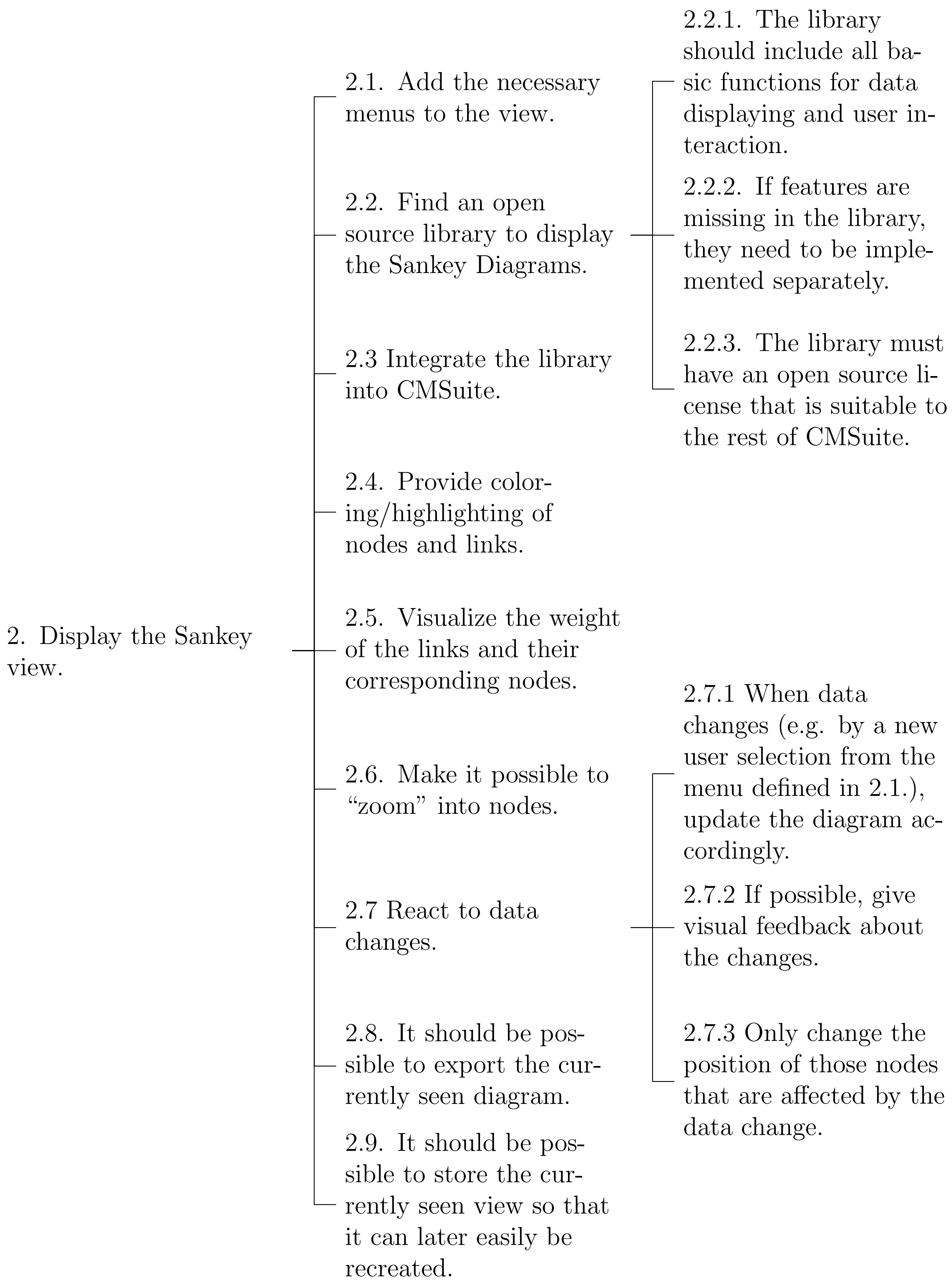
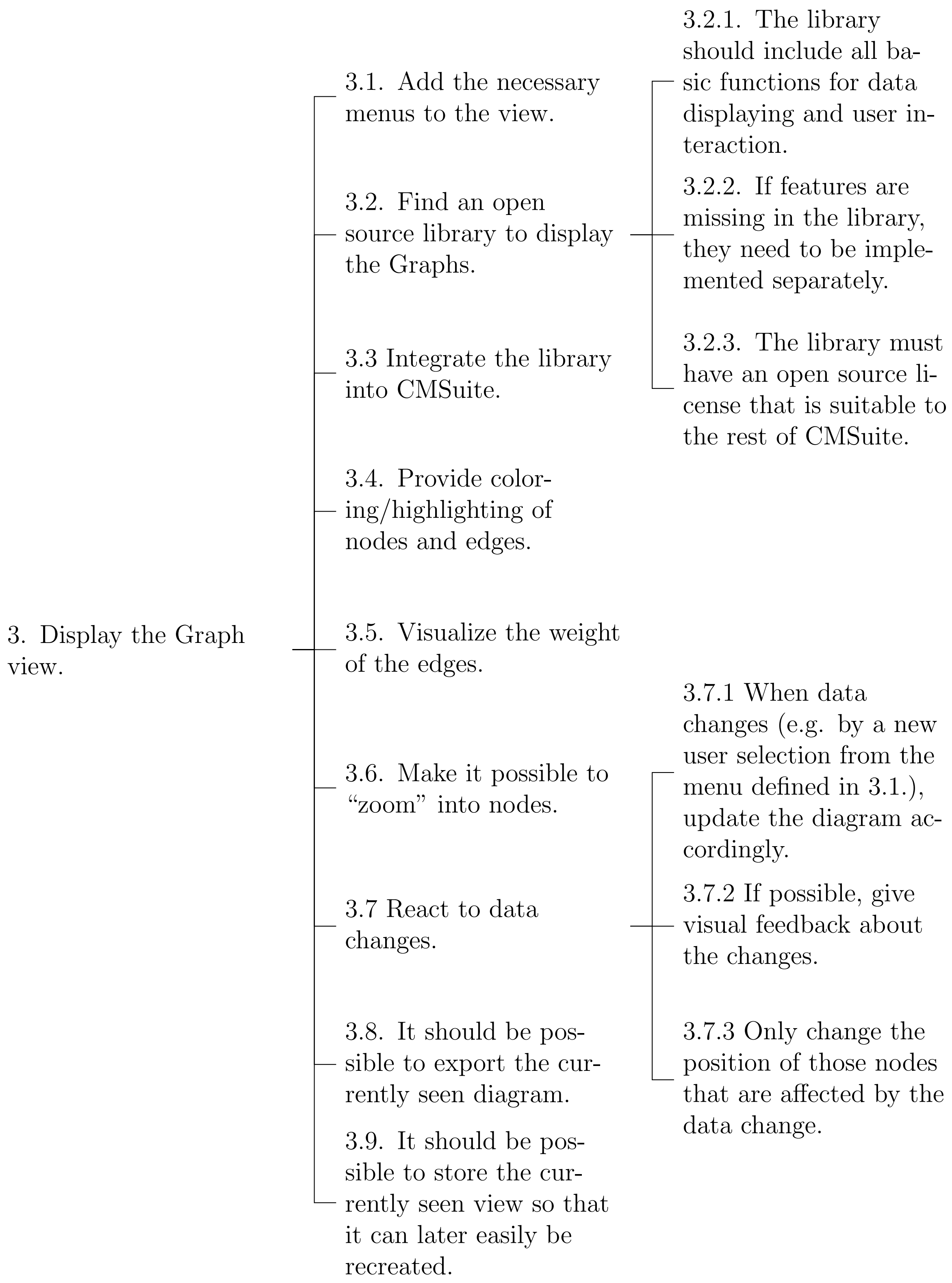

2. Display the Sankey view.
2.1. Add the necessary menus to the view.
2.2. Find an open source library to display the Sankey Diagrams.
2.2.1. The library should include all basic functions for data displaying and user interaction.
2.2.2. If features are missing in the library, they need to be implemented separately.
2.2.3. The library must have an open source license that is suitable to the rest of CMSuite.
2.3 Integrate the library into CMSuite.
2.4. Provide coloring/highlighting of nodes and links.
2.5. Visualize the weight of the links and their corresponding nodes.
2.6. Make it possible to "zoom" into nodes.
2.7 React to data changes.
2.7.1 When data changes (e.g. by a new user selection from the menu defined in 2.1.), update the diagram accordingly.
2.7.2 If possible, give visual feedback about the changes.
2.7.3 Only change the position of those nodes that are affected by the data change.
2.8. It should be possible to export the currently seen diagram.
2.9. It should be possible to store the currently seen view so that it can later easily be recreated.

Figure 3.2: The requirements for the Sankey view visualized using a tree representation.



- 3. Display the Graph view.
  - 3.1. Add the necessary menus to the view.
  - 3.2. Find an open source library to display the Graphs.
    - 3.2.1. The library should include all basic functions for data displaying and user interaction.
    - 3.2.2. If features are missing in the library, they need to be implemented separately.
    - 3.2.3. The library must have an open source license that is suitable to the rest of CMSuite.
  - 3.3 Integrate the library into CMSuite.
  - 3.4. Provide coloring/highlighting of nodes and edges.
  - 3.5. Visualize the weight of the edges.
  - 3.6. Make it possible to “zoom” into nodes.
  - 3.7 React to data changes.
    - 3.7.1 When data changes (e.g. by a new user selection from the menu defined in 3.1.), update the diagram accordingly.
    - 3.7.2 If possible, give visual feedback about the changes.
    - 3.7.3 Only change the position of those nodes that are affected by the data change.
  - 3.8. It should be possible to export the currently seen diagram.
  - 3.9. It should be possible to store the currently seen view so that it can later easily be recreated.

Figure 3.3: The requirements for the Graph view visualized using a tree representation.

4 Architecture and Design

In the following chapter, the general structure and important decisions regarding the software components needed for this thesis are described.

4.1 Selection of a library for Sankey Diagrams and Graphs

In order to display Sankey Diagrams and Graphs, it is necessary to decide if a library should be written from scratch, or whether open source software can be used for this task. The latter could potentially save development time and later reduce maintenance efforts. Because of this, it is preferable to look for a ready-to-use open source library. Since the two types of diagrams most likely aren't supported by one single library, the selection will be made independently for both of them.

4.1.1 Library for Sankey Diagrams

With requirements 2.2.1 and 2.2.3, as stated in section 3.2, we have already defined the minimum required functionality of a Sankey Diagram library. They are as follows:

- Have an open source license that is compatible with CMSuite.
- Allow basic styling (e.g. setting the background colors) of nodes.
- Allow basic styling (e.g. setting the background colors) of links.
- Visualize link weights by using different heights.
- Allow user interactions (e.g. clicking, right clicking and hovering) of nodes and links.

Since CMSuite utilizes the Angular framework, compatibility between the Sankey library and Angular could reduce the development time even more. However, this is not a strict requirement since creating a “wrapper”, acting as a bridge between the selected library and Angular, should be possible in most cases.

Potential libraries

Many libraries were checked for compatibility with the requirements listed above. Those pieces of software that did not provide functionality to display Sankey Diagrams, or that did not have a proper open source license, such as Google Charts[1], were omitted from the list of potential “candidates”. In the end, a decision between the libraries D3.js[2] (later referred to as “D3”) and angular-plotly.js[3] had to be made.

A comparison of D3 and angular-plotly.js can be found in table 4.1. It is important to note that both libraries are constantly updated, which could make the comparison not fully valid for future versions.

