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

NILS HÄUSLER

MASTER THESIS

VISUALIZATION OF CODE COMPONENT ARCHITECTURE IN OPEN SOURCE SOFTWARE PRODUCTS

DEVELOPMENT OF A WEB-UI

Submitted on 15 May 2019

Supervisors: Andreas Bauer, M.Sc.
Prof. Dr. Dirk Riehle, M.B.A.

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, 15 May 2019

License

Erlangen, 15 May 2019

Abstract

Components are fundamental building blocks in software engineering. Modern day software products are often a complex composition of many components from entirely different vendors. Some are proprietary software, while others are open source components. The reuse of open source components has many many advantages but also brings up challenges, like license non-compliance, copyright issues or the potential risk of security weaknesses introduced through dependencies. These issues can be identified and managed by understanding the component architecture and the relationship between components. We call one representation of a software product architecture the product model, which is an enhanced version of what is known as the bill of materials. A useful visual representation of the product model doesn't exist at this point. Therefore the focus of this thesis is the implementation of an interactive tool the visualization of the underlying component graph, intending to help software developers to get a better understanding of the code component architecture (product model) of their software products.

Contents

1 Introduction

At the beginning the motivation for the present work and its structure and overview will be outlined.

1.1 Motivation

Modular software architectures and the development of generic software components allow extensive code reuse and the development of software products as complex compositions of many individual functional pieces. Especially the ongoing rise of open source software (OSS) has significantly contributed to the reuse of software. Today open source components (OSC) have become so relevant, that they can be found in most software products. Either directly or indirectly included, as transitive dependencies of other components. Deshpande and Riehle, 2008 have shown that the total amount of open source code and the total number of projects have been growing exponentially.

The usage of OSS has many benefits for both open source software developers and developers in firms. The possibility to review the source code, a large base of developers and testers, and the decreased risk of vendor lock-in are just a few of them (Morgan & Finnegan, 2007). Developers can also greatly save on development costs and time through code reuse (Haefliger, Von Krogh & Spaeth, 2008). OSS has also proven to be an economically viable alternative to proprietary software with a diverse set of working business models (Riehle, 2007).

But in spite of all benefits, there are also some drawbacks that developers have to be aware of when they incorporate OSS into their product. Legal aspects like license non-compliance and copyright are major issues that are challenging to work out. Copyleft licenses for example, like the GNU General Public License (GPL), have strong constraints that enforce derivative works to be subject to the terms of the same license. When it comes to copyright it is an often overlooked fact, that copyright protection comes into force at the moment in which the software is created, even if no explicit license is provided (Laurent, 2004).

Another issue is the potential risk of security weaknesses introduced through dependencies (Morgan & Finnegan, 2007). Vulnerabilities can easily be passed

through the dependency graph. Some known and easy to prevent vulnerabilities may go unnoticed because projects unknowingly include third-party components as transitive dependencies. A dependency that is several projects removed, is not easily visible and can be a possible hazard.

To manage these issues, it is first of all important for developers to be aware of them and then to understand the interactions between the different components. But because of the complexity of the relationships, this can be overwhelming. Assisting tools have potential to reduce the overhead for software engineers and analysts. Providing visualizations for a variety of viewpoints of a software system has many benefits (Councill & Heineman, 2001).

1.2 Overview and Structure

At the beginning, the underlying basics of graph theory are explained in chapter 2 (Foundations). In addition, the term software component is defined and the corresponding component model, the product model, is clarified. In the following chapter 3 (Technologies) the most important technologies, which were used for drawing the graph and for implementing the front-end, are presented. Chapter 4 (Requirements) presents the goal of the visualization tool and addresses the requirements for the implemented application. Here, mainly functional requirements and their purposes are described. The general architecture and the front-end design are explained in chapter 5 (Architecture and Design). It is followed by chapter 6 (Implementation), which describes the implementation of the parser, graph drawing, and the graphical user interface. This chapter also covers the publish-subscribe pattern, which is used for communication purposes. Chapter 7 (Evaluation) reviews and evaluates all previously defined requirements. In the last chapter, chapter 8 (Summary and Outlook) the essential parts of this thesis are summarized and possible further development steps are suggested.

2 Foundations

This chapter serves as a short basis of topics that are important in order to fully understand the rest of this thesis. Section 2.1 touches on the basics of graph theory by showing important representations, graph layouts and different types of graphs. In section 2.2 the term software component is defined, and the Product Model is introduced.

2.1 Graph Theory

In graph theory, a graph or network is used to model the pairwise relationships of entities. Each graph $G = (N, E)$ consists of two finite sets, one for the entities (nodes) $N = \{n_1, n_2, \dots n_i\}$ and one for the relationships (edges) $E = \{e_1, e_2, \dots e_j\}$, where each edge e_j is a two-node subset describing their relationship. A simple example with four nodes and four edges would be defined like this:

$$
\begin{aligned}
&N = \{n_1, n_2, n_3, n_4\} \\
&E = \{e_1 = \{n_1, n_2\}, e_2 = \{n_1, n_3\}, e_3 = \{n_4, n_1\}, e_4 = \{n_3, n_4\}\} \\
&G = (\{n_1, n_2, n_3, n_4\}, \{\{n_1, n_2\}, \{n_1, n_3\}, \{n_4, n_1\}, \{n_3, n_4\}\})
\end{aligned}
$$

Remark. many different terms denote the elements of a graph in literature. Nodes are often referred to as vertices or points. Edges are also called links, lines or arcs.

2.1.1 Graph Representations

There are many different ways to represent graph structures, but two of the most common 2D-representations are matrix-based representations (figure 2.1) and the so called node-link diagram (figure 2.2).

	n_1	n_2	n_3	n_4
n_1	0	1	1	1
n_2	1	0	0	0
n_3	1	0	0	1
n_4	1	0	1	0

	e_1	e_2	e_3	e_4
n_1	1	1	1	0
n_2	1	0	0	0
n_3	0	1	0	1
n_4	0	0	1	1

Figure 2.1: Adjacency matrix(left) and incidence matrix(right) of graph G

Matrix-based representations are usually two dimensional. In the adjacency matrix, each node of the underlying graph is represented by a row and a column. An entry in the adjacency matrix means, that there is an edge between the node represented by the column of the matrix, to the node represented by the row. Another matrix-based format is the incidence matrix, where each row represents a node of the graph and every column represents an edge. Each column has two entries, one for each node, that the respective edge connects. The number of entries in each row is the same as the represented node has edges. The order of the elements in the matrices is generally alphabetically but is often changed to highlight specific features of the graph.

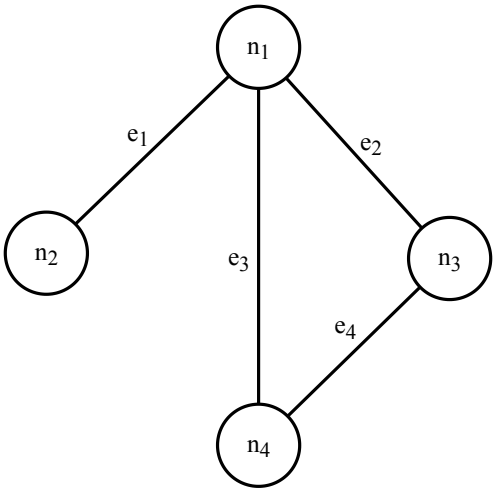


Figure 2.2: Node-link diagram of graph G

The appearance of a node-link diagram is affected by the topology, geometry and visual features of the elements. The topology is given by the graph description and can't be changed without changing the graph itself. The geometry of

a graph refers to the positions of the nodes. These can be chosen at will, but are typically computed with a specific layout in mind (see: subsection 2.1.2). Visual features of the elements, such as shape, size or color can also be chosen freely to enhance the perception of the graph.

There are many differences between node-link diagrams and matrix-based representations in terms of readability and comprehensibility. The task for a user to find paths between two nodes in a graph is, for example, usually achieved much faster on node-link diagrams. Matrix-based representations, on the other hand, tend to perform better, when graphs become larger and denser. One key point for both representations is the degree of familiarity with the underlying data set. Experience has a substantial impact on the ability to quickly understand the graph representations (Keller, Eckert & Clarkson, 2006) (Ghoniem, Fekete & Castagliola, 2004).

2.1.2 Graph Layouts

The positioning of the nodes in node-link diagrams is crucial for the comprehensibility of the visualized graph. Especially, when graphs get bigger and more complex visualizations tend to get overwhelming. Some basic rules of good graph layouts are a uniform distribution of nodes, minimal edge crossings, uniform edge length and highlighting of the underlying structure (Fruchterman & Reingold, 1991).

One widely used approach is the force directed layout introduced by Fruchterman and Reingold, 1991. It has been highly optimized since then, but the basic idea is truly powerful and easy to explain. Starting from random positions the iterative algorithm computes repulsing and contracting forces for each node in the graph. These forces vary depending on the relative position of the node to all other nodes and the relative position to connected neighbors. This iterative simulation runs until an equilibrium is reached and the graph is in a relaxed state. Layouts like this are called physics based layouts.

Layered graph layouts (also called Sugiyama-style graph layouts after Sugiyama, Tagawa and Toda, 1981) are based on the fact, that typically there is a general direction for edges in directed graphs and directed acyclic graphs (DAGs). First a rank is assigned to each node, with some heuristics. Then nodes can be arranged into horizontal layers, with a depth-first search method, which exposes the underlying hierarchical structures in the graph (Gansner, Koutsofios, North & Vo, 1993).

Other kinds of graph layouts are the circle, concentric circle or grid-based layouts. The choice for the appropriate layout always depends on the type and specific characteristics of the graph, as well as the general goal of the visualization.

2.1.3 Types of Directed Graphs

Graphs can be classified into many different types, based on their characteristics. In general, a distinction is made between directed and undirected graphs. A graph is called directed, if the subsets of edges e_j in the graph are ordered and have a direction associated. This means, that the nodes which are connected by edges have a transitive relation. Figure 2.3 shows the same graph as before, but with directed edges.

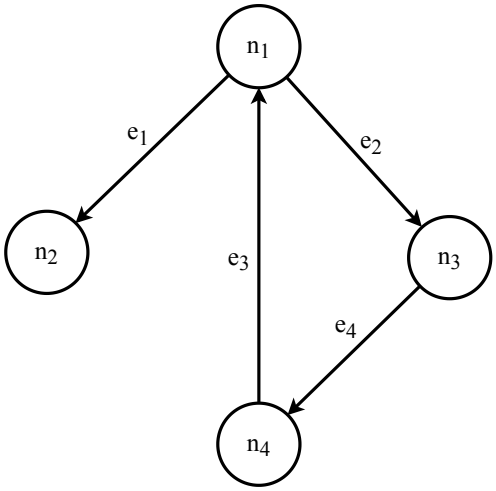


Figure 2.3: Directed node-link diagram of graph G

Directed acyclic graphs (DAGs) are special forms of directed graphs that have no directed cycles. This means that there is no path of successive directed edges that starts and ends at the same node in the graph (figure 2.4). DAGs have many applications in computer science but are also relevant in general. One example is the citations graph of publications in academia. Each publication is a node and each citation is a directed edge. Because it is only possible to cite works, that have already been published in the past, resulting citation graphs are DAGs.

