TU WIEN Informatics

Comprehensive Visualizations for the Historical Analysis of Issue Tracking Systems in Software Engineering Education

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Markus Lupinek, BSc
Matrikelnummer 11776820

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Thomas Grechenig

Wien, 10. Jänner 2025

Unterschrift Verfasser	Unterschrift Betreuung

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Comprehensive Visualizations for the Historical Analysis of Issue Tracking Systems in Software Engineering Education

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Markus Lupinek, BSc
Registration Number 11776820

to the Faculty of Informatics

at the TU Wien

Advisor: Thomas Grechenig

Vienna, 10th January, 2025

Signature Author

Signature Advisor

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Comprehensive Visualizations for the Historical Analysis of Issue Tracking Systems in Software Engineering Education

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Markus Lupinek, BSc
Matrikelnummer 11776820

ausgeführt am
Institut für Information Systems Engineering
Forschungsbereich Business Informatics
Forschungsgruppe Industrielle Software
der Fakultät für Informatik der Technischen Universität Wien

Betreuung: Thomas Grechenig

Wien, 10. Jänner 2025

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Markus Lupinek, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 10. Jänner 2025

Markus Lupinek

Danksagung

Ich möchte meinem Betreuer meinen herzlichen Dank aussprechen. Ohne seine Unterstützung wäre der erfolgreiche Abschluss dieser Arbeit nicht möglich gewesen. Außerdem danke ich den Teilnehmern der Interviewrunden für ihre Zeit und ihre wertvollen Anmerkungen. Ebenso danke ich meiner Familie und meinen Freunden für ihre ständige Unterstützung während meines Studiums.

Acknowledgements

I would like to express my sincere gratitude to my supervisor. Without his support, the successful completion of this work would not have been possible. I also want to thank the participants of the interviews for their time and valuable feedback. Additionally, I am grateful to my family and friends for their constant support throughout my studies.

Kurzfassung

Die Softwareentwicklung und ihre Methoden haben sich in den letzten Jahren weiterentwickelt. So hat sich in der Vergangenheit bemerkbar gemacht, dass Systeme zur Erfassung von Tickets für Funktionsanfragen und Softwarefehler sowie Plattformen zur Verteilung des Quellcodes an Beliebtheit gewonnen haben. Es ist daher nicht verwunderlich, das vermehrt Softwareengineering-Lehrveranstaltungen auf solche Systeme zurückgreifen und diese für unterschiedliche Zwecke nutzen. Während der Entwicklung eines Softwareprojekts kann sich in diesen Systemen, auch als Software-Repositories bekannt, eine Vielzahl an Daten ansammeln. Diese Daten stellen eine wahre Goldmine an Wissen dar, welche eine solide Entscheidungsgrundlage bieten kann. Allerdings erschwert die Tatsache, dass diese Information nur in textuelle Form zu Verfügung stehen, den Analyseprozess erheblich. Visualisierungen haben sich bei der Interpretation solcher Daten als hilfreich erwiesen, jedoch besteht eine Nische, wenn es um die Visualisierungen von historischen Ticketdaten im Kontext der Softwareengineering-Lehre geht.

In dieser Diplomarbeit wird diesbezüglich der bestehende Informationsbedarf untersucht und ein Ansatz vorgestellt, der das Visualisieren von Software-Repositories ermöglicht. Um dies zu erreichen, wurde zunächst eine Literaturrecherche durchgeführt, um den Informationsbedarf der Entwickler festzustellen. Diese dienten als Grundlage für die Gestaltung mehrerer Konzepte, die anschließend mithilfe von semi-strukturierten Experteninterviews validiert und priorisiert wurden. Anhand der Ergebnisse wurde in den darauffolgenden Phasen ein Prototyp in mehreren Iterationen entwickelt und mithilfe von szenariobasierten Interviews evaluiert.

Die befragten Experten bestätigten die Nützlichkeit des Prototyps im Kontext der Lehre. Die Visualisierungen der Metriken, wie die Verteilung von Tickets innerhalb eines Projekts, die Statusänderungen der Tickets und die Beteiligungen an Tickets veranschaulichen den evolutionären Verlauf eines Projekts. Dadurch erhalten sowohl Studierende als auch Lehrkräfte wertvolle Einblicke in die Entwicklung des Projekts bis hin zum finalen Stand. Allerdings müssen noch Design- und Usability-Probleme behoben werden, damit ein solches System für Projekte in der Softwareengineering-Lehre von Nutzen sein kann.

Keywords: *Softwarevisualisierung, Softwareevolutionsvisualisierung, Problemverfolgungssystem, Versionskontrollsystem, Softwareengineering-Lehre*

Abstract

Software engineering and its methods have evolved in recent years. In the past, it has become increasingly popular to use systems to create tickets for feature requests and software defects, as well as platforms for distributing source code. It is therefore not surprising that more and more software engineering courses are using such systems for various purposes. During the development of a software project, a large amount of data can accumulate in these systems, also known as software repositories. This data represents a veritable gold mine of knowledge that can provide a solid basis for decision-making. However, the fact that this information is only available in textual form makes the analysis process considerably more difficult. Visualizations have proven to be helpful in interpreting such data, but there is a niche when it comes to visualizing historical ticket data in the context of software engineering education.

This thesis investigates the existing information needs and presents an approach for the visualization of software repositories. To achieve this, a literature review was conducted to determine the information needs of developers. This served as a basis for the design of several concepts, which were then validated and prioritized with the help of semi-structured expert interviews. In the subsequent phases, based on the results, a prototype was developed in several iterations and evaluated with the help of scenario-based interviews.

The interviewed experts confirmed the prototype's usefulness in an educational context. The visualizations of the metrics, such as the ticket distribution within a project, ticket status changes and the participation in tickets, effectively illustrate the evolution of a project. This provides both students and teaching staff valuable insights into the development of the project up to its final stage. However, design and usability issues must be resolved to ensure the system's applicability for projects in software engineering education.

Keywords: *Software Visualization, Software Evolution Visualization, Issue Tracking System, Version Control System, Software Engineering Education*

Contents

CHAPTER 1

Introduction

The following chapter provides an introduction to the topic of this master thesis. At the beginning, the problem is described, which is followed by the expected results of this thesis. Subsequently, the structure of this thesis is detailed.

1.1 Problem Description

Agile Project Management (APM) is becoming increasingly popular in software development as with this sort of methods, projects are developed over multiple iterations and can be adapted to the rapidly changing needs of the software world [26]. In order to make students aware of the software project development process, the Project Based Learning (PBL) approach has proven to be helpful, as it emphasizes applying theoretical foundations of software engineering in practice [16, 32, 66].

Projects in software engineering courses are often used with the intention of exposing students to the challenges of industrial software development. Although software engineering education projects aim to simulate these conditions, projects in an educational context differ from professional software projects. On the one hand, student projects are usually small to medium-sized, limited in scope, and limited in length. While professional software development projects often involve large teams, educational projects are usually limited to a manageable number of students. The focus is on meeting academic requirements, applying theoretical concepts in practice, and learning new technologies [28, 49, 72].

At the Vienna University of Technology, in software engineering courses such as Software Engineering Project (SE PR) and Advanced Software Engineering (ASE), groups of up to six students are formed after they have passed the individual phase. Over the course of the semester, these student groups work on a software project consisting of multiple parts, which they must successfully complete over several iterations using APM

methods. To support team collaboration, Issue Tracking Systems (ITSs) are utilized to track functional requirements and software defects. On the other hand, Version Control Systems (VCSs) are used to distribute the source code within the team. Over the lifespan of a project, the repositories of these systems can accumulate a large amount of data [11, 23, 57]. This gold mine of knowledge can be valuable for both students and teaching staff, either as a solid foundation for decision-making during the project's development or to produce more objective estimations during its evaluation. However, the amount and complexity of this data, makes the historical analysis challenging, as the data and the relationships between entities (e.g. software issues and their corresponding commits) is typically available in textual form. Although the data in the repositories reflects the historical progression of a project, exploring its evolution remains difficult due to a lack of adequate aids.

In order to facilitate the derivation of information from software repositories, previous research in the field of software visualization has shown that illustrations of software data are an efficient way to provide this information. In the past, attention has been primarily paid to the visualization of data which can be extracted from the software repositories of VCSs [19, 21, 30, 36]. However, in the context of software engineering education, data from ITSs received less attention. Furthermore, scientific visualizations often rely on simple representations that are sufficient for overviews, but when detailed historical information needs to be explored, these are no longer suitable. This master thesis aims to close this gap by developing comprehensive visualizations that can be used for the historical analysis of a software project. The developed visualizations are designed to visualize various aspects, including the distribution of software issues, work distribution based on contributions to software issues, and the lifecycle of software issues, reflected by their status changes—all data which can be extracted from ITSs.

1.2 Expected Results

In general, this thesis aims to determine how a visualization tool can facilitate the analysis of a software project in an educational context. To explore this, visualizations are created to display the project's history based on the data accumulated in ITSs. The following hypothesized benefits are expected:

- Identification of Code Hotspots: Determine which areas of the codebase undergo frequent changes and are prone to defects.
- Identification of Bottlenecks: Analyze which issue statuses have the longest ticket dwell times.
- Distribution of Work: Assess which team members have worked on specific software issues to understand workload distribution and contributions.
- Trend Recognition: Detect whether a process or workflow has a measurable impact on the project's progress and outcomes.

The proposed key features of this prototype are:

- A historical view providing information on which software issues have caused changes in which source code areas (e.g., packages, files, etc.) and how these changes have been distributed across the entire project.
- A historical view providing information about the lifecycle of software issues by showing how long each issue ticket remained in each issue status.
- A historical view providing information about contributions to software issues by showing which individuals participated to which tickets by making changes to them.

1.3 Structure

The structure of this thesis is divided as follows: Chapter 1 provides an introductory overview of the problem and the expected results. In Chapter 2, the foundations for this thesis and the related work are detailed. Chapter 3 defines the research questions and the methodology to answer these research questions. Chapter 4 describes the design of the visualizations and the results of the semi-structured interviews. The implementation of the visualizations can be found in Chapter 5, and the subsequent evaluation of the illustrations is found in Chapter 6. After discussing the results in Chapter 7, this thesis and future work are concluded in Chapter 8.

CHAPTER 2

Theoretical Background

This chapter provides the theoretical knowledge relevant to this thesis. Section 2.1 covers the fundamental concepts, while Section 2.2 provides an overview of the current state of research.

2.1 Fundamentals

This section provides fundamental information about the terminology and concepts that are relevant to this thesis. First, Section 2.1.1 describes the basics of Project Management (PM) and how APM differs from Traditional Project Management (TPM). Section 2.1.2 and Section 2.1.3 detail functionalities as well as basic metrics of VCSs and ITSs. Finally, essential information about the data visualization is given in Section 2.1.4.

2.1.1 Project Management

According to Project Management Institute (PMI) [58], a project is a temporary group activity designed to produce a unique product, service or outcome. It has a beginning and an end, described by the delivery of the expected result [60]. A project is characterized by a prescribed time for completion and a limited budget. It further defines goals and a number of activities to achieve these goals [63]. As defined by PMI [58], PM is the accumulation of knowledge, expertise and methodology used to bring a project to its objectives safely, efficiently and successfully and to meet the requirements and expectations of the project's stakeholders. Kerzner [41] describes PM as the planning and controlling of resources over a short period of time in order to achieve specific goals. Both TPM and APM methods have been derived from the best practices of the engineering and management disciplines [25].

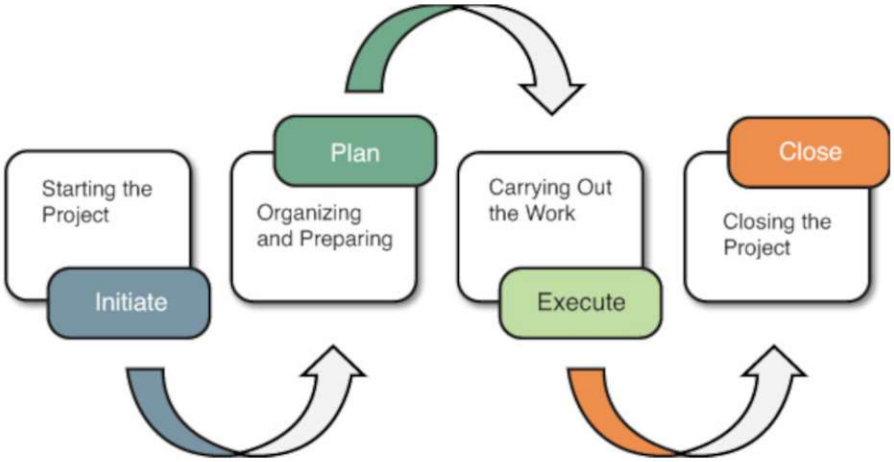


Figure 2.1: Life cycle of TPM [60]

Traditional Project Management

Usually, TPM is a linear process where, in the most cases, the phases are executed consecutively [26, 60]. As seen in Figure 2.1 it consists of multiple phases. In the initiation phase, the project scope and constraints on resources in terms of time, manpower, and budget are described. In the planning phase, the project documents needed for the implementation as well as the control of the project are defined. During the execution phase, the project activities are carried out, building the path to the project's success. Monitoring and controlling compares the planned results with the real results to identify areas where adjustments are necessary. Lastly, in the closing phase, it is determined whether all planned work has been completed and accepted by all participants [41, 60, 63].

In the traditional approach, it is expected that each phase will be executed only once and that already completed phases will not be revisited [8]. The well-structured process and the importance of requirements are among the advantages of this approach. However, in software engineering, TPM quickly reaches its limits, as a project can lose its sequential flow. In addition, it can be difficult for customers to define all requirements at the beginning of the project, leading to potential problems [8, 63]. It assumes flawless knowledge of the goals and their solutions. The lack of flexibility and the impossibility to adjust schedule, resources and scope are disadvantages in today's complex and fast-paced project environment [8, 26].

Agile Project Management

APM is an iterative and incremental process where stakeholders and developers work closely together to understand the scope, identify requirements, and prioritize features

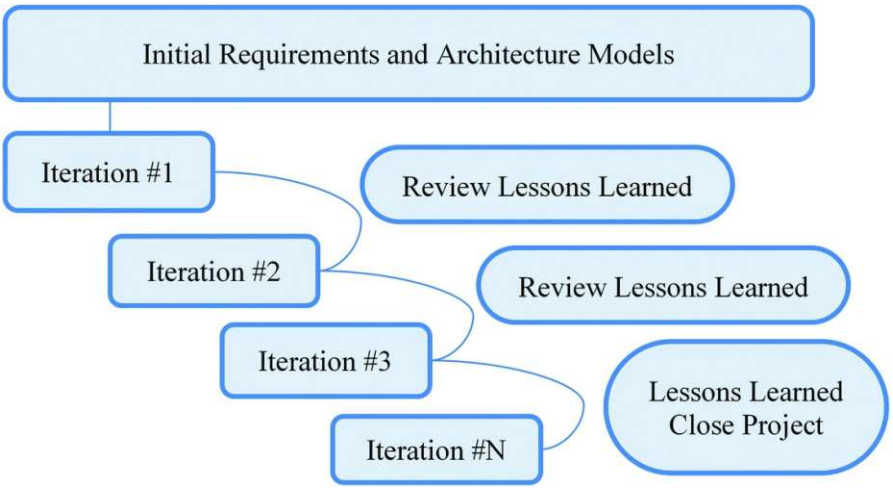


Figure 2.2: Life cycle of APM [8]

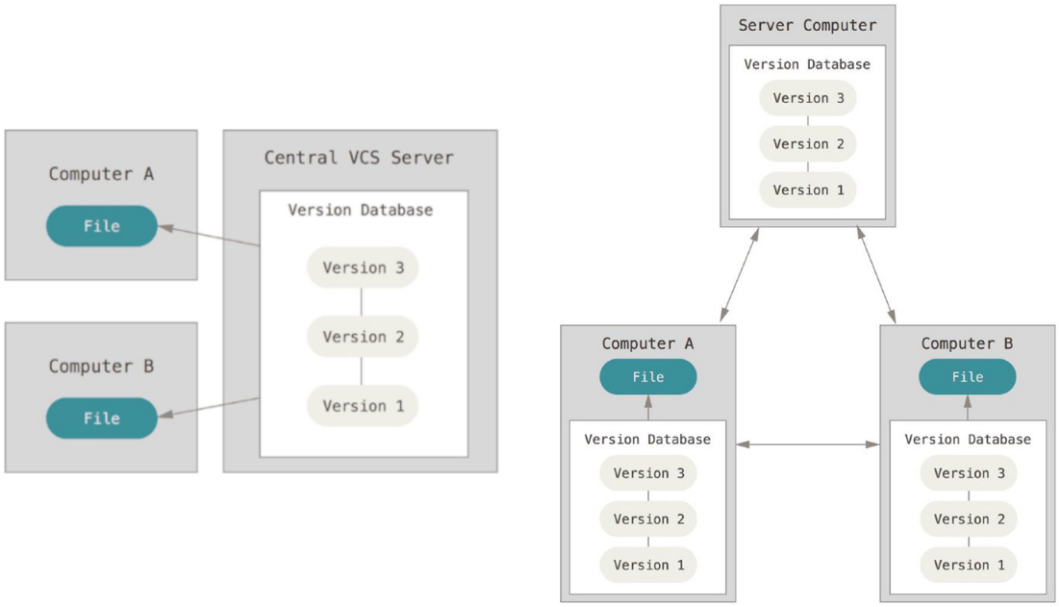
[8]. APM practices are flexible, light and collaborative and have been strongly affected by the concepts of agile software development [63]. It is based on the four core principles found in the Agile Manifesto, which are paraphrased as follows [2, 26, 46]:

1. *"Individuals and interactions over processes and tools"*
2. *"Working software over comprehensive documentation"*
3. *"Customer collaboration over contract negotiation"*
4. *"Responding to change over following a plan"*

APM consists of a number of short iterative planning and development cycles, called iteration, as seen in Figure 2.2. An iteration is a short period of time during which an agile team works on a predefined number of tasks. These are used to divide a long project into smaller pieces [8]. At project start, planning and requirement definition are done. Following the feature-driven management approach, the project scope and the specifications are adapted to a prioritization of project features. In subsequent iterations, planned, implemented and tested functionalities are delivered to the customer [63]. In contrast to TPM, APM tries to adapt to uncertainty, allowing changes to be incorporated even in the late stages of project [2]. Due to its iterative nature, APM allows continuous reviews and adjustments to any adaptation requests from stakeholders [8].

2.1.2 Version Control System

A VCS, also often called Revision Control System (RCS), is a widely used system for the development of software products. It saves changes made to the source code, makes historical versions of the software available to the user [11, 42, 62, 71, 79] and stores



(a) CVCS (b) DVCS

Figure 2.3: Types of VCSs [15]

versioned data for each file [62]. A VCS also allows changes to be made in the form of deleting or adding code without affecting the current, functioning version. If there are breaking changes, they can be reverted [11, 79]. Since VCS manages all versions of the code, users are encouraged to work together on one project [79].

Types

According to Chacon and Straub [15], in addition to Local Version Control Systems (LVCSs), there are two different types of VCSs, namely Centralized Version Control Systems (CVCSs) and Distributed Version Control Systems (DVCSs).

Centralized Version Control System A CVCS (Figure 2.3a) is a type of VCS where the files are hosted centrally on a single server [15, 22, 71, 79] and where users check out the latest version of the files. With CVCSs, team members can get an approximate overview of what other people in the team are doing. A CVCS also requires less administrative effort, as it does not manage local repositories of individuals. However, it has some disadvantages as in the event of a system failure, nobody can make changes to the central version, or if the server's memory fails without backup, the entire history can be lost [15].

Distributed Version Control System A DVCS is a type of VCS where users have a complete copy of the repository in addition to the server, as seen in Figure 2.3b [15, 22, 71, 79]. Contributors make changes to the local repository and share them with the

server if they consider it necessary [79] and since the changes are made locally, it is faster than CVCSs [71]. In the event of a complete system failure, a local repository can be used to restore the old state of the server [15, 79]. Furthermore, contributors do not have to be core members of a repository to perform basic tasks such as merging branches or reverting states [15, 59, 79].

A well known DVCS is Git, which available on all development systems due to its free software license [67, 71]. Compared to its predecessors, software revisions in Git have become first-class citizens, as every user has a copy of the repository and manage these revisions [67]. Unlike other VCSs, Git works with a stream of snapshots, where each time a change is made and saved, a snapshot of the entire project is created. Rather than saving files that have not changed, Git creates links to the original files [15].

Principles

This section introduces the fundamental concepts of VCSs which form the backbone of collaborative software development workflows.

Software Repository A software repository is a central location where information about various activities is recorded, including both VCSs and ITSs. Due to the large amount of data stored in a software repository, it can be efficiently used for data analysis [11, 23, 57].

Commit A commit is the process of saving changes, deletions and additions made by a user to source files in a software repository [3]. In Git, a snapshot of the entire project is taken, with the latest commit representing the current version of the project [33, 71]. In general, files can be in the following states [71]:

- Modified: Changes have been made to a file.
- Staged: Changes made are prepared for the commit.
- Committed A file is located in a snapshot of the project.

Each commit is linked to its predecessor and successor, and together all commits represent the history of a software project [33].

Branch A branch is a separate development line of program code that evolves independently of the stable version of the project [33, 56, 70]. A single repository can consist of several branches [33] which allows users to split tasks among themselves without interfering with other tightly coupled tasks [70, 79]. Modifications to one branch have no impact on other branches [56]. Branching also makes software easier to maintain by reducing the risk of unwanted changes [70]. The most recent commit is referred to as the head of the branch (HEAD), and it moves forward as new commits are added to the branch [33].