	D3	angular-plotly.js
Tested version	5.7.0	0.1.15
Have an open source license that is compatible with CM-Suite.	Yes (BSD 3-Clause “New” or “Revised” License)	Yes (MIT License)
Allow basic styling (e.g. setting the background colors) of nodes.	Yes	Yes
Allow basic styling (e.g. setting the background colors) of links.	Yes	Yes
Visualize link weights by using different heights.	Yes	Yes
Allow user interactions (clicking, right clicking and hovering) of nodes and edges.	Yes	Click and hover possible. Right click not supported.
(Optional) Include a ready-to-use wrapper for Angular.	No	Yes

Table 4.1: Comparison of potential libraries for Sankey Diagrams.

[1]https://developers.google.com/chart, accessed on March 27, 2019
[2]https://d3js.org, accessed on March 27, 2019
[3]https://github.com/plotly/angular-plotly.js, accessed on March 27, 2019

As it turns out, both of the libraries fulfill (nearly) all minimum requirements. The only disadvantage of angular-plotly.js is the missing right click support. However, it already provides an Angular wrapper, that could make the integration in CMSuite easier compared to D3.

Since the direct comparison did not clearly reveal the best-suited library for the tasks of this thesis, another point had to be considered: The flexibility, i.e. the possibility to change the individual pieces of the diagram.

D3 doesn't create DOM[4] elements on its own. Instead, it only provides the results of the calculations that were made to find positions for the individual nodes and links. Also, it includes several helper functions making the creation of actual DOM elements easier. Since no wrapper for Angular is provided, it has to be written before being able to use the library in CMSuite.

Angular-plotly.js, on the other hand, already includes an Angular wrapper. Changing the style of individual nodes or links can only be done by setting values to pre-defined attributes in the configuration object. Directly adding custom elements or changing the general layout/behavior of the diagram is not possible - it is necessary to change the actual source code of the library.

Because of this, using D3 and creating a custom wrapper for Angular has been considered the better, more flexible choice for the work in this thesis; so D3 has been chosen to display the Sankey Diagrams.

4.1.2 Library for Graphs

Similar to how it was done for the Sankey Diagrams, an open source library needs to be chosen for Graphs. If no suitable library can be found, all necessary algorithms need to be implemented from scratch.

With requirements 3.2.1 and 3.2.3, as stated in section 3.2, we have already defined the minimum required functionality of a Graph library. They are analogical to the ones selected for the Sankey Diagrams:

- Have an open source license that is compatible with CMSuite.
- Allow basic styling (e.g. setting the background colors) of nodes.
- Allow basic styling (e.g. setting the colors) of edges.
- Visualize edge weights by using different widths.

[4] "DOM" is an abbreviation for "Document Object Model" and refers to the representation of an HTML or XML document within a browser, see https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction (accessed on April 20, 2019)

- Allow user interactions (e.g. clicking, right clicking and hovering) of nodes and edges.

Potential libraries

The following four open source libraries were considered in the comparison:

- Sigma.js[5]
- Cytoscape.js[6]
- D3 (using D3-force[7])
- ngx-graph[8]

It should be noted that D3 does not explicitly provide a Graph library. Instead, the D3 Force Simulation can be used. This allows the programmer to create nodes that "react" to each other. For instance, it is possible to make nodes collide with others or get drawn together when an edge exists between them.

A comparison can be found in table 4.2. Like for the Sankey Diagram libraries, it is important to note that all libraries are constantly updated, which could make the comparison not fully valid for future versions.

Since all four libraries fulfill the minimum requirements, other aspects need to be considered to choose the best fitting one.

D3 has already been selected for Sankey Diagrams. Most likely, the API is similar and thus makes the code more consistent. Also, it might be possible to share some parts of the code between the Sankey Diagram and the Graph wrapper.

ngx-graph has the advantage that it already includes a wrapper for Angular.

The other two libraries do not provide any of the advantages mentioned above. Because of this, only D3 and ngx-graph are considered hereinafter.

As described for Sankey Diagrams, D3 offers a lot of flexibility since it only provides the results of its calculations and does not automatically create any DOM elements. The disadvantage is that a wrapper for Angular needs to be created.

Ngx-graph already includes an Angular wrapper. However, there are much less options for customizations compared to D3. It is possible to change individual

[5]http://sigmajs.org, accessed on March 27, 2019
[6]http://js.cytoscape.org, accessed on March 27, 2019
[7]https://github.com/d3/d3-force, accessed on March 27, 2019
[8]https://github.com/swimlane/ngx-graph, accessed on March 27, 2019

	Sigma.js	Cytoscape.js	D3	ngx-graph
Tested version	1.2.1	3.2.20	D3: 5.7.0 D3-force: 1.1.2	5.1.0
Have an open source license that is compatible with CMSuite.	Yes (MIT License)	Yes (MIT License)	Yes (BSD 3-Clause "New" or "Revised" License)	Yes (MIT License)
Allow basic styling (e.g. setting the background colors) of nodes.	Yes	Yes	Yes	Yes
Allow basic styling (e.g. setting the colors) of edges.	Yes	Yes	Yes	Yes
Visualize edge weights by using different widths.	Yes	Yes	Yes	Yes
Allow user interactions (e.g. clicking, right clicking and hovering) of nodes and edges.	Yes	Yes	Yes	Yes
(Optional) Include a ready-to-use wrapper for Angular.	No	No	No	Yes

Table 4.2: Comparison of potential libraries for Graphs.

properties of the used SVG elements. Yet, the programmer cannot explicitly influence the exact placement of the nodes, precisely animate individual properties (for instance to make the nodes visually "collide" with each other), define how edges are positioned (e.g. all starting in one pre-defined point or evenly distributed over the nodes), or add more advanced custom elements to the Graph. This lack of flexibility could quickly become a problem.

Because of this, D3 has been chosen as it is the most flexible library and is already in use for the Sankey Diagram. An Angular wrapper needs to be created to be fully compatible with CMSuite.

4.2 Structural changes to relevant existing components

As described before, the Sankey Diagrams and Graphs will be shown as an alternative to the already existing journal view. The journal view already required a set of Services and Components so that the user is able to select and filter the data as he wishes. Thus, most of the structures providing the form elements and interactions with the user, as well as the Services that enable the communication with the backend and make the data available to the actual journal view, already exist. Still, the existing structure had to be changed in many places so that the Sankey Diagrams and Graphs can be displayed.

Hereinafter, we only show components responsible for the handling of organizational elements ("OrgElements"). Besides the organizational elements, there is only one other type of economic agent that can be analyzed, which is the inner source project ("Isp"). The structure needed for inner source projects is analogical to the one required by the organizational elements, which is why we are limiting the explanations to the latter one.

4.2.1 Current structure

The structure that existed prior to this thesis is illustrated in figure 4.1. Here, only the most relevant components are shown. When the user updates filter settings in the menu ("OrgElementEventJournalMenuComponent"), the embedding component ("OrgElementAccountancyComponent") is informed about the (new) menu settings. The "OrgElementAccountancyComponent" then makes the Services fetch the new economic events. It tells the event journal ("EventJournalComponent") that new data is currently loading and provides a handle allowing the event journal to retrieve the new economic events as soon as they have been received from the server by the Services.

4.2.2 Modified structure

To add the Sankey Diagram and Graph event views, we had to change multiple parts of the components related to the Accountancy page. The updated structure, based on the one illustrated in figure 4.1, is shown in figure 4.2. Multiple changes in the naming of the components were introduced, so that the menu is now called "OrgElementAnalysisMenuComponent", inheriting from the abstract "AnalysisMenuComponent". Also, a "TabbedMenuComponent" was introduced to allow users to switch between the different event views.

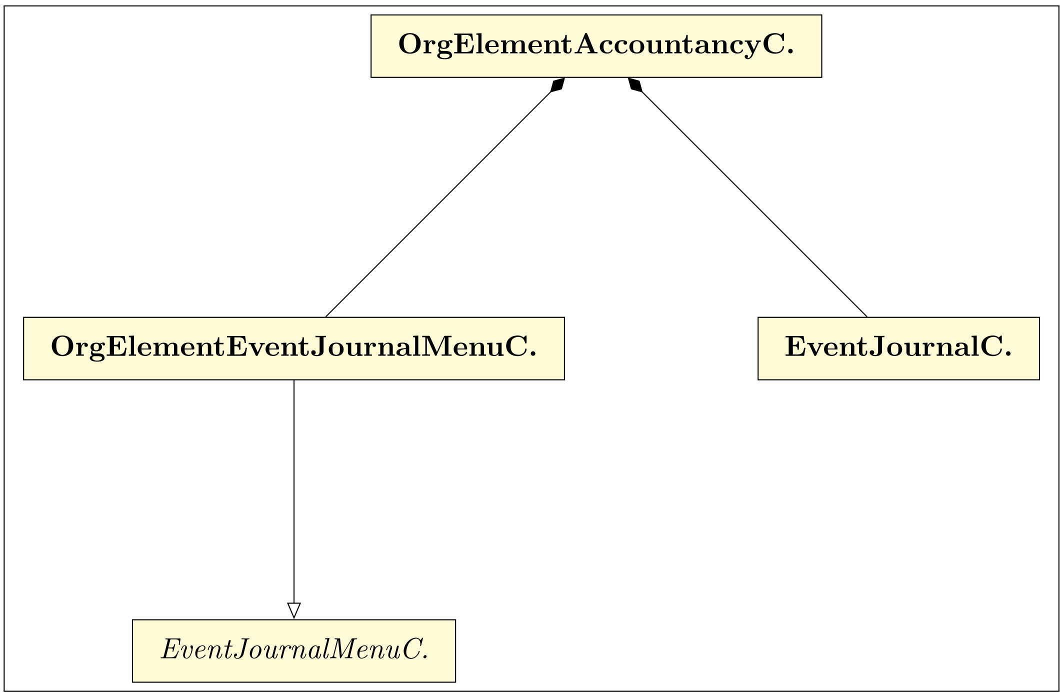


Figure 4.1: The structure of the Accountancy page prior to this thesis. Only the most relevant elements are shown to display data for organizational elements. "C." is an abbreviation of "Component".

Additionally, we are now using the Router[9] to inject the requested event view into the template of the "OrgElementAccountancyComponent". When the user selects a new tab in the "TabbedMenuComponent", the Router reacts to it and replaces the currently shown event view with the one matching the selection in the tabbed menu.

When the user updates filter settings in the menu ("OrgElementAnalysisMenu-Component"), the process is, at first, similar to how it used to be in the old structure: The embedding component ("OrgElementAccountancyComponent") is notified about the (new) menu settings and fetches the new economic events using the Services. Now, however, it informs the "AccountancyStateService" (not shown in figure 4.2) that new data is loading and passes the handle to it. The "AccountancyStateService" informs the currently active event view that new data is loading and provides the handle for the new data, so that the event view is able to react to it.

[9]In CMSuite, we are using UIRouter for Angular 2+, see https://github.com/ui-router/angular (accessed on April 22, 2019).