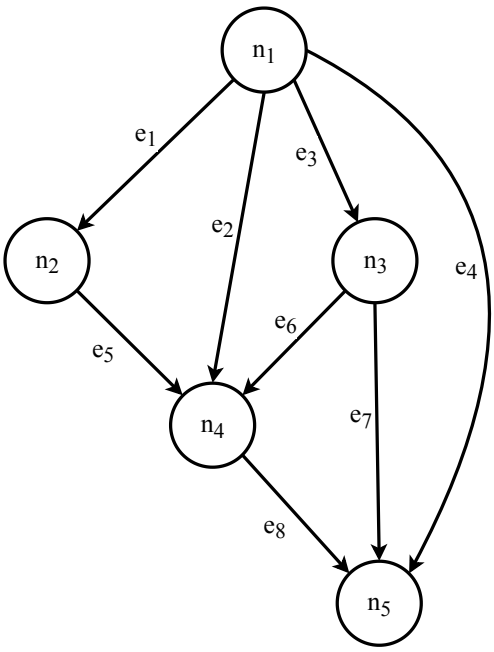


Figure 2.4: Node-link diagram of a directed acyclic graph (DAG)

2.2 Software Components

2.2.1 Definition

Modern software engineering has a strong focus on separation, with the goals of reuse, flexibility, extensibility, and maintainability of the individual parts. Applications are broken down into important parts based on functional behavior and build from independently working pieces, which are referred to as components. The term *software component* has no clear and precise meaning in literature, and the given definitions differ with context and specific domain (Szyperski, Bosch & Weck, 1999). To have a common ground with preceding and future works this thesis follows the given definition:

> A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.
>
> —Councill and Heineman, 2001

2.2.2 The Product Model

The component model that underlies this thesis was developed at the Professorship for Open Source Software at the Friedrich-Alexander-University Erlangen-Nürnberg and is called the Product Model. It provides a general way to model the component architecture of software products and results from the necessity to fulfill the requirements on compliance tools (Harutyunyan, Bauer & Riehle, 2018).

The starting point of this model is the software project itself. A project can contain several different root components, which in turn can depend on several different other components. These dependency relationships between components can be of different types (e.g., dynamically or statically linked), but they always form a hierarchical structure, that represents a directed acyclic graph. Each component contains information about the software artifact it represents and a list of metadata, such as license or interface data. The diagram in figure 2.5 shows the general structure of the Product Model.

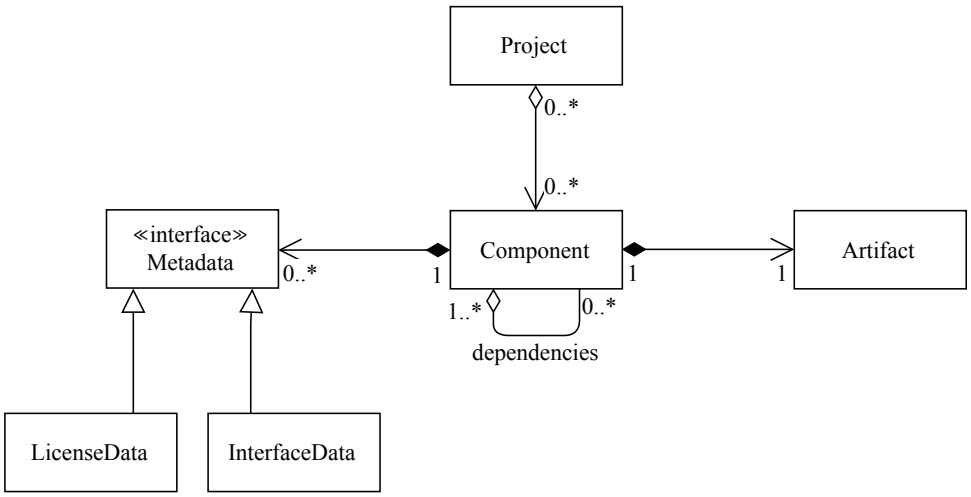


Figure 2.5: General structure of the Product Model

The Product Model is intended to be processed as common machine-readable file formats such as JSON, XML or YAML. In listing 2.1 exemplary JSON code is shown, which depicts the component architecture of a specific project via the Product Model.

Listing 2.1: JSON code that describes the component architecture of an exemplary project

```
{
  "name": "ProjectName",
  "homepageUrl": "https://github.com/user/ProjectName",
  "vcs": "https://github.com/user/ProjectName",
  "declaredLicenses": ["Apache 2.0"],
  "buildTool": "MAVEN",
  "rootComponents": [
    {
      "name": "Component A",
      "namespace": "",
      "version": "2.0",
      "artifact": {
        "filePath": "/ProjectName/jars/component_a.jar",
        "hashAlgorithm": "SHA512",
        "hash": "B1B556692341A240F8B81F8F71B8B5C0225CCF"
      },
      "metaData": [
        {
          "description": "License",
          "value": "MIT"
        },
        {
          "description" : "Interface",
          "value" : { ... }
        }
      ],
      "dependencies": [
        {
          "target": {
            "name": "Component B",
            "namespace": "",
            "version": "3.4",
            "artifact": { ... },
            "metaData": [ ... ],
            "dependencies": [ ... ]
          },
          "type": "DYNAMIC_IMPORT"
        },
        ...
      ]
    },
    ...
  ],
}
```

Dependencies in software projects are usually managed by build systems or package managers like Maven or the Node.js package manager (NPM). The

necessary information to construct the Product Model of a specific software project can be extracted from these tools. A crawler that does this for Maven based Java projects was implemented as part of a preceding master thesis at the Professorship for Open Source Software at the Friedrich-Alexander-University Erlangen-Nürnberg (Scheffer, 2018).

3 Technologies

This chapter introduces the main technologies used for the implementation of the dependency graph visualization tool. In section 3.1 the choice of the graph drawing library based on an evaluation of existing libraries is explained. Section 3.2 covers some essential aspects of the front-end framework and state management, that has been used.

3.1 Graph Drawing Library

A variety of web-based open source visualization libraries have been evaluated before the design and implementation of the tool began. Five of the libraries that have been identified as the most promising ones, have been looked at more closely. These libraries are Cytoscape.js (Lopes et al., 2015), D3.js (Bostock, Ogievetsky & Heer, 2011), Sigma.js[1], Vega (Satyanarayan, Moritz, Wongsuphasawat & Heer, 2017) / Vega-Lite (Satyanarayan et al., 2017) and VivaGraph.js[2]. All of them are licensed under permissive open source licenses. Either under the BSD 3-clause license (D3.js, Vega / Vega-Lite, and VivaGraph.js) or the MIT license (Cytoscape.js and Sigma.js).

Cytoscape.js, Sigma.js, and VivaGraph.js are dedicated graph drawing libraries, which allow the user to render a graph as an interactive node-link diagram. But they are not able to visualize a graph in a different representation. D3.js and Vega / Vega-Lite, on the other hand, are general data visualization libraries. They are not restricted to relational data or specific representations and can be used to render graphs as a node-link diagram and other representations, like the adjacency matrix. The flexibility of D3.js and Vega / Vega-Lite comes with a trade-off in complexity, which makes them harder to use and understand. Other compromises are derived from the rendering engine technology (Scalable Vector Graphics (SVG), Hypertext Markup Language (HTML) Canvas or WebGL) that is used by the library to render the graph. The performance, for instance, scales differently. SVG performance depends on the size and complexity of the scenegraph (which relates to the number of elements in the graph), while HTML Canvas performance is more dependent on the size of the view area. The best performance can be achieved with WebGL based rendering. But performance

[1] https://github.com/jacomyal/sigma.js
[2] https://github.com/anvaka/VivaGraphJS

gains generally come with the loss of support for interactivity features (i.e. event listeners on the graph elements). A comparison of important features and other criteria that were evaluated to choose a fitting visualization library is shown in table 3.1.

	Graph Specific	Graph Analytics	Performance	Interaction API	Documentation
Cytoscape.js	Yes	Yes	Satisfying	Very Satisfying	Very Satisfying
D3.js	No	No	Neutral	Very Satisfying	Satisfying
Sigma.js	Yes	No	Satisfying	Very Satisfying	Satisfying
Vega / Vega-Lite	No	No	Neutral	Satisfying	Very Satisfying
VivaGraph.js	Yes	No	Very Satisfying	Unsatisfying	Unsatisfying

Table 3.1: Comparison of graph visualization libraries

From this comparison, Cytoscape.js stands out as a full graph theory library, that is developed not only for graph visualization but also for analysis. A rich API for graph analytics and interactions with the graph elements allows for graph manipulations in every way. With HTML Canvas as underlying rendering engine technology, the performance is good, and thousands of graph elements can be rendered on standard hardware (Lopes et al., 2015).

Other reasons that speak in favour of Cytoscape.js are:

- A sophisticated graph model
- Support for directed graphs and compound nodes
- Separation of graph style and graph data
- Access to numerous layout algorithms
- An extension API for UI, layout and the core
- Many existing extensions (first and third-party)
- Export of the graph data in JSON format

3.2 Front-End

This section covers the front-end framework Vue.js and the state management mechanism of the Vuex library.

3.2.1 Framework

The initial implementation of the application was done in vanilla JavaScript without any particular framework. But because the code lacked structure and state management became an issue, the front-end was rewritten in Vue.js[3]. The main reason that Vue.js was used instead of other well-known frameworks, like React or Angular, was vue-cytoscape[4]. Which is a well-maintained wrapper library, that helps with the integration of Vue.js and the used graph drawing library Cytoscape.js.

Vue.js describes itself as a progressive framework, that can be adopted incrementally. The core is just a single library without any dependencies to other libraries. It takes care of displaying data on an HTML page (view binding), handling user interactions (event binding) and takes input values from forms (input binding). Single-file components (vue-components) structure the application in a concise way and keep all the relevant source code (markup, styling, and functionality) of UI elements together in one place (listing 3.1).

Listing 3.1: Structure of a single file component in Vue.js

```
<template>
  <v-tab-item class="tab-content">
    <v-card-text>
      <pre>{{ $store.state.sidebar.dataTabContent }}</pre>
    </v-card-text>
  </v-tab-item>
</template>

<script lang="ts">
  import { Vue, Component } from 'vue-property-decorator';

  @Component
  export default class DataTab extends Vue {}
</script>

<style scoped>
  pre { font-size: 2em; }
</style>
```

3.2.2 State Management

Vuex[5] is a state management library for Vue.js applications. It provides a centralized way to store the application state and ensures that changes to the state are correctly propagated to all relevant UI components. The shared application

[3] https://github.com/vuejs/vue
[4] https://github.com/rcarcasses/vue-cytoscape
[5] https://github.com/vuejs/vuex

state is stored and managed by Vuex in a global singleton. The underlying state management pattern can be seen in figure 3.1.

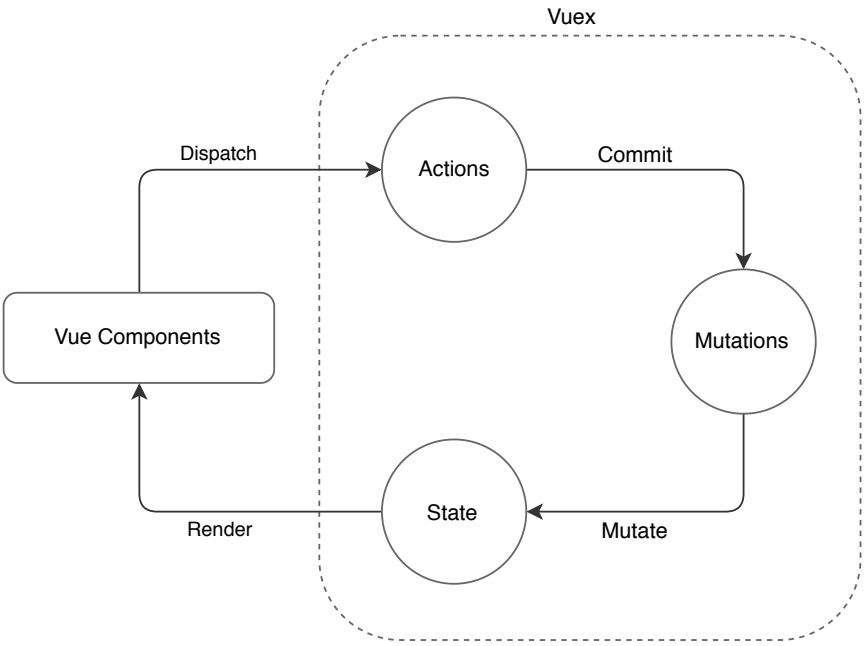


Figure 3.1: The Vuex state management pattern[6]

To change the state vue-components dispatch actions based on user interactions. These actions then commit one or multiple mutations with the desired changes. Committing mutations is the only method to modify the state of the global store. Mutations in contrast to actions have to be synchronous. When the state in the store is mutated, the changes are automatically propagated across the application, and all relevant UI components are getting updated and rerendered. This is done efficiently via the reactivity mechanisms of Vue.js.