Merge A merge is the process of combining changes of two branches into one branch [33, 56], a so-called three-way merge. This involves three versions of the project: two independently developed branches and their common ancestor. The changes from both branches are applied to create a merged version of the software [14, 78]. If changes were made to the same parts of the code, a conflict arises, which must be manually resolved by the developer [14, 33, 78]. Merging can be divided into three groups [14, 78]:

1. Unstructured merge: This approach recognizes differences based on plain text. A conflict occurs when changes have been made at the same location in the code, which is then passed on to the user for manual resolution [14, 78].

2. Structured merge: To increase merge accuracy, the structured merge attempts to resolve conflicts automatically by analyzing the structure of the artifacts. Approaches may use context-free and context-sensitive syntax [14] or represent artifacts as graphs or abstract syntax trees [14, 78].

3. Semi-structured merge: This approach is a combination of unstructured and structural merge. Software artifacts are represented as trees, providing information about how nodes and subtrees can be merged. With abstraction, the approach can discover ordered elements. If the structured merge is not able to resolve a conflict automatically, the unstructured merge is used to resolve the conflict [14].

Merge Request A Merge Request (MR) in GitLab[1], also known as Pull Request in GitHub[2] or BitBucket[3] [53], is a process to integrate changes of the source branch into the target branch [4, 27, 34]. This typically involves merging a feature branch into the master branch, or merging a master branch into a release branch [53]. MRs are often used to review changes that have been made to the code. They consist of a request description, changes to the code, possible reviews to the changes, status about pipelines to see if the tests passed and if the software can build, a comment section to discuss the changes and a list of commits with a comparison view to see the differences between the source and the target branch [27, 34].

2.1.3 Issue Tracking System

A software issue ticket is a report in natural language often used for various changes made in a software project. These changes can include feature requests, bug reports, common development tasks or any work related to a project [45, 47, 48, 73]. The information in an issue ticket is stored in multiple fields, such as title, description, comments, etc. To each software issue, deadlines and developers responsible can be added. Due to the rich amount of information stored in issue tickets throughout the development of a software

[1] `https://about.gitlab.com/`, Accessed: *23.08.2024*
[2] `https://github.com/`, Accessed: *23.08.2024*
[3] `https://bitbucket.org/`, Accessed: *23.08.2024*

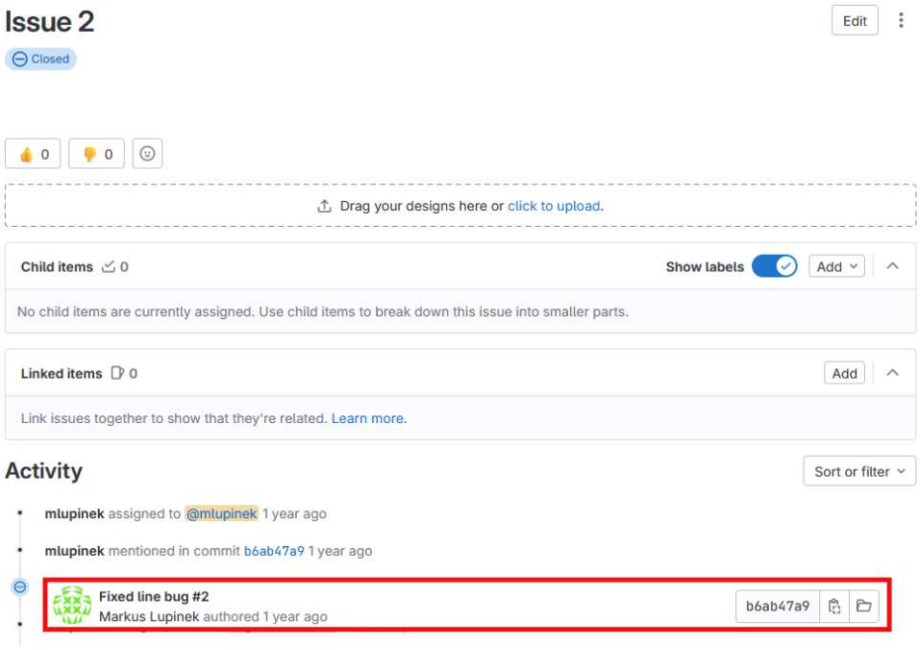


Figure 2.4: Example of a referenced issue in a commit message in GitLab[1]

project, they serve as a good source of documentation [45, 47]. Each ticket is normally assigned a status, describing its current implementation phase [74].

An ITS supports the software development process by tracking the large number of software issues that arise during development and maintenance. This is achieved by providing a feature-rich environment for reporting and managing issues [10]. Proper use of an ITS can bring many benefits, such as increased software quality and user satisfaction. Furthermore, ITSs improve communication and productivity within the team and thus reduce the costs of a project. From an agile process perspective, ITSs support agile development practices. Usually, boards are used to visualize the status of the issues and thus the status of the project, which adds further advantages [74]. Well-known ITSs include Jira[4], Bugzilla[5], GitHub[2] and GitLab[1]. These tools are used by large open source projects such as Apache[6] or Spring[7], with GitHub[2] and GitLab[1] also providing a VCS in addition to the issue management system [10, 73].

Software issues and commits are different artifacts created in separate systems. To establish a link between an issue ticket and the associated commits, the issue identifier is entered into the commit message [52, 61, 68]. An example of this in GitLab[1] can be seen in Figure 2.4. References between issues and commits are necessary for various software engineering tasks such as bug and feature localization and the analysis of commits [61].

[4] https://www.atlassian.com/de/software/jira, Accessed: *23.08.2024*
[5] https://www.bugzilla.org/, Accessed: *23.08.2024*
[6] https://www.apache.org/, Accessed: *23.08.2024*
[7] https://spring.io/, Accessed: *23.08.2024*

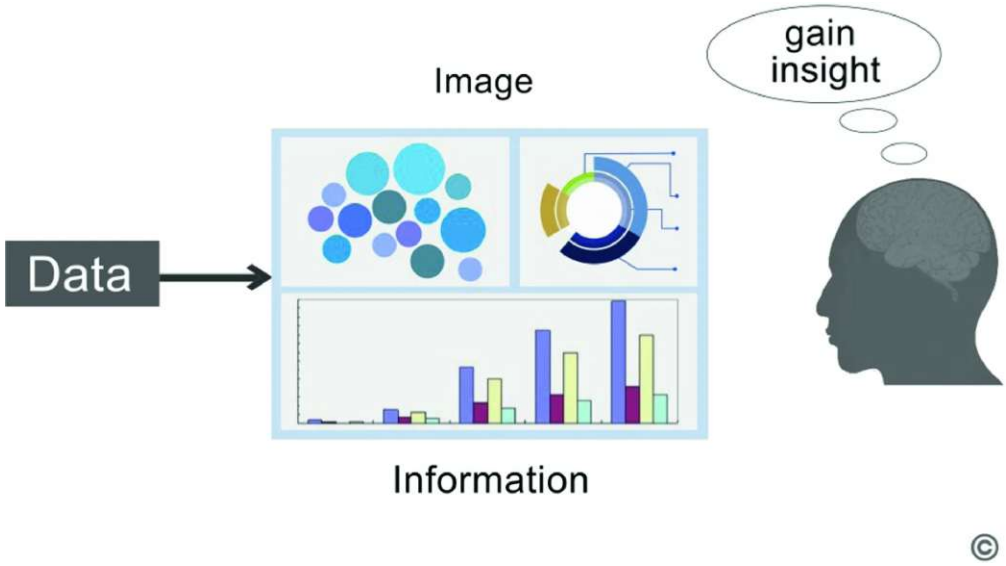


Figure 2.5: Information process of data visualization [44]

They are also often used to evaluate effort and to understand software development [68]. However, a link may not be available for various reasons. In such cases, it can be useful to recover them with the help of suitable tools and thus improve the link quality [61, 68].

2.1.4 Data Visualization

Data, in various forms, is usually raw, unprocessed information that has no meaning [45]. There are two types of data: primary and secondary data. Primary data is data that is collected directly for a specific study purpose, while secondary data is processed and analyzed data, used for a different purpose [1]. Information, on the other hand, is data that has been processed in such a way that it can be understood by the viewer [44].

Data visualization involves using computers to transform data into a visual form. For example, Kard et al. [40] defines the visualization of data as *"The use of computer-supported, interactive, visual representations of data to amplify cognition"*. It is a technique that allows to describe a large amount of data in a comparable and simple way [51] and makes it possible to represent large amounts of data very quickly. Additionally, it helps users quickly recognize patterns by gaining a basic understanding of either small or large datasets. The cognitive system enables the viewer of a visualization to interpret the data and derive useful information. This process is shown in Figure 2.5 [44].

Visualization applications provide users with intuitive ways to analyze and explore data, recognize patterns, explore correlations, and support various activities [9]. According to Bikakis [9] they should deal with the following aspects:

- Real-Time Interaction: A system that responds in an acceptable time, enabling interactions with countless data points from a dataset through efficient and scalable techniques.
- On-the-Fly Visualization: Support for on-the-fly visualization of data is required, since in many cases pre-processing is not possible.
- Visual Scalability: Problems such as information overloading should be circumvented using data abstraction mechanisms.
- User Assistance and Personalization: The capability to adapt to different user exploration scenarios and assist users in understanding the data.

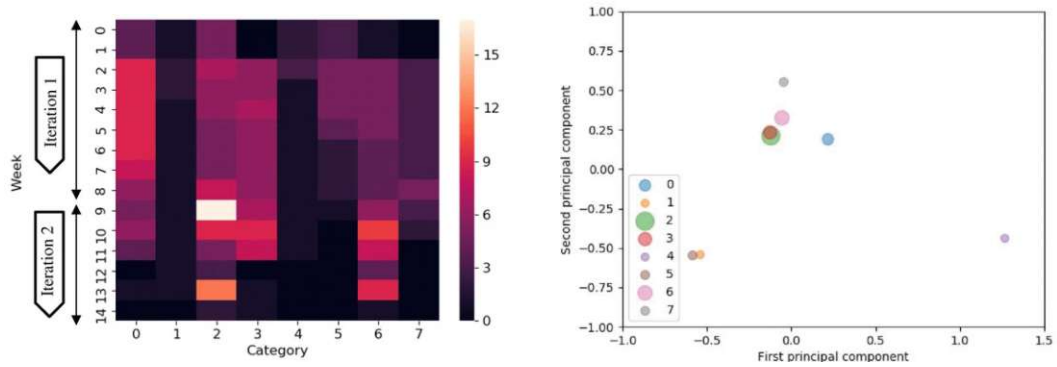
Data visualization can be classified into two categories: information visualization and scientific visualization. Both fields aim to make raw data more understandable by transforming it into a visual form [44, 50].

- Information visualization: Specializes in illustrating abstract data (e.g., business data). Common methods include tables, diagrams, trees, maps or scatter plots [44, 51].
- Scientific visualization: Focuses on representing physical data (e.g., data about the human body or environment). Simulations, wave forms or volumes are commonly used to represent the data [44, 51].

To create meaningful data visualizations, for either scientific or informatic purposes, certain criteria must be met to answer questions and provide insights. First, the visualization must be readable in such a way that the data can be easily understood and interpreted. Additionally, users should be able to distinguish past knowledge from the visualized data. Lastly, the data should be presented in a way that it supports the user in exploring the data [44].

Software Visualization

Software visualization is a scientific field that focuses on the illustration of data that can be derived from software artifacts (e.g., the source code of a program) [43], giving shape to the otherwise invisible and intangible software [35]. Especially in large and complex systems, knowledge about the code can be lost due to the software's life span and maintenance activities. Original developers may have taken on new tasks, and related documentation might not be updated. Small changes can lead to errors, which



(a) Ticket lifetime visualization (b) Ticket feature visualization

Figure 2.6: Software issue visualization tool by [38]

further leads to low productivity. Software visualization can help to increase developers' productivity and aid understanding complexity. It provides new developers with insights into how code works and helps developers remember details [5]. The representation of software data is often used in the areas of software maintenance and reverse engineering, where large amounts of data need to be analyzed [43]. According to Koschke [43], the simplest form of software visualization is the textual representation of software artifacts.

2.2 Related Work

The following section provides an overview of the current state of the areas *Software Visualization*[8], *Information Needs* and *Software Engineering Education*[9]. After outlining the academic state of research for the aforementioned fields, the differences between the existing works and this thesis are described.

2.2.1 Current State of Research

In the following section, the current state of research in the fields of *Software Visualization*[8], *Information Needs* and *Software Engineering Education*[9] are described.

Software Visualization

In the domain of *Software Visualization*[8], extensive research has been done on visualizing data from software collaboration tools. The software issue illustration tool of Ishizuka et al. [38] provide developers with insights into the evolution of a software project and the effects of changes made during the development process by categorizing and visualizing software issue tickets. The categorization process first extracts the title, description, and commit comments, then uses a clustering algorithm to group them. Two views

[8] `https://vissoft.info/`, Accessed: *23.08.2024*
[9] `https://conferences.computer.org/cseet/`, Accessed: *23.08.2024*

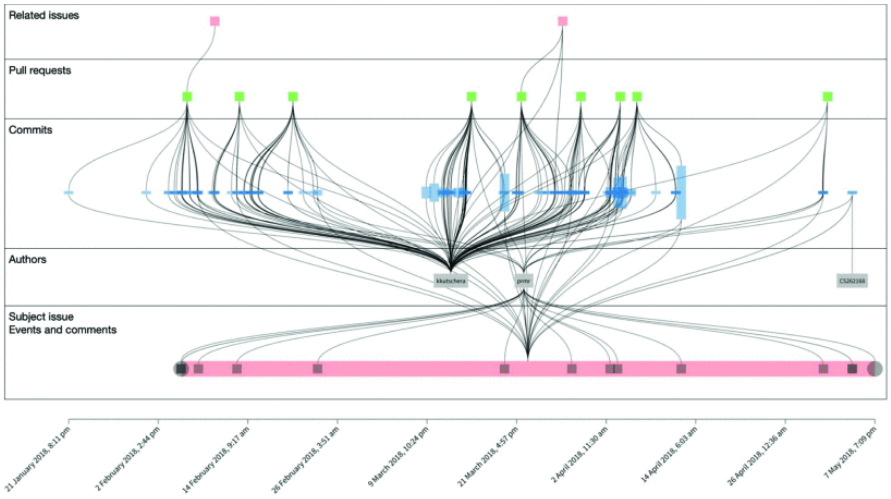


Figure 2.7: Fine-grained issue tale view [31]

are available to display the data: a ticket lifetime visualization and a ticket feature visualization. Figure 2.6a shows the heatmap of the ticket lifetime visualization. This visualization helps prioritize categories by highlighting ticket features with higher request rates. The brightness of a cell corresponds to the number of open tickets. The ticket feature visualization (Figure 2.6b), on the other hand, shows relationships between ticket categories. Both visualizations are designed to facilitate new team members in reviewing tickets and prioritizing which features to work on first [37, 38].

The visual analytic tool of Fiechter et al. [31] enables users to view the evolutionary process of software issues. This is done by modeling the project as object nodes and edges. The nodes represent commits, issues, pull requests, contributors, comments, labels and events, whereas the edges describe the relationship between the nodes, such as authorship relations to indicate the association between a node and its creator. Together, all events and actors then represent, as Fiechter et al. say, the tale of a software issue. Two visualizations can be used to analyze either individual or all software issue tickets within a project: a fine-grained and a coarse-grained view. Figure 2.7 shows the detailed view of one software issue. Over the course of three months, three developers carried out more than 150 commits and nine pull requests. In the coarse-grained illustration, each issue is visualized by a striped rectangle, where each stripe represents an object node. The height describes the number of objects, while the width of a rectangle indicates the duration of the issue [31].

The bug analytics platform in*Bug from Dal Sassc and Lanza [20] uses interactive visualizations to compare, understand and explore bug repositories. This tool provides both an overview of the data in a repository and fine-grained visualizations for detailed defect analysis. The overview offers insights into the lifetime of the bugs and includes a panel to filter and sort bug reports and another panel for project switching. The fine-grained view supplies the user with information about the persons involved in a

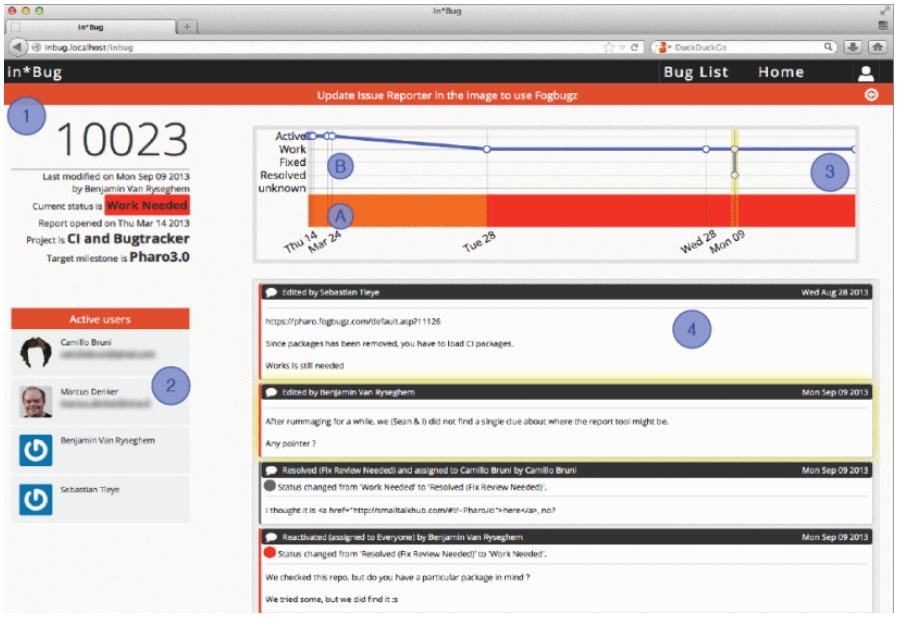


Figure 2.8: Fine-grained visualization of in*Bug [20]

bug (Figure 2.8, Number 2). Additionally, the detailed bug lifetime visualization gives information about the creation date of a bug, when it has been worked on, and whether it has been fixed or not (Figure 2.8, Number 3). In addition to a list of events that happened in connection with the bug report (Figure 2.8, Number 4), the tool also provides an area summarizing the most important metadata (Figure 2.8, Number 1) [20, 21].

BugMaps [36] is another software defect visualization tool that allows for the analysis and exploration of bugs in software projects with the goal of answering questions about lifetime, distribution, evolutionary behavior and stability. The architecture of the system consists of two components: a mapping module and a visualization module. The mapping module collects data in the form of log messages from VCSs and ITSs. Extensible Markup Languages (XMLs) parsers are used to extract the necessary information and to assign the bugs to the relevant classes. The visualization module then gets the aforementioned data, computes measures on bugs, and provides the data to the user with interactive visualizations. BugMaps offers two analysis browsers. The history browser allows users to display various bug measurements and the number of defects per class or package over their lifetime. The snapshot browser can be used to visualize the number of bugs per class and the lifetime of the bugs, as shown in Figure 2.9. Furthermore, the source code of the clicked class is displayed [36].

Like BugMaps, the approach of D'Ambros et al. [19] aims to visualize data accumulated in bug reports. The authors focus on the bug states in which a software defect can be found. The tool offers two visualizations for the analysis of a system: the System Radiography View and the Bug Watch View. The System Radiography View aims to identify where many defects are located within a system by illustrating the distribution of

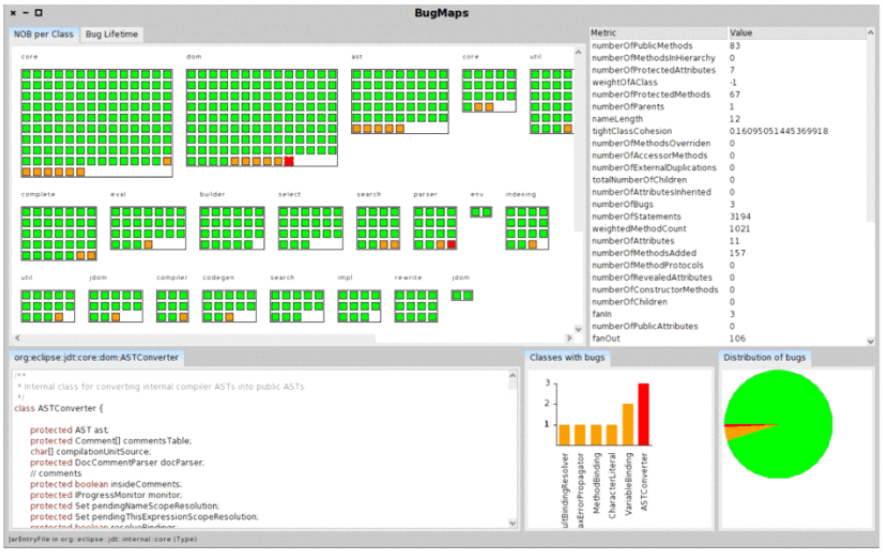


Figure 2.9: Bug lifetime and number of bugs visualization [36]

issues, highlighting components with the most open (not yet fixed) bugs, and showing the lifetime of a bug (first to last mention of the bug). When analyzing a subset or a single defect, the Bug Watch View can be used to display bugs affecting a set of components over time, allowing users to find the most significant ones [19].

The interactive visualization dashboard for provenance of software development from Schreiber et al. [64] aims to provide software project managers with insights to development process changes, development progress and interactions between developers and external participants. The tool achieves this by extracting data from ITSs and release systems. The dashboard includes the following views:

1. A tree map visualization that gives insights into the commits made by users to repositories.
2. A timeline that shows the distribution of weekly summarized events over time.
3. An event timeline that represents the development activities of a project over time.