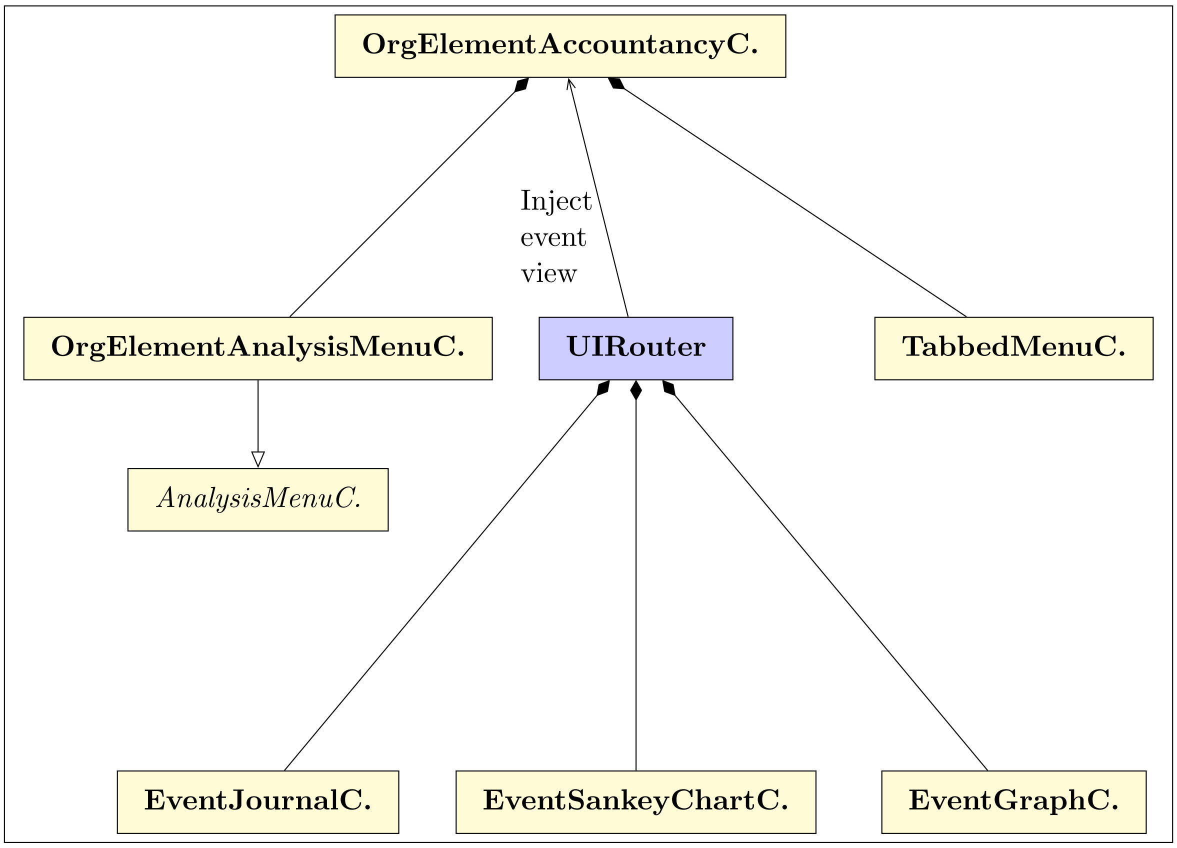


Figure 4.2: The modified structure of the Accountancy page. Only the most relevant elements are shown to display data for organizational elements. "C." is an abbreviation of "Component". We are using UIRouter for Angular 2+ (see https://github.com/ui-router/angular (accessed on April 22, 2019)) to inject the selected event view into the template.

4.3 Wrappers for D3

In order to use D3 in CMSuite, wrappers need to be created that integrate D3's API into Angular. Even though for both the Sankey Charts and the Graphs the same library is used, the wrappers still need to be created separately. This is due to the fact that D3 itself provides different libraries for Sankey Diagrams and Graphs. Only small parts of the code can be shared among the two wrapper modules.

In the following, the basic structure of the Graph Module is described, which is analogical to the Sankey Diagram Module.

Hereinafter, "edges" are called "links" to be compliant with the naming used by

D3.[10]

4.3.1 General structure

The Graph Chart Module consists of two parts: The actual Graph Chart Component and multiple Draw Controllers. When using the Graph Chart Module, the developer only interacts with the Component's interface and never with the Draw Controllers. This is also illustrated in figure 4.3, where the Graph Chart Component is embedded into the “EventGraphComponent” (which, as shown in figure 4.2, is the event view for Graphs).

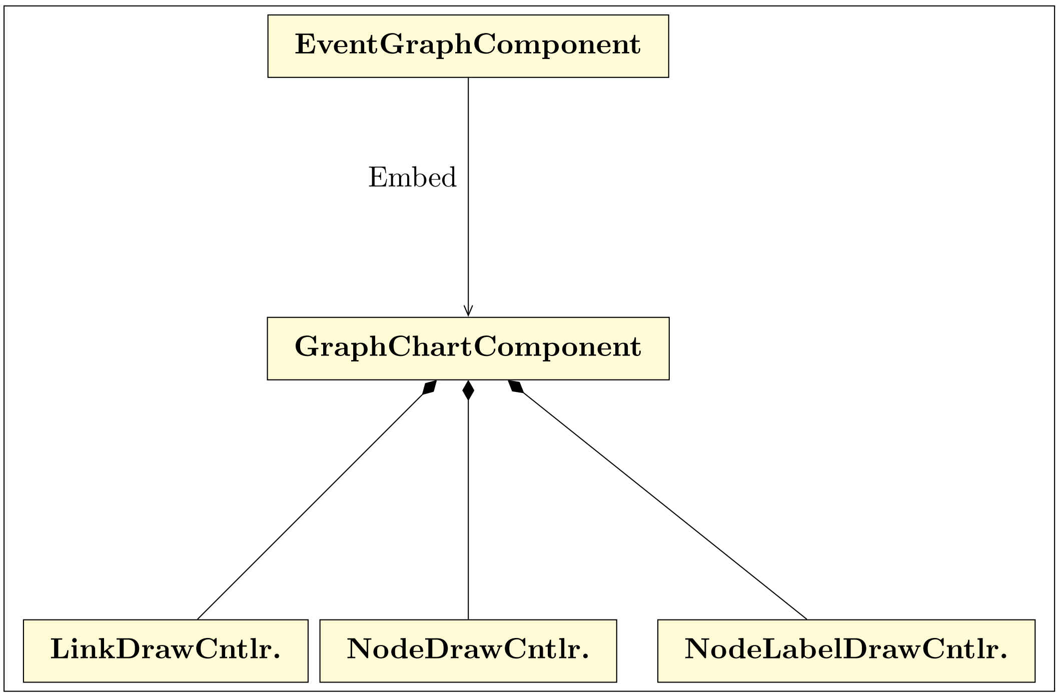


Figure 4.3: The Graph Chart Component and its Draw Controllers. The Graph Chart Component is the only element that communicates with the component it is embedded in (here: the “EventGraphComponent”). “Cntlr.” is an abbreviation of “Controller”.

Graph Chart Component

The Component receives an array of nodes, as well as an array of links. It is responsible for the handling of data changes, basic configurations and data

[10]Example from the README file: “...sets the array of links associated with this force ...”. See https://github.com/d3/d3-force/blob/455c0bad8f0f6c3d65be85706b601850342a5520/README.md, accessed on March 27, 2019

structures. When a new diagram has to be rendered for the user, it calls the Draw Controllers.

Draw Controllers

All visible diagram elements that are shown to the user, are drawn using the Draw Controllers.

Each Draw Controller is responsible for one specific type of diagram content: The Node Draw Controller, for instance, only manages the nodes that are shown to the user. It does not create links or the text that is written inside the nodes. The positions of the nodes are calculated by D3. D3 also provides methods to generate and update the DOM elements.

There are two more Draw Controllers that are needed to display the complete Graph: The Link Draw Controller handles the links between the nodes, while the Node Label Draw Controller solely has the responsibility for the text that is placed inside the nodes.

This structure allows a clear separation of concerns and thus makes the code better to maintain.

Each Draw Controller has two methods that are called by the Graph Chart Component: *initialize()* and *draw()*. The first is called only once and can be used by the individual Draw Controllers to set up necessary DOM elements (e.g. containers) or data structures. The latter is called every time data changes. When this is the case, all Draw Controllers need to perform three tasks:

- Find out which DOM elements need to be newly created (and create them).
- Find out which DOM elements need to be updated (and update them).
- Find out which DOM elements need to be removed (and remove them).

4.3.2 Basic sequence of events

Figure 4.4 shows a simplified sequence of events when the Graph Chart Component is embedded into another Component "A".

As soon as Component A decides to display the Graph, it passes the link and node data to the Graph Chart Component. The Graph Chart Component now initializes the Draw Controllers. After this has been completed, the actual diagram will be rendered: Each Draw Controller draws the elements it is responsible for. For instance, the Link Draw Controller draws all lines to the DOM. Now, the user is able to see the diagram that has been generated.

When data (nodes or links) change in Component A, the Graph Chart Component will be notified. The Graph Chart Component now makes the Draw Controllers find the differences in the old and new data sets using D3's helper methods and render the changes. Eventually, the updated diagram will be shown to the user.

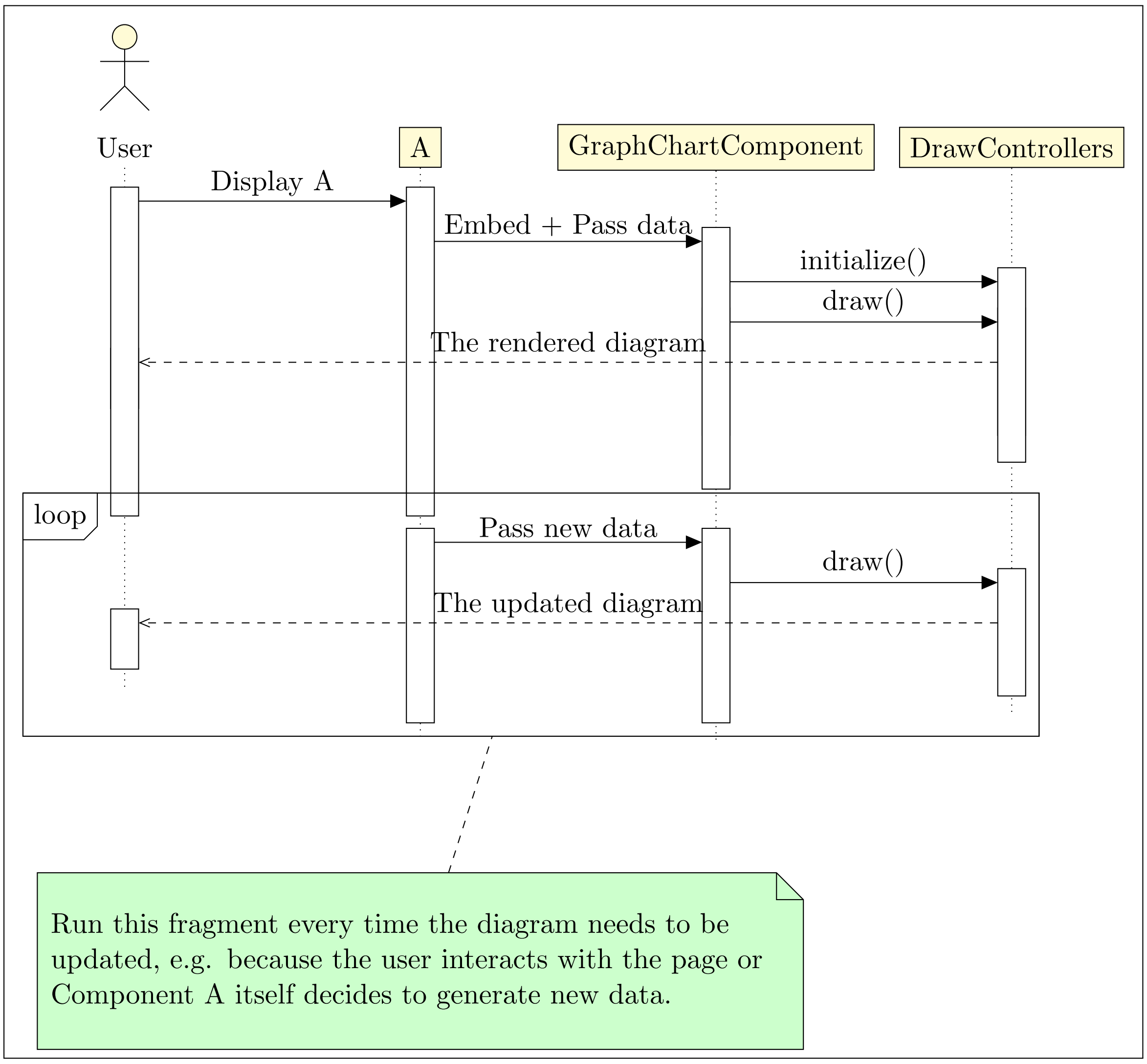

User
A
GraphChartComponent
DrawControllers
Display A
Embed + Pass data
initialize()
draw()
The rendered diagram
loop
Pass new data
draw()
The updated diagram
Run this fragment every time the diagram needs to be updated, e.g. because the user interacts with the page or Component A itself decides to generate new data.