[6]Image taken(edited) from https://vuex.vuejs.org/

4 Requirements

The objective of the visualization tool that was implemented as part of this thesis is clarified in section 4.1. Section 4.2 specifies the functional and qualitative requirements, that were set to create a useful application, which accomplishes the goal of interactive dependency graph visualization.

4.1 Purpose of the Tool

The reuse of software has been a general goal of software engineering since the beginning. Today software projects are built from many parts, with many different libraries, which often stem from entirely different sources. This is only possible because of modern tooling and dependency managers, that make publishing and consuming dependencies incredibly easy, by reducing the overhead for developers to approximately zero. But because it is so easy to introduce new dependencies, the count of imported software components in a software project can grow quickly to a relatively high number. One reason for this is, that an included dependency can additionally have multiple other transitive dependencies.

Simple static visualizations of all software components and their dependencies are often overwhelming and insufficient. The aim of the application/visualization tool is therefore, to visualize the software component graph of software projects in a dynamic and interactive way, so that the users are enabled to navigate and explore the graph as a whole, or to focus on selected parts. With the overall goal to provide better insight and a deeper understanding of the component architecture of the project.

4.2 Application Requirements

Some essential requirements have been identified before the design of the architecture and the implementation of the tool started. The requirements are divided into functional (**F**) and qualitative (**Q**) requirements. Functional requirements specify specific operations, that can be performed by the application or the user. Qualitative requirements refer to specific quality criteria, that can be measured in numerical values to be evaluated later.

F1: Parsing

The basis of the visualization is the input data (crawler artifact). The application must be able to parse the given input data and to generate an internal representation, from which in turn the visualization can be created. The output of the parser is also the foundation for further internal operations, like manipulation and the analysis of subgraphs. A successful implementation of the parser generates output, that conforms to the chosen graph model and includes all software components, their dependency relationships and the metadata.

F2: Visual Representation

The most important requirement is the correct visualization of the component graph itself. This requires the software to be able to generate a complete visual representation of the underlying dependency graph, with all transitive relationships between the components. Correct visualization means, that the number of elements displayed in the graph representation is equal to the number of elements contained in the input data and the output from the parsing step.

F3: Navigation and Exploration

Navigation allows the user to move freely within the graph representation and to explore the structure of the graph itself in order to develop an understanding of its complexity. To enable this, the visual representation has to support basic navigation functionality with gestures, like zooming, panning, and dragging of selected elements.

F4: Focus

The tool should allow the user to highlight and focus on selected subgraphs or specific components with all their dependencies, to reduce the visual complexity of large component graphs. This enables the user to concentrate on the parts of the dependency graph that are relevant to him.

F5: Data View

Each software component (node) and each dependency relationship (edge) that is visualized in the graph representation can have associated data. To make this information available to the user, a data view which displays the attached information of a selected graph element should be implemented as part of the application. The displayed information has to include associated metadata, if the selected element is a node (software component).

F6: Filtering

Another important functional requirement is filtering. The UI should have the functionality to filter the represented graph, based on the components names and attached metadata. This helps the user to reduce the complexity of the graph, to find relevant software components or to only inspect partial structures of the graph.

Q1: Performance

The only qualitative requirement that was identified is performance. The implementation of the application should address the performance of the parsing step, the application load time and the scalability of the graph representation with an increasing number of elements. To assess the performance of the visualization tool, common action should be performed on different dependency graphs while measuring the execution time and the resulting delays. The evaluated graphs should be of different size and with different element counts (number of nodes/edges).

5 Architecture and Design

This chapter provides an overview of the overall architecture and design of the application. First, the primary building blocks are introduced in section 5.1. Then, in section 5.2, the concept and structure of the UI is described in detail, and the different UI components and their respective tasks are introduced.

5.1 Architecture Overview

A good way to understand the general architecture of the Product Model Graph application is by following the flow of the data. The input data (crawler output) is generated beforehand by external tools, like the Maven Crawler (Scheffer, 2018) or the analyzer of the OSS Review Toolkit (ORT)[1]. These tools analyze repositories or build artifacts and extract dependencies and corresponding metadata of software components. The resulting data is then served to the application via a simple HTTP server. This could be changed later to a more sophisticated server, which provides a proper REST API in order to additionally serve partial data when requested from the client.

The transfer of the input data marks the actual entry point into the Product Model Graph application. The first step, after receiving the data is to create a representation, that describes the dependency graph and can be rendered. This means the data has to be conform to the graph model, that is expected by the graph drawing library. The generation of this graph model conform representation is the task of the parser. Since the input data can be generated by different tools and thus have different data and file formats, one parser implementation is not sufficient. Each combination of data format and extraction tool needs a special parser implementation, that generates the graph model conform representation with the correct dependency graph. The implementation details of the parser are explained in more depth in section 6.1.

Once the parser has generated the graph model representation, it can be transferred to the front-end, where the actual graph is rendered and displayed in a view. The architecture diagram in figure 5.1 gives a general overview and shows all elements and their connections.

[1] https://github.com/heremaps/oss-review-toolkit

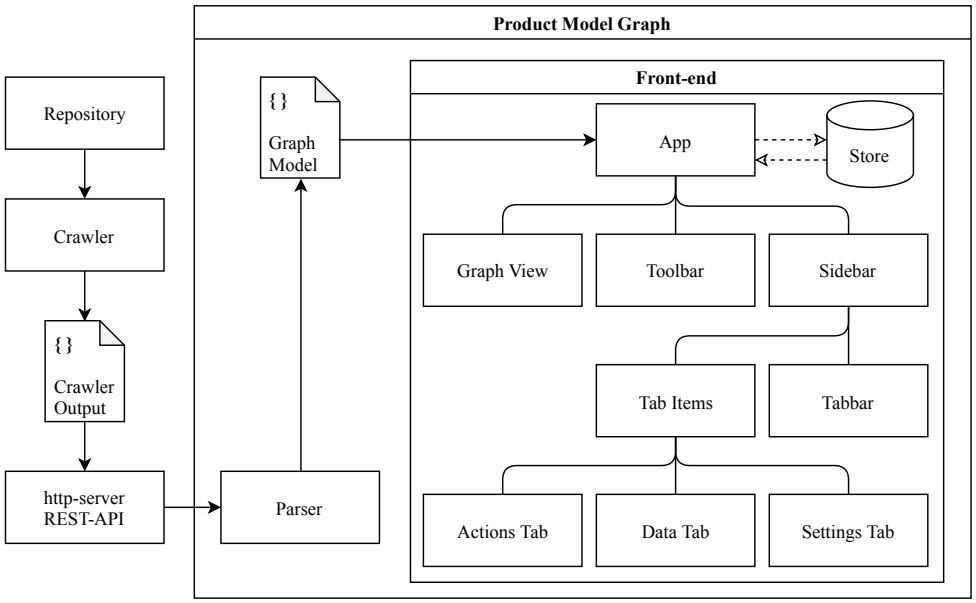


Figure 5.1: Architecture overview of the Product Model Graph application

5.2 Front-End Design

The graphical user interface (GUI) of the Product Model Graph application should enable users to interactively inspect the dependency graph of their software project. An important part of accomplishing this, and the implementation of the requirements listed in section 4.2, is the concept and structure of the UI components. The main UI components are the Graph View, the Toolbar and the Sidebar which consists of the Tabbar and three Tab Items (Actions Tab, Data Tab and Settings Tab).

The Graph View is the largest and most important part of the application. It serves as the canvas for the graph drawing library to render the dependency graph on. The Toolbar provides quick access to essential features like the filter input field and other functions, that are commonly performed. Anchored to the side of the window is the Sidebar, which contains the Tabbar and the associated Tab Items. The Tabbar displays several buttons for switching between the different Tab Items (subviews) of the Sidebar. Controls and other GUI elements, with which the user can issue commands to manipulate the graph and its elements are located within the Actions Tab. The Data Tab is mainly a container for textual metadata of the currently selected graph element. Settings for both the graph's appearance and the application as a whole, can be found in the Settings Tab. Figure 5.2 shows the relationship between the UI component hierarchy and the layout of the graphical user interface (GUI).

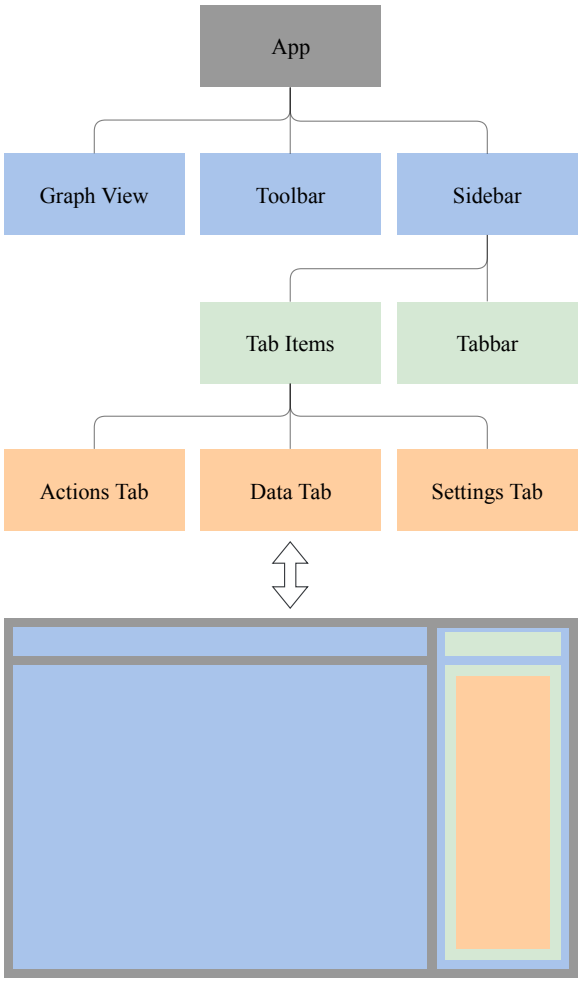


Figure 5.2: Context between the hierarchical and GUI layout of the UI components

6 Implementation

This chapter describes important aspects and details of the implementation. The parsing step and data generation for Cytoscape.js is explained in section 6.1. Next, in section 6.2, the rendering and use of Cytoscape.js for graph drawing are described in more detail. Section 6.3 describes all elements of the GUI in detail. And finally, section 6.4 explains how the communication is handled in order to overcome issues related to storing the graph instance in the global state.