RepoVis by Feiner and Andrews [30] is a tool that offers a visual overview of the structure, evolution and status of Git repositories. The system consists of a backend and a frontend, where the backend clones a Git repository and extracts source code, commit messages, and other metadata, making them available to the frontend via Representational State Transfers (RESTs) requests. For the visualization, four predefined modules are available: last modifications, developers, file types and issues. Each entry from a module is then represented by a different color and assigned to the corresponding files, folders and lines

Figure 2.10: Comprehensive visual overview of RepoVis [30]

of code, as seen in Figure 2.10. Additionally, the system provides a legend for the color resolution and a timeline with which the Git repository can be visualized at different timestamps. Furthermore, this system includes a full text search function, allowing the user to search specific terms or use pre-defined search shortcuts containing multiple terms. The results are then shown by highlighting the corresponding files [30].

Gitinspector[10] is a visual analytical tool for software repositories. It primarily shows statistics about the authors of a software repository, as seen in Figure 2.11. Additionally, a timeline analysis shows the workload and activity of each author, and by default, only source files are included in the analysis. Initially, the tool was developed to retrieve repository information from student projects in the object-oriented programming project course at Chalmers University of Technology[11] and Gothenburg University[12]. Some of Gitinspector[10]'s features include views about cumulative work by each author, statistical timeline analysis and scans for file types found in the repository.

Information Needs

In a software engineering context, information needs were analyzed by Buse and Zimmermann [13]. In their survey, they interviewed 110 developers and managers about analytical decision-making. It turned out that data and metrics about a software project are among the most important factors in the decision-making process of managers. In

[10] https://github.com/ejwa/gitinspector, *Accessed: 23.08.2024*
[11] https://www.chalmers.se/en/, Accessed: *23.08.2024*
[12] https://www.gu.se/en, Accessed: *23.08.2024*

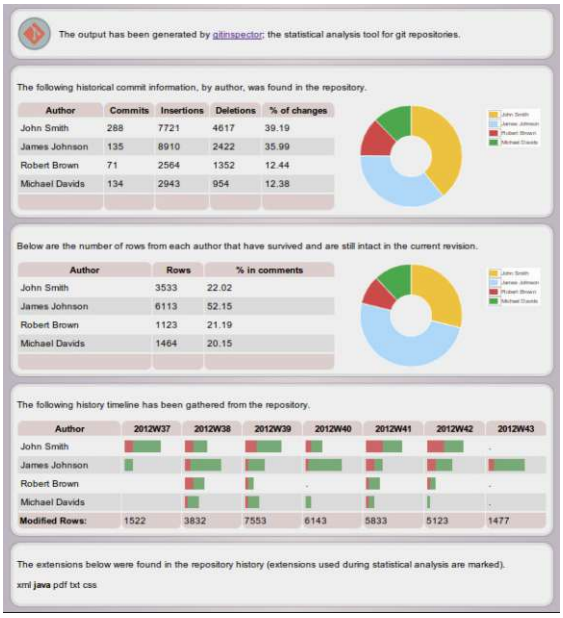


Figure 2.11: Gitinspector's[10] contributor visualization

contrast, developers rate their personal experience as an important factor. As for artifacts, they indicated that many provide valuable information. Features, components and bug reports were among the most important artifacts for managers. Developers rated classes, functions and tests as very important. Additionally, the participants described scenarios in which they would use analytical tools to make decisions; most of them can be assigned to testing, refactoring, release planning and inspection. Furthermore, the authors of the study mentioned important characteristics that any analytical tool should follow. These tools should be easy to use, produce fast and concise output, and be interactive. Common types of analysis include examining trends, addressing urgent events, mapping trends into the future, and summarizing the vast amount of data generated by the software product [13].

Buse and Zimmermann [13] also noted that participants considered information about the past of a project to be more important than trying to predict the future of a project. Moreover, predicting the future is significantly more difficult than finding information about a past or current status [13]. Codoban et al. [17] further examined developers' motivations for researching software history and the challenges they face in doing so. The study consisted of two phases: a qualitative and a quantitative method. For the former, 14 experienced developers were interviewed to explore the above points. For the latter, a survey of 217 people was conducted to quantify the responses. The results showed that recent history and old history are used for different purposes. Recent history becomes more relevant for gaining insights into changes made to the project or understanding work in progress, while old history becomes more relevant for recovering lost knowledge in software history or learning how a project evolved over time. The study also revealed that,

besides non-informative commit messages and tangled changes, information overload is one of the top challenges developers face when researching software history[17].

Tao et al. [69] examined the understanding of code changes in the software development process and the tools used by developers. The data was collected through a large online survey, followed by a series of email interviews. In the online survey, participants were asked about the most common scenarios in which understanding of code changes is required and how often they experience these scenarios. Participants indicated that they most often need to understand code changes when reviewing other people's code for comments or approval, or to quality-check their own code. Another common scenario for understanding changed code is when a new defect is introduced. Furthermore, the study identified 15 information needs, with the following two being of particular interest [69]:

- *"Is this changed location a hotspot for past changes?"*
- *"Is this changed location a hotspot for past bug-fixes?"*

Begel and Zimmermann [7] present the results of two surveys on questions software engineers would like data scientists to investigate. In the first survey, 1500 software engineers were asked to list five questions they would like data scientists to investigate. The responses of 203 software engineers were summarized into 145 questions and grouped into 12 categories. The questions then got split into subsets of questions and sent to a new sample of 2500 software engineers to rate the most important questions to work on. Among the top-rated questions, the following two are the most promising ones related to the topic of this thesis [7]:

- *"Which test strategies find the most impactful bugs (e.g., assertions, in-circuit testing, A/B testing)?"*
- *"In what places in their software code do developers make the most mistakes?"*

Bomström et al. [12] explored information needs and their presentation in agile software development by surveying three small to medium-sized companies. The results of the study show that it is important for the participants to have detailed information about the functional requests they receive. This includes details such as when a request was received, who received it, and the underlying problem. Additionally, information about the structure of the software and its evolution is considered useful for answering questions about the system and its development over time. Finally, the paper points out that information should be presented in such a way that allows for easy interpretation, enabling users to examine details and connections [12].

Software Engineering Education

Software engineering education, like software engineering, is a young discipline that combines theory and practice to equip undergraduates with the necessary concepts and principles to solve real-world problems. Both hard and soft skills are required in software engineering education. The capability to communicate and to collaborate with other people is just as important as developing and testing software [54].

ITSs and VCSs have been widely used in software engineering education in the past and have proven particularly useful in courses where students work in small groups on a project over an extended period of time [24, 45, 65, 72]. The data that accumulates in the repositories makes it easier for the teaching staff to analyze groups, thus allowing for fair grading within the group [18]. Liu [45] and Tushev et al. [72] explored in their classes that ITSs and VCSs need a learning curve before they can be used effectively in a project. Liu [45] also reported that due to the use of ITSs, the number of failing teams within a class has been reduced, and teams with significant problems can no longer hide them until the last day of delivery. Collaboration tools were also used to coordinate teams in courses where team members come from different universities, simulating the software industry globalization [29].

Jones [39] and Coppit [18] described how they used ITSs and VCSs to evaluate the work of individual students in group projects. Jones [39] used the commit history of Subversion (SVN) to compare the weekly work summaries each student submitted in order to rate the completeness and accuracy of the report. Coppit [18] described how a large project was successfully implemented in a university course. The course participants were divided into subgroups and assigned task areas. The team members chose a group leader who, in addition to planning the team's tasks, also had weekly meetings with the other group leaders. This simulated a real-world environment. An ITS and VCS were used to distribute the code and coordinate the work packages. Students were given points for each completed task, and these points were used to calculate the grade [18]. Data from ITSs and VCSs have also been displayed to indicate the distribution of work within teams, although the intervals at which they are presented to the teams vary [16, 28, 55]. Chen et al. [16] developed a system that provides weekly insights into the efforts and participation of a team.

The work of Eraslan et al. [28] presented the effects of using GitLab[13] metrics to show students their performance. They focused on the number of assigned and complete issue tickets and the number of commits made to the repository. At two checkpoints during the project, the teams were shown their report, whereby each team got only its own data. At the first checkpoint session, the report included the number of assigned and completed tasks and the number of commits for each student. For the second checkpoint session, the report also showed the number of commits for each student per week. The data was visualized in the form of tables and interactive graphs. Figure 2.12 shows an example of such a report. Students on the course were, with few exceptions, satisfied with the use of

[13] `https://about.gitlab.com/`, Accessed: *23.08.2024*

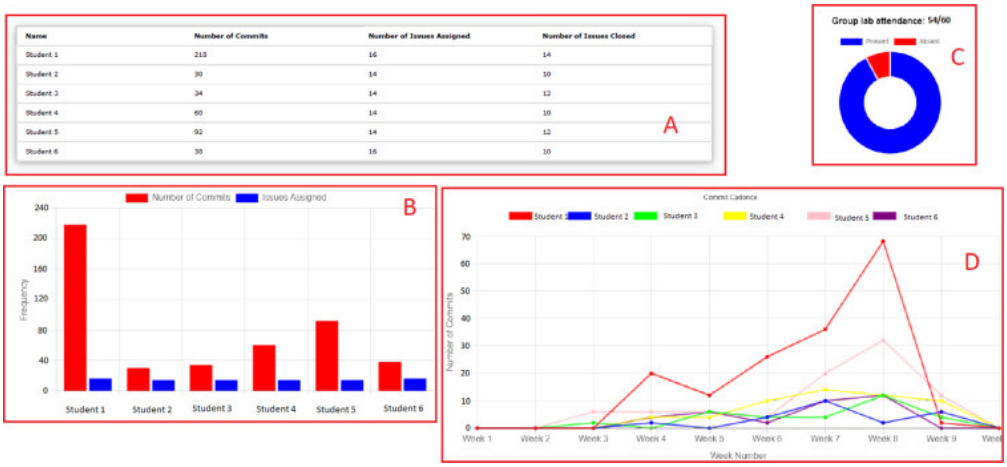
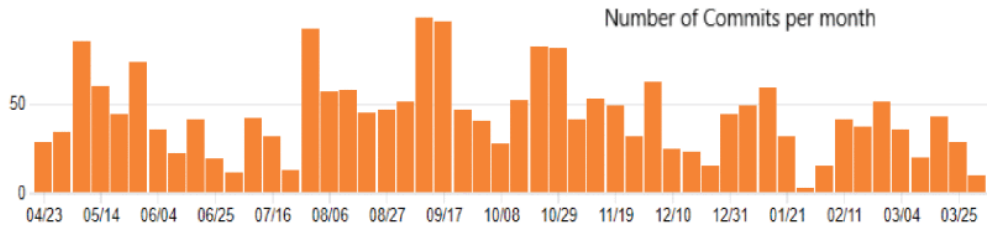
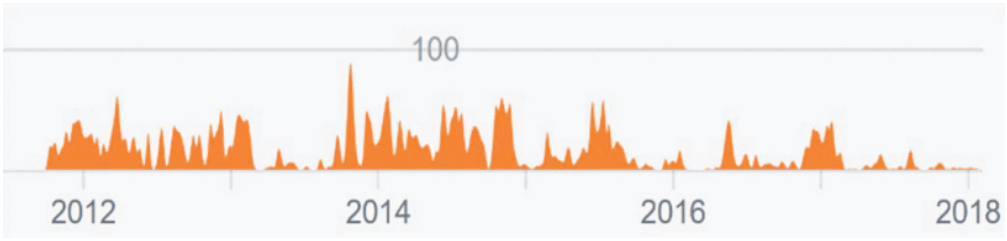


Figure 2.12: Checkpoint session report of [28]



(a) Commits per month



(b) Commits per year

Figure 2.13: The number of commits per month and year [55]

checkpoint sessions and the metrics used in these sessions. However, they also noted that the number of issues assigned and completed was the more useful metric [28].

Similarly, Parizi et al. [55] built a multidimensional monitoring service that enables stack holders to view team contributions at any time by providing the necessary information to a Git repository, as seen in Figure 2.13. The system then creates a series of reports containing visualized metrics such as (*number of commits*, *number of merge pull requests*, *number of files*, *total lines*, *time spent*) for each team member and the team as a whole.

Additionally, the collaborators are evaluated and assigned into the following categories based on their performance: *Excellent*, *Good*, *Satisfactory*, *Poor* and *Unacceptable*. The author emphasizes the importance of further data analysis as seemingly poorer performance, such as a lower number of commits, might be due to solving a more challenging problem [55].

2.2.2 Distinction from Current Research

Based on the literature research conducted in this chapter, to my knowledge, there is no similar solution to the idea proposed in this master thesis that has been performed in the field of software engineering education. Most of the systems described in Section 2.2.1 focus on giving users insight into contributor participation in a project based on basic metrics such as the number of commits contributed or number of resolved issues [16, 18, 28, 55]. However, no work addresses metrics like the local distribution of issues, participation (any form of interaction with an issue beyond assigning it to a person, such as referencing an issue in a commit message or changing the issue description) in issues, and the lifecycle (representation of the status changes) of issues.

Furthermore, no tool was found in the area of software visualization that is comparable to the solution developed in this thesis, though the work of Fiechter et al. [31] and Hora et al. [36] are the most similar. BugMaps [36] provides statistics on the local distribution of software defects but does not show these for functional requests or other project-related tasks, which can also be tracked with software issues. Furthermore, due to the chosen visualization method, it is not clear if a bug affects multiple classes. The visualizations of Fiechter et al. [31], on the other hand, focus on any type of software issue. Although these illustrations could show participation and status changes, they do not give an insight into the distribution of software issues within a project. Additionally, the information about the entire history of a software issue or the project is visualized, but there is no possibility to narrow it down to show visualizations for a specific period (e.g., for one project management iteration). Finally, both tools use data from either GitHub[14] or Jira[15]/Bugzilla[16], but not from GitLab[13] repositories, the collaboration tool used for the courses SE PR and ASE.

The visualizations created in this master thesis fulfill the information needs to be described in Section 2.2.1 in multiple forms. First, the tool presents the history of a software project by illustrating the software issues on a timeline (e.g., the date an issue was created), which is important for various reasons, as addressed in [13, 17]. Second, the study of Tao et al. [69] and Begel and Zimmermann [7] has shown that there is an interest in finding out where clustering occurs. Due to the distribution visualization of the software issues and locations (packages, classes), accumulations can be assessed by the tool. Lastly, by displaying historical data, the tool also makes it possible to identify trends and thus analyze possible strategies, as mentioned in [7].

[14] `https://github.com/`, Accessed: *23.08.2024*
[15] `https://www.atlassian.com/de/software/jira`, Accessed: *23.08.2024*
[16] `https://www.bugzilla.org/`, Accessed: *23.08.2024*

CHAPTER 3

Research Design

This chapter describes the research questions defined in this thesis and the methodology used to answer these questions

3.1 Research Questions

This section defines the research questions RQ1 - RQ3 that form and guide the scope of this thesis.

RQ1 a) What information needs exists for a detailed analysis of software issues that can be used for software engineering education?

b) What visual aspects are necessary to support detailed analysis of software issues in software engineering education?

RQ2 How do experts rate the proposed visualization concepts for use in software engineering education?

RQ3 a) How do the experts evaluate the visualization developed with regard to its hypothesized benefits in software engineering education?

b) How effective is the developed visualization compared to traditional methods in software engineering education?

3.2 Methodology

The following section describes a five-phased methodological approach, derived from Wieringa's [75] engineering cycle, used to obtain the results of this master's thesis. This engineering cycle extends the design cycle, which consists of the tasks of problem

investigation, treatment design, and treatment validation, by implementing and evaluating a validated treatment in the real world. In the first phase (Section 3.2.1) a literature and tool research were conducted in order to investigate the problem. Section 3.2.2 describes the steps that were taken to design the concept, and Section 3.2.3 outlines the validation, which was done by carrying out semi-structured expert interviews. Subsequently, the validated design was implemented with an iterative process (Section 3.2.4) and evaluated with scenario-based expert interviews (Section 3.2.5).

3.2.1 Literature Research

Phase 1 was concerned with the investigation of the problem. As suggested by Wieringa [75], a systematic scientific literature review was performed to obtain an overview of the current state of the art in the following research areas:

1. *Software Visualization*[1]
2. *Information Needs*
3. *Software Engineering Education*[2].

As a starting point, software visualization approaches that attempt to solve a similar problem were researched. The search engines Google Scholar[3] and IEEE Xplore[4] were used to find academic papers related to the research fields of this thesis. The majority of them were found in ACM[5] and IEEE[6] databases. Further publications were found through the cited sources of the found papers or by papers referencing the papers already found. Google Scholar[3] and Connected Papers[7] were used for this purpose. With the results of the literature research, RQ1a was answered.

3.2.2 Conceptual Design

In Phase 2, the treatment was designed. Artifacts, for this master thesis visualizations, were created that could solve the investigated problems from the literature review described in Section 3.2.1. These were established on the basis of necessary functionalities that were missing in similar works. First, the main features were sketched, and then mock-ups were designed. These visualizations were then used to present the features in the semi-structured expert interviews. The results of this phase were taken to answer RQ1b.

[1] `https://vissoft.info/`, Accessed: *23.08.2024*
[2] `https://conferences.computer.org/cseet/`, Accessed: *23.08.2024*
[3] `https://scholar.google.com/`, Accessed: *23.08.2024*
[4] `https://ieeexplore.ieee.org/Xplore/home.jsp`, Accessed: *23.08.2024*
[5] `https://dl.acm.org/`, Accessed: *23.08.2024*
[6] `https://www.ieee.org/`, Accessed: *23.08.2024*
[7] `https://www.connectedpapers.com/`, Accessed: *23.08.2024*

3.2.3 Semi-Structured Expert Interviews

In Phase 3, the designed treatment was validated to examine whether the artifacts created could help achieve the goals. As mentioned by Wieringa [75], the easiest way of validating a treatment is to get expert opinions on how the artifacts interact with the problem context and what effects they would have. Therefore, a qualitative method with the support of a questionnaire was conducted. For the questionnaire, the survey software Google Forms[8] was used. The participants were asked to evaluate the features illustrated by the mockups. To do this, respondents had to rate the usefulness of the visualizations. Participants of this interview were mainly software engineers with educational experience. Feedback received through interviews was then incorporated into the design of the visualizations, and features rated more important were implemented earlier than lower-rated features. To answer RQ2, the results of the interview session were taken into account.

3.2.4 Implementation

In Phase 4, the visualizations were developed using an iterative development approach. A prototype was developed based on the mock-ups mentioned in Section 3.2.2 and the requirements identified in Phase 3. Due to the iterative development process, after each iteration, assumptions and possible extensions were discussed. The final visualizations were then defined as the working prototype and utilized for the scenario-based expert evaluation.

3.2.5 Scenario-Based Expert Evaluation

In Phase 5, after the completion of the prototype, the purposefulness of the developed visualizations was evaluated. Again, interviews with experts, mainly with experience in software engineering education, were chosen to rate the implementation. Scenarios were defined in advance and solved in a given context using the developed illustrations. In addition to the usefulness of the visualizations, the interviewees also had to describe how they would solve these scenarios using other technologies. In order to minimize incorrect results, each participant received a short introduction before executing the scenarios and answering the questions. Subsequently, the results were analyzed and interpreted to answer RQ3.

[8] https://www.google.com/intl/de_at/forms/about/, Accessed: *23.08.2024*

CHAPTER 4

Semi-Structured Expert Interviews

The following chapter details the process of developing and validating the visualization concepts. The concepts are first described in more detail in Section 4.1. Further, Section 4.2 outlines the design of the first interview session before Section 4.3 - 4.4 concludes the chapter with the results of the interviews and the determined requirements of the prototype.

4.1 Concept

Based on the scientific literature review of the current state of the art and additional data available in ITS repositories, potential features for historical analysis of software issues were defined. The features are summarized in Table 4.1 and realized by the visualization concepts in this section.

Feature	Description	Source
F-1	Show which areas are hotspots for frequent changes	[13, 69]
F-2	Show the working time of changes	[13]
F-3	Show the issue status changes of software issues	
F-4	Show contributions of team members	[16, 28, 55]
F-5	Show general information on software issues	[28]
F-6	Show historical and evolutionary information	[12, 13, 17]
F-7	Filter results based on multiple criteria	[17]
F-8	Compare multiple versions of history	[17]

Table 4.1: Overview of the proposed features

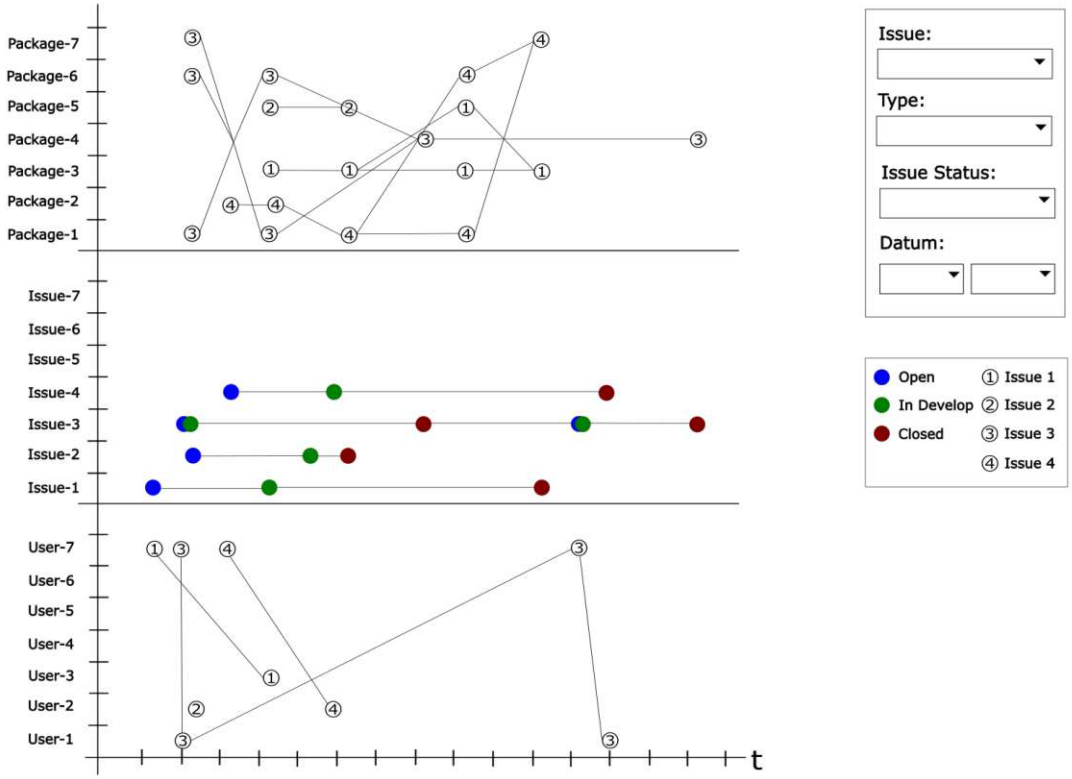


Figure 4.1: First vision that implements the features from Table 4.1

The conceptualization phase consisted of two stages. In the first stage, rough sketches of the illustrations were created using the drawing tool Inkspace[1], a tool for creating two-dimensional vector graphics. These initial visualizations, shown in Figure 4.1, consist of three sub-graphs: the top one showing the distribution of software issues within a project, the middle one showing the status changes of all issues, and the bottom one showing the developers involved in each issue. Additionally, there is an option to filter the issue tickets according to different criteria and a legend to resolve the meaning of the points and their colors. This graphic served as a preliminary draft of the idea, but was not sufficient for the detailed concepts needed for the interviews. It quickly became apparent that distinguishing tickets was difficult and that the visualization lacked meaning. However, as these and potential further changes would have required significant effort, it was decided to implement the sketches with the help of a visualization library.