Figure 4.4: A simplified sequence of events when the Graph Chart Component is embedded into another Component “A”.

4.4 Additional Visualizations

After the implementation of the Graph view, we observed that the generated diagrams are easier to interpret compared to the tables shown in the journal view. However, for large data sets, the Graphs are still difficult to comprehend. To solve this problem, we decided to implement another kind of diagram that we named "Circular Graph".

Since the Circular Graph was not part of the original requirements, the details are not subject to the evaluation done in chapter 6.

In the following, the most relevant parts of the implementation is covered. We decided to use D3 like we already did for the Sankey Diagrams and the standard Graphs. An additional wrapper was created for the Circular Graphs and integrated into CMSuite similarly to how it was done for the other two visualization modules.

4.4.1 Creation of nodes and links

Like a standard Graph, the Circular Graph consists of nodes and links.

Links are created as Hierarchical Edge Bundles as described by Holten (2006). Here, a tree hierarchy, in our case the hierarchy of the organizational units, projects and persons, is used to generate control points for our link polygons. The links are then drawn as interpolations of the given control points. Luckily, D3 already includes support both for the computation of the control points as well as for the necessary interpolations.[11]

Nodes are drawn as arcs. Instead of just showing the source/target nodes of the created links, we visualize the whole tree hierarchy of the involved economic agents. Only the root node is not shown since it would create a full circle without any more meaning. Each node label (text drawn inside the node) is bent so that it follows the curve of the respective node. Also, overflowing text is cut and ellipses are added as indicator when not the whole label is shown. To accomplish this, we use the same algorithm as for the standard Graph that is explained in section 5.1.

An example of the generated nodes, links and labels is shown in figure 4.5.

4.4.2 Interactivity and Animations

When a user hovers a leaf node (a node that has no children in the hierarchy) with the mouse cursor, the corresponding links are highlighted. When a user hovers

[11] Also, there are multiple examples of implementations of Hierarchical Edge Bundle visualizations using D3. One example can be found on https://bl.ocks.org/mbostock/1044242 (accessed on April 24, 2019).

a non-leaf node, all descendants and their corresponding links are highlighted recursively as well.

We use transitioning effects to make data updates in the diagrams look smooth. For the arcs, we had to implement additional data structures and methods since the animation of the generated polygons (having varying numbers of control points) is not trivial.

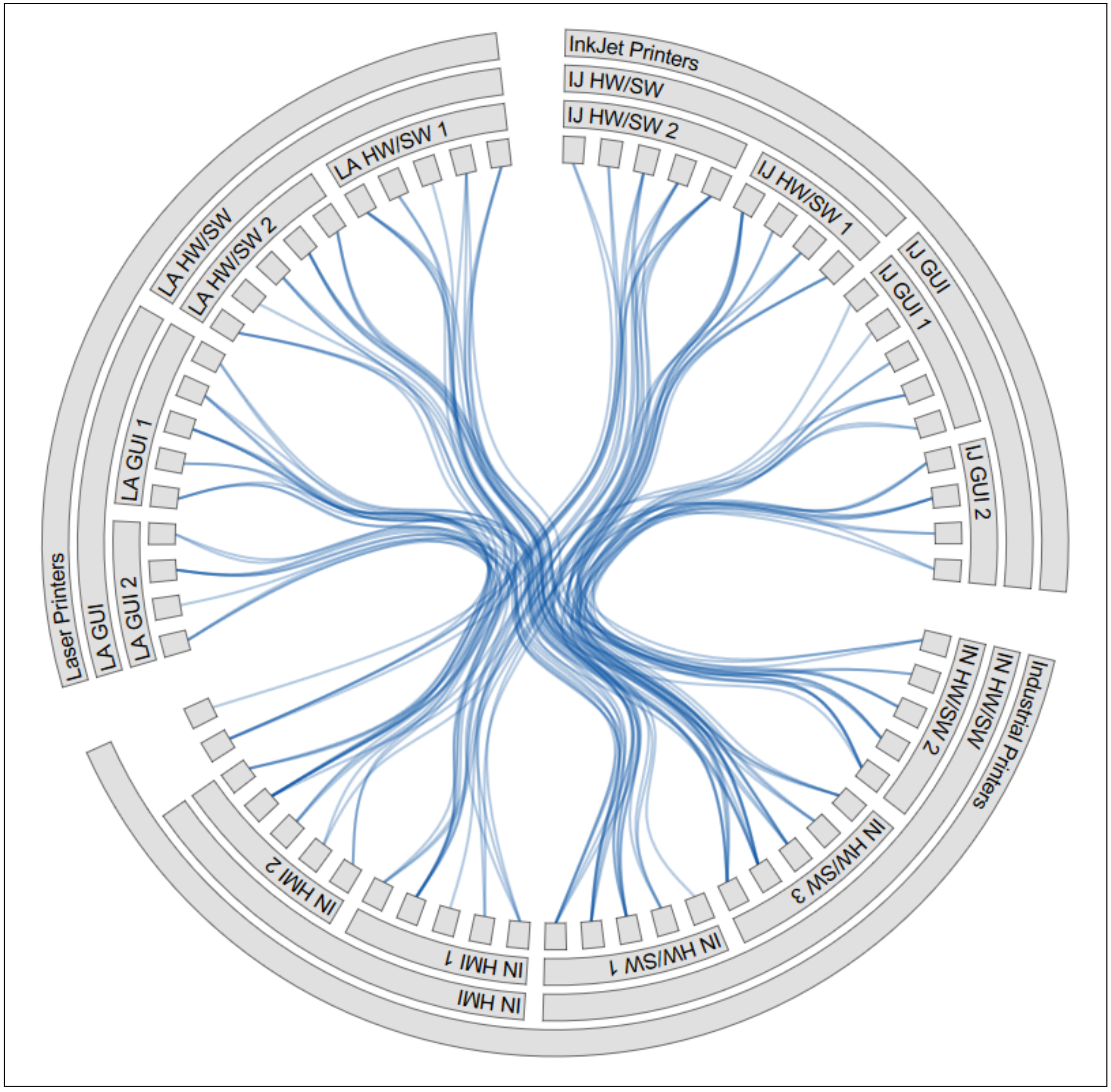


Figure 4.5: An example of a generated Circular Graph.

5 Implementation

This chapter covers some of the most interesting (algorithmic) details of the code developed for this thesis. The main focus relies on the D3 wrappers as they made up most of the work and contain multiple non-trivial implementations.

5.1 Placing text inside a Graph Node

During the implementation of the Graph wrapper, multiple challenges had to be overcome. One was the rendering of the node texts, which is described in the following section.

5.1.1 Fundamentals

For all diagrams, we generate SVG[1] elements using our D3 wrappers.

The basic structure of the generated SVG for a Graph can be found in figure 5.1.

5.1.2 Objective of implementation

Our first objective was to create simple rectangular nodes with text placed inside them. These nodes should have a fixed width. Figure 5.2 shows an example of three nodes that are connected with links. Each node has a dark border, dark text and white background. The simplified structure of the SVG elements that were required to generate the nodes and texts for figure 5.2 can be found in figure 5.3.

[1] "Scalable Vector Graphics (SVG) is an XML-based markup language for describing two dimensional based vector graphics. SVG is essentially to graphics what HTML is to text." (https://developer.mozilla.org/en-US/docs/Web/SVG, accessed on March 28, 2019)

```
<svg height="400" width="800">
  <defs>
    <!-- Elements that can be referenced by others, such as
    "markers", can be found here. -->
  </defs>
  <g class="links-container">
    <!-- All the links generated using the Link Draw Controller
    can be found here. -->
  </g>
  <g class="nodes-container">
    <!-- All the nodes generated using the Node Draw Controller
    can be found here. -->
  </g>
  <g class="node-labels-container">
    <!-- All the node labels generated using the Node Label Draw
    Controller can be found here. -->
  </g>
</svg>
```

Figure 5.1: Basic structure of an SVG for a Graph.

5.1.3 Problems and solutions

In order to clearly emphasize which text belongs to which node, the text should always be placed inside the nodes and never exceed the borders. Otherwise, the result would not look visually appealing and could lead to confusion among the users: When many nodes with overflowing text exist next to each other, it might not be clear which part of the text is related to which node. Figure 5.4a shows an example of overflowing text that has not been truncated.

Additionally, if the text was too long for a node and had to be cut off, some kind of indicator, for instance ellipses ("..."), should be shown. This tells the user that the text he sees is not complete. Figure 5.4b shows an example of a node where the text "A long text inside a node" has been cut off without any visual indicator. The user is not able to recognize that the text he currently sees is not complete. Figure 5.4c demonstrates a possible solution for this problem: Right next to the text that had to be cut off, ellipses are shown so that the user is informed about the fact that this text is not complete.

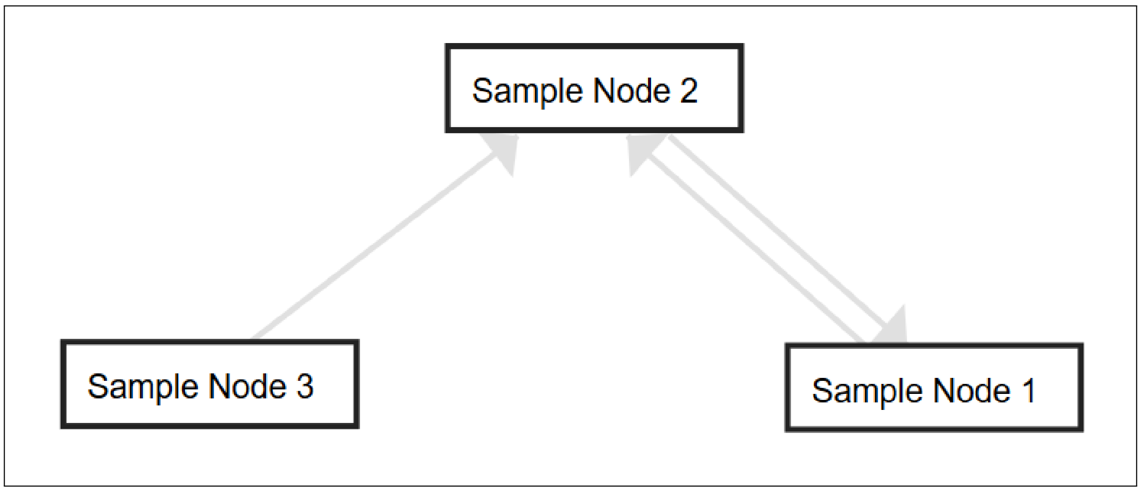


Figure 5.2: An example of a Graph with rectangular nodes.

5.1.4 Fixed text width in SVG

It turned out that dealing with fixed-width text in SVGs is not a trivial task. There are two attributes that can be added to a text element which make the content not exceed the bounds of the node: **“textLength”**[2] and **“text-overflow”**[3]. Both of them are covered hereinafter. We did not want to make the text break into multiple lines. All nodes should have the same height so that the resulting diagram looks more organized. Because of this, properties that make the text break into multiple lines are not covered here. Also, it is possible to embed HTML elements into an SVG using “foreignObject”[4]. This would enable us to use the CSS property “text-overflow: ellipsis;” that displays ellipses at the end of the text if it was cut off. However, this could later become a problem if advanced styling, that is only supported by SVG, should be added to the text. Thus, a solution without “foreignObject” is desirable.