6.1 Parsing Component Graphs

The goal of the parsing step is to generate the data representation, that can be processed and rendered by Cytoscape.js. This is accomplished by extracting the dependency graph and other relevant data from the input data. Since the input data may come from different sources in different formats, a separate parser has to be implemented for each format and data source. In the process of this thesis, two parser implementations were realized. One for the output of the Maven Crawler and one for the analyzer output of the OSS Review Toolkit. This demonstrates that the application can be easily extended to support other tools as an import source. The implementations differ in details, but the general approach is the same in both cases (see figure 6.1). The recursive identification of software components with their dependencies and the creation of graph model compliant elements for each of them. Nodes (listing 6.1) are created for every software component and edges (listing 6.2) for their dependencies.

Listing 6.1: Graph model conform notation for a node

```
"data": {
  "id": "name.space:Component A:2.0",
  "name": "Component A",
  "namespace": "name.space",
  ... // attached data (artifact, license, ...)
}
```

Listing 6.2: Graph model conform notation for an edge

```
"data": {
  "id": "name.space:Component A:2.0->
         name.space:Component A:2.0:Component B:3.4",
  "source": "name.space:Component A:2.0",
  "target": "name.space:Component B:3.4",
  "type": "DYNAMIC_IMPORT"
}
```

The Cytoscape.js graph model notation requires only a *data* object with an *id* field to create a valid node. If the fields *source* and *target* are also defined, then the element is interpreted as an edge between two nodes. Other data such as metadata can be attached to the object by creating custom fields with names, that are not reserved by the graph model notation. The complete graph model description[1] can be found in the documentation of Cytoscape.js.

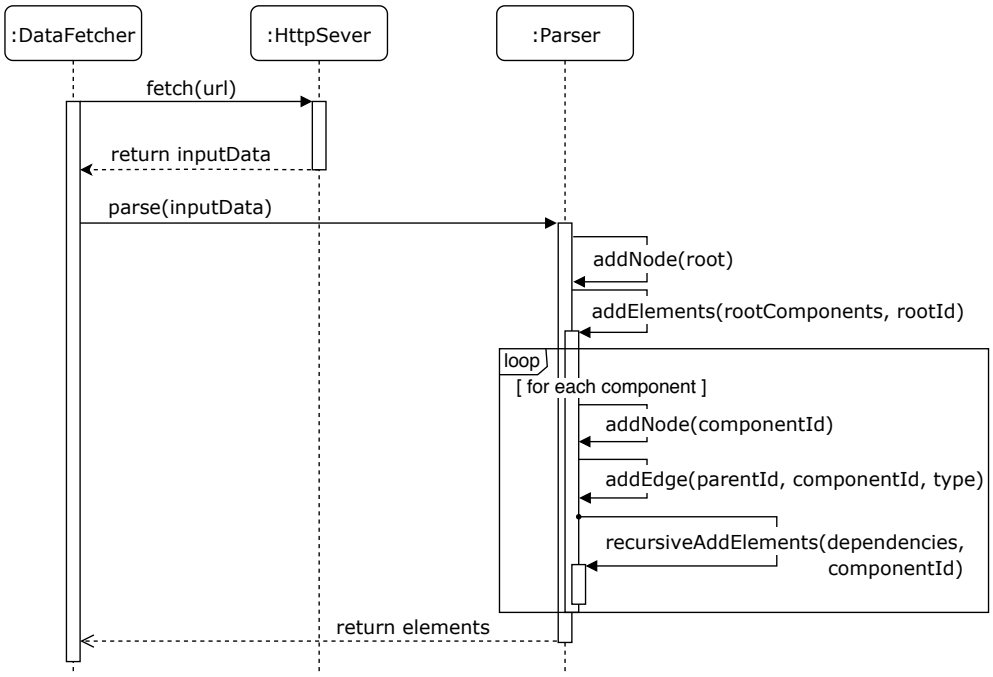


Figure 6.1: Sequence diagram of the data fetching and parsing step

The input data is requested from the server and after it has been received, the parsing starts with the selection of the suitable parser implementation. This selection is based on the structure of the data and the existence or lack of certain data fields.

[1] http://js.cytoscape.org/#notation/elements-json

In *addNode()*, the first node object is created for the root, and its associated data is attached. Then the *addElements()* function is called, with the list of all root components and the root as parent ID. Now for each root component, a new node with metadata and a directed edge from root to the current component is created. Every root component has a list of dependencies, which all have the same fields and a list of their own dependencies. The implementation of *recursiveAddElements()* is analogous to the process just described, but makes use of the recursive structure of the remaining data, by calling itself at the end. After the whole dependency tree is recursively parsed, duplicated entries get removed. At last a large array of *data* objects gets returned as the result of the parser

6.2 Graph Drawing

In Cytoscape.js a graph corresponds to a cytoscape instance. To create and visualize a cytoscape instance four things have to be provided: the *container*, the *style*, the *elements*, and the initial *layout*. Listing 6.3 shows the complete initialization of the cytoscape instance as vue-component with the vue-cytoscape wrapper library.

Listing 6.3: Initialization of the cytoscape instance in the Graph View vue-component

```
<template>
  <div id="cytoscape-container" v-hotkey="keymap">
    <cytoscape :config="config" :preConfig="preConfig"
               :afterCreated="afterCreated">
    </cytoscape>
  </div>
</template>

<script lang="ts">
...
@Component
export default class GraphView extends Vue {
  private config = { style: defaultStyle };
  ...
  private async afterCreated(cy) {
    const elements = await fetchData(RESOURCE_URL);
    this.replaceGraph(elements, initialLayout);
    ...
  }
  ...
}
</script>
```

The *container* is a simple HTML <*div*> element, that contains the cytoscape instance. It is important to note, that the dimensions of the *container* element also define the size of the canvas on which the graph is rendered. The *style* is set to the default style, which was predefined and imported. The *elements* and the *layout* cannot be set in the configuration, because of the asynchronous fetching of the data and the parsing step. For this reason, the *elements* get added in the *afterCreated()* function. After all graph *elements* have been added, the initial *layout* is created from a predefined default layout and then applied.

6.3 Graphical User Interface

The following section describes all implemented elements of the graphical user interface. An overview of most elements can be seen in figure 6.2.

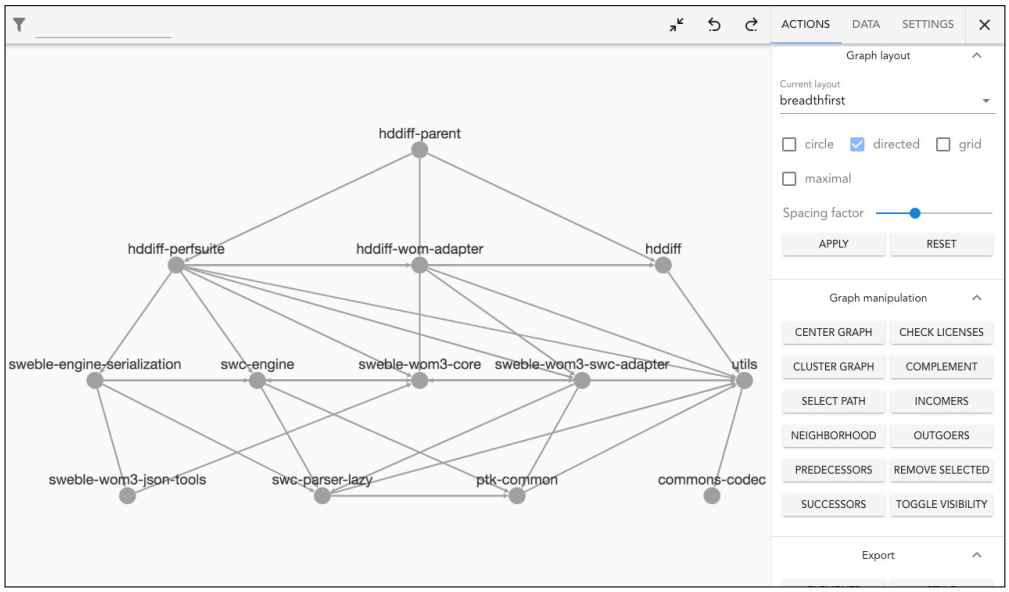


Figure 6.2: The graphical user interface of the application

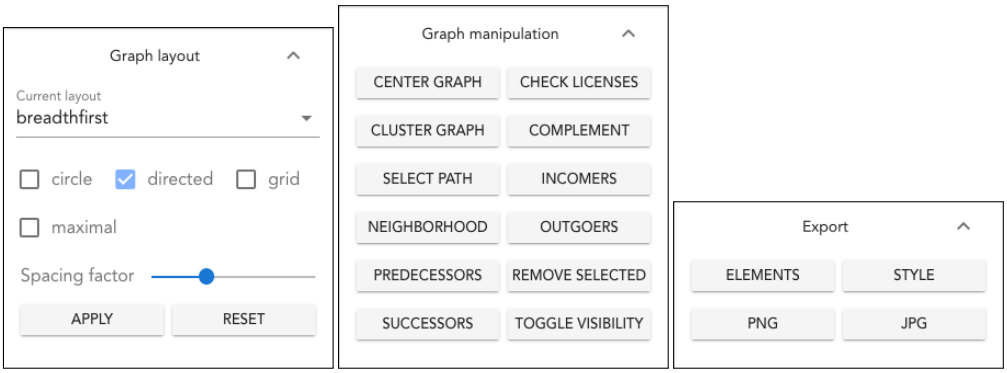
6.3.1 Graph View

The graph view takes up the most space of the user interface and extends across the entire width of the application, when the Sidebar is unopened. It serves as a canvas to display the graph representation. The drawing of the graph itself has already been described in section 6.2 and is therefore not discussed any further here.

6.3.2 Sidebar

The Sidebar, which is located on the right side of the interface, is composed of the Tabbar (figure 6.3) in the top part and the area for the Tab Items, that occupies the remaining space. The currently active tab is displayed in the Tab Items area. Thereby a distinction is made between the Actions Tab, Data Tab, and Settings Tab. The former is divided into the sections graph manipulation, graph layout, and export (figure 6.4).

Figure 6.3: The Tabbar



(a) Graph layout (b) Graph manipulation (c) Export

Figure 6.4: The three sections of the Actions Tab

In the graph layout section, the current layout is selected from various predefined options (e.g. breadthfirst, concentric, dagre). Depending on the selected layout option, the user is provided with further configuration settings. In the area graph manipulation, different buttons are displayed which trigger specific actions. The last section named export allows to download the graph data as JSON format, or a snapshot of the currently displayed elements in the viewport as png or jpg.

The second tab is the Data Tab. It allows the user to obtain further information by displaying all the attached data of the currently selected graph element. This is done as text, because the size and attributes of the attached data can be different for each graph. The implementation is realized by putting the actual JavaScript object, that holds the data in an HTML <*pre*> element. This preserves both spaces and line breaks and presents the data precisely as it is.

The last Tab Items is the Settings Tab. Here adjustments can be made to the visual appearance of the graph, by changing settings related to nodes and edges. Also general application settings can be configured (see appendix A). Figure 6.5 shows the to sections graph settings and application settings with all possible setting options. All three Tab Items can be seen as full-size images in appendix B.