In the second phase, the mock-ups for the interviews were created using the React[2] graph library Recharts[3], which offers an easy way to create charts with React[2] components,

[1] `https://inkscape.org/`, Accessed: *23.08.2024*
[2] `https://react.dev/`, Accessed: *23.08.2024*
[3] `https://recharts.org/en-US/`, Accessed: *23.08.2024*

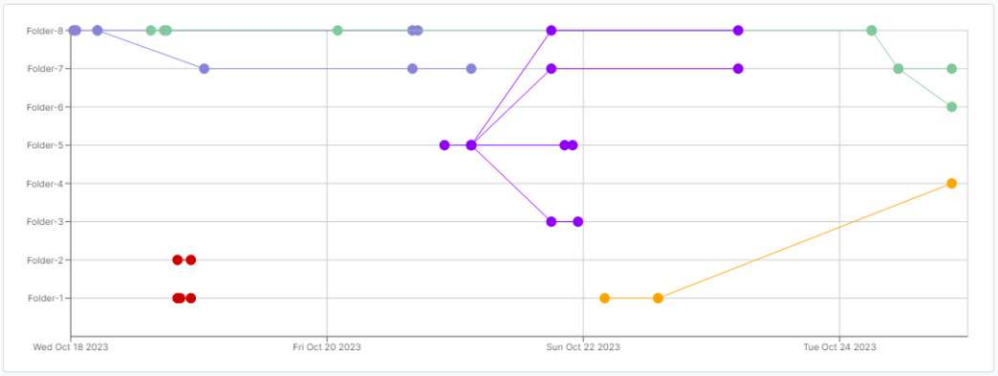


Figure 4.2: Local distribution of software issues with a Git-like graph

leveraging lightweight dependencies on D3.js[4] submodules. Recharts[3] provides several chart types and allows for testing these charts on the fly by customizing the data. The following sections 4.1.1 - 4.1.4 describe the visualizations that were created during the second phase.

4.1.1 Issue Distribution *(F-1, F-2, F-6)*

This section describes the views that represent the local distribution of software issue tickets within a project. The purpose of these graphics is to provide the user with a way to better assess which areas of the software project change frequently. Modifications can result from change requests, tasks, discovered bugs or other types of changes that can be tracked within an issue ticket.

Figure 4.2 shows a visualization representing the local distribution of software issues in a Git-like graph. In this view, the x-axis corresponds to the time, showing the project's timeline, while the y-axis represents the software locations, which can be either folders, packages or files. Each software issue is denoted by a unique color and may consist of several points, each point representing a change to a location made as part of a commit. However, it should be noted that a commit can only be associated with an issue if the issue identifier is mentioned in the commit message. When a commit affects multiple locations, each location is assigned its own point. The x-axis shows the temporal occurrence of the commits, allowing the viewer to determine the approximate processing time of the software issue. Each commit is linked to its predecessor. If a location has already been changed in the course of an issue, the predecessor is the previous change to that location. If the location is changed for the first time, the closest commit is selected, considering both temporal and spatial proximity.

This creates a Git-like graph, which is often used to interpret the history of a VCS and is easy to understand for those familiar with Git. Another advantage of this visualization is that it minimizes confusion between issue tickets that introduce changes in multiple

[4] `https://d3js.org/`, Accessed: *23.08.2024*

places, as each issue ticket is uniquely colored. Furthermore, this graphic has a practical side effect as it offers the possibility of early detecting merge conflicts since it is common practice in software development to create a dedicated branch for an issue.

The next visual representations, shown in Figure 4.3 provide a slightly different view of the distribution of software issue tickets in a project. Again, folders, packages, and files are shown on the y-axis, with each location represented by a horizontal bar, which may consist of multiple sub-bars. The sub-bars are coded in different colors to represent different software issues. The width of the bars is determined by various attributes displayed on the x-axis. For this concept, the number of attributes that influence the size of the bar was limited to the following three:

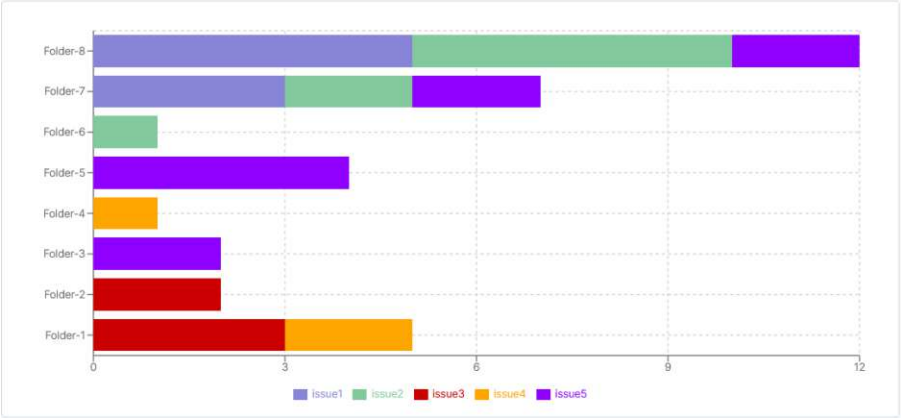
1. Number of commits: The width of the bar depends on the number of commits associated with a location and issue, as shown in Figure 4.3a.
2. Duration to resolve an issue: In Figure 4.3b the time taken to resolve an issue affects the size of the bar. More precisely, it is affected by the time elapsed between two commits, which are then summed up to give a total length.
3. Number of changed lines of code: The width of a bar is affected by the number of lines of code that have been modified during the course of an issue. This includes added and removed code lines, as illustrated in Figure 4.3c.

Due to the simplicity of this visualization, it should be easy to compare the locations within the software and x-values can be adapted to the needs of the viewer. Again, due to the different colors representing the issues, one issue ticket cannot be mistaken with another issue ticket. Compared to the visualization in Figure 4.2, this approach does not provide a detailed overview in a single graph, but collectively, all three visualizations offer a comprehensive overview of the local distribution of software issues in a project.

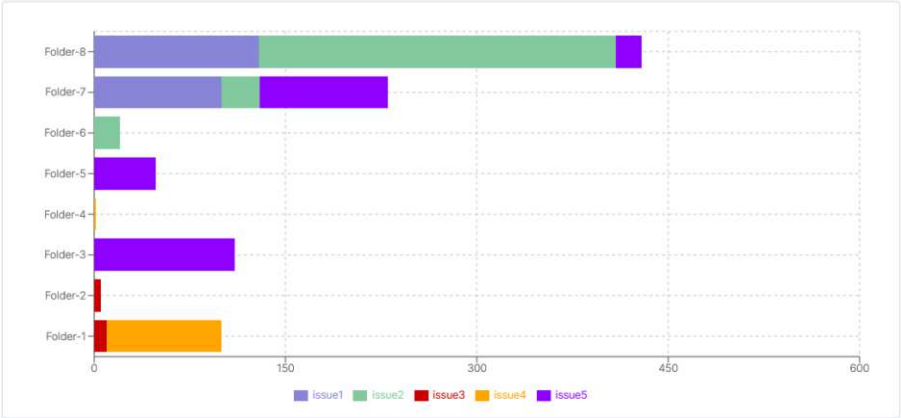
Lastly, the illustration in Figure 4.4 also provides an overview of the local distribution of software issues. Compared to the visualizations in Figure 4.3, the same attributes are considered here, but the values are of relative nature. Each attribute corresponds to its own pie chart, with the number of commits in yellow, the time required to solve an issue in blue, and the number of changed lines of code in red. Each location is represented by a segment with an associated average value. This visualization provides the coarsest overview among all visualizations in this section.

4.1.2 Issue Status Changes *(F-3, F-6)*

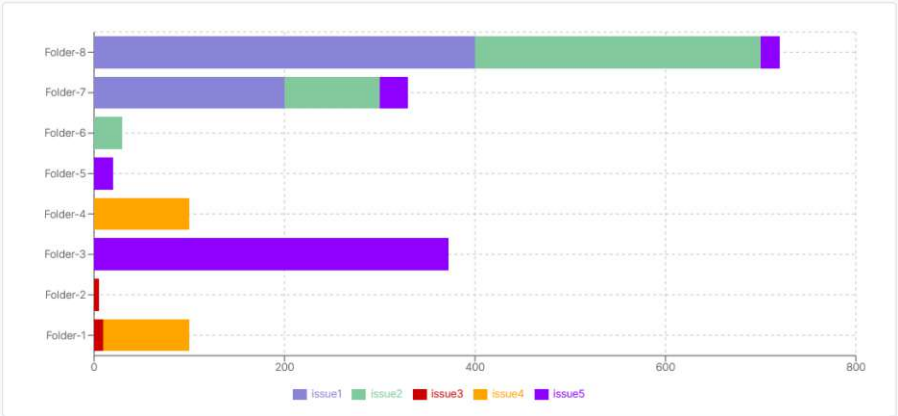
This section considers views that visualize the status changes of all software issues in a project, allowing the user to analyze the status a particular software issue has been in, how long it has remained in a particular status, and the average duration a software issue stays in each status. The following status are considered in these views: *Open*, *Develop*, *Test*, *Review* and *Closed*.



(a) Local distribution of software issues (number of commits)



(b) Local distribution of software issues (duration of issue)



(c) Local distribution of software issues (number of changed code lines)

Figure 4.3: Local distribution of software issues with different attributes

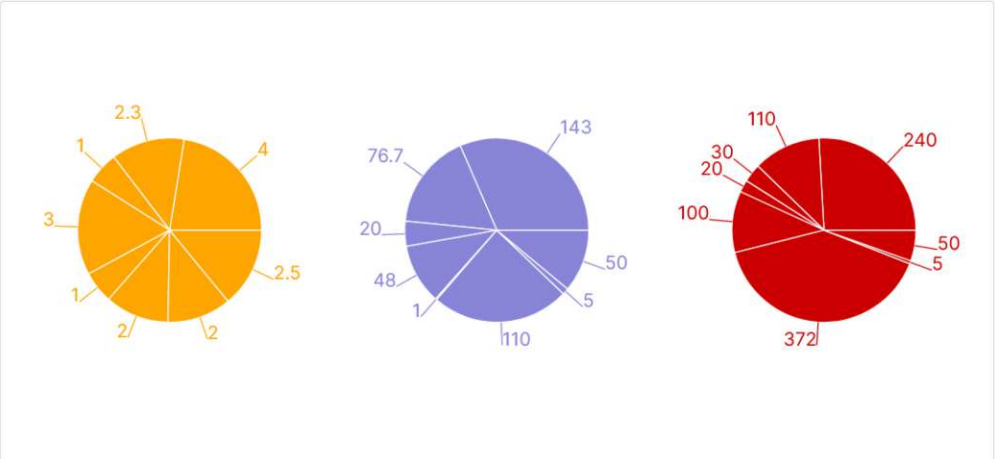


Figure 4.4: Local distribution of software issues with the attributes *Number of commits*, *Duration to resolve issue*, *Number of changed code lines* as relative values

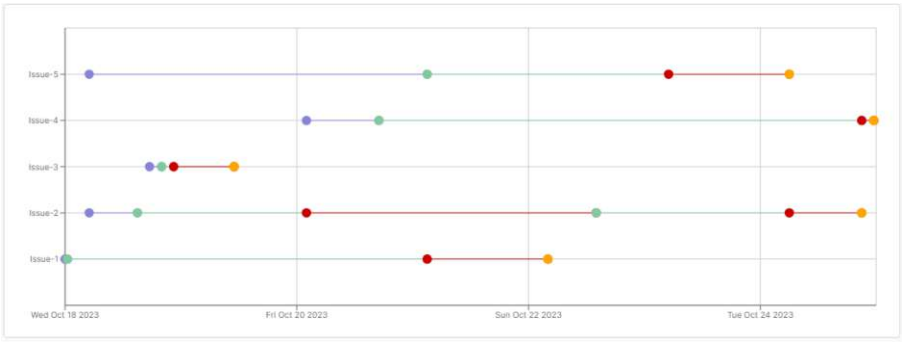
The representation in Figure 4.5a illustrates the evolution of issue status changes for each software issue ticket in a given project in the form of a line diagram. The y-axis shows the distinct issue identifiers, and the x-axis displays the timeline. The line chart visualizes the states a software issue has been in, with each phase represented by a unique color. In this view, the status *Open* is displayed in blue, the status *Develop* is shown in green and status *Test* and *Review* are presented in red and orange respectively. A transition from one status to the next is represented by a point, with the color of the point corresponding to the color of the new status. This makes it clear at which point in time an issue changed to which status.

In comparison, the visualization in Figure 4.5b provides information about how long an issue remains in a status on average. This is done using a radar chart where the different issue states are displayed in the corners of the chart. The size of an entry depends on the average issue processing time in a status. Furthermore, a distinction can be made between different types of issues, such as change request or a software defect, as shown in the example of Figure 4.5b. These two visualizations allow the user to analyze the lifecycle for individual issues as well as for all issues in the entire project.

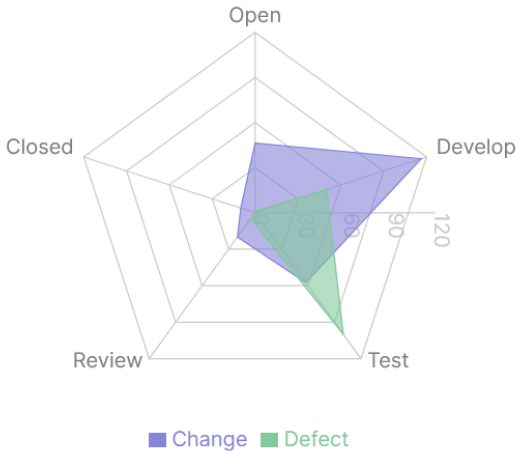
4.1.3 Issue Contributions *(F-2, F-4, F-6)*

The views in this section are intended to support analysis of the contributions per student. Viewers can find out which team members have worked on which software issue and how much time they spent on them. These visualizations also provide an overview of the distribution of work within the project team. Contributions to an issue can be made in several ways, including adding commits that reference the issue identifier in the commit message or by editing the description of the software issue.

The visualization in Figure 4.6 shows student contributions using a line graph. Team



(a) Visualization of software issues and their duration in a software issue status



(b) Visualization of the average duration in a software issue status

Figure 4.5: Software issue status visualizations

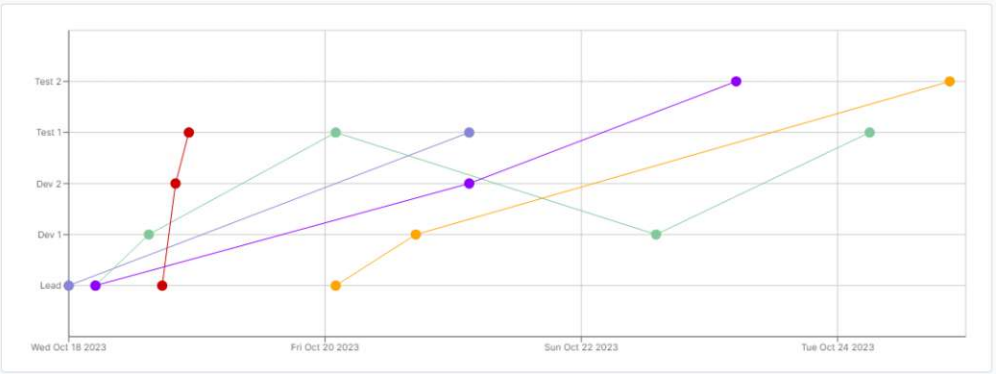
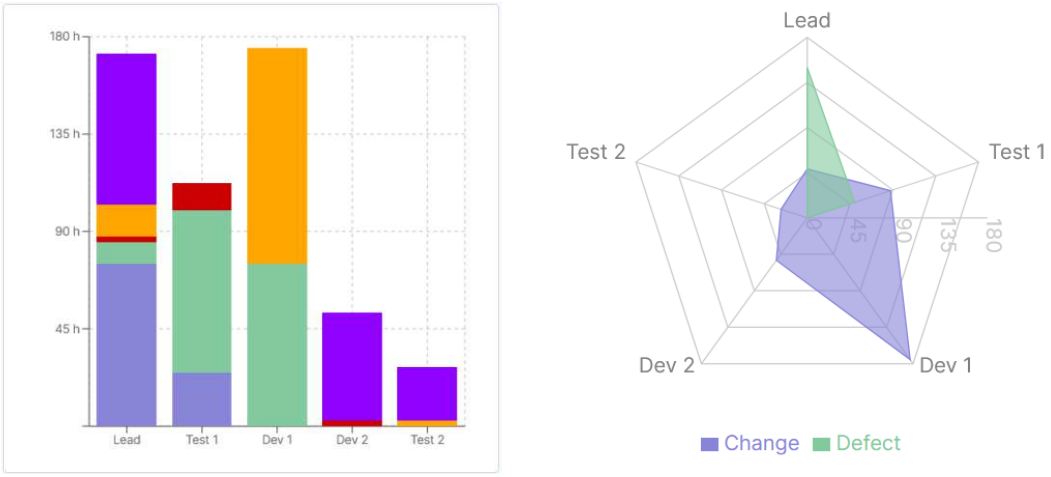


Figure 4.6: Students contributions to software issues



(a) Students and the software issues they worked on

(b) The average tracked time on software issues for each student

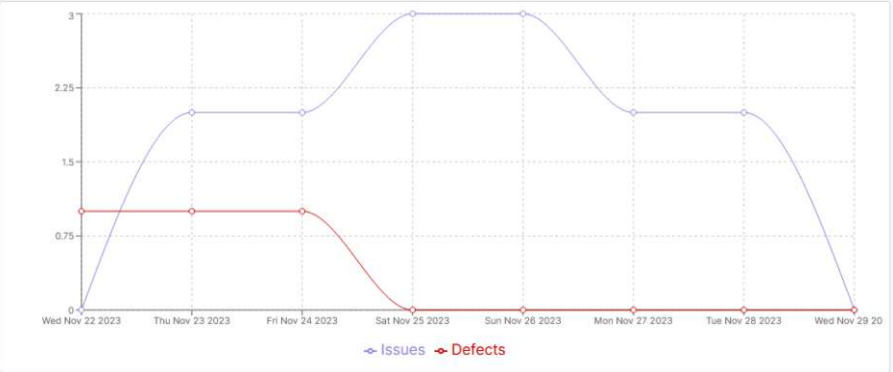
Figure 4.7: Work distribution visualization

members who have contributed to a software project are listed on the y-axis, while the project's timeline is shown on the x-axis. Each software issue ticket is represented by a line, with each line displayed in a unique color assigned to a software issue. Whenever a developer contributes a change to an issue, a point is added to the corresponding line at the time the change was made. This visualization enables viewers to see not only the contributors of a specific issue ticket, but also the contributions to the entire project, highlighting how work has been distributed among the students.

An alternative approach was taken for the illustrations in Figure 4.7. The visualization in Figure 4.7a shows the students who have contributed changes to a project on the x-axis, while the y-axis shows the hours each contributor has tracked. Each team member is represented by a bar, divided into sub-bars, while a sub-bar corresponds to a software issue in a unique color. The height of each sub-bar depends on the number of hours spent on that ticket. The total bar represents the total time a user has spent on all issues. The illustration in Figure 4.7b uses a radar chart to provide insight into the average processing time in hours for an issue type for each student. In this example, a distinction is made between change requests and bug tickets. Users of this visualization can compare team members based on the time spent and the involvement with the tickets. Additionally, they can also see which developers were involved in which types of issues and how much time they spent on them on average.