The “textLength” property

The “textLength” attribute can be used to force text to render with a fixed width. An example of how the attribute can be applied to a text element can be seen in figure 5.5. Unfortunately, for long text, the result becomes hard to read since the browser tries to reduce the spacing between the letters so that the whole text fits

[2]Specification of the textLength attribute: https://svgwg.org/svg2-draft/text.html#TextElementTextLengthAttribute, accessed on April 01, 2019

[3]Specification of the text-overflow attribute: https://svgwg.org/svg2-draft/text.html#TextOverflowProperty, accessed on April 01, 2019

[4]See https://developer.mozilla.org/en-US/docs/Web/SVG/Element/foreignObject, accessed on April 01, 2019

```
<g class="nodes-container">
  <rect fill="#ffffff" stroke="#222222" stroke-width="2"
  width="98" height="28" x="296" y="136"></rect>
  <rect fill="#ffffff" stroke="#222222" stroke-width="2"
  width="98" height="28" x="182" y="36"></rect>
  <rect fill="#ffffff" stroke="#222222" stroke-width="2"
  width="98" height="28" x="54" y="135"></rect>
</g>
<g class="node-labels-container" style="font: 11px sans-serif;">
  <text dy="0.35em" x="304" y="151">Sample Node 1</text>
  <text dy="0.35em" x="190" y="51">Sample Node 2</text>
  <text dy="0.35em" x="62" y="150">Sample Node 3</text>
</g>
```

Figure 5.3: A simplified structure of the SVG elements required to output the nodes and texts of figure 5.2.

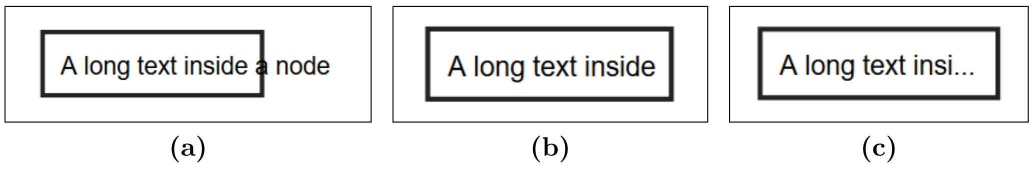


Figure 5.4: Multiple ways to deal with overflowing text in nodes: (a) let the text overflow the node, (b) cut the text off when reaching the border of the node without any visual indicator that the text is not complete, (c) cut the text off and show ellipses.

inside the given bounds. Figure 5.6 shows how the properties defined in figure 5.5 are rendered within a node. Obviously, using “textLength” does not always produce readable results. Because of this, another solution should be looked for.

The “text-overflow” property

The “text-overflow” property allows text to be cut off when it has exceeded the bounds of its container. It also allows us to automatically add ellipses to the end of the text if it had to be truncated. Unfortunately, at the time of writing this thesis, the property was not supported by the majority of the currently used browsers. We tested it with Google Chrome[5] in version 73 and also with Mozilla Firefox[6] in version 65. These two browsers combined own a market share of

[5]https://www.google.com/intl/en_us/chrome/, accessed on April 01, 2019
[6]https://www.mozilla.org/en-US/firefox, accessed on April 01, 2019

nearly 80 percent.[7] In our tests, both browsers did not support the attribute as defined in the official SVG specification.[8] This fact makes it impossible to use the attribute for now.

```
<g class="node-labels-container" style="font: 11px sans-serif;">
  <text dy="0.35em" x="304" y="151" textLength="84">A long text
  inside a node</text>
</g>
```

Figure 5.5: Example of the usage of the "textLength" attribute. The resulting text can be seen in figure 5.6.

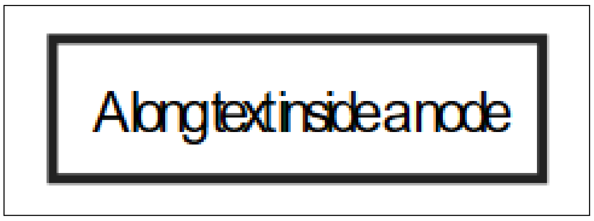


Figure 5.6: An example of a node with a long text and a small value for the "textLength" property.

5.1.5 Implementation using JavaScript

Since the SVG implementation in the current browsers does not provide any way to solve our problem, we decided to use JavaScript functions to solve the problem.

Cut after a fixed number of characters

A relatively simple approach is to cut the text after a pre-defined number of characters. However, most of the fonts available on modern browsers have symbols with different widths. When truncating the text after a fixed number of characters, it might get cut off too "early", leaving a large amount of whitespace, or get cut off too "late", making the text overflow the container. Figures 5.7a and 5.7b show an example of this problem: For both of them, the text was truncated after

[7]In March 2019, Chrome had a worldwide market share of about 69% and Firefox owned a market share of about 9% when comparing desktop browsers, according to http://gs.statcounter.com/browser-market-share, accessed on April 01, 2019.

[8]Additionally, a feature request (https://bugs.chromium.org/p/chromium/issues/detail?id=366550) for the implementation of the attribute in Chrome was not fulfilled, last checked on April 01, 2019.

the same number of letters. While the text in figure 5.7a extends over the whole node before being cut off, the text in figure 5.7b gets truncated before even covering half of the available space. This is due to the fact that the character “W” takes up much more horizontal space compared to the letter “i”. This problem could be solved by using a monospaced font, which is a font where all characters have the same horizontal width. However, displaying a font that is already in use in other parts of the application makes the overall design look much more consistent. Because of this, the decision was made not to use a monospaced font.

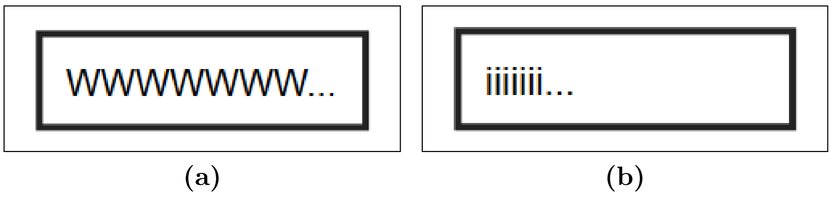


Figure 5.7: Two nodes with the same number of characters. Since the characters in (a) take up more horizontal space compared to (b), the horizontal space of the node in (b) is not fully used.

Compute the perfect number of characters

To perfectly fit the texts into the nodes, a slightly more complex approach needs to be taken. An implementation in pseudo code is shown in figure 5.8. At first, the whole text is put into the node, without ellipses added to it. If it fits into the node, nothing else is done. Otherwise, if it does not fit into the node, characters are removed one by one until the text doesn’t overflow the node anymore. Also, ellipses are added to the end of the text. The advantage of this algorithm is that the result can be considered “perfect”: It fits as many characters into the node as possible. However, the computational complexity could become a problem with larger amounts of nodes: For an average number n of characters of the input strings and m nodes, the algorithm has a computational complexity (in the average case) of $O(n \cdot m)$. If n is, in most cases, a very large number, we can perform the algorithm in lines 4–11 the other way round: First fit one character into the node, then increase this number until the text does not fit anymore. For an average number of c fitting characters, the computational complexity is $O(c \cdot m)$.

However, we wanted to implement a quicker algorithm based on the first one (having the computational complexity of $O(n \cdot m)$). Our new approach uses a binary-search-like procedure, reducing the computational complexity to $O((\log_2 n) \cdot m)$. A simplified version of the algorithm as pseudo code can be found in figure 5.9. It does not try to reduce the number of characters one by one. Instead, it cuts the text in half and checks whether the result fits into the node. If it doesn't, it cuts the text in half again. If the text fits into the node, half of the string that was removed in the last iteration is added again. This procedure is repeated as long as there is potential for a longer (and still fitting) string.

A huge advantage of this implementation, like with any other JavaScript solution, is the fact that it provides a lot of control over the output. Since the last characters of an organizational unit's name may be as important as the first ones, we decided not to add the ellipses to the end of the text like in figure 5.4c, but to place them in the middle as illustrated in figure 5.10.

```
let text = 'A long text inside a node';
renderTextIntoNode(text);
if (textDidNotFit()) {
  while (text.length > 0) {
    text = removeLastCharacter(text);
    renderTextIntoNode(text + '...');

    if (textDidFit()) {
      break;
    }
  }
}
```

Figure 5.8: JavaScript-like pseudo code for a simple implementation that removes one character after the other until the text fits into the node.

```javascript
const text = 'A long text inside a node';
renderTextIntoNode(text);
if (textDidNotFit()) {
  let lowerBound = 0;
  let upperBound = Math.floor(text.length / 2);
  let best;

  while (true) {
    // Terminate when the lower bound "shot" over the upper one.
    if (lowerBound > upperBound) {
      break;
    }

    const middle = Math.floor((lowerBound + upperBound) / 2);
    renderTextIntoNode(truncateText(text, middle));

    if (textDidNotFit()) {
      upperBound = middle - 1;
    } else {
      best = middle;
      lowerBound = middle + 1;
    }
  }

  // Now, the variable "best" includes the perfect number of
  characters.
}
```

Figure 5.9: JavaScript-like pseudo code for an algorithm that uses a binary-search-like approach to fit text into a node.

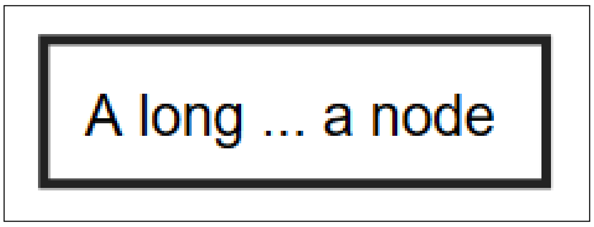


Figure 5.10: A node with ellipses in the middle of the text.

5.2 Calculating collisions between links and nodes

Another challenge to overcome was the placement of links, i.e. the calculations that are required to make the links "start"/"stop" right next to a node.

5.2.1 Fundamentals

Inside our SVG, we create links using "line" elements. These elements are used to display straight lines from a starting point $(x1, y1)$ to a target point $(x2, y2)$. Hereinafter, we call the node at the starting point "start node" and the one at the target point "target node". To place an arrow head at the end of the link, we use a "marker", which can be added to the end of a line using the "marker-end" property.

Figure 5.11 shows a simplified example of the code that is necessary to generate links with arrow heads at the end of them, as shown in figure 5.21b.

```
<defs>
  <marker id="graph-marker-arrow-default-1" orient="auto"
  refX="4" refY="5" viewBox="0 0 10 10"
  markerUnits="userSpaceOnUse" markerWidth="20"
  markerHeight="20">
    <polygon points="0,10 5,5 0,0" fill="#e0e0e0"></polygon>
  </marker>
</defs>

<g class="links-container">
  <line marker-end="url(#graph-marker-arrow-default-1)"
  stroke="#e0e0e0" y1="168" x2="193" y2="85" stroke-width="4"
  x1="243"></line>
  <line marker-end="url(#graph-marker-arrow-default-1)"
  stroke="#e0e0e0" x1="207" y1="85" y2="168" stroke-width="1"
  x2="256"></line>
  <line marker-end="url(#graph-marker-arrow-default-1)"
  stroke="#e0e0e0" x1="125" y1="169" x2="175" y2="85"
  stroke-width="2"></line>
</g>
```

Figure 5.11: A simplified structure of the SVG elements required to output the links of figure 5.21b

5.2.2 Objective of implementation

Links should always start/end next to a node and never have their starting/ending points (in the following referred to as “link coordinates”) “inside” a node. Figure 5.12 illustrates one of the negative consequences of the placement of the link coordinates near to the center of the nodes: The text becomes hard to read. The more links are drawn, the more illegible the text becomes. Because of this, it is essential to make sure the links don’t overlap the nodes. Also, the link coordinates should not be set too far apart from the corresponding nodes so that for each link it is clear which node it is pointing to. The best solution for this task is to make the links start/end right at the border of the nodes.