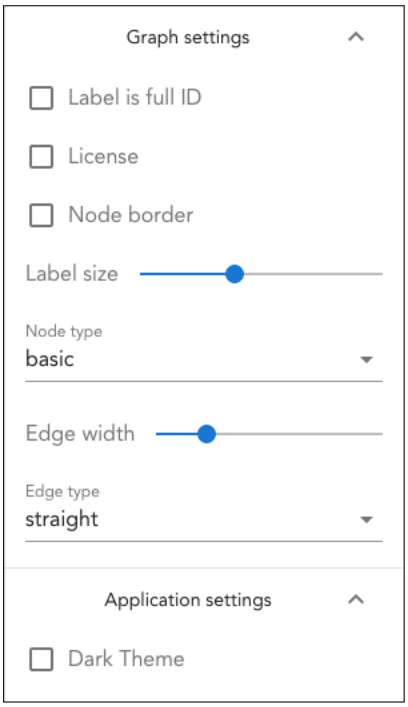


Figure 6.5: The Settings Tab

6.3.3 Toolbar

The Toolbar is located in the upper part of the application and extends from the left edge to the Sidebar on the right. It serves as a container for frequently used functions and provides quick access to them. Currently, it contains the input field for the filter functionality and buttons for undo/redo and to fit the graph to the size of the Graph View.

6.4 Communication

Storing the application state with the Vuex library and managing state changes as described in subsection 3.2.2 works fine for most parts of the application. But unfortunately, when the cytoscape instance, which holds the actual rendered graph object, is managed with Vuex, the application breaks down. This is not a big problem, because the cytoscape instance can easily be stored in the Graph View component and is treated as local state.

The question is now how to communicate actions, that are triggered somewhere else in the application to the Graph View component. Using the built-in data flow mechanics of Vue.js, by emitting events to parent components and passing data down to child components with props is tedious and cumbersome because

many different vue-components would be involved. A more fitting approach for the use case of communication actions to the Graph View component is the publish-subscribe pattern (figure 6.6).

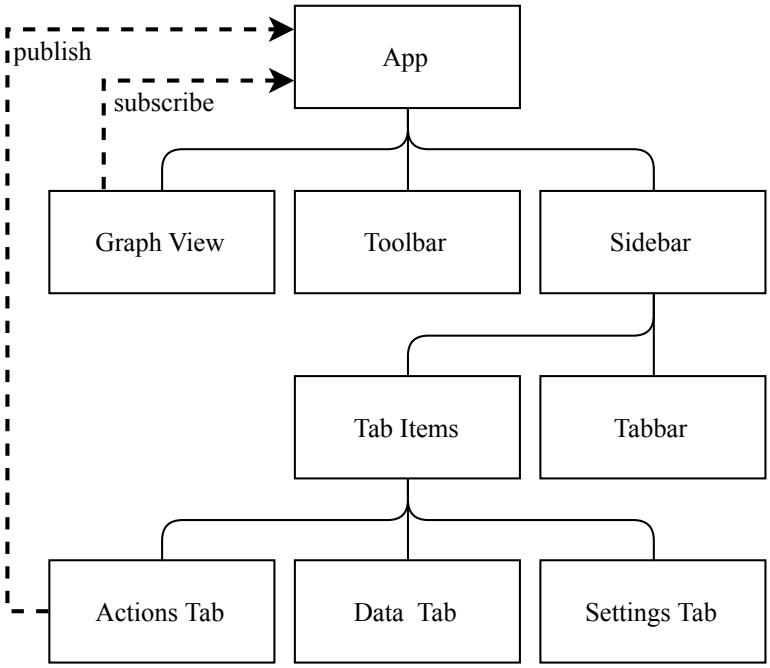


Figure 6.6: The publish–subscribe pattern for communication

This pattern is widely used is for communicating messages without coupling the involved parts (Eugster, Felber, Guerraoui & Kermarrec, 2003). When an action is triggered, or a change happens, an event is emitted directly to a global object, the event hub. Other vue-components register listeners on the event hub to be notified when a specific event occurs. The event hub could be a separate vue-component, that is globally accessible, but the App as the root vue-component is an obvious choice. Listing 6.4 shows the relevant source code for the registration of listeners and listing 6.5 shows the emissions of an event.

Listing 6.4: Registration of an event listener for the apply-layout event

```
// in GraphView.vue

private mounted() {
// register listeners on event hub
  this.$root.$on('apply-layout', () => {
    this.runLayout();
  });
}
```

Listing 6.5: Emission of the apply-layout event

```
// in ActionsTab.vue

<v-btn small block class="elevation-1"
@click="$root.$emit('apply-layout')">Apply
</v-btn>
```

Using this simple way to convey messages across the application separates responsibilities from the start and keeps the source code maintainable, by retaining the relevant source code within the responsible vue-components. Adding and removing listeners at runtime is also possible if needed.

7 Evaluation

In this chapter, the developed application is assessed by evaluation of all requirements that have been defined in chapter 4.

Evaluation F1: Parsing

In the course of this work two parsers were implemented. The parsers convert the given input data to a format, which is conform to the graph model notation provided by the graph drawing library Cytoscape.js. The requirement demanded, that all software components and their dependencies should be correctly processed and included in the output of the parser. To ensure this, the number of elements before and after the parsing step were compared. First, several representative input files were selected for the two implemented parsers. The input files were evaluated manually to obtain the initial number of software components and dependency relationships. These values were then compared to the number of elements generated by the parsers. No deviations were found in any of the tested input files. A detailed investigation of the runtime of the parsing step was also carried out and is described in the section of the performance evaluation (Q1).

The support of multiple input models by different parser implementations allows to compare them in a graphical way. Doing this for the Maven Crawler and the analyser of the OSS Review Toolkit has revealed some differences in the number of dependency relationships found by them. This is an interesting finding, but was not further investigated in the context of this work.

Evaluation F2: Visual Representation

The requirements for the visual representation of the component graph were fulfilled by using the graph drawing library (Cytoscape.js) to implement the Graph View as core of the application. It is the element of the graphical user interface (GUI) that allows the users to view and interact with the dependency graph of their projects. The choice to use Cytoscape.js as the visualization library, allowed the visualization of the dependency graphs as completely dynamic and interactive node-link diagrams with many different graph layout options (see figure 7.1). Especially the dagre[1] option delivers great results, with computed layouts that emphasize the underlying hierarchical structure of the dependency graphs.

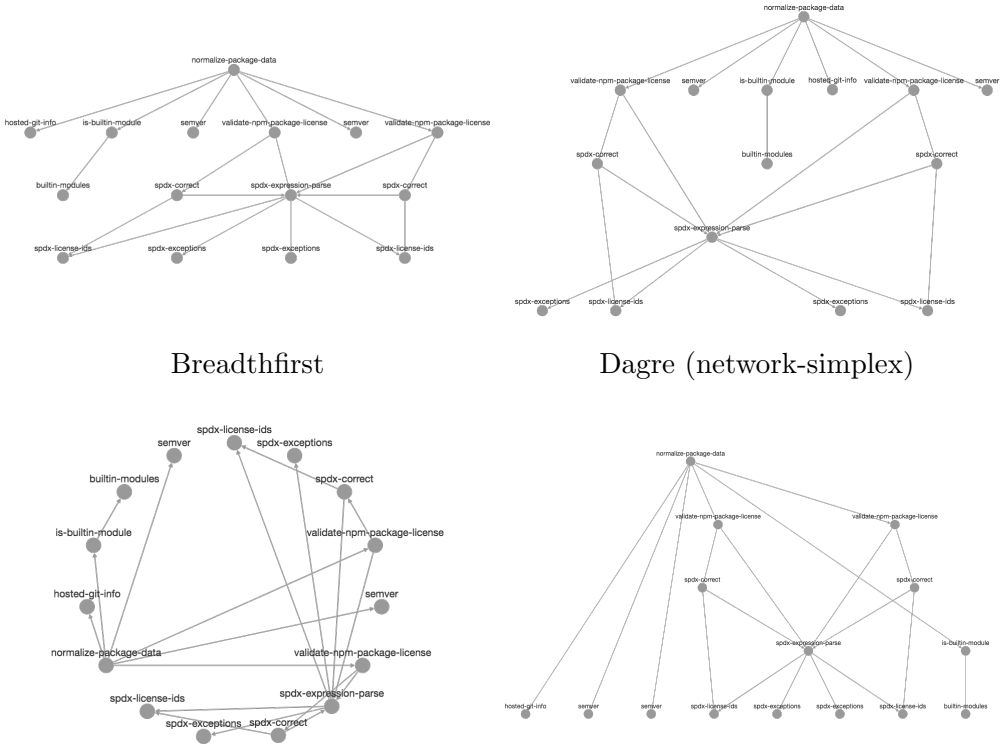


Figure 7.1: Different computed layouts for the same graph data

The correctness of the rendered graph is handled by Cytoscape.js as long as the provided data includes all elements and conforms to the graph model. To ensure the completeness of the elements, the number of rendered nodes and edges was programmatically compared to the number of elements from the parser output.

[1] https://github.com/cytoscape/cytoscape.js-dagre

The style and appearance of the rendered graph can also be adjusted in the settings tab. Figure 7.2 shows the same graph four times, but with different style-settings and different style-classes activated each time. Different styles can reduce visual clutter and allow a user to export and share graphs in the style he needs to be consistent with other visual architectural representations.

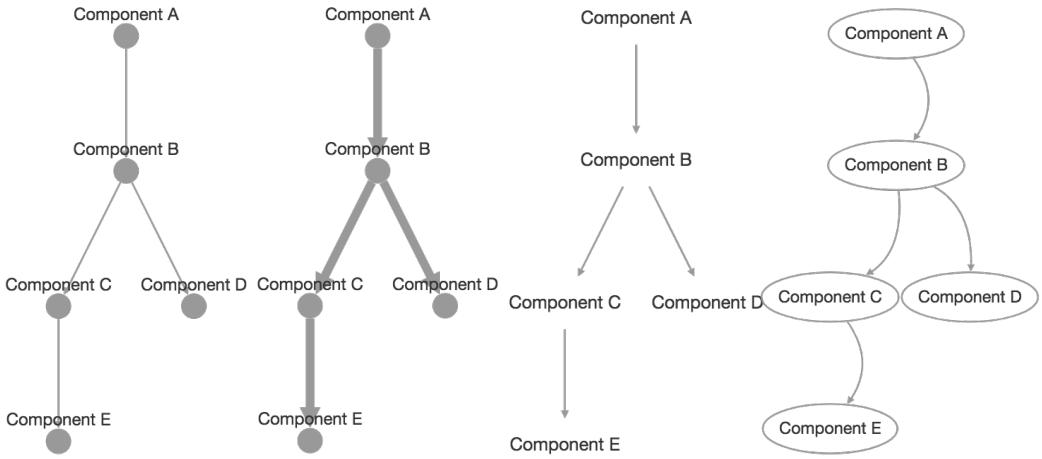


Figure 7.2: Different visual styles of the same graph

Evaluation F3: Navigation and Exploration

All gestures (zooming, panning and dragging of selected elements) required in the requirement description F3 in section 4.2 are supported by the chosen graph drawing library (Cytoscape.js) by default and work as expected. Additional navigation features, such as centering and fitting, have been implemented and integrated into the GUI. The former is used to move the displayed elements and the selected elements to the center of the view. The latter scales the graphs according to the dimensions of the viewport. In the further process of implementation support for undo/redo[2] of simple actions and gestures has been added. Figure 7.3 shows a partial screenshot of the GUI, in which the relevant buttons for centering, fitting, and undo/redo are highlighted.