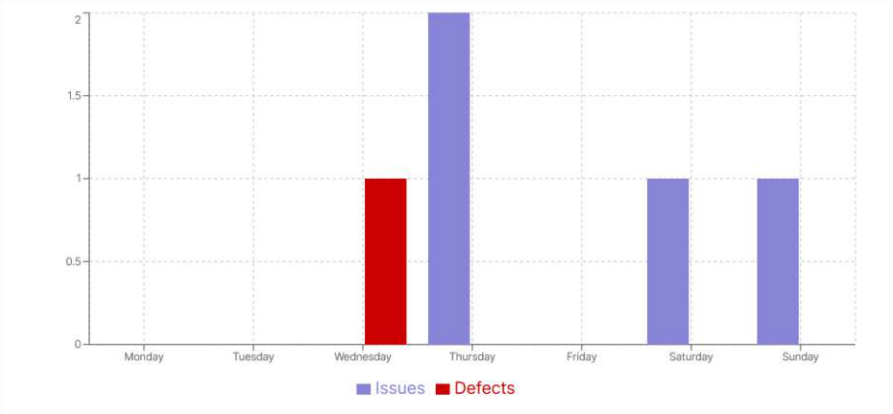
4.1.4 General Issue Information *(F-5)*

The visualizations in Figure 4.8 are designed to provide a general overview of the number of software issues in a project over a given period. The line graph in 4.8a allows the



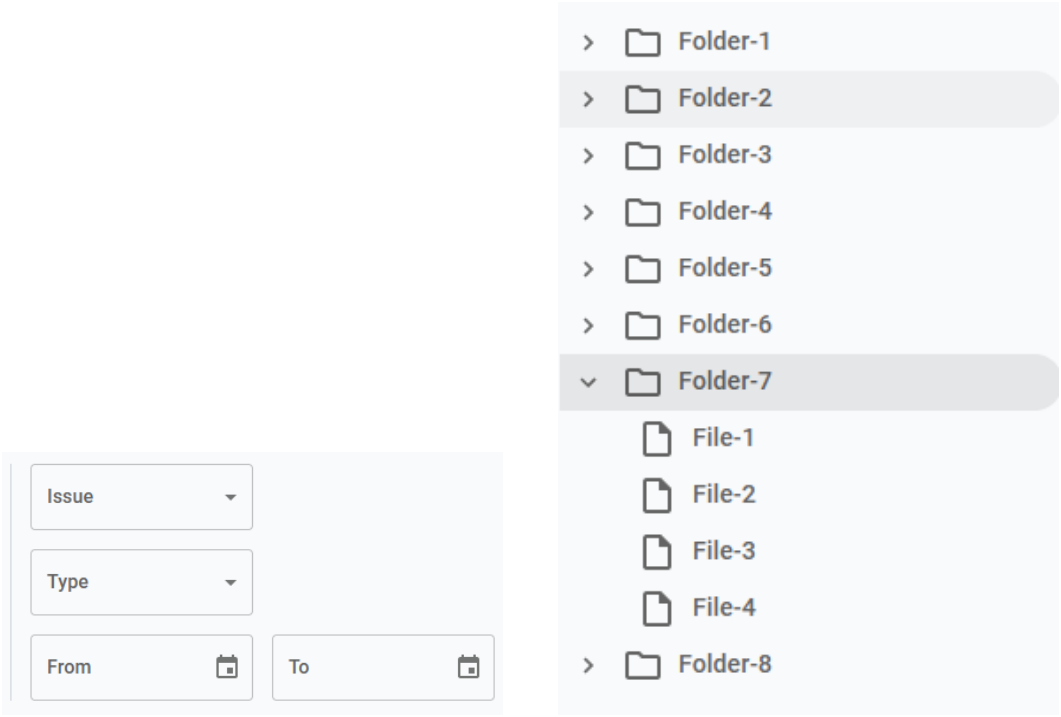
(a) Number of software issues in a project

(b) Number of issues resolved in a project



(c) Average number of issues created, grouped by weekday

Figure 4.8: General issue information



(a) Filter options (b) Collapsible folder structure

Figure 4.9: Possible filter options

viewer to see how many issue tickets were active in a project at any given time, referring to all issues whose status was not *Closed*. The x-axis represents the time, while the y-axis plots the number of software issues. Each time an issue is created or closed, it is represented in this graph with a new point, increasing or decreasing the number of issue tickets by one. The next graph in Figure 4.8b is a cumulative representation of the resolved software issues, which is increased with each closed issue ticket. The last chart in Figure 4.8c uses bars to show the average number of issues created on a weekday over a given period of time. All of these illustrations distinguish between different types of issues, in these examples between software defects and software issues that are not bugs.

4.1.5 Filters and Version Comparison *(F-6, F-7, F-8)*

During the development of a project, a large amount of data can accumulate in the repositories of ITSs. This makes it difficult to narrow down the appropriate subset, especially when analyzing a specific area. To counteract this and to adapt to the search the user needs, the visualizations provide functionality to filter the data. A possible set is shown in Figure 4.9. The filters should be able to display software issues only for a particular selection, a type (e.g., only bug tickets) or in a selected time range. It should also be possible to only show issues that contain a minimum number of commits, tracked time and lines of code changed. In addition, there should be an option to compare two

areas of a project with each other (e.g., two visualizations from different time periods side by side). The collapsible folder structure in Figure 4.9b allows users to view software issues for a specific location by only visualizing changes for the folders, packages, and files in that location.

4.2 Interview Design

A series of semi-structured expert interviews were conducted to determine the importance and order of the implementation of the visualizations. A qualitative approach was chosen, supported by a set of guiding questions asked during the interview session. These questions were divided into three sections, each area dealing with a different topic and containing both open and closed questions. The first section covered the background of the person. Questions were asked about demographics, experience in software engineering, software engineering education, ITSs and VCSs. The second part of the survey assessed the relevance of the proposed features. The questionnaire can be found in Appendix 8.1.

To introduce the participants to the topic of this master thesis, a short explanation of the problem and the proposed solution approach was provided. In total, five individuals working in software engineering education were questioned during the interviews. The first meeting served as a rehearsal to test the quality of the questions and make improvements. The feedback showed that the purpose of the interviews needed to be explained more clearly in the introduction, and some visualizations needed more detailed descriptions. It was not clear to the expert that the visualization of the software issue included not only bug tickets, but also issues created for other purposes, such as change requests. Additionally, for the visualization in Figure 4.2, there was confusion as the expert thought the locations on the y-axis refer only to folders, and it was not described in enough detail that the visualizations should also work for packages or files.

In order to provide participants with as much scheduling flexibility as possible, the interviews were conducted remotely using Zoom[5]. This tool also allowed the interviews to be recorded, enabling responses to open-ended questions to be added after the interview sessions, with the interviews lasting an average of 35 minutes.

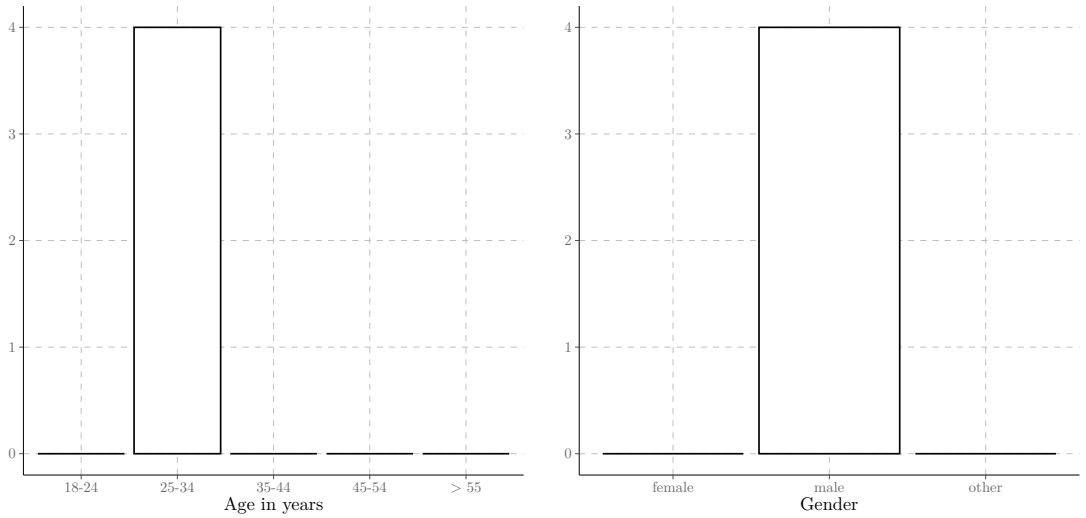
4.3 Results

The following section provides a detailed summary of the results of the expert interviews. A total of four developers, with experience in software engineering and software engineering education, participated in the interviews.

4.3.1 Demographics

The first two questions addressed the demographic information of the interviewees. Answer options were provided for both questions. Participants could also specify their

[5] `https://zoom.us/`, Accessed: *23.08.2024*



(a) Age of participants (b) Gender of participants

Figure 4.10: Demographics of participants

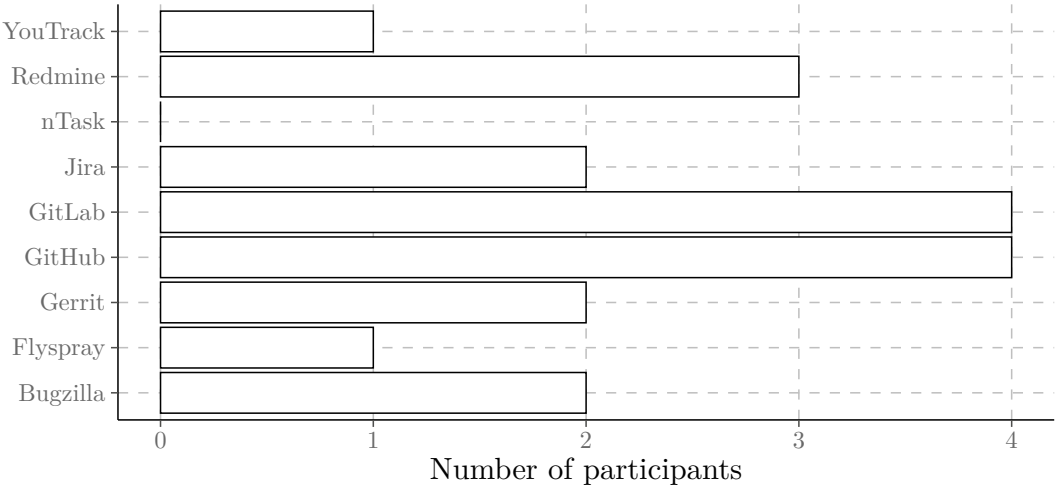
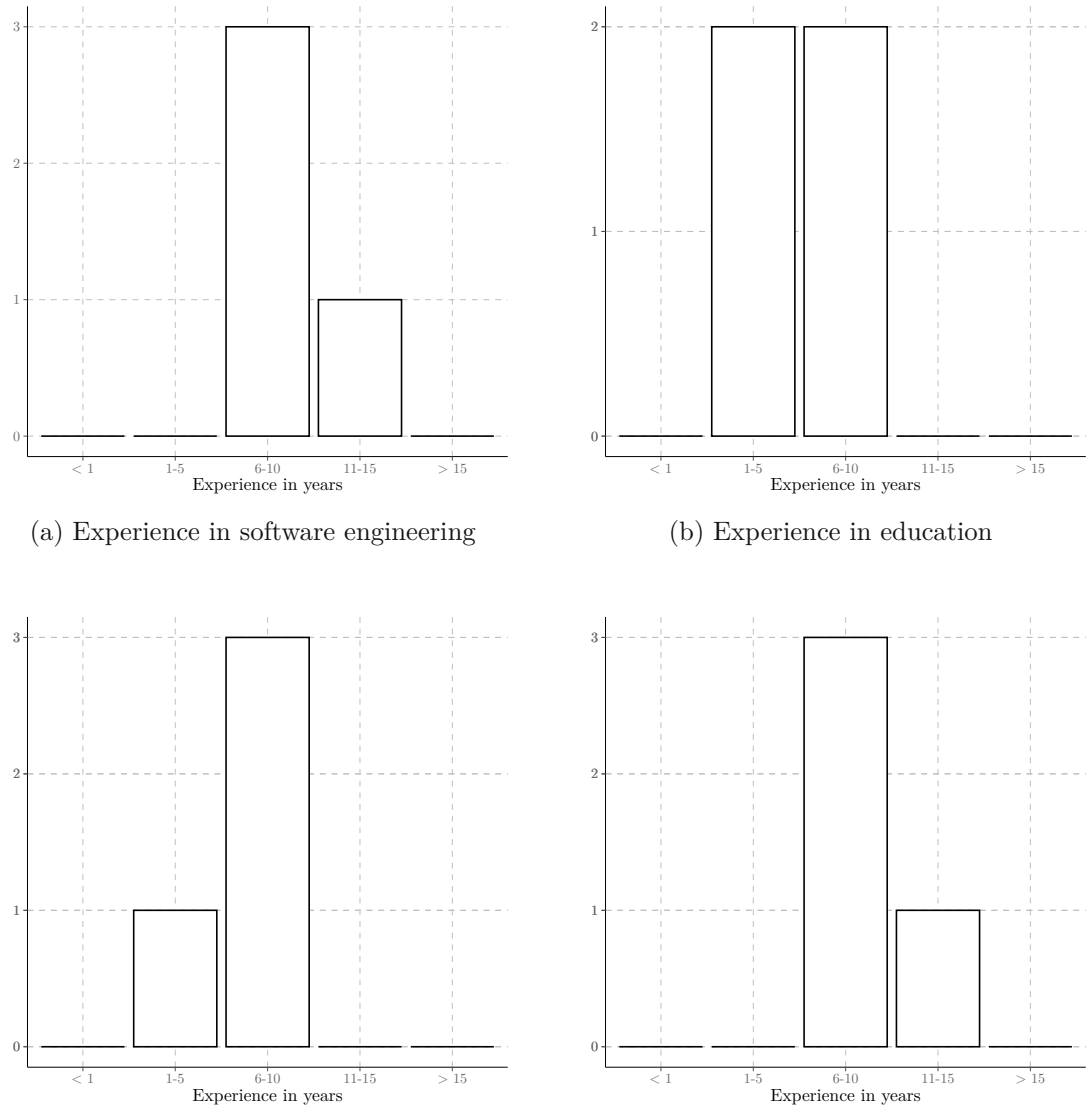


Figure 4.11: Used ITSs and VCSs

gender identity. As shown in Figure 4.10, all participants indicated that they were between *25 and 34 years* old, and all of them identified themselves as *male*.



(a) Experience in software engineering

(b) Experience in education

(c) Experience with ITSs

(d) Experience with VCSs

Figure 4.12: Experiences of participants

4.3.2 Experiences

The next four questions focused on the participants' experience with the areas of software engineering, software engineering education, ITSs and VCSs. Answer options reflected their experience in years, with ranges of *less than one year*, *1-5 years*, *6-10 years*, *11-15 years* and *more than 15 years*. Figure 4.12 shows the responses to those questions. All participants, except Participant 1 (*11-15 years*), reported *6-10 years* of experience in software engineering (Figure 4.12a). In terms of software engineering education, both Interviewees 1 and 4 reported *6-10 years* experience, while Interviewee 2 and 3 answered this question with *1-5 years* experience. Regarding experience with software maintenance tools, Developer 3 had the least experience with ITSs (*1-5 years*) and Developer 4 had the most experience with VCSs (*11-15 years*). All other interviewees mentioned *6-10 years* experience for both ITSs and VCSs.

Q7 aimed to identify which ITSs and VCSs have been used by the software engineers. Participants were given a selection of the most popular tools in these categories. As shown in figure 4.11, all participants were used with GitHub[6] and GitLab[7]. Redmine[8] was used by all participants except Participant 3, and the most interviewees were also familiar with Jira[9], Gerrit[10], and Bugzilla[11]. Finally, YoutTrack[12], Flyspray[13], and nTask[14] were among the least used software collaboration tools.

4.3.3 Importance Evaluation

The questions in this section focused on determining the usefulness of the proposed visualizations and prioritizing their implementation. Participants rated each visualization on a scale from 1 to 5, with 1 being *Not useful* and 5 being *Very useful*.

State of the Art

Before evaluating the effectiveness of the proposed features, Q8 and Q9 assessed the usefulness of the issue boards provided by ITSs, with Q9 focusing on the effectiveness of historical data analysis. Interviewees were given a screenshot of an issue board from a project in software engineering education, where all tickets were in the column *Done*. The results of these questions are presented in Figure 4.13. For Q8, the respondents gave a mean score of 4.25. Developer 1 noted that even if tickets are only in the *Done* column, it is still useful information. Participant 3 added that, issue boards should be used in every project, as they clearly show which person is working on which task and which

[6] `https://github.com/`, Accessed: *23.08.2024*
[7] `https://about.gitlab.com/`, Accessed: *23.08.2024*
[8] `https://www.redmine.org/`, Accessed: *23.08.2024*
[9] `https://www.atlassian.com/de/software/jira`, Accessed: *23.08.2024*
[10] `https://www.gerritcodereview.com/`, Accessed: *23.08.2024*
[11] `https://www.bugzilla.org/`, Accessed: *23.08.2024*
[12] `https://www.jetbrains.com/de-de/youtrack/`, Accessed: *23.08.2024*
[13] `https://github.com/flyspray/flyspray`, Accessed: *23.08.2024*
[14] `https://www.ntaskmanager.com/`, Accessed: *23.08.2024*

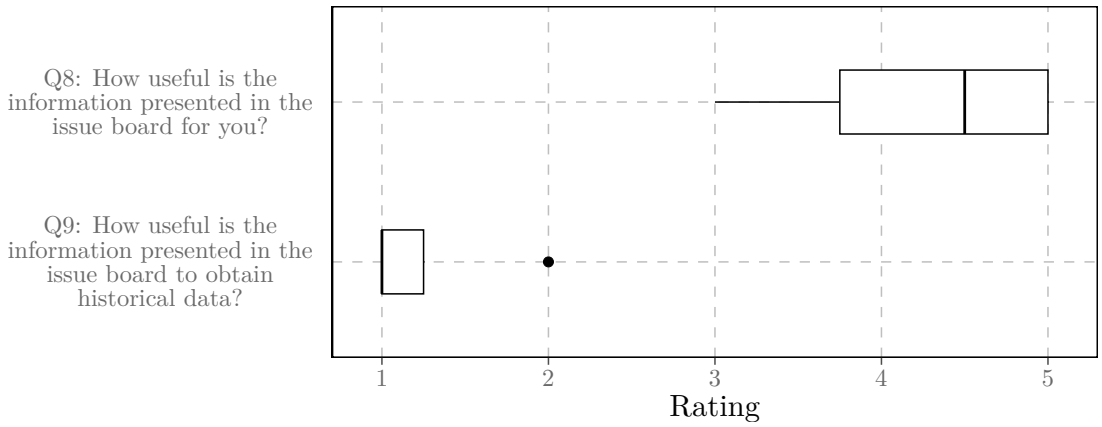


Figure 4.13: Score of issue board questions

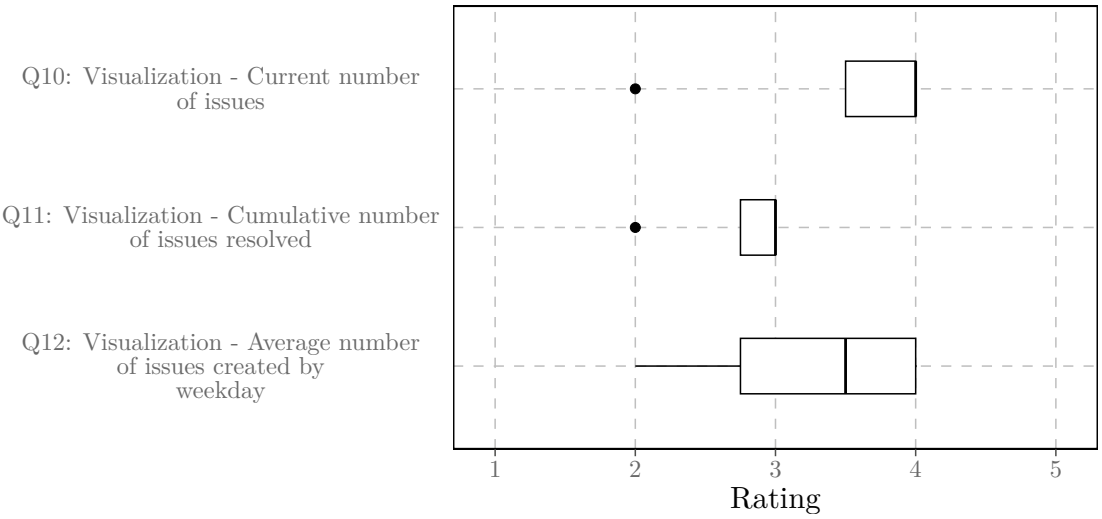


Figure 4.14: Score of general issue information visualizations

tasks need to be completed. However, except for Interviewee 3, who thinks that some historical insights can be gathered from issue boards (e.g. by sorting the closed issues), all participants rated the usefulness of issue boards for historical data with *Not useful*, resulting in a mean score of 1.25. These results indicate that issue boards are not very useful for historical data, and that additional presentation methods are needed for such purposes.