It is trivial to create undirected Graphs as shown in figure 5.13. Here, the links all start/end in the center of the respective nodes. The nodes are placed on top of the links so that the links don’t cover the text.

However, we also wanted to show the direction of the patch-flow using triangular arrow heads. If we used the same approach as with the undirected Graph, the arrow heads would be drawn underneath the nodes and become invisible to the user.

Because of this, we had to develop an algorithm to calculate the link coordinates, taking into account all of the observations mentioned above.

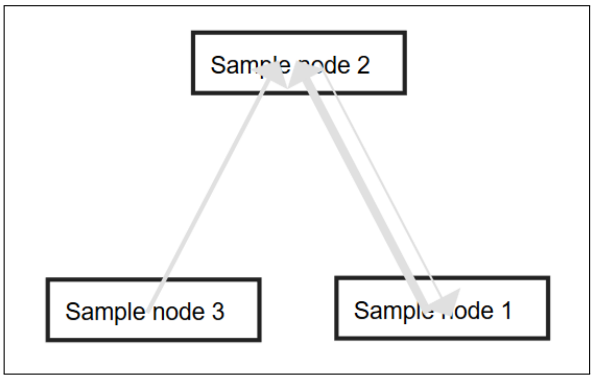


Figure 5.12: A Graph with links that start/end in the middle of the respective nodes, making the text hard to read.

5.2.3 Implementation

The algorithm is divided into three parts. At first, we determine which node border the currently processed link should stick to. Then, the actual “collision coordinate” is calculated. In the end, the collision coordinates are moved clockwise/counterclockwise to reduce ambiguities.

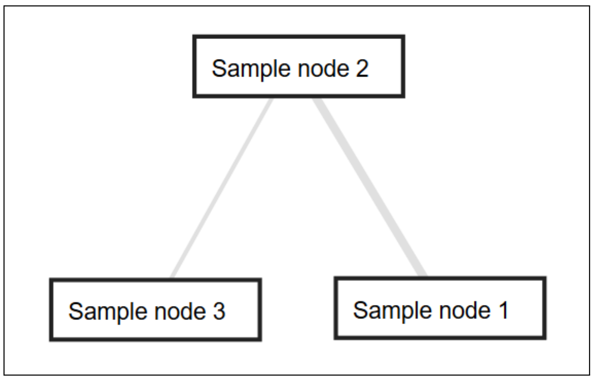


Figure 5.13: An undirected Graph where the links start/end in the center of the respective nodes. Since the nodes were drawn on top of the links, they appear to start/end at the border of the nodes.

Determination of the affected node border

The first part of the algorithm finds out which side of a node (top, right, bottom or left) a link will "collide" with. For two nodes A and B, with A being the source and B the target node of link l, we always want to select the sides that are facing the respective other node.

If, for instance, A is on the right of B, then l collides with the left side of A and with the right side of B. This example is illustrated in figure 5.14. Here, the source node ("Sample node 1") has been placed to the right of the target node ("Sample node 2"). Because of this, the link collision takes place on the left side of the source node and on the right side of the target node.

Figure 5.15a shows a bad implementation: Here, the source node has been placed below the target one. However, the collision side is set to "left" for both the source and the target nodes. This makes the arrow head get hidden behind the target node, which makes the interpretation of the diagram hard for the user. The result of the correct implementation can be seen in figure 5.15b: The source collision side is set to "top" and the target collision side to "bottom".

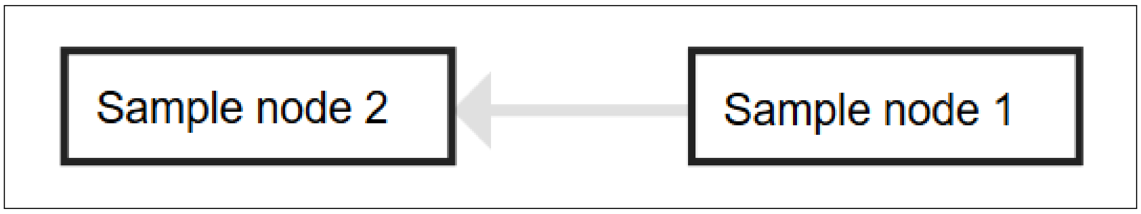
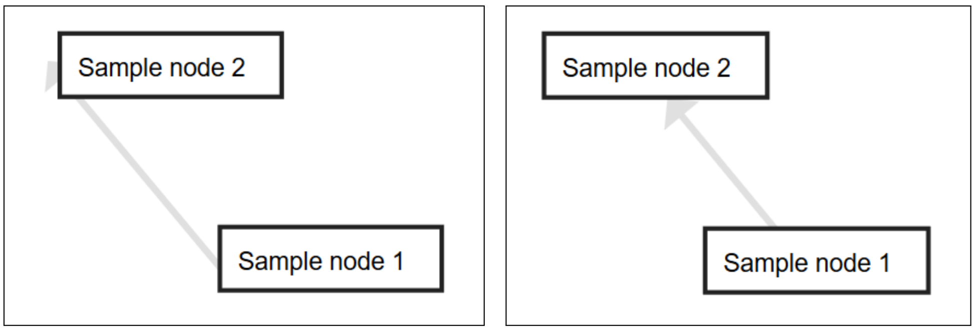


Figure 5.14: A Graph consisting of one link and two nodes (source on the right, target on the left). The link collision side is set to "left" for the source and "right" for the target node.



(a) **(b)**

Figure 5.15: Two Graphs, both consisting of one link and two nodes (source on the bottom, target on the top). In (a), the source and target collision sides are both set to "left". In (b), the source collision side is set to "top" and the target one to "bottom".

In the following, we only calculate the source collision side. The calculations for the target one are analogical. To better explain the algorithm, additional lines and other elements are drawn into the nodes. Of course, these elements would not be displayed with the actual implementation.

At first, we divide the source node into four parts by drawing one line $c1$ from the bottom left to the top right corner and another line $c2$ from the top left to the bottom right corner. Now, four triangles are visible: One at the top, one at the right, one at the bottom and the last on the left side of the node. This is illustrated in figure 5.16.

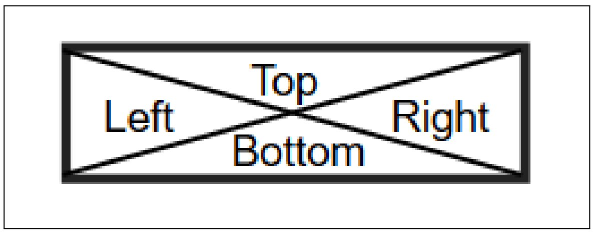


Figure 5.16: A node divided diagonally into four parts, representing the four possible collision sides.

Now, we draw a straight line m from the center of A to the center of B, which is illustrated in figure 5.17. Our goal is to compare m with the four triangles: If it was, for instance, drawn starting within the "top" triangle, then the source collision side is the top one. We can determine the collision side by comparing the slope of m with the slope of $c1$ or $c2$. There are multiple ways to achieve the desired result. One can be found in figure 5.18. Please note that in SVGs, the origin of the coordinate system is at the left top; x values increase to the right and y values to the bottom.

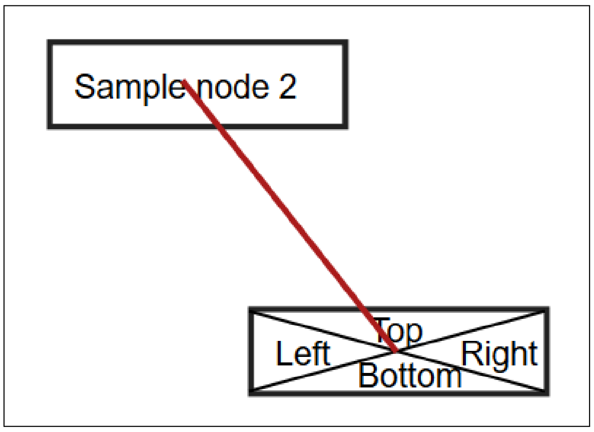


Figure 5.17: The node shown in 5.16, connected to another node using a (red) line starting/ending in the center of the respective node.

Calculation of collision coordinates

After having obtained the collision side, we can now proceed to determine the actual coordinates of the link that has been drawn from the center of A to the center of B. Similarly to how it was done in the first step of the algorithm, we only cover the calculations for the source side, since it is trivial to apply the algorithm to the target one as well.

We want to analyze line m, that has been drawn from the center of A to the center of B, and find out the exact coordinates where it collides with node A.

Depending on the previously determined collision side, either the x or the y coordinate can be easily calculated: If the collision takes place on the top or bottom of A, the y coordinate of the collision is equal to the y coordinate of the top or bottom border. On the other hand, if the collision takes place on the left or right, the collision x coordinate is equal to the one of the left or right border of the node. An example of an implementation can be found in figure 5.19.

To find the second coordinate, we need to perform some additional calculations that are shown (in a simplified way) in figure 5.20. Here, we represent m in the form of $y = slope * x + t$. If m is a straight vertical line, we cannot compute the slope. Thus, this special case needs to be treated separately, which is also done below.

If the collision takes place on the top or bottom of the node, we already know the y coordinate and need to find x. This can be achieved by a simple transformation as follows:

$$y = slope \cdot x + t$$
$$slope \cdot x = y - t$$
$$x = \frac{y - t}{slope}$$

Of course, if m is a straight vertical line, we cannot perform this transformation since the slope is not computable. In this case, the resulting x is equal to the x coordinate of the center of A or B.

If the collision takes place on the left or right of the node, we already know x and need to find y, which can be calculated using the equation $y = slope * x + t$.

Now, we have found both the x and y coordinates where m collides with A. There is a last step we need to consider before drawing the diagram, which is explained in the next section.