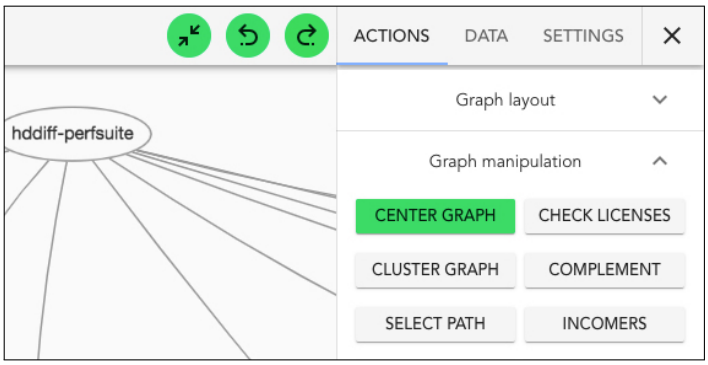


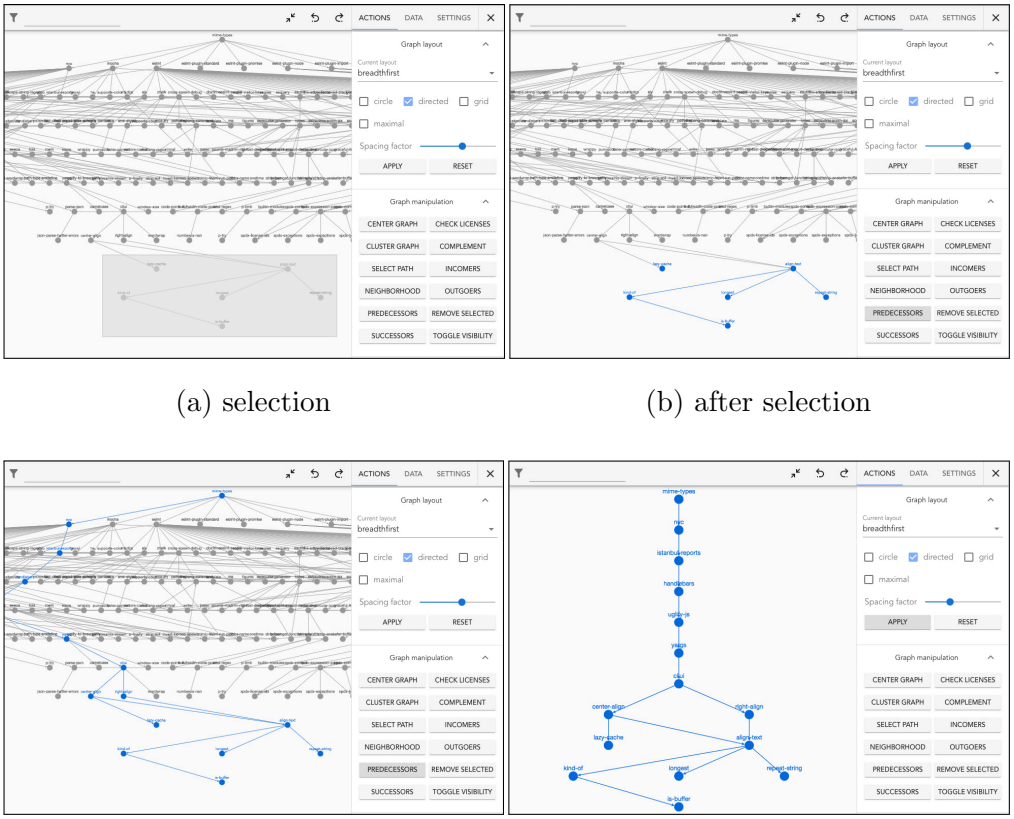
Figure 7.3: A screen-shot of the GUI with the center, fit, and undo/redo buttons highlighted (green)

The undo/redo feature allows the user to explore various ways of displaying the graph elements by allowing the user to reverse performed gestures easily. Currently reverting changes to the graph layout is not yet possible.

[2]https://github.com/iVis-at-Bilkent/cytoscape.js-undo-redo

Evaluation F4: Focus

Allowing users to view and focus on only a part of the dependency graph was one of the major requirements from the beginning. It was fulfilled by giving the user different possibilities to select and view a subgraph. Viewing a subgraph is probably the best way to reduce complexity and highlight the important aspects of relevant dependency structures in large graphs. Figure 7.4 shows the relevant steps, to view a partial dependency hierarchy. First, the software components that are of interest are selected (a and b). Then the dependency hierarchy is added to the selection, by executing the predecessors action from the graph actions area in the sidebar (c). In the last step (d) a new layout is computed and applied to the selected subgraph.



(a) selection (b) after selection

(c) get predecessors (d) apply breadthfirst layout

Figure 7.4: Steps to focus on a partial dependency hierarchy

To focus on a specific component with all direct dependencies as described in section 4.2 requirement F4, a node corresponding to the component of interest has to be selected first. Then all direct dependencies can be added to the selection by, executing the outgoers action from the actions tab. At last a new

layout (e.g. circular layout) should be computed to highlight and separate the selection from the whole dependency graph. Figure 7.5 shows an exemplary graph as a result of executing these steps.

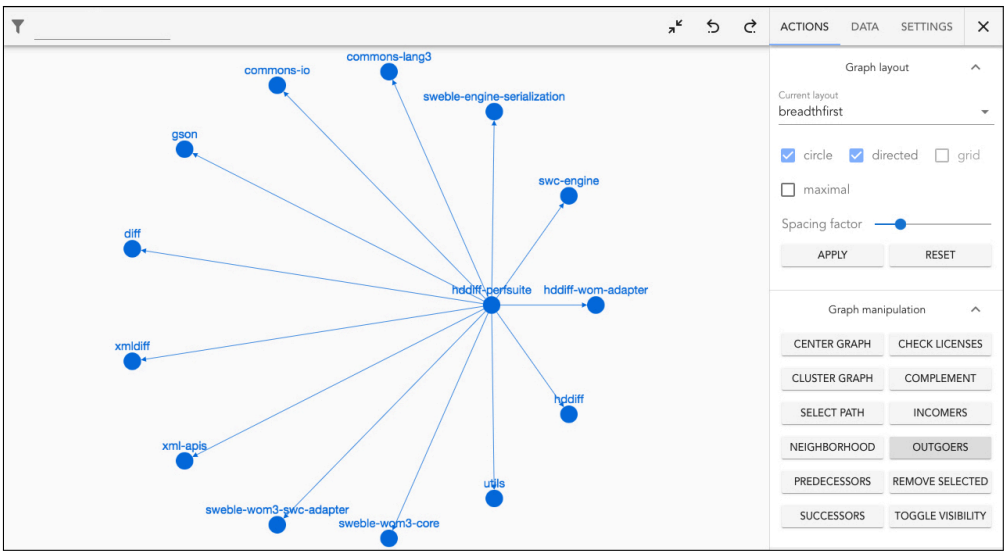


Figure 7.5: Focus of one specific software component with all direct dependencies in a circular layout

Evaluation F5: Data View

The graph representation visualizes software components as nodes and their dependency relationships as directed edges. Both software components and relationships can carry more information, which is usually not shown in the graph representation. For example elementary data like a version number or the namespace of the software component, but also more complex data as defined by the Product Model in subsection 2.2.2. To make this data available to the user, as required and described, a data view has been implemented and integrated as a separate tab in the sidebar. When the data tab is active, it shows the attached information of the graph element that was selected last. In figure 7.6 a software component is selected and the corresponding data is displayed on the right side in the data tab. The data includes information about declared licenses that were found by the crawler and information about the software artifact from which the software component was deduced.

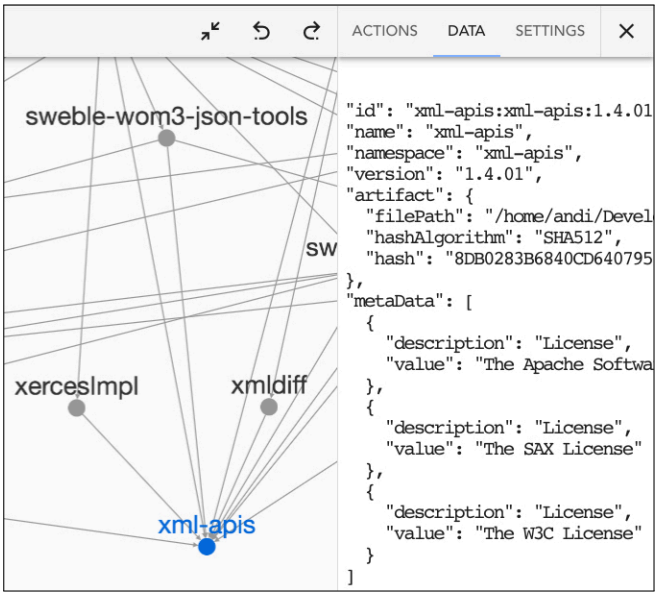


Figure 7.6: A selected node with the attached information displayed in the data tab

The data view is implemented as a text view, that displays the JavaScript object which holds the data of the selected element. A specialised view is not feasible, because the different tools, that can be used as sources for input data do not have a common notation for their metadata. This text view approach is quite simple, but extremely flexible, so it always displays all attached data, regardless of format. This also applies to possible future structural changes to the Product Model, for example, if additional information is added to the metadata object.

Evaluation F6: Filtering

To enable filtering an input field for the search has been added to the GUI. It is positioned isolated from the other navigation elements on the left side in the toolbar, which is located in the upper part of the interface. By entering an input text, the matching elements get selected and highlighted in the graph representation. This includes both the software components and their dependency relationships.

The usage of the filter functionality is illustrated in figure 7.7 with a small graph. But the actual benefit is most noticeable in graphs with a high number of elements. Appendix C contains a full screenshot of the filter functionality used on a large graph.

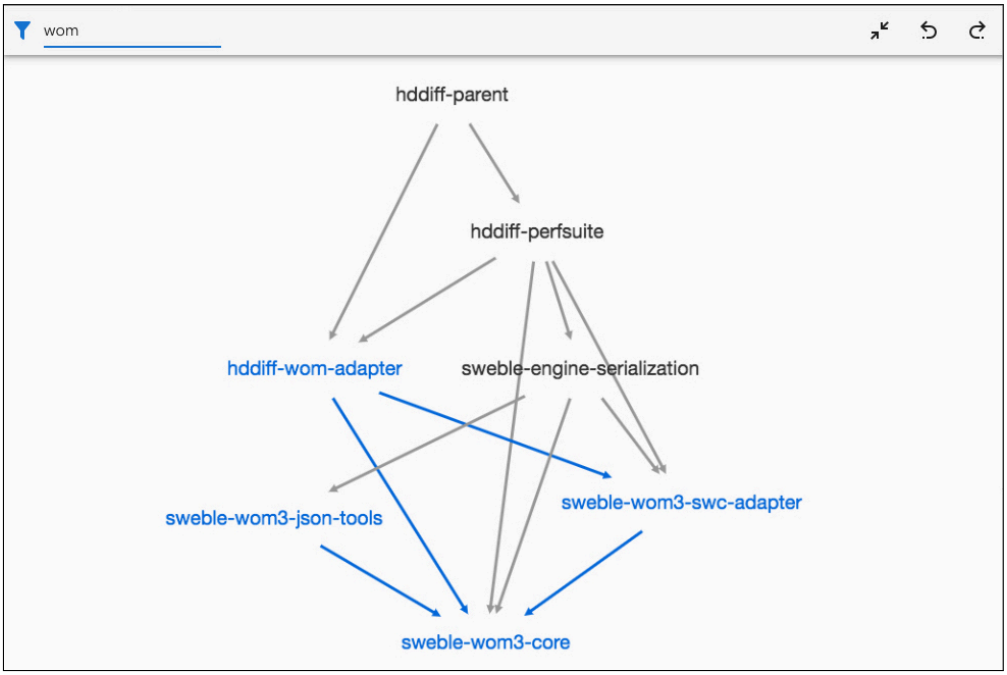


Figure 7.7: Selection of graph elements matching the input text "wom" in the filter input form

Currently, it is possible to filter by name and ID using the input field, which was part of the requirement. In future implementation steps, it should be possible to filter by name and ID as well as the attached metadata. Due to the reimplementation with the Vue.js front-end framework, this step could not be implemented within the given time frame.