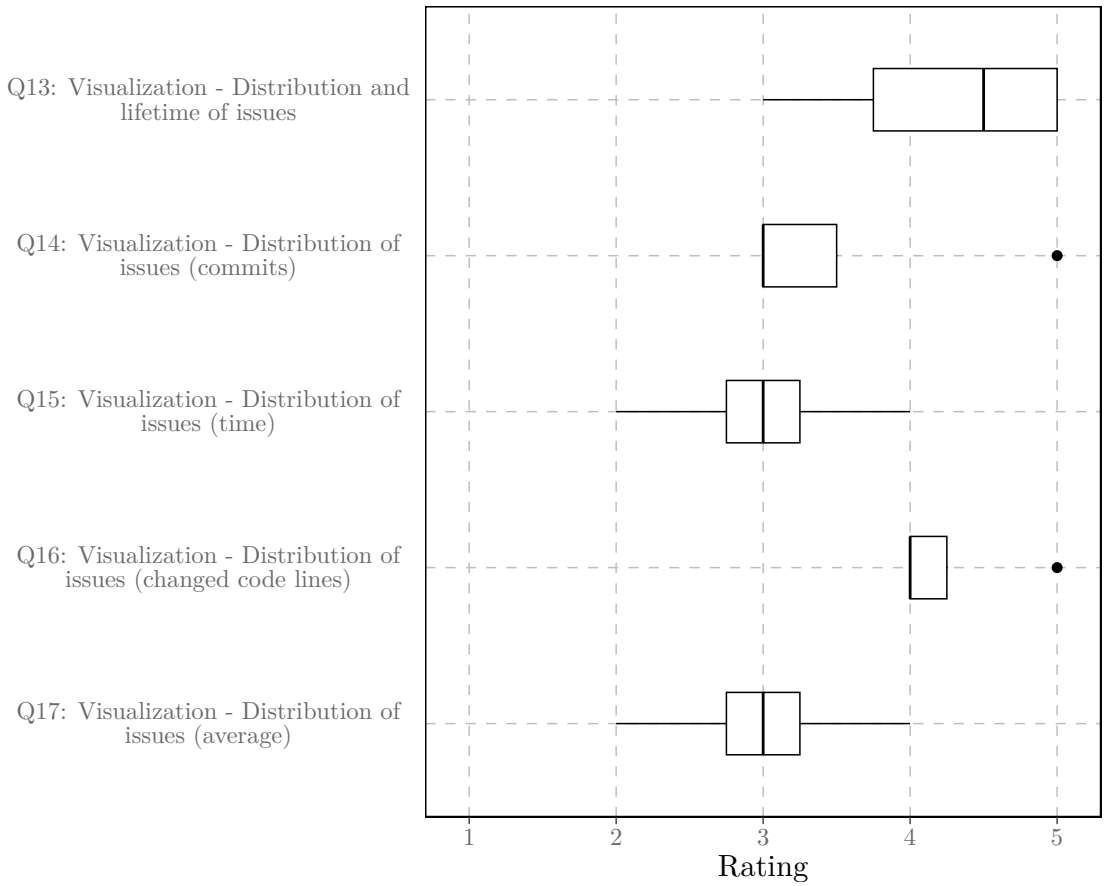


Figure 4.15: Score of issue distribution visualizations

General Issue Information *(F-5)*

Figure 4.14 shows the interviewees' responses to Q10-Q12, rating the purposefulness of the visualizations that provide general issue information (Figure 4.8). Developers 1, 3, and 4 rated the feature from Q10 (Figure 4.8a) with a score of 4. Interviewee 1 noted that distinguishing between the different issue types could be helpful, and Interviewee 3 felt that it could be helpful to see how the groups have worked in software engineering courses. With a mean rating of 3.5, this visualization performed better than the other two visualizations from Q11 (Figure 4.8b, mean score: 2.75) and Q12 (Figure 4.8c, mean score: 3.25). According to Participant 3, the visualization from Q11 does not provide any additional information, and the same information can be obtained from the chart in Q10. The software engineers found the information in the visualization from Q12 interesting, but only Developer 4 knew how he would utilize it.

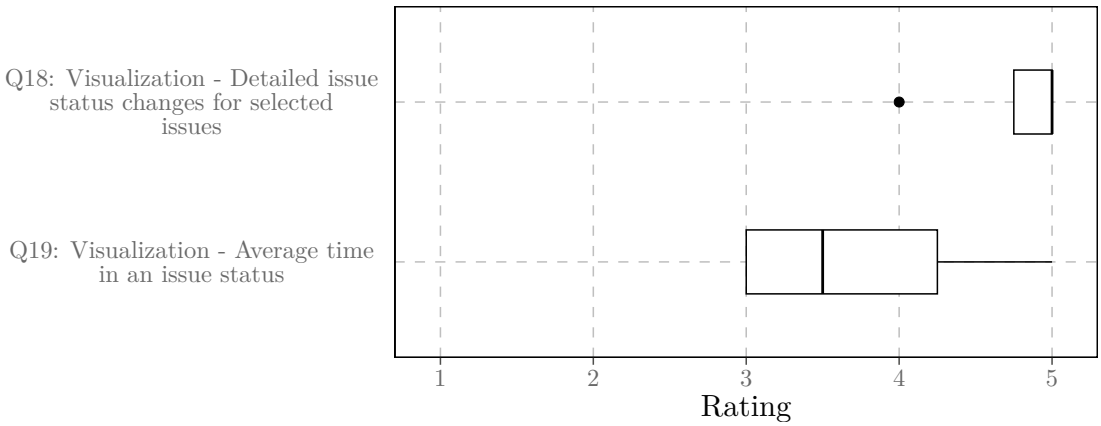


Figure 4.16: Score of issue status changes visualizations

Issue Distribution *(F-1)*

Q13-17 asked about the usefulness of visualizations that provide information on the local distribution of software issues within a project. The evaluation results are shown in Figure 4.15. The developers rated the visualization from Q13 (Figure 4.2) with a mean score of 4.25. All participants agreed that this chart could be useful, but both Interviewees 3 and 4 noted that the visualized information might be overwhelming in larger projects. Participant 1 suggested that information on whether an issue was referenced in multiple branches would also be useful. Among the graphs from Q14-Q16, which differ in the values shown on the x-axis, the chart of Q16 (Figure 4.3c) received the highest score, with a mean rating of 4.25. The interviewees criticized the charts from Q14 (Figure 4.3a, mean rating: 3.5) and Q15 (Figure 4.3b, mean rating: 3), noting that the values could be influenced by undesirable parameters. On the one hand, the commit behavior can vary greatly within a team, and on the other hand, the time elapsed between the commits does not necessarily reflect how long the issue was worked on. This is not the case with the visualization from Q16, though Interviewee 1 added that major refactoring changes should be filtered out as they could skew the results. The diagram in Figure 4.4 referred to in Q17 was rated a 3 by all participants except Participant 1. Here, they agreed that the information was better represented by the visualizations in the previous questions (Q14-Q16).

Issue Status Changes *(F-2)*

The next two questions, Q18 (Figure 4.5a) and Q19 (Figure 4.5b), focused on the visualizations that provide users with information about the status changes of issues. The rating results of these functionalities can be seen in Figure 4.16. All respondents except Participant 4, who gave it the rating 4, deemed the chart from Q18 as *Very Useful*. Participant 1 and Participant 3 added that this could address the issue raised in Q9

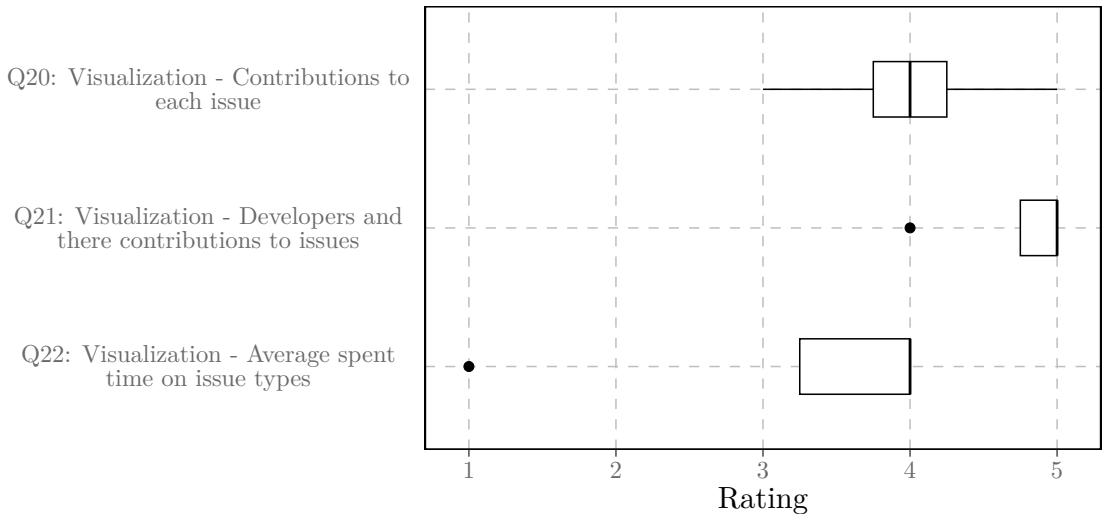


Figure 4.17: Score of issue contributions visualizations

regarding the historical analysis. Interviewee 1 also noted that this visualization could provide insights into the issue tracking behavior of a project group and how it evolved over the project's duration. Regarding Q19, Participant 1 felt that the illustration was too coarse, and prefers to view information for individual issues. However, Interviewee 4, gave this visualization a score of 5, highlighting that confidence intervals could provide additional valuable information. Q18 received a mean rating of 4.75, which was one point higher than Q19.

Issue Contributions *(F-3, F-4)*

The next three questions focused on the purposefulness of the visualizations, providing insights into software issues and developers' involvement. The results for these questions are shown in Figure 4.17. Q21 (Figure 4.7a) received the highest mean score of 4.75. The interviewees found this illustration particularly helpful in an educational context for assessing various groups. Participant 1 suggested that such a visualization could be helpful with other metrics, such as the number of commits or changes. The chart in Q20 (Figure 4.6) received a mean score of 4. Developer 3 mentioned concerns about the lines, as they could suggest connections between users that do not exist. Q22, focusing on the visualization shown in Figure 4.7b received the lowest mean score with 3.5. Participants noted distortions in reality, such as one contributor working on many issues briefly while another working on a few issues over a longer period of time. They further mentioned that the information could be skewed by too many developers in a project.

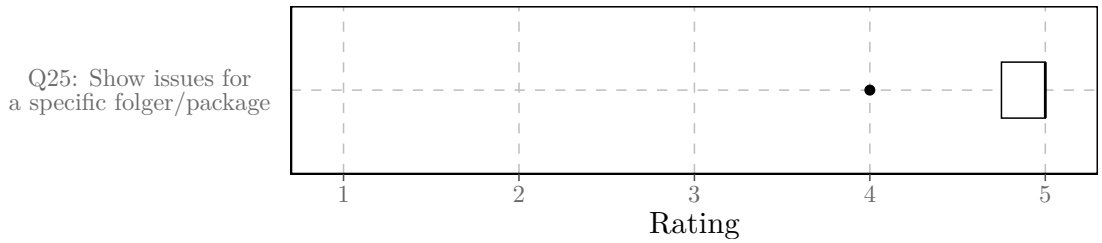


Figure 4.18: Score of folder/package selection

Filters and Version Comparison *(F-6, F-7)*

The questions in this section focused on determining the most useful filters. All developers, except for Developer 2, rated the filter in Q25 (Figure 4.9b) as *Very useful* (Figure 4.18). All of them were of the opinion that such functionality is particularly necessary when it comes to analyzing large software repositories. The results of the other filters can be seen in Figure 4.19. The interviewees varied in their opinions on the filters. Participant 3 considered displaying a certain subset of issues (Q27) essential for large projects, while Participant 2 rated this as neutral. Filters from Q26 and Q28 received the highest rating of 4.75, deeming them crucial. Filters from Q29 and Q30 were noted to depend on the project culture, such as how issues are managed (reopened or recreated) and developers commit behavior. Regarding Q32, Participant 1 emphasized that the type of changes that matter must be taken into account.

Q33, focused on whether it makes sense to compare two different filtered versions of a project (Figure 4.20). For this question, the opinions of the interviewees varied. Participants 2 and 3 found such a feature to be very useful. On the other hand, Participant 1 mentioned that while this view is especially important in management, comparing two versions that differs too much makes no sense.

Additional remarks

Finally, the respondents had the opportunity to add remarks on the provided visualization or suggest additional features that could be helpful for the historical analysis of software issues. Comments relevant to previous sections (Section 4.3.3 - 4.3.3) have already been incorporated there. Participant 3 mentioned that including information about Continuous Integration/Continuous Deployment (CI/CD), a paradigm aimed at automating the build process and enabling shorter release cycles [77], could provide additional value. Participant 4 suggested that a visualization similar to the one in Figure 4.2 could also be interesting for different programming languages. Regarding additional filters, every participant who answered this question expressed a desire for the ability to filter by specific contributors or by the number of contributors.

After the interviews were completed, the participants' answers were transferred to a spreadsheet, and the mean score for the answers was calculated. The values were used as

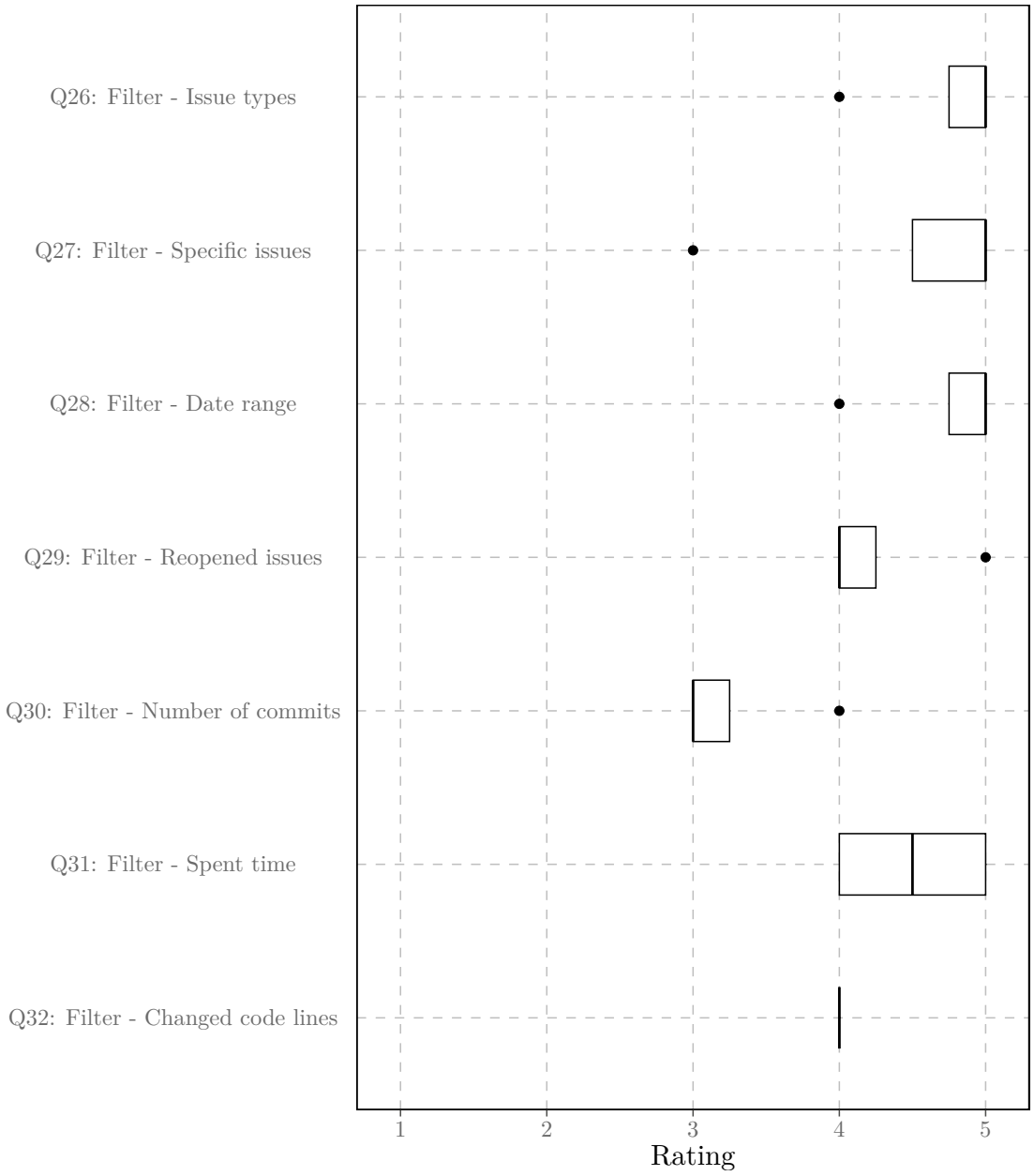


Figure 4.19: Score of filters

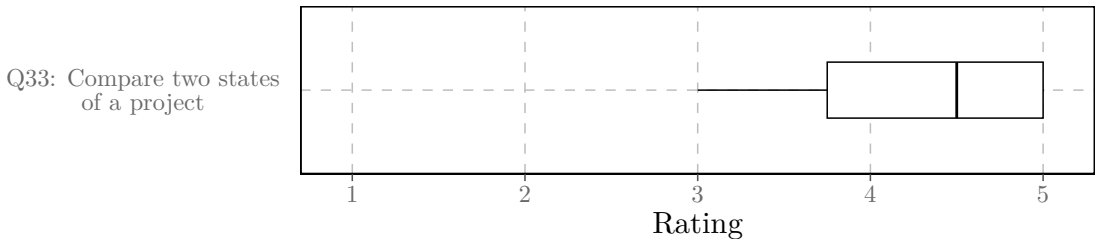


Figure 4.20: Score of compare functionality

a guide to prioritize the development of the proposed features and can be found in Table 4.2 and Table 4.3. It was decided to implement all visualizations and filters with a rating of 4

Visualizations	**Mean Value**
Q18 - Status changes	4.75
Q21 - Contributions per individual	4.75
Q13 - Distribution and time of software issues	4.25
Q16 - Distribution of software issues - lines of code	4.25
Q20 - Contributions to software issues	4
Q19 - Average issue status duration	3.75
Q10 - Number of software issues	3.5
Q14 - Distribution of software issues - commits	3.5
Q12 - Software issues per weekday	3.25
Q17 - Average commits, time and lines of code	3.25
Q22 - Average contributions	3.25
Q15 - Distribution of software issues - time	3
Q11 - Cumulative number of software issues	2.75

Table 4.2: Visualizations ranked by mean value

4.3.4 Threats to Validity

This section describes a set of threats that could affect the validity of the proposed visualizations and filters.

Number of Participants

The first threat to validity is the number of participants selected for the semi-structured expert interviews. Four people participated in the interviews. This sample may be too small to generalize the quantitative results of this interview, as this would normally require a larger number of randomly selected participants [6].

Filters	Mean Value
Q18 - Folder structure	4.75
Q21 - Software issue type	4.75
Q13 - Software issue selection	4.75
Q16 - Date range	4.5
Q20 - Spent time	4.5
Q19 - Reopened software issues	4.25
Q10 - Comparison	4.25
Q14 - Lines of code changed	4
Q12 - Number of commits	3.25

Table 4.3: Filters ranked by mean value

Participant Selection

Next, the selection of participants could be a potential threat to validity. For the interviews, all participants were members of the staff team of SE PR and ASE. Since the visualizations to be developed will be evaluated using data from these courses, the selection of participants was limited. This led to a selection bias and should be taken into account in a possible future extension to other courses. Furthermore, the demographics of the selected participants could be a threat as they are all in the same age range and from the same gender.

Questionnaire

Lastly, the questions asked during the interview may have been misunderstood. Although each participant had the opportunity to ask questions in case of misunderstanding, they could have given random ratings. This would not have been noticed because of the predetermined set of answers. By using mock-ups, it is also possible that visualizations rather than the usefulness of the features were rated.

4.4 Requirements

The following section describes the list of functional and non-functional requirements created for the software issue visualizations. These requirements were constructed based on the deficiencies of existing work and the results of the interviews from Section 4.3. They served as the basis for the further steps of this thesis. The requirements were used to develop the prototype and also served as the foundation for deriving the scenarios used to evaluate the implementation in Chapter 6.

Issue Distribution *(R-1)*

The prototype must assign each software issue to a local area (e.g., file, folder, package) based on the commits referenced in the issue. Additionally, it must be possible to see how

many changes (lines of code added and removed) are made by a commit, and hence how many changes are made by an issue in an area. It should also provide insight into the processing time of an issue based on the time passed between the first and last referenced commit. This should clarify which areas of the project are subject to frequent changes.

Issue Status Changes *(R-2)*

The prototype must visualize each status change of each software issue. In addition, the visualization should provide insight into the time of change. These changes should then reflect the life cycle of the issues.

Issue Contribution *(R-3)*

The prototype must assign each software issue to project members based on the involvement referenced in the issue. The visualization should also show which issues a developer was involved in, and how much time the developer spent on it, providing an overview of the distribution of work within the project team.

Issue Details *(R-4)*

The prototype must be able to provide additional detailed information for all visualizations. Furthermore, selected configuration values and identification features, such as colors, should be clarified using a legend.

Issue Filtering *(R-5)*

The prototype must be able to filter the software issues displayed by the visualizations according to different criteria, and thus adapt the visualized result to the user's needs.

Version Comparison *(R-6)*

The prototype must be able to compare two filtered versions of a software project, allowing users to see the differences between two versions and providing a foundation for trend analysis.

Historical information *(R-7)*

The prototype must display data from the past to the current state for visualizations of issue distribution, issue status changes, and issue contribution to show the evolution of a project.

CHAPTER 5

Implementation

This chapter describes the implementation of the visualizations. Section 5.1 first outlines the layers of the visualization tool and the architecture used to implement these layers. Section 5.2 and 5.3 explain the process used to retrieve the necessary data from the repositories of VCS and ITS. Finally, Section 5.4 describes the experiences made in developing the different views for analyzing software issues.

5.1 Architecture

This section provides an overview of the basic components of the proof of concept and their relationships. It also describes which technologies used to implement these components. As shown in figure 5.1, the prototype consists of three layers: the Extraction Layer (EL), the Processing Layer (PL), and the Visualization Layer (VL). The EL and PL are responsible for the data preparation, divided into two layers to provide data from both archived and server-hosted GitLab[1] software repositories, while the VL displays the data using the validated illustrations from Chapter 4.