Move the collision coordinates

We want to visualize the weight of the links using different widths of the lines. With the calculations performed above, one problem arises that can be seen in figure 5.21a: If a line is drawn from node A to node B and another one is drawn from node B back to node A, they are displayed on top of each other, making it impossible to distinguish them. We solve this problem by moving the collision coordinates on the source side counterclockwise and on the target side clockwise. The result is visualized in figure 5.21b: The links are not drawn on top of each other anymore.

```
// Handle special cases, e.g. a not computable slope of line m.
handleSpecialCases();

// Calculate the "corner slope" of the source node (so the absolute slope of c1 or c2).
const nodeCornerSlope = nodeHeight / nodeWidth;

// Calculate the absolute slope of line m.
const absoluteLinkSlope = Math.abs(calculateSlope(lineM));

// The side where, on the source node, the collision takes place.
let sourceCollidedEdge;

if (absoluteLinkSlope > nodeCornerSlope) {
  // The colliding edge is either the top or bottom one.

  if (source.y > target.y) {
    sourceCollidedEdge = 'top';
  } else {
    sourceCollidedEdge = 'bottom';
  }
} else {
  // The colliding edge is eiher the right or the left one.

  if (source.x > target.x) {
    sourceCollidedEdge = 'left';
  } else {
    sourceCollidedEdge = 'right';
  }
}
```

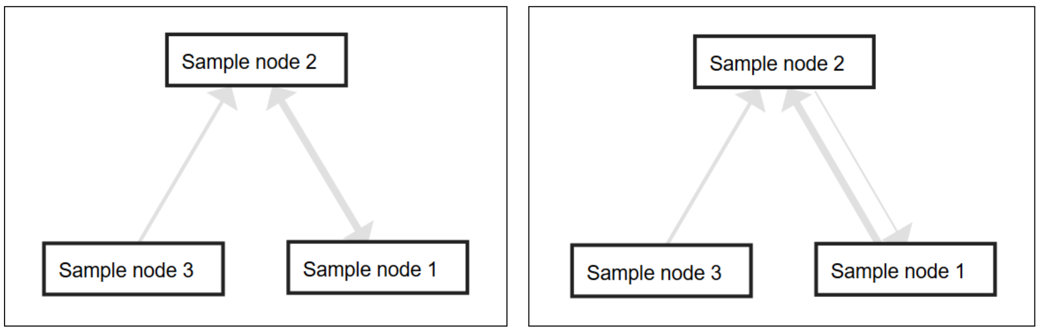
Figure 5.18: JavaScript-like pseudo code for the calculation of the source collision side.

```
// The coordinates where, on the source node, the collision
takes place.
const result = {
  x: undefined,
  y: undefined
};

switch (sourceCollidedEdge) {
  case 'top': {
    // The x and y values point to the center of the node.
    result.y = sourceNode.y - nodeHeight / 2;
    break;
  }
  case 'bottom': {
    result.y = sourceNode.y + nodeHeight / 2;
    break;
  }
  case 'left': {
    result.x = sourceNode.x - nodeWidth / 2;
  }
  case 'right': {
    result.x = sourceNode.x + nodeWidth / 2;
  }
}
```

Figure 5.19: JavaScript-like pseudo code for the calculation of the first collision coordinate. This code is supposed to be run after the source collision side has been determined using the code shown in figure 5.18.

```javascript
const linkXDistance = sourceNode.x - targetNode.x;
const linkYDistance = sourceNode.y - targetNode.y;

// Represent the link in the form "y = slope * x + t".
let slope;
let t;

if (linkXDistance !== 0) {
  slope = linkYDistance / linkXDistance;
  /*
    y = slope * x + t
    t = y - slope * x
  */
  t = sourceNode.y - slope * sourceNode.x;
}

if (sourceCollidedEdge === 'top' || sourceCollidedEdge === 'bottom') {
  // We already know the y coordinate of the result.

  if (linkXDistance === 0) {
    result.x = sourceNode.x;
  } else {
    /*
      y = slope * x + t
      slope * x = y - t
      x = (y - t) / slope
    */
    result.x = (result.y - t) / slope;
  }
} else {
  // We already know the x coordinate of the result.

  // y = slope * x + t
  result.y = slope * result.x + t;
}
```

Figure 5.20: JavaScript-like pseudo code for the calculation of the second collision coordinate, based on the already determined first coordinate. This code is supposed to be run after the code shown in figure 5.19.



(a) (b)

Figure 5.21: Two directed and weighted Graphs, both consisting of three nodes and three links connecting the nodes. In (a), two links are drawn on top of each other, making it impossible to see the individual weights. In (b), the links have been moved clockwise/counterclockwise. Because of this, the individual weights can be distinguished.

6 Evaluation

In this chapter, the requirements defined in chapter 3 are evaluated.

The evaluation is done using four-column tables. For each task from 3.2, we first write down its number, followed in the next column by the title. In the third column, the received score is shown. For requirements without children, the score is either 0 or 100%. Any other task receives a "computed score" based on the formula defined in section 3.3. Then, in the last column, we give an explanation of the score.

The overall requirements were fulfilled to 79%. Occasionally, during the implementation of the various requirements, we decided to focus on other tasks that were not defined in chapter 3 (e.g. because we found that some functionality not defined in the requirements became very important so that the user is now able to interact with the generated diagram in a more intuitive way). This lead to a relatively low score.

Information regarding the evaluation of the four basic tasks 1 to 4 is shown in table 6.1. The results regarding task 1 can be found in detail in table 6.2, task 2 is evaluated in table 6.3. Table 6.4 shows the evaluation of task 3 and in table 6.5, the results of task 4 can be found.

Table 6.1: Evaluation of the general requirements.

Number	Title	Score	Details
1.	Make it possible for users to switch between the different presentation types.	100% (computed)	*For details, see table 6.2.*
2.	Display the Sankey view.	66% (computed)	An example of the generated Sankey diagrams is shown in figure 6.1. *For details, see table 6.3.*

Continued on next page

Table 6.1 (continued)

Number	Title	Score	Details
3.	Display the Graph view.	51% (computed)	An example of the generated Graphs is shown in figure 6.2. *For details, see table 6.4.*
4.	The code should have a high quality.	100% (computed)	*For details, see table 6.5.*

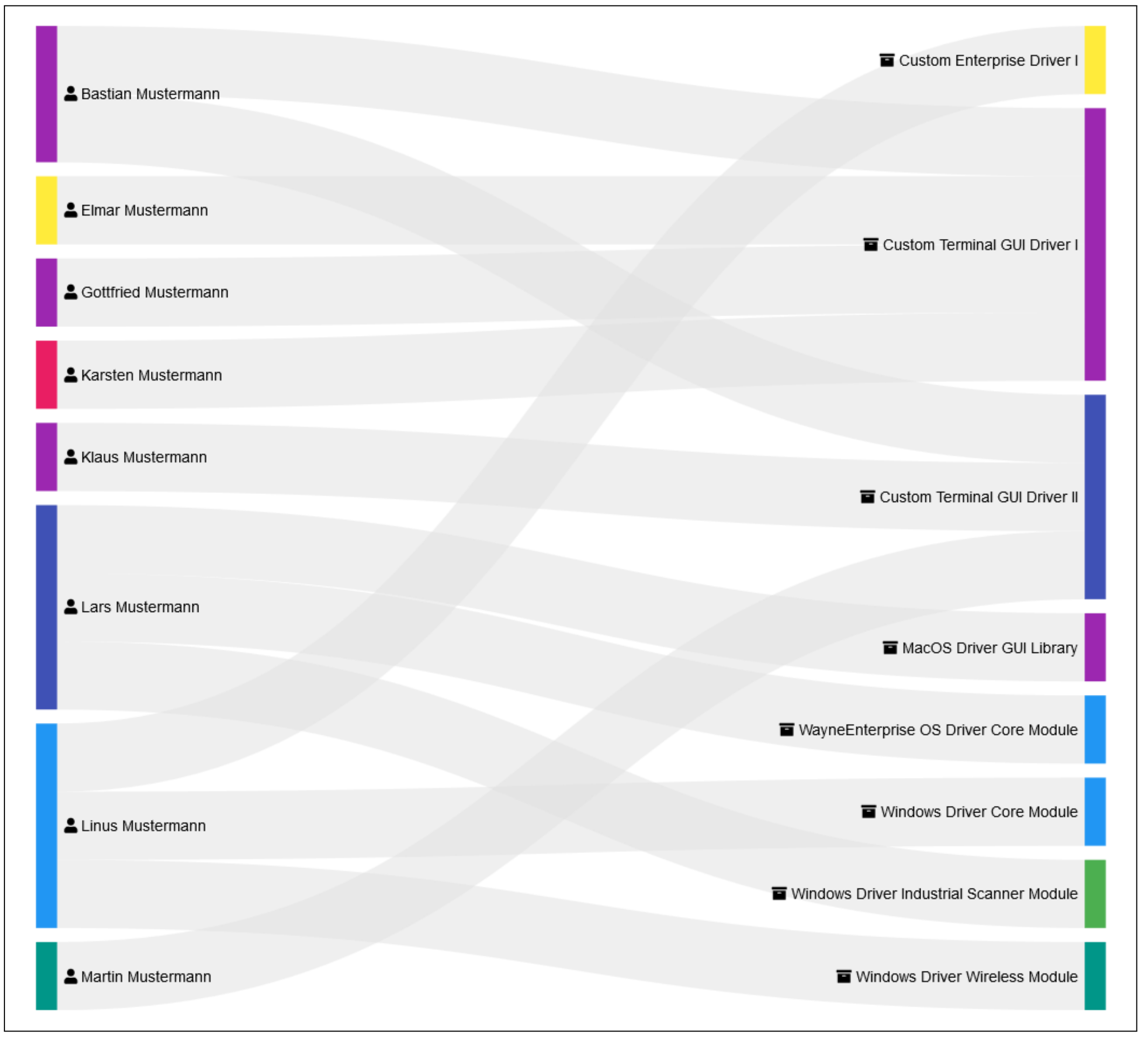


Figure 6.1: An example of a generated Sankey diagram. The icons are part of FontAwesome, see https://fontawesome.com (accessed on April 06, 2019).

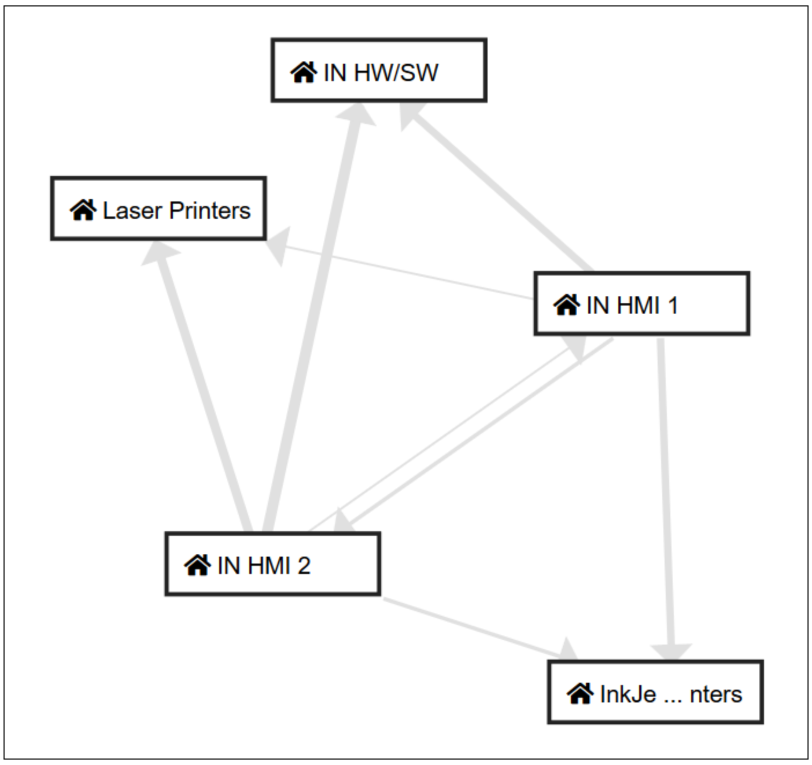


Figure 6.2: An example of a generated Graph. The icons are part of FontAwesome, see https://fontawesome.com (accessed on April 06, 2019).

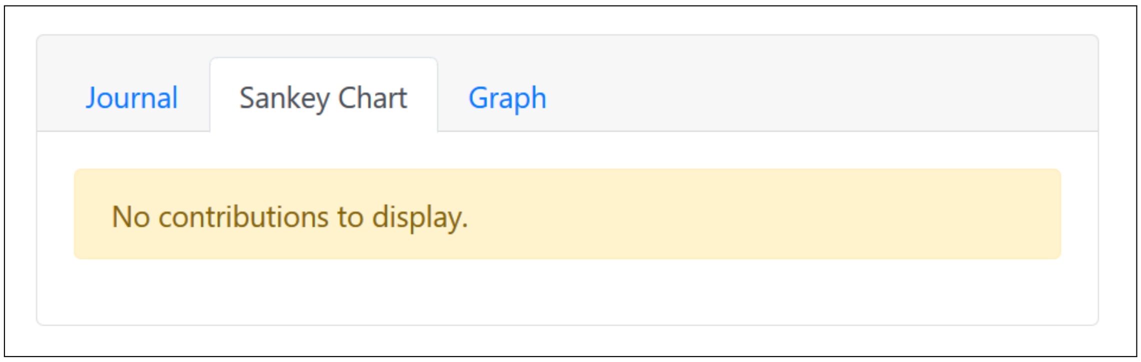
Table 6.2: Evaluation of the requirements for the functionality to switch between the different presentation types.