Evaluation Q1: Performance

To gain an understanding of the performance of the application, time measurements were taken of the initial loading time of the application and the duration of the parsing step. In addition, the scalability of the graph representation was examined with generated test data of different sizes. All measurements and results were performed and generated on a 13-inch, Mid 2012 MacBook Pro with a 2.5 GHz Intel Core i5 processor. The full hard- and software configuration of the test computer is listed in table 7.1.

Model	13-inch, Mid 2012 MacBook Pro
CPU	2.5 GHz Intel Core i5
GPU	Intel HD Graphics 4000 1536 MB
RAM	16 GB (2x8GB) 1600 MHz DDR3
OS	macOS Sierra (Version 10.12.6)
Browser	Google Chrome (Version 74.0)

Table 7.1: Full hard- and software configuration of the test computer

The analysis of the initial application load time was done with the performance profiler of the Chrome DevTools. Figure 7.8 shows the results obtained using two charts and corresponding legends. The legend on the left refers to both, the upper graphic and the circular chart. The representation of the timeline is supplemented by an additional legend that corresponds to the three vertical markers.

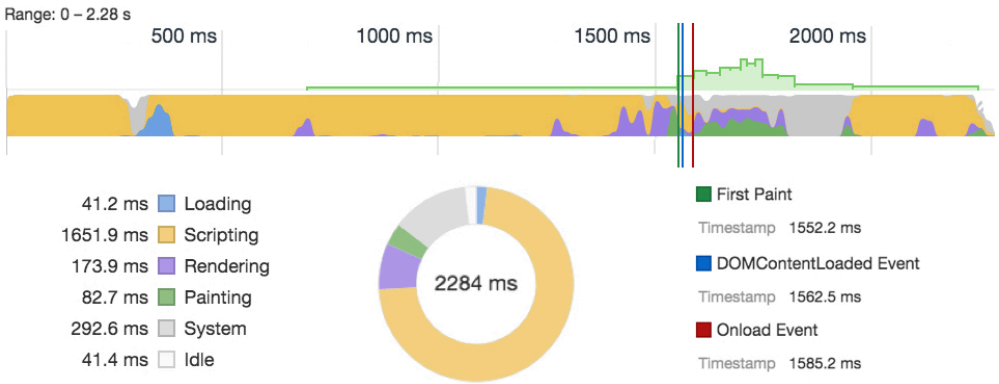


Figure 7.8: Overview of the initial application load time[3]

The initial HTML document, without taking into account the included resources, was fully loaded and parsed at 1562.5 milliseconds, as indicated by the DOMContentLoaded event (blue). The painting of the first frames on screen started at 1552.2 milliseconds as marked by the green vertical line. At

[3]Generated with the Chrome DevTools

timestamp 1585.2 milliseconds the Onload event (red) is triggered, which indicates, that the entire application has finished loading, including all dependent resources such as stylesheets, symbols, and required fonts.

In order to evaluate the parsing step and the time required for it, a total of four test input data sets, with an increasing number of elements (software components and their relationships), were thoroughly tested. In total all data sets were parsed 10000 times and the mean value was calculated from the results. Table 7.2 shows the average parsing time required for each data set, in the first row. Furthermore the following row lists the found nodes and edges with duplicate elements still included. The last row contains the values without duplicates.

Input data	A	B	C	D
Parsing time in milliseconds	0.02527	0.09361	0.57910	9.18197
Parsed Nodes/Edges (with duplicates)	46/42	371/370	667/666	7419/7193
Parsed Nodes/Edges (duplicates removed)	26/36	26/67	293/422	854/1863

Table 7.2: Parsing times needed for different input data sets

The values in the table show that with a higher number of parsed elements there is no significant increase in runtime, as was expected beforehand. A comparison of the duration from the smallest sample (A), with the duration required for the largest data set (D) shows, that an increase of 9,1567 milliseconds in parsing time is present. But this is due to the high difference in size of the data sets, which can be seen by the number of found elements (with duplicates). All measured parsing times are acceptable and small enough to be neglected in the total load time.

In order to evaluate the scalability of the graph representation, the most important layout options for graphs of different sizes were calculated and their computation times were measured. Table 7.3 shows the results obtained.

Number of Nodes/Edges	**26/36**	**26/67**	**293/422**	**854/1863**	**1561/3081**
Breadthfirst	526.53	538.98	584.01	864.13	1670.43
Circle	522.18	540.53	605.48	686.21	1091.24
Concentric	531.35	536.34	612.68	814.98	1235.24
Dagre	536.01	539.08	4443.84	55356.29	127415.81
Grid	529.58	538.71	622.30	838.17	1241.99

Table 7.3: Layout computation times for different graph sizes in milliseconds

Most of the used layout options scale reasonably well, and have acceptable computation times even for graphs with a higher number of elements. The biggest tested graph had 4642 elements (1561 nodes and 3081 edges) and most computation times were in a range between 1091.24 and 1670.43 milliseconds. However, the dagre layout option, which delivered the best visual results, does not scale well at all. Its computation time for the biggest graph was with 127415.81 milliseconds about 76 times longer than the time needed for the breadthfirst layout, which was the second longest time.

8 Summary and Outlook

The extensive reuse of software has become increasingly common, especially through widespread usage of open source components (OSC). But the increasing number of software components from different sources has led to very complex code component architectures of software projects. In the context of this thesis a visualization tool was developed, which is able to represent the complete software component dependency graph of projects, in an interactive and navigable way, with the goal of helping developers to manage and understand these complex relationships between software components of their projects.

The core of the application is the Graph View, which displays the dependency graph as a dynamic node-link diagram. This is done with the graph drawing library Cytoscape.js, which was selected after an exhaustive evaluation of existing libraries. Various layouts and actions to manipulate the graph allow the user to reduce the visual complexity and to concentrate on the areas of the graph that are of interest to him. Other implemented functionalities like filtering help to navigate the graph and to search for specific elements. More detailed information on the individual elements, which is only available as metadata, is displayed in a separate view.

The sources for the input data from which the component graphs are generated can gathered through various tools. To show this, two parsers were implemented. One for an internal tool, the Maven Crawler and another for the analyser of the OSS Review Toolkit (ORT).

Possible improvements and additions that could be made in future implementation steps are among others. The extension of the filter functionality with the option to also filter by attached metadata of elements such as declared licenses; making all layout changes reversible via undo/redo and overall optimizations in the implementation of graph manipulation actions, to decrease rendering costs to a minimum, which could be done by using batch processing. A more major enhancement would be the additional support of a further graph representation, for example a matrix-based representation, like the Dependency Structure Matrix (Sangal, Jordan, Sinha & Jackson, 2005). Keller et al. (2006) argue, that both node-link diagrams and matrix-based representations have their respective advantages and that a multiple graph view strategy could greatly enhance the user experience, by enabling the user to switch representations at fitting times.

All in all, this thesis has shown that software tools that help users to understand/analyze complex relationships between software components can be of great value.

Appendix A Dark Mode

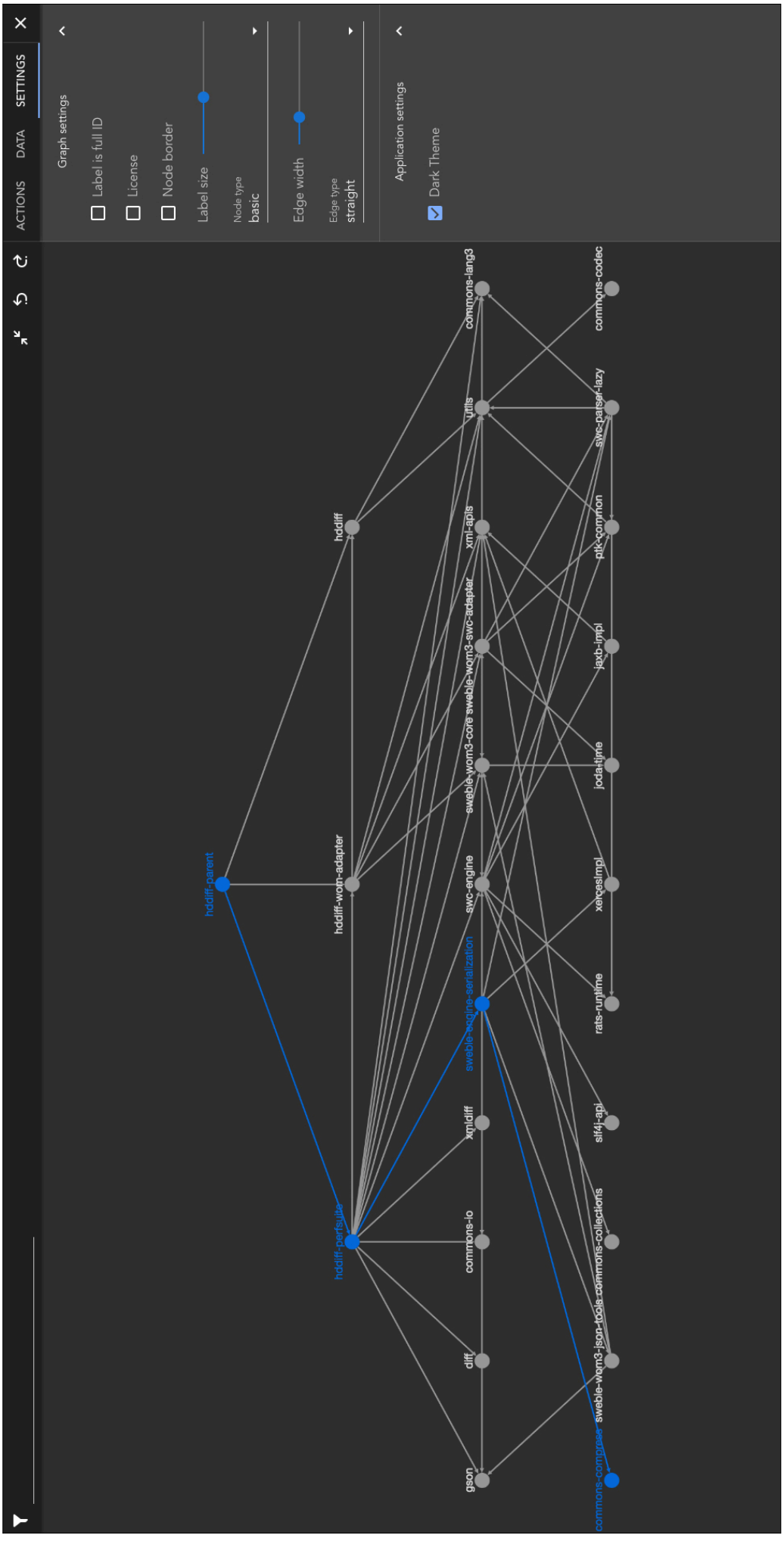
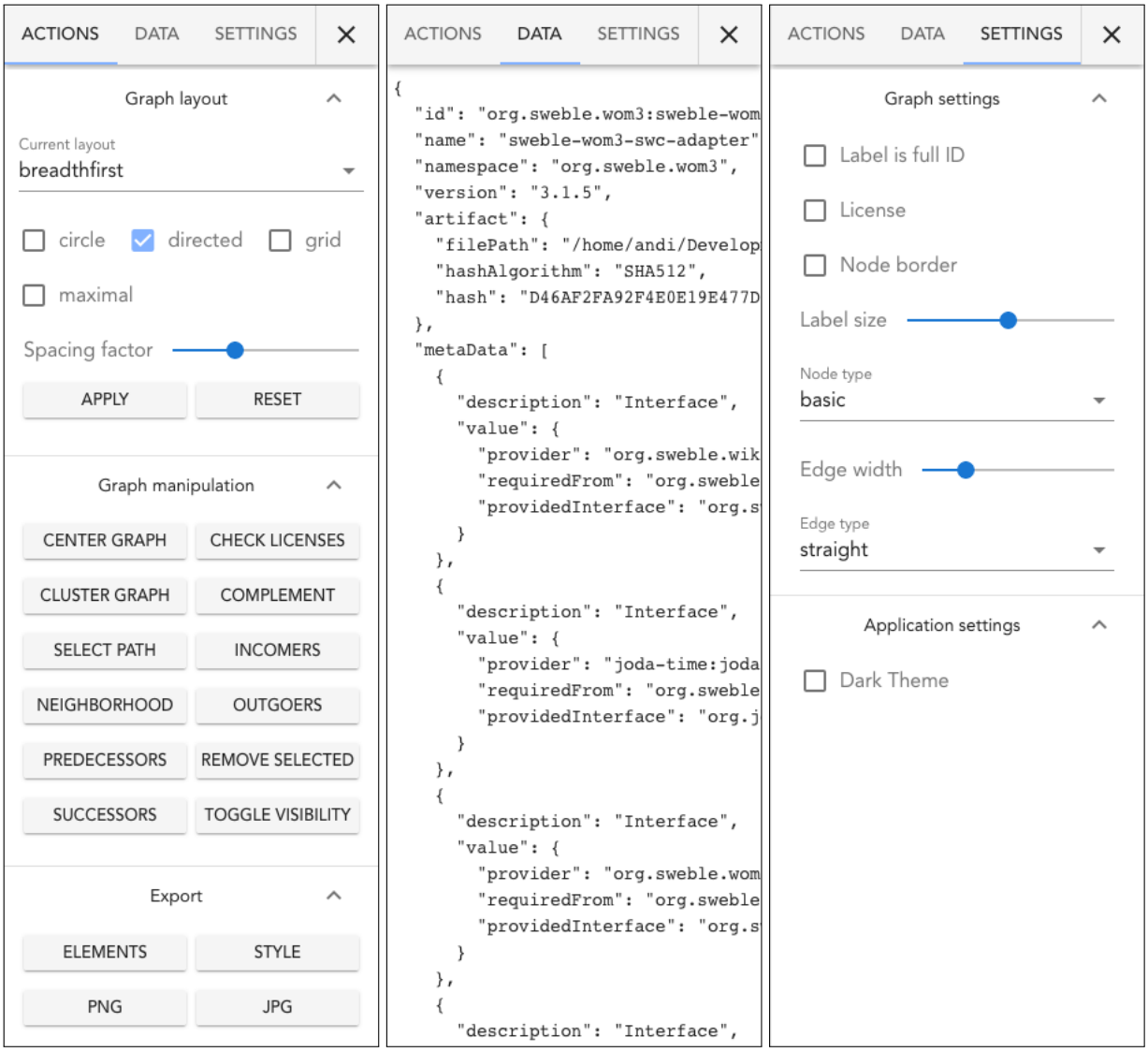


Figure 8.1: The graphical user interface (GUI) in a dark theme

Appendix B Tab Items



(a) Actions Tab (b) Data Tab (c) Settings Tab

Figure 8.2: The there Tab Items contained in the Sidebar

Appendix C Large Graph Filtering

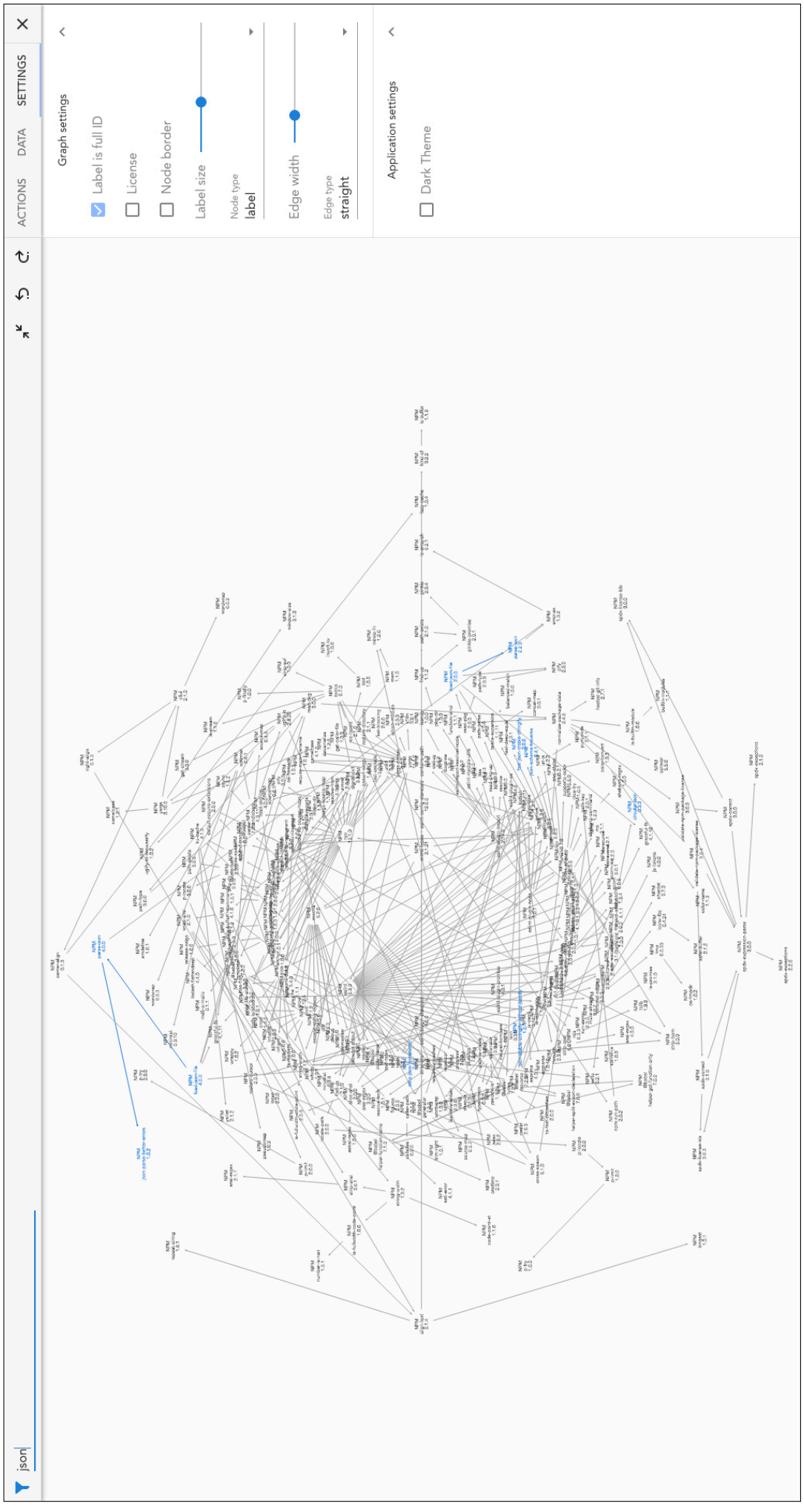


Figure 8.3: The filter functionality shown with a large graph

List of Abbreviations

API Application Programming Interface. 12, 19

DAG directed acyclic graph. 7, 8, 53

DAGs directed acyclic graphs. 5, 6

GPL GNU General Public License. 1

GUI graphical user interface. 20, 21, 27, 34, 36, 47, 53

HTML Hypertext Markup Language. 11–13, 26, 28, 41

JSON JavaScript Object Notation. 8, 9, 12, 55

NPM Node.js package manager. 9

ORT OSS Review Toolkit. 19, 23, 33, 45

OSC open source components. 1, 45

OSS open source software. 1

REST Representational State Transfer. 19

SVG Scalable Vector Graphics. 11

UI user interface. 12–14, 19–21, 53

XML Extensible Markup Language. 8

YAML YAML Ain't Markup Language. 8

List of Figures

List of Listings

List of Tables

References

Bostock, M., Ogievetsky, V. & Heer, J. (2011). D^3 data-driven documents. *IEEE transactions on visualization and computer graphics*, *17*(12), 2301–2309.

Councill, B. & Heineman, G. T. (2001). Component-based software engineering. In G. T. Heineman & W. T. Councill (Eds.), (Chap. Definition of a Software Component and Its Elements, pp. 5–19). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Deshpande, A. & Riehle, D. (2008). The total growth of open source. In *Ifip international conference on open source systems* (pp. 197–209). Springer.

Eugster, P. T., Felber, P. A., Guerraoui, R. & Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, *35*(2), 114–131.

Fruchterman, T. M. & Reingold, E. M. (1991). Graph drawing by force-directed placement. *Software: Practice and experience*, *21*(11), 1129–1164.

Gansner, E. R., Koutsofios, E., North, S. C. & Vo, K.-P. (1993). A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, *19*(3), 214–230.

Ghoniem, M., Fekete, J.-D. & Castagliola, P. (2004). A comparison of the readability of graphs using node-link and matrix-based representations. In *Ieee symposium on information visualization* (pp. 17–24). IEEE.

Haefliger, S., Von Krogh, G. & Spaeth, S. (2008). Code reuse in open source software. *Management science*, *54*(1), 180–193.

Harutyunyan, N., Bauer, A. & Riehle, D. (2018). Understanding industry requirements for floss governance tools. In I. Stamelos, J. M. Gonzalez-Barahoña, I. Varlamis & D. Anagnostopoulos (Eds.), *Open source systems: Enterprise software and solutions* (pp. 151–167). Cham: Springer International Publishing.

Keller, R., Eckert, C. M. & Clarkson, P. J. (2006). Matrices or node-link diagrams: Which visual representation is better for visualising connectivity models? *Information Visualization*, *5*(1), 62–76.

Laurent, A. M. S. (2004). *Understanding open source and free software licensing: Guide to navigating licensing issues in existing & new software.* " O'Reilly Media, Inc."

Lopes, C. T., Bader, G. D., Huck, G., Franz, M., Sumer, O. & Dong, Y. (2015). Cytoscape.js: a graph theory library for visualisation and analysis. *Bioinformatics*, *32*(2), 309–311.

Morgan, L. & Finnegan, P. (2007). Benefits and drawbacks of open source software: An exploratory study of secondary software firms. In *Ifip international conference on open source systems* (pp. 307–312). Springer.

Riehle, D. (2007). The economic motivation of open source software: Stakeholder perspectives. *Computer*, *40*(4), 25–32.

Sangal, N., Jordan, E., Sinha, V. & Jackson, D. (2005). Using dependency models to manage complex software architecture. In *Acm sigplan notices* (Vol. 40, *10*, pp. 167–176). ACM.

Satyanarayan, A., Moritz, D., Wongsuphasawat, K. & Heer, J. (2017). Vega-lite: A grammar of interactive graphics. *IEEE transactions on visualization and computer graphics*, *23*(1), 341–350.

Scheffer, D. (2018). *An artifact crawler for determining code component architectures* (Master's thesis, Friedrich-Alexander University Erlangen-Nürnberg, Germany).

Sugiyama, K., Tagawa, S. & Toda, M. (1981). Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, *11*(2), 109–125.

Szyperski, C., Bosch, J. & Weck, W. (1999). Component-oriented programming. In *Moreira a. (eds) object-oriented technology ecoop'99 workshop reader. ecoop 1999. lecture notes in computer science* (Vol. 1743, pp. 184–192). Springer, Berlin, Heidelberg.