5.1.1 Extraction Layer

The task of EL is to extract the data required for visualization from archived software repositories and make it available to the PL via a REST interface. Care was taken to ensure that the interface is similar to the REST endpoints of GitLab[2], so that both archived repositories can be visualized using EL and server-hosted repositories without EL. Furthermore, it was ensured that the data of a project is stored in a separate database, isolated from the data of other projects. The following technologies have been used to implement this layer:

[1] `https://about.gitlab.com/`, Accessed: *23.08.2024*
[2] `https://docs.gitlab.com/ee/api/rest/`, Accessed: *23.08.2024*

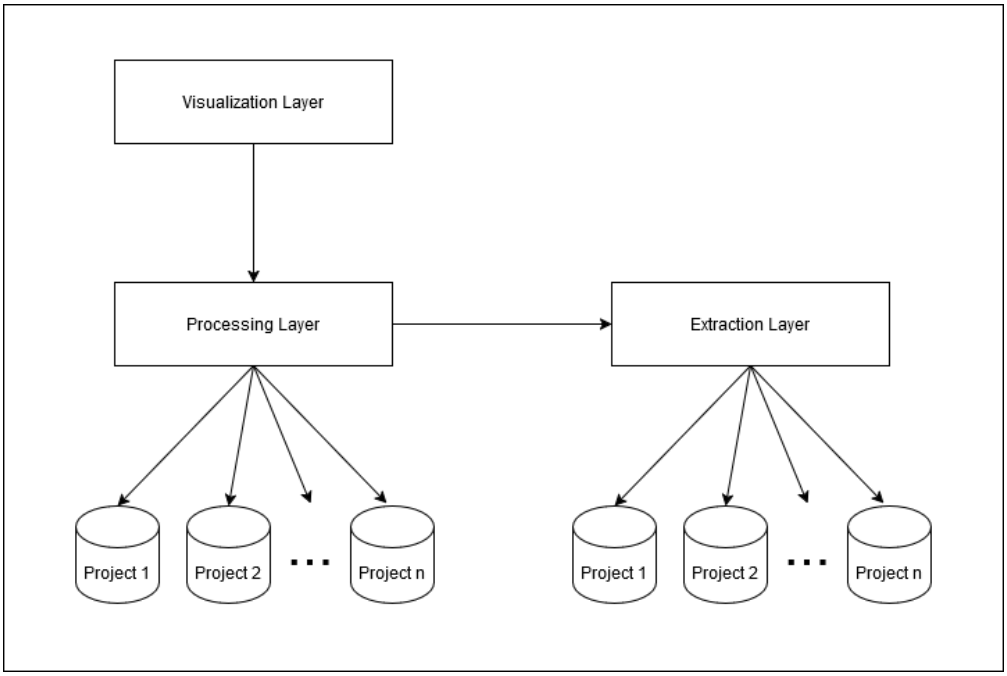


Figure 5.1: Architecture used to visualize software issues

- *Kotlin*[3]/Java[4], which was used as the main programming language in the backend layers to minimize the need for boilerplate code compared to Java[3].
- *Maven*[5], which was used as a build and dependency management tool.
- *Spring Boot*[6], which was used to develop the RESTful services in the backend. More precisely, the packages *Spring Boot Starter Data JPA*[7] and *Spring Boot Starter Web*[8] were used for this implementation.
- *SQLite*[9], which was used as a file based Structured Query Language (SQL) database. To use SQLite[9] with Hibernate, the packages *SQLite JDBC*[10] and *Hibernate Community Dialects*[11] were necessary.

[3] https://kotlinlang.org/, Accessed: *23.08.2024*
[4] https://www.java.com/en/, Accessed: *23.08.2024*
[5] https://maven.apache.org/, Accessed: *23.08.2024*
[6] https://spring.io/projects/spring-boot, Accessed: *23.08.2024*
[7] https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-data-jpa, Accessed: *23.08.2024*
[8] https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-web, Accessed: *23.08.2024*
[9] https://www.sqlite.org/, Accessed: *23.08.2024*
[10] https://github.com/xerial/sqlite-jdbc, Accessed: *23.08.2024*
[11] https://mvnrepository.com/artifact/org.hibernate.orm/hibernate-community-dialects, Accessed: *23.08.2024*

- *FasterXML/jackson*[12] which was used for serializing Java objects to JavaScript Object Notation (JSON) and vice versa.

5.1.2 Processing Layer

The PL is responsible for fetching the data from a software repository using the provided REST interface. The data is processed, related and made available to the VL via another REST interface. As with the EL, project data is stored in a separate database, and as both have similar functionality, the same technology stack as described in Section 5.1.1 was used for this layer.

5.1.3 Visualization Layer

The task of VL is to read the pre-processed data from PL and forward it to the charts for visualization. Additionally, this layer provides various filters to adapt the results to the user's needs. The following technology stack was used to implement this layer:

- *TypeScript*[13] which was used as the main programming language in the frontend to add type support to JavaScript.
- *Next.js*[14], which allows for building web applications quickly and easily.
- *Recharts*[15], which were used to build the illustrations.
- *NPM*[16], which was used to manage the packages needed for this layer.
- *MUI*[17], which contains a wide range of *Material UI*[18] components used for user input of any form.

5.2 Extraction Layer

The EL is responsible for selectively retrieving data from archived software repositories and making it available to the PL via a REST interface. The special characteristic of this interface is its similarity to the REST interfaces defined by GitLab[1]. This makes it possible to analyze repositories that are hosted on a server in addition to archived repositories by replacing the EL with the active repository. Section 5.2.1 first examines the data and its structure, which is mined by EL. The components of this layer are then discussed in Section 5.2.2 - Section 5.2.4.

[12] `https://github.com/FasterXML/jackson`, Accessed: *23.08.2024*
[13] `https://www.typescriptlang.org/`, Accessed: *23.08.2024*
[14] `https://nextjs.org/`, Accessed: *23.08.2024*
[15] `https://recharts.org/en-US/`, Accessed: *23.08.2024*
[16] `https://www.npmjs.com/`, Accessed: *23.08.2024*
[17] `https://mui.com/`, Accessed: *23.08.2024*
[18] `https://m3.material.io/`, Accessed: *23.08.2024*

Figure 5.2: Structure of the data

5.2.1 Data Structure

As a starting point, a couple of archived student projects from the course SE PR were provided, and in a first step, these were analyzed. Each project consisted of two parts: a `export` archive and a `repo` folder. The former contained all the information about the software repository such as software issues, project members, milestones or general project information, which was created by GitLab's[1] export functionality[19]. The `repo` folder, on the other hand, provided, besides the project's source code, Git-related data such as commits and branches.

Next, the `export` archive was decompressed so that the extracted archive and the `repo` folder were the root level of the project. Figure 5.2 shows the directory structure and the files contained in the directory. It should be noted that `export` contains many more files, but only those relevant to the implementation of the visualization are shown. This includes the files `issues.ndjson`, `project_members.ndjson` and `milestones.ndjson`. All of these were available in the Newline Delimited JavaScript Object Notation (NDJSON) format, a format in which each line of the NDJSON file is mapped to an object in an array. For this reason, in this section, they are presented in a prettified form for better readability.

Issues

All tracked software issue tickets from a project are stored in the `issues.ndjson` file, with each line of this file corresponding to one software issue. An issue ticket object

[19] `https://docs.gitlab.com/ee/user/project/settings/import_export.html`, Accessed: *23.08.2024*

contains numerous pieces of information, some of which are particularly relevant for the implementation of the prototype. Listing 5.1 provides an overview of what such a JSON object looks like. Besides title, issue identifier (`iid`) and assigned milestone, each issue stores the time of creation (`created_at`) and closing (`closed_at`). The `notes` object provides rich data, as all events created in connection with software issues are stored here. This includes all commits where the issue identifier was referenced in the commit message, as well as the tracked time for an issue ticket. Furthermore, any changes made to the issue, such as changing the issue title, are also stored here. For each of these entries, the user who made the change and the timestamp at which the change was made are saved.

```
{
  ...
  "title": "User Story ...",
  "created_at": "2020-04-21T11:43:55.496Z",
  "iid": 3,
  "closed_at": "2020-05-08T...",
  "milestone": {
    "title": "M1"
  },
  "notes": [
    {
      "note": "mentioned in commit a...",
      "author_id": 468,
      "created_at": "2020-05-05T...",
    },
    {
      "note": "added 3h of time spent",
    },
    {
      "note": "changed title ...",
    },
  ...
  ]
}
```

Listing 5.1: Example of an exported issue object

Two areas where GitLab[1] lacks functionality are identifying the type of issues and tracking issue status changes. In order to determine the type of issue, it was decided to search the description and the title for defined keywords such as *Bug* or *Defect*, or to determine the commit behavior (e.g., if there have been no commits, then it is most likely an organizational issue). For the issue status changes, the `resource_label_events` object in the `issues` file, as shown in Listing 5.2, provides valuable information. During

the analysis of the projects, it was recognized that the students used these labels to mark the phase a software issue was in. Each addition and removal of a label is stored, along with the developer who made the change and the timestamp of when it was added or removed.

```
{
  "resource_label_events": [
    {
      "action": "add",
      "created_at": "2020-04-27T...",
      "label": {
        "title": "In Progress",
      }
    },
    {
      "action": "remove",
      "created_at": "2020-05-04T...",
      "label": {
        "title": "In Progress",
      }
    }
  ]
}
```

Listing 5.2: Example labels to indicate status changes

Project Members and Milestones

Information about project members can be found in `project_members.ndjson`. It lists all users assigned to a GitLab[1] project, including their email, username, and the time they were added to the project. An example of such an entry is shown in Listing 5.3. Additionally, all milestones of a project are stored in `milestones.ndjson`, as shown in Listing 5.4. Besides the title and the description of the milestone, it also contains the date (`due_date`) until when the milestone was planned.

```
{
  "created_at": "2020-04-22T...",
  "user": {
    "email": "student1@student.tuwien...",
    "username": "student1"
  }
}
```

Listing 5.3: Example of an exported project member

```
{
  "title": "M1",
  "description": "Issue description",
  "due_date": "2020-05-07",
  ...
}
```

Listing 5.4: Example of an exported milestone

Commits

The repo folder can be used to obtain commits referenced in the issues in Listing 5.1. In addition to the source code of the software, the folder also contains a `.git` directory, which can be used to query Git-related data. The project's detailed information about the commits can be extracted from the repository using the command `git log -p -all`. Relevant information includes the commit hash used to identify a commit, the commit message, the source files changed as part of the commit, and the changes themselves, indicated by a plus (+) for added lines and a minus (−) for removed lines. An example of such a commit can be seen in Listing 5.5.

```
!!commit d0bd6483a93eeb67b...!!
...
--- a/.../Component.ts
+++ b/.../Component.ts
@@ -83,7 +83,7 @@ export class Component ...
<-  const example = ...>
$+  let example = ...$
    ...
@@ -144,7 +144,7 @@ export class Component ...
<-  const example2 = ...>
$+  let example2 = ...$
    ...
```

Listing 5.5: Example of extracted commit

Files and Folders

As a final source of information, all the source files and the folders in which they are located can be found with the command `git ls-files`, which prints out the path and name of every file in the Git repository. This is needed in order to assign the location of the changed files to the issues. An example of the result of this command can be seen in Listing 5.6.

```
backend/mvnw.cmd
backend/pom.xml
backend/src/main/.../BackendApplication.java
...
frontend/src/app/app.module.ts
frontend/src/app/.../component1.css
frontend/src/app/.../component1.html
...
```

Listing 5.6: Example of extracted files and folders

5.2.2 Routing and Data Source

To ensure that the data of one repository is isolated from the data of other repositories, a multi-tenant approach was chosen for the EL. All requests reach the same application, which then selects the correct database based on a value set in the request and executes the operation [76]. To implement this, the data was stored in a file-based SQLite database, and routing to the appropriate database was enabled using Spring's

`AbstractRoutingDataSource`[20]. The latter is a Java interface that can dynamically determine the actual database in which the data persists. To distinguish the respective databases, the project names of the exported software repositories were used. The Java mechanism `ThreadLocal`[21] is used to ensure that REST requests, bound to a thread, actually address the correct database by making sure that each thread accesses its own copied instance of a variable. Together, these two components form the routing mechanism of this application.

Listing 5.7 provides an idea of how the routing was implemented. Two classes were created: `DBManager` and `DBInterceptor`. The first class provides the `ThreadLocal` variable, stores the keys to the data sources, and is also responsible for managing the databases. New databases can be added using a method that adds a data source to a `ConcurrentHashMap`. Additionally, it was declared as a `Bean`, an object managed by the Spring Inversion of Control (IoC) container, so that it can be used to select the data source. Therefore, the `determineCurrentLookupKey` method has to be implemented. When a request reaches the EL, it must be ensured that the appropriate key is assigned to the `ThreadLocal`. This task is handled by the `DBInterceptor`, a class derived from `HandlerInterceptor`[22], which reads the project name from the request path and assigns it to the `ThreadLocal`. Spring will then ensure that the operation reaches the correct database.

[20] `https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/datasource/lookup/AbstractRoutingDataSource.html`, Accessed: *23.08.2024*

[21] `https://docs.oracle.com/javase/8/docs/api/java/lang/ThreadLocal.html`, Accessed: *23.08.2024*

[22] `https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/HandlerInterceptor.html`, Accessed: *23.08.2024*

```
class DBManager {
  val context = ThreadLocal<String>()
  val dataSources: MutableMap<Any, Any> = ConcurrentHashMap()

  @Bean
  fun dataSource(): DataSource {
    sourceRouting = object : AbstractRoutingDataSource() {
      override fun determineCurrentLookupKey(): Any {
        return context.get()
      }
    }.apply{setTargetDataSources(dataSources) ... }

    return sourceRouting
  }
  fun addDB(name: String) { ... }
  // Addidional methods
  ...
}

class DBInterceptor(db: DBManager): HandlerInterceptor {
  override fun preHandle(...) {
    ...
    db.context.set(groupeId)
  }
}
```

Listing 5.7: Database routing with ThreadLocal and AbstractRoutingDataSource

5.2.3 Initialization

The initializers are responsible for populating the databases on startup of the application. As mentioned in Section 5.2.1, the data for a project is extracted from various sources: the exported NDJSON files for `issues`, `project_members` and `milestones`, and the Git commands to gather the commit, folder and file information. Five initializers have been developed to perform this task and are executed when the layer is started. In Spring, this behavior can be implemented using the `CommandLineRunner`[23] interface. This `Bean` is called when the application context is loaded and is used for tasks that should be executed when the application starts, such as initializing databases in this case.

Listing 5.8 shows the structure of the implemented `CommandLineRunner`. It takes the path to the root directory containing the exported repositories from a configurable

[23] `https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/CommandLineRunner.html`, Accessed: *23.08.2024*

variable in the `application.yml` file. After checking whether the folder for a project has the same structure as shown in Figure 5.2, a new thread is started to initialize the data. If no database exists for this project, it is added to the available data sources using the `DBManager` mentioned in Section 5.2.2 and the initialization processes for issues, project members, milestones, commits, folders and files are started. By binding the data source information to the thread, this process can be run in parallel.

```
1  @Bean
2  fun runner(
3    ... // injection of initializers and dbManager
4  ): CommandLineRunner {
5
6    return CommandLineRunner {
7      val path = ... // get configured path
8      // check whether structure of the data is valid
9
10     Thread {
11       dbManager.addDB(it.fileName.toString())
12       issueInitializer.initIssues(path)
13       fileInitializer.initFiles(path)
14       userInitializer.initUsers(path)
15       commitInitializer.initCommits(path)
16     }.start()
17   }
18 }
```

Listing 5.8: Intitialization of database

Issue, Project Member and Milestone Initializer

The information about software issues, team members and milestones of a software repository are provided by the corresponding NDJSON files. FasterXML was used to extract this data. The class `ObjectMapper`[24] from this library was used to convert the JSON objects into a list of Plain Old Java Object (POJO). Kotlin's Data Classes[25] were used to build these POJO. To ensure seamless conversion, it was necessary that the Data Classes have the same structure as the exported `issues`, `project_members`, and `milestones` files. The modules `KotlinModule`[26] and `JavaTimeModule`[27], had to

[24] `https://fasterxml.github.io/jackson-databind/javadoc/2.7/com/fasterxml/jackson/databind/ObjectMapper.html`, Accessed: *23.08.2024*

[25] `https://kotlinlang.org/docs/data-classes.html`, Accessed: *23.08.2024*

[26] `https://javadoc.io/doc/com.fasterxml.jackson.module/jackson-module-kotlin/2.12.2/index.html`, Accessed: *23.08.2024*

[27] `https://fasterxml.github.io/jackson-modules-java8/javadoc/datetime/2.9/com/fasterxml/jackson/datatype/jsr310/JavaTimeModule.html`, Accessed: *23.08.2024*

be registered to allow the `ObjectMapper` to correctly use the Data Classes and convert the timestamps. An example of this can be seen in Listing 5.9.

```
1  @Bean
2  fun objectMapper(): ObjectMapper =
3    ObjectMapper()
4      .registerModule(KotlinModule.Builder().build())
5      .registerModule(JavaTimeModule())
6
7  fun init(
8    path: String,
9    objectMapper: ObjectMapper
10 ) {
11
12   objectMapper
13     .readerFor(Dto::class.java)
14     .readValues<Dto>(File("$path/...")) // Path to file
15     .also {
16       // store data
17     }
18 }
```

Listing 5.9: NDJSON file converter

Commit, Folder and File Initializer

Since the commit, file and folder information was not available in NDJSON format, a different approach had to be taken for the respective initializers to extract this data. Therefore, the class `ProcessBuilder`[28] (Listing 5.10) was used to execute the Git-related commands, mentioned in Section 5.2.1 and 5.2.1. The results of these commands were then iterated line by line. For the commits, the corresponding commit information was extracted using defined regex patterns (e.g., to determine the commit author). Care had to be taken in distinguishing the commits, as the commit's information was completed with the hash of a subsequent commit. This was not needed for the files, as each line of the result represented a separate file. However, the necessary information about the folders in which the files were located had to be extracted from the file paths. Additionally, files and folders were given different flags to distinguish them from each other and an identifier entry pointing to the parent node.

[28] `https://docs.oracle.com/javase/8/docs/api/java/lang/ProcessBuilder.html`, Accessed: *23.08.2024*

```
1 fun init(path: String) {
2
3   ProcessBuilder("git", "ls-files")
4     .apply { directory(File("$path/repo"))}
5     .start().inputStream().bufferedReader().useLines {
6       // Process results
7     }
8 }
```

Listing 5.10: Git-related command execution

5.2.4 REST Interfaces

As already mentioned in the beginning of this chapter, the goal of the EL was to provide an archived software repository with an interface in such a way, that the EL can be swapped with an active hosted repository. Since various information about GitLab[1] repositories can be queried via REST endpoints, it was decided to implement interfaces that are similar to GitLab's[1]. Spring's `RestController`[29] was used to create these interfaces, which can be used to create restful web services with little configuration. The entry point of the following eight operations were defined using `GetMapping`[30]:

- `GET /repository/tree`: Returns all folders and files in a Git repository. All child nodes are connected to the parent node by the parent identifier.
- `GET /users`: Returns all users of a project.
- `GET /milestones`: Returns all milestones of a project.
- `GET /repository/commits`: Returns all commits of a repository. This includes the commit message, author, and time of the commit. Changes introduced by a commit are not listed here.
- `GET /repository/commits/<commit_hash>/diff`: Returns the changes made by a commit. The hash of the commit can be retrieved by the previous REST request.
- `GET /issues`: Returns all tracked issues of a project.
- `GET /issues/<issue_id>/notes`: Returns all notes associated with an issue. These notes include the commits and the time tracked referenced to a software issue.

[29] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html, Accessed: *23.08.2024*

[30] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/GetMapping.html, Accessed: *23.08.2024*

- `GET /issues/<issue_id>/resource_label_events`: Returns all resource labels for a software issue.

5.3 Processing Layer

The task of the PL is to fetch the data from software repositories, process the information, and establish relationships among the data. This layer accesses the data through the REST endpoints provided by GitLab[1] repositories. These interfaces offer a variety of information, which had to be tailored to the needs of the visualizations of this thesis. The development of this layer and the VL builds upon the EL due to the available archived repositories from the course SE PR. However, since EL builds an identical REST interface as GitLab[1], any publicly accessible GitLab[1] repository could be visualized. Theoretically, the extraction layer could also be replaced by another mining backend that extracts data from other ITS. The PL should still be able to prepare the data for visualization, as long as it can access the data similarly.

Like the EL, this layer was implemented using Spring. The application uses a multitenant approach, where data from each repository is stored in its own database and all data sources can be reached through the same application. PL is divided into three layers: a database and routing mechanism, several initializers that gather and process the data, and various REST endpoints that pass the processed data to the visualizations. Besides the possibility to configure the username, passwords, and the location for the databases, users can also configure custom issue states for which the application will search. For these configurations, a `application.yml` file was provided.

5.3.1 Routing and Initialization

As with EL and as described in more detail in Section 5.2.2, the routing of this service was implemented using Spring's `AbstractRoutingDataSource`, the Java mechanism `ThreadLocal` and an interceptor were used. The interceptor assigns a value sent with the REST request to the `ThreadLocal`. This ensures that the correct data source is determined when the operation is applied. SQLite was chosen as the database engine for PL. However, the key for routing in this layer is no longer the repository name, but the base URL of the project where the information is available.

Five initializers were developed to retrieve the project information from the endpoints described in Section 5.2.4. In order to access the data, Spring's `RestTemplate`[31], a client for sending Hypertext Transfer Protocol (HTTP) requests, was used. Kotlin's Data Classes were also used to convert the responses from JSON format into POJOs. Listing 5.11 provides a snippet of how this was implemented.

[31] `https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html`, Accessed: *23.08.2024*

```
fun init(
  url: String,
  restTemplate: RestTemplate,
) {

  restTemplate.getForObject(
    url,
    Array<Dto>::class.java
  )?.forEach {
    // store data
  }
}
```

Listing 5.11: Data gathering of PL

In contrast to EL, in PL the initializers for populating the database are not executed automatically when the application is started. In this layer, the initializers are triggered by the following REST command:

- `POST /initRepo`: Which expects the repository URL and starts the initialization of the data sources.

File, Folder, Milestone and User Initializer

The initializers for files, folders, milestones, and users gather information required by the other initializers but do not reference other information themselves, which is why these must be run at the beginning. To achieve this, the corresponding REST endpoints are called. The `FileInitializer` iterates over the results and checks for each file/folder whether the parent folder has already been saved. If this is the case, the identifier of the parent folder is referenced to the current file/folder; otherwise, the folder is at root level and no folder is referenced. This approach is feasible, due to the nature of the data, no child node will be processed until the parent node has been processed. The `UserInitializer` and the `MilestoneInitializer` simply retrieve their information and store it in the database, with the former filtering out all users whose username is not a matriculation number (a seven-digit unique identification number for students).

Commit and Issue Initializer

The initializers for fetching commits and issues have to be launched after the aforementioned initializers. The `CommitInitializer` first fetches all of a project's commits, takes each commit hash, and then calls the appropriate endpoint that returns the changes to a particular commit. These changes are returned as a list, where changes to a file are grouped into the same list entry. For each change, a database record is created that

references the changed file, including the number of lines of code added and removed. Each commit then references its changes and contains the time at which the commit were made.

The last initializer retrieves all the issues, extracts the issue identifier, and then requests the `notes` and `resource_label_events` from their corresponding endpoints. The information in the notes is used to find the commits that reference the issue in the commit message. The commits found are then associated with the appropriate issue and user. As the commits do not have a user record at this point, they are updated with the information from the issue ticket. The tracked time for each user and other changes made to the issue can also be extracted from the notes. Reopened issues can be recognized by the fact that they have an entry in notes that indicates that the issue has been opened. This is not the case for software issues that were not reopened. Issue status can vary from project to project, but each issue ticket can be assigned an *Open* and *Closed* status, as information is available about when the issue was created and when it was closed. As already mentioned in Section 5.2.1, additional status can be read via the retrieved labels, which contain the times when a particular label was added to or removed from a software issue. Since these labels do not necessarily have to reflect the status of an issue, the labels that represent a status can be configured before the application is launched. Another piece of information that cannot be derived directly from the issue is the issue type. However, to distinguish between different types of issues, the title and description of each issue are searched for specific keywords. If they contain a keyword, the issue is labeled as a software defect. If no code changes were made by the issue, it is labeled as an organizational issue.

5.3.2 Data Model

Figure 5.3 provides an overview of the tables and their connections to each other, along with the cardinalities used to store the repository data:

- *IssueEntity*: This table provides access to all information related to software issues. In addition to the issue identifier, title, and creation date, each issue contains a list of commits, time logs, and status changes.
- *UserEntity*: This table stores all usernames and user identifiers.
- *MilestoneEntity*: This table stores all milestones and their start and end dates.
- *LocationEntity*: This table stores all files and folders, their paths, and an entry for the parent folder.
- *TimeEntity*: This table provides all the time logs that users have tracked for an issue.
- *ChangeEntity*: This table lists all changes made to an issue via `notes` and the user responsible for the change.

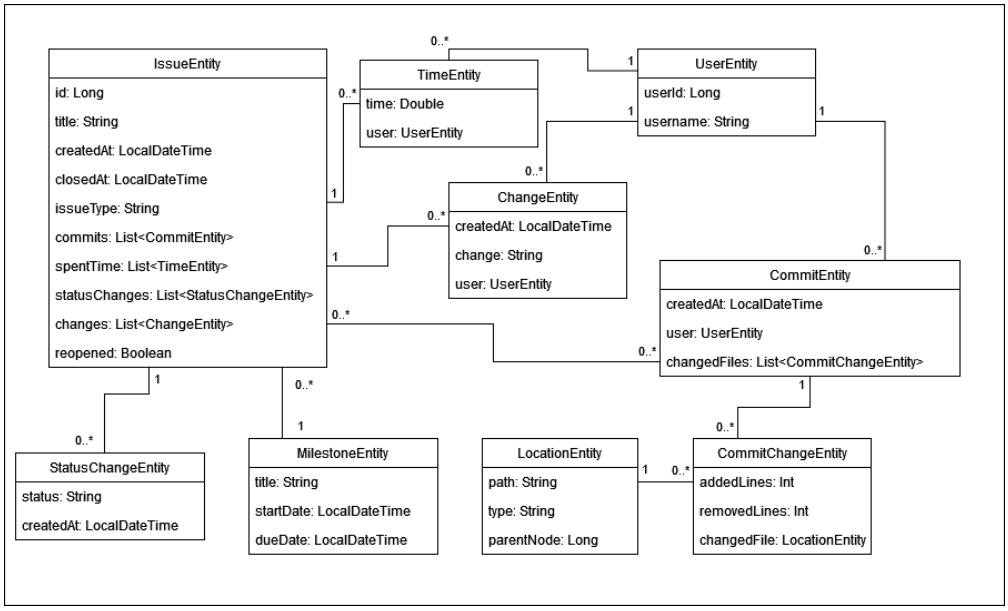


Figure 5.3: Data model of PL

- *StatusChangeEntity*: This table stores all issue status changes to a ticket and when they were made.
- *CommitEntity*: All commits are provided in this table, and a commit is associated with a user. It also stores the time the commit was made, and a list of the changes made by the commit.
- *CommitChangeEntity*: This table contains the number of lines of code added and removed by a commit for each file and commit.

5.4 Visualization Layer

This section describes the steps taken to create the visualization layer for the prototype. The conceptual ideas in Section 4.1 served as the basis for the development of the visualizations, although they were adapted when necessary due to technical limitations of the charting library. Figure 5.4 shows the structure of the VL. It consists of three parts. On the left side of the dashboard (Figure 5.4, Number 1), the user has a sidebar where the visualizations can be controlled with the following elements:

- Repository (Figure 5.5, Number 1): This is the initialization point for the software issue charts. The PL to process the data is started via the URL to a software repository that provides the raw data of a software repository using REST endpoints. When the initialization process is complete, the user is notified. Figure 5.5, Number 1 shows the URL to a project made queryable via the EL.

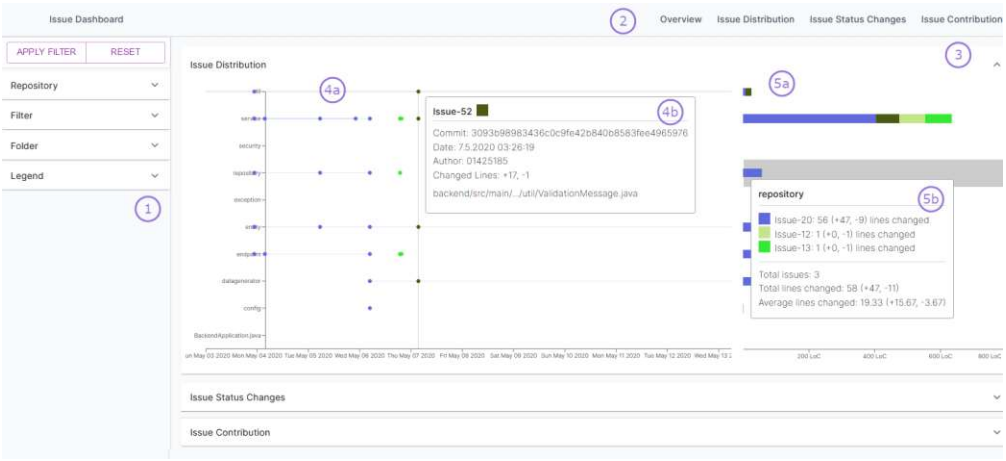


Figure 5.4: Distribution of issues

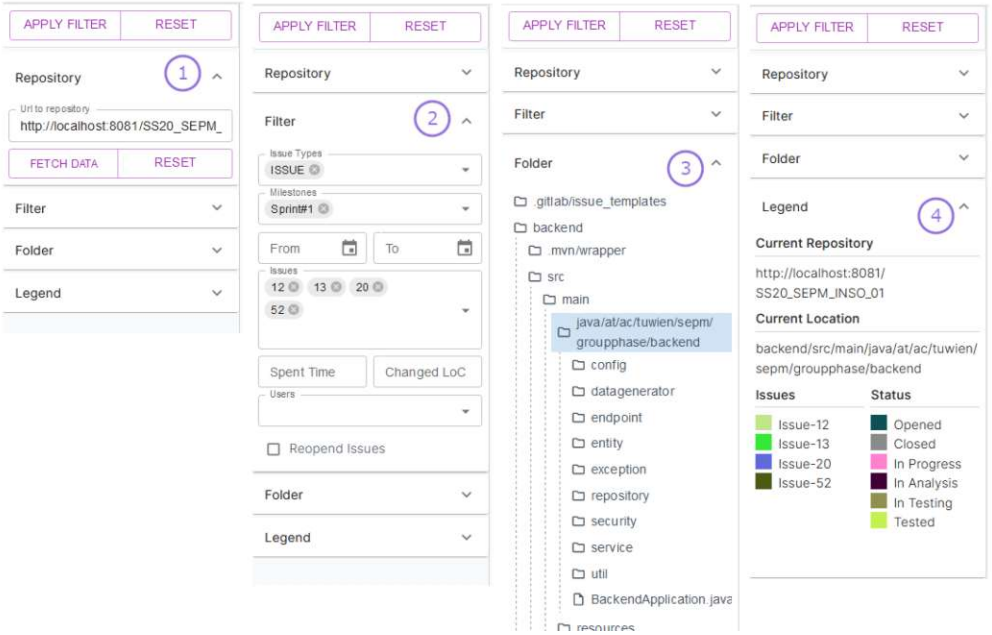


Figure 5.5: Information and control elements in the sidebar

- Filter (Figure 5.5, Number 2): In order to control the visualizations according to the user's needs and to narrow down the results, the user has several filter criteria available. Software issues can be filtered by issue type, referenced project milestones, a timeframe in which the issue was closed, issue identifiers, time tracked, number of lines of code changed, and contributors involved. Each time the user clicks the *Apply Filter* button, an object is updated with the values set by the filter criteria. The visualization component responds to this update by sending a HTTP request and passing the filter object to the PL. The visualization is then re-rendered based on the response received from the server.

- Folder (Figure 5.5, Number 3): The visualization result can be further restricted using an expandable tree view. If a directory is clicked on, only the issues that have caused changes in this folder are displayed. A directory in which there is only one other subdirectory is skipped to ensure better usability.

- Legend (Figure 5.5, Number 4): The legend provides the user with additional information about the visualizations. It shows the software repository from which the visualized data comes and the area of the software project that is being visualized. It also shows the corresponding color for all issues and for each issue status.

In the top bar (Figure 5.4, Number 2), the user can navigate to each visualization and compare two filtered versions. The remaining space of the VL (Figure 5.4, Number 3) is dedicated to the illustrations.

5.4.1 Issue Distribution

Figure 5.4 shows the final result of the illustration showing the distribution of software issues. It is based on the ideas of the visualizations in Figure 4.2 and Figure 4.3c. The graph consists of two sub-charts, a scatter chart (Figure 5.4, Number 4a) and a vertical stacked bar chart (Figure 5.4, Number 5a). On the y-axis, the folders/files of the current directory are listed and can be changed using the folder structure (Figure 5.5, Number 3). In contrast to the concept, the left subchart displays the timestamps on the x-axis, ensuring that an entry is created for each day and that it is at least 100 pixels wide, as it was difficult to keep the points apart in a project that extends over a longer period of time. As a result, the left partial chart can become very wide when displaying a long period of time, and the right partial chart can extend beyond the right edge of the screen. CSS[32] with the `overflow: auto`[33] property were used to prevent this behavior. However, this meant that the y-axis was only visible at the beginning of the chart. Since Recharts does not provide sticky axes that move with the scrollbar, this behavior had to be implemented in a roundabout way. An element can be moved using the CSS

[32] `https://developer.mozilla.org/en-US/docs/Web/CSS`, Accessed *23.08.2024*
[33] `https://developer.mozilla.org/en-US/docs/Web/CSS/overflow`, Accessed *23.08.2024*

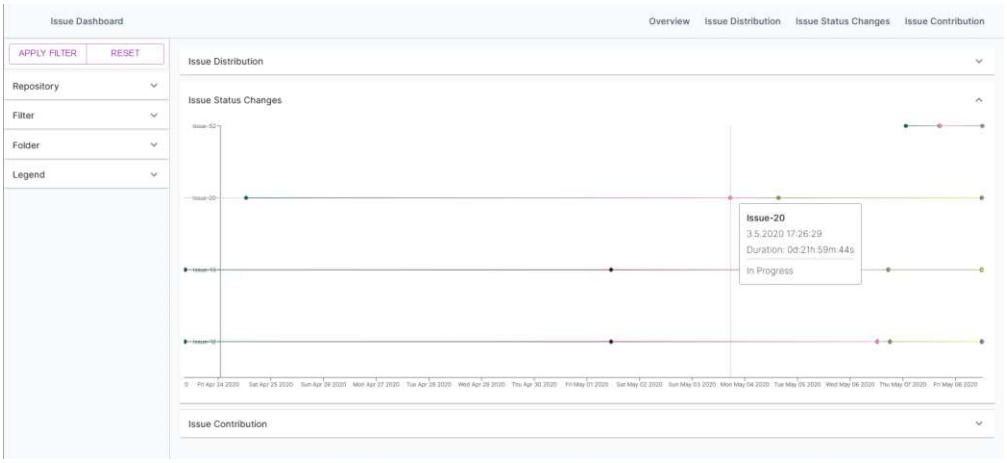


Figure 5.6: Issue status changes

function `transform: translate`[34], and the `element.scrollLeft`[35] property sets the number of pixels to scroll from the left edge of the element. The container's `scroll` event could then be used to move the axis using the aforementioned function and property.

As in the concept, the dots represent commits, with the color indicating which issue they belong to. While in the concept the first commit of a location was linked to the last commit of the closest location, this has been abandoned for better visibility. Compared to the concept, little has changed in the right subchart. The x-axis still shows the number of code changes, and changes to an issue are highlighted in color. However, the width of the bars has been adjusted so that they can be combined with the lines in the left sub-chart. Both the left and right subchart offer the option of clicking on the drawn entries to provide the viewer with additional information in the form of a tooltip. In the scatter plot (Figure 5.4, Number 5a), the hash, which clearly identifies the commit, the time and person who made the commit, the number of code changes, and the files that were changed are shown. The bar plot (Figure 5.4, Number 5b) lists all issues and the number of changes per issue that can be assigned to an area. It also adds up the changes to show the total and the average number of changes.

5.4.2 Issue Status Changes

Figure 5.6 shows the final visualization, which shows the status changes for a software issue. Again, the final result of the implementation phase differs only minimally from the idea in the concept shown in figure 4.5a due to the high rating in the interviews. The issues are shown on the y-axis and the time on the x-axis. The component assigns a height or width to each value on the axis based on the number of issues and project

[34] `https://developer.mozilla.org/en-US/docs/Web/CSS/transform-function/translate`, Accessed *23.08.2024*

[35] `https://developer.mozilla.org/en-US/docs/Web/API/Element/scrollLeft`, Accessed *23.08.2024*

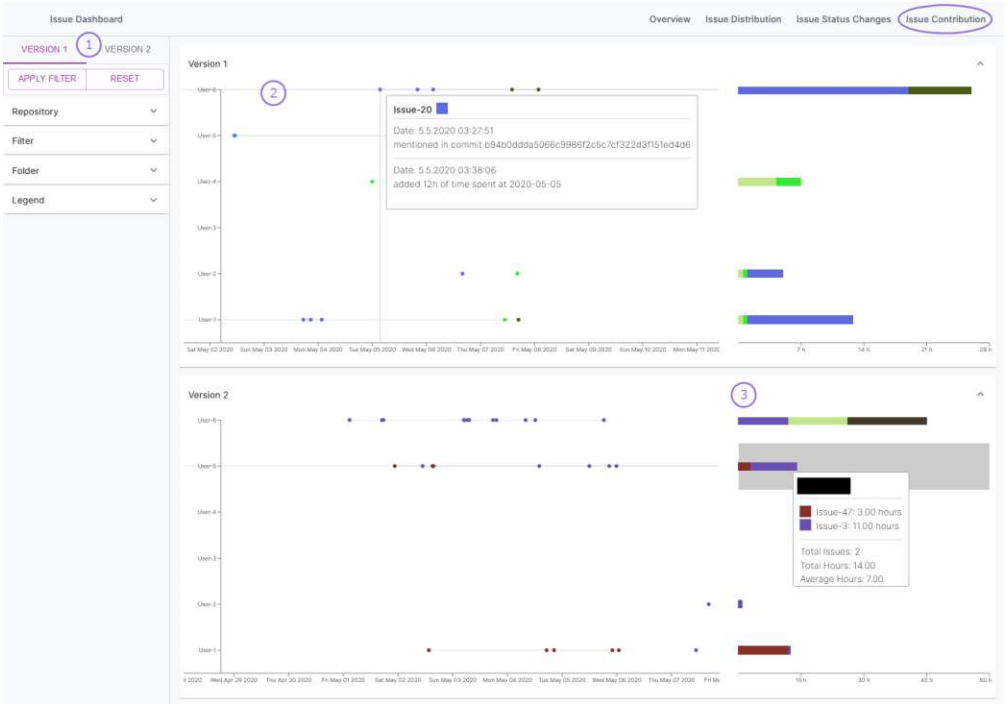


Figure 5.7: Issue contribution and compare functionality

duration, again ensuring that both the height and width of the chart do not exceed the assigned space. The tooltip for this chart provides additional information about an entry. The viewer is informed about the exact time when the status change occurred, the length of time the issue was in that status, and the status. Each status change is represented by a point, and if there is another state change, this point is connected to the next point so that the line represents the time the issue has been in an issue status.

A color gradient has been defined for drawing the line so that the line becomes more of the same color as it gets closer to a point, which makes the transition from one status to the next more smooth. This also has the pleasant side effect that several colors can be assigned to a line and its points, which would not be possible without a color gradient due to another limitation of the Recharts. The performance advantages over an implementation where the color change is realized by multiple lines, each representing an issue state, are obvious. However, when defining a color gradient for a chart in which the data differ by only one value (as with this chart, a straight line is drawn in which only the x-values change and the y-values remain the same), care must be taken to ensure that at least one value differs. Otherwise, the line will not be drawn. Therefore, during the rendering process, the y-value of the first entry in the data set was changed minimally (in this case, it was reduced by 0.001).

5.4.3 Issue Contribution

Figure 5.7 shows the final illustration of project team members and their contributions to each issue ticket. This is the combined implementation of the concept ideas from Figure 4.6 and Figure 4.7a. All users who contributed changes to software issues during the project's implementation are plotted on the y-axis. In the left part of the graph (Figure 5.7, Number 2), as in the previous visualizations, care has been taken to ensure that each entry on the x-axis has a minimum width and that the graph does not extend beyond the defined area for better visibility. The colored points associated with an issue correspond to each type of participation, but unlike the concept idea, only the dots of a team member are connected. The stacked bar chart (Figure 5.7, Number 3), which shows time in hours on the x-axis and where the width of a bar can be used to determine how much time a person has tracked for an issue, has two differences from the concept. Again, as with the bar graph to visualize the distribution, care was taken to ensure that the bars could be combined with the lines. Therefore, the width of the bars and the orientation of the chart had to be changed from horizontal to vertical.

Both graphs have tooltips that can be used to display detailed information about an entry. In the scatter plot, the exact time of the change and information about what changed is displayed for each point (in the example in Figure 5.7, Number 2, both a commit and a time entry for an issue were referenced). The bar chart, on the other hand, shows all issues and the cumulative time logged for each user. In addition, all of a person's hours are summed up to show the total time and average time per issue worked.

Figure 5.7 also illustrates the comparison functionality of the dashboard, which can be accessed for all visualizations via the top bar. In this view, the graph is rendered twice, one for *Version 1* and *Version 2*. Both project versions have their own sidebar that can be used to control the respective graphs, and the desired sidebar can be selected via a tab, as shown in 5.7, Number 1.

CHAPTER 6

Scenario-Based Evaluation

This chapter describes the evaluation process for the developed issue visualizations. This involved a series of expert interviews, where scenarios were derived based on the requirements outlined in Section 4.4. Section 6.1 first describes the interview procedure and the example project used to apply the scenarios. The outcomes of the interviews are then summarized in Section 6.2.

6.1 Design

This section provides an overview of how the interviews were conducted for the evaluation of the visualizations by first outlining the framework of the interview round (Section 6.1.1) and then describing the example project on which the scenarios were performed (Section 6.1.2).

6.1.1 Procedure

In order to evaluate the developed visualizations and thus answer RQ3, a series of scenario-based evaluations were conducted. During the interview sessions, the final implementation was used to solve the pre-defined scenarios, guided by a questionnaire created with Google Forms[1], and can be found in Appendix 8.1. The latter consisted of several parts, with the first section dealing with the collection of demographic data and experience in the field of software engineering and software maintenance systems.

[1] https://www.google.com/intl/de_at/forms/about/, Accessed: *23.08.2024*

The remaining sections focused on the scenarios, which included:

- A description of the scenario and the problem.
- The questions to be answered using the final implementation.
- Alternative state-of-the-art approaches to