Number	Title	Score	Details
1.1.	Add the necessary menu(s).	100%	We have added a menu consisting of three tabs (as shown in figure 6.3), making it possible for the user to switch the views.
1.2.	The navigation should be easy to use.	100%	The user only needs to perform one click to change the views, making the menu easy to use.

Continued on next page

Table 6.2 (continued)

Number	Title	Score	Details
1.3.	The user should easily (i.e. without having to perform an interaction with the UI) get a visual feedback on which type of view he is currently using.	100%	The currently active tab is highlighted, which enables the user to see what the current view (e.g. the Graph view) is called.



No contributions to display.

Figure 6.3: The menu we added to switch between the different presentation types. (In this image, the actual diagram is not shown because the user defined settings that resulted in an empty data set.)

Table 6.3: Evaluation of the requirements for the Sankey view.

Number	Title	Score	Details
2.1.	Add the necessary menus to the view.	100%	It was possible to re-use the already existing menu components that were utilized in the journal view.
2.2.	Find an open source library to display the Sankey Diagrams.	100% (computed)	*For details, see the evaluation of requirements 2.2.1 to 2.2.3.*

Continued on next page

Table 6.3 (continued)

Number	Title	Score	Details
2.2.1.	The library should include all basic functions for data displaying and user interaction.	100%	As described in section 4.1.1, we have found a suitable library (D3) that fulfills all minimum requirements.
2.2.2.	If features are missing in the library, they need to be implemented separately.	100%	D3 already includes support for all the functionality we defined for 2.2.1. Additionally, during the creation of the wrapper, we also implemented advanced visual elements, e.g. the possibility to show icons next to the node titles.
2.2.3.	The library must have an open source license that is suitable to the rest of CMSuite.	100%	Table 4.1 of section 4.1.1 shows that the license (BSD 3-Clause "New" or "Revised" License) is compatible with CMSuite.
2.3.	Integrate the library into CMSuite.	100%	We have created all necessary wrapper functions to integrate D3's Sankey library into Angular (and CMSuite). Section 4.3 contains a brief summary of how such a wrapper was structured.

Continued on next page

Table 6.3 (continued)

Number	Title	Score	Details
2.4.	Provide coloring/highlighting of nodes and links.	100%	Nodes are colored using the colors provided by the “Color Palette Service”. In the wrapper, we also made it possible to color the links. However, the decision was made to use the default (grey) color for the links in order to make the diagram look more consistent. Even though in the end, only the nodes are colored and the links all have the same color, we consider this task as “successfully completed” since the wrapper also includes the option to color links and it was an intentional design choice to set all link colors to the same value.
2.5.	Visualize the weight of the links and their corresponding nodes.	100%	If a link has a larger weight, a greater height is assigned to it. Nodes automatically inherit the sizes of the links connected to them.
2.6.	Make it possible to “zoom” into nodes	0	We have not implemented this feature in order to focus on other tasks.
2.7.	React to data changes.	100% (computed)	*For details, see the evaluation of requirements 2.7.1 to 2.7.3.*
2.7.1	When data changes (e.g. by a new user selection from the menu defined in 2.1.), update the diagram accordingly.	100%	As soon as a change in the data is detected, the diagram is updated.
2.7.2.	If possible, give visual feedback about the changes.	100%	We use animations and transitioning effects to emphasize which elements were updated when the data changes.

Continued on next page

Table 6.3 (continued)

Number	Title	Score	Details
2.7.3.	Only change the position of those nodes that are affected by the data change.	100%	Of course, when new nodes are added, the already existing ones might be slightly shifted or downsized to create space for the new ones. However, the order of the already existing nodes is not changed in this case, making it easy for the user to understand which elements were added (or removed).
2.8.	It should be possible to export the currently seen diagram	0	We have not implemented this feature in order to focus on other tasks.
2.9.	It should be possible to store the currently seen view so that it can later easily be recreated.	0	We have not implemented this feature in order to focus on other tasks.

Table 6.4: Evaluation of the requirements for the Graph view.

Number	Title	Score	Details
3.1.	Add the necessary menus to the view.	100%	Like for the Sankey view, it was possible to re-use the already existing menu components that were utilized in the journal view.
3.2.	Find an open source library to display the Graphs.	100% (computed)	*For details, see the evaluation of requirements 3.2.1 to 3.2.3.*
3.2.1.	The library should include all basic functions for data displaying and user interaction.	100%	As described in section 4.1.2, we have found a suitable library (D3 in conjunction with D3-force) that fulfills all minimum requirements.

Continued on next page

Table 6.4 (continued)

Number	Title	Score	Details
3.2.2.	If features are missing in the library, they need to be implemented separately.	100%	D3 already includes support for all the functionality we defined for 3.2.1. Additionally, during the creation of the wrapper, we also implemented advanced features, e.g. a method to force nodes to stay within a bounding box so that they cannot be dragged out of the viewport.
3.2.3.	The library must have an open source license that is suitable to the rest of CMSuite.	100%	Table 4.2 of section 4.1.2 shows that the license of D3 and D3-force (BSD 3-Clause "New" or "Revised" License) is compatible with CMSuite.
3.3.	Integrate the library into CMSuite.	100%	We have created all necessary wrapper functions to integrate D3's Sankey library into Angular (and CMSuite). Section 4.3 contains a brief summary of how such a wrapper was structured.
3.4.	Provide coloring/highlighting of nodes and edges.	0	We added the possibility to control the color of the edges and the border and background color of the nodes. However, the decision was made not to support a fine-grained control over the node or edge colors (e.g. on a per-node/per-edge basis). Thus, we consider this task "failed".
3.5.	Visualize the weight of the edges.	100%	If an edge has a larger weight, a greater width is assigned to it.
3.6.	Make it possible to "zoom" into nodes	0	We have not implemented this feature in order to focus on other tasks.
3.7.	React to data changes.	66% (computed)	*For details, see the evaluation of requirements 3.7.1 to 3.7.3.*

Continued on next page

Table 6.4 (continued)

Number	Title	Score	Details
3.7.1	When data changes (e.g. by a new user selection from the menu defined in 3.1.), update the diagram accordingly.	100%	As soon as a change in the data is detected, the diagram is updated.
3.7.2.	If possible, give visual feedback about the changes.	100%	We completely re-render the diagram. This makes the nodes all get placed in the center of the screen and get animated as if they are repelling each other (and later attracting the ones connected with edges). Even though this is a relatively simple approach, the user is still informed about the fact that the data changed, making this task "successfully completed".
3.7.3.	Only change the position of those nodes that are affected by the data change.	0	As described for 3.7.2, we completely re-render the whole diagram, making the nodes all move to random positions. Thus, we consider this task "failed".
3.8.	It should be possible to export the currently seen diagram	0	We have not implemented this feature in order to focus on other tasks.
3.9.	It should be possible to store the currently seen view so that it can later easily be recreated.	0	We have not implemented this feature in order to focus on other tasks.

Table 6.5: Evaluation of the requirements regarding the code quality.

Number	Title	Score	Details
4.1.	The code should be formatted according to the guidelines of CMSuite.	100%	We have not only tested the code against the linter, but also introduced new linter rules to enforce a more consistent coding style in CMSuite.
4.2.	All important functions/methods should have a comment documenting their parameters and purpose.	100%	We have added comments to all important functions/methods and also to model (and similar) files. If no comment was added to an element, this (conscious) decision was made because the usage is trivial.

Continued on next page

Table 6.5 (continued)

Number	Title	Score	Details
4.3.	For all components and services, there should be a test coverage of more than 90%.	100%	For almost all completely newly created components and services, we added tests providing a test coverage of more than 90%. It is possible to verify this using the code coverage reports that can be generated in Angular using the command *ng test --code-coverage --watch=false* In some rare special cases, we did not receive a test coverage of more than 90%. This was always due to the fact that the respective code contains parts that cannot easily be tested. For instance, a "switch" statement within a private method that includes a "default" case which cannot be reached with the current control flow (and only exists to make sure future changes to the code cannot lead to unwanted behavior) did lead to a relatively low "branches" coverage in the respective file. Also, we did not test the "TabbedMenuComponent" since the content is trivial (a list of links). Since there are only small exceptions to the "90% rule" that were also always a conscious decision, we consider this task "successfully completed"
4.4.	Files should not exceed a length of 400 lines to be better readable.	100%	We have successfully split all files in a way that none of them exceeds the length of 400 lines. It is possible to verify this by temporarily adding the linter rule "max-file-line-count" with the maximum line number set to 400.

6.1 Conclusion

Prior to this thesis, only the journal view existed. It was hard for users to understand the relationships within large data sets. The added Sankey and Graph views enable even non-technical individuals to explore the information about the patch-flow intuitively. The generated results provide a clean and organized look, ready to be used within presentations or reports.

The code developed for this thesis has been thoroughly tested and is designed in a way that additional features can be easily added in the future. Thus, the Sankey and Graph views can be altered to fit even the most complicated needs of the respective company.

7 Future work

We have successfully added rich visualizations of the patch-flow to CMSuite, making it possible even for non-technical users to explore the data sets intuitively. However, there are many parts that still can be improved.

As described in chapter 6, not all requirements were fully implemented. Especially features to export the diagrams as images or the possibility to "zoom" into nodes (in order to reveal the underlying organizational units belonging to the respective nodes) could greatly enhance the user experience. Also, there would be a better user experience if the Graph was not always redrawn when data changes.

Additionally, it would be great to enable the user to interact with the nodes, for instance to show additional information about a node (e.g. the number of ingoing/outgoing contributions) when it has been clicked.

The Graph nodes are currently ordered randomly and can be reorganized by the user. If they were automatically sorted in a more "intelligent" way, we could save the user a lot of time. For instance, the nodes could be grouped based on the hierarchy of the organizational unit(s) they belong to.

If it turns out that the generated diagrams are often exported, e.g. to be used in presentations, features that enable users to customize the diagrams are desirable. For instance, users could customize the colors and sizes of nodes and links, change or add captions, set the overall size of the exported diagram and (in the Sankey Chart) reorder the nodes. This could make it easier to make the visualizations fit the needs of the individual user.

References

Capraro, M., Dorner, M. & Riehle, D. (2018). The patch-flow method for measuring inner source collaboration. In *Proceedings of the 15th international conference on mining software repositories* (pp. 515–525). MSR '18. Gothenburg, Sweden: ACM. doi:10.1145/3196398.3196417

Capraro, M. & Riehle, D. [Dirk]. (2017). Inner source definition, benefits, and challenges. *ACM Comput. Surv. 49*(4), 67:1–67:36. doi:10.1145/2856821

Dinkelacker, J., Garg, P. K., Miller, R. & Nelson, D. (2002). Progressive open source. In *Proceedings of the 24th international conference on software engineering* (pp. 177–184). ICSE '02. Orlando, Florida: ACM. doi:10.1145/581339.581363

Holten, D. (2006). Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, *12*(5), 741–748. doi:10.1109/TVCG.2006.147

Riehle, D. [D.], Capraro, M., Kips, D. & Horn, L. (2016). Inner source in platform-based product engineering. *IEEE Transactions on Software Engineering*, *42*(12), 1162–1177. doi:10.1109/TSE.2016.2554553

Stol, K.-J., Avgeriou, P., Babar, M. A., Lucas, Y. & Fitzgerald, B. (2014). Key factors for adopting inner source. *ACM Trans. Softw. Eng. Methodol. 23*(2), 18:1–18:35. doi:10.1145/2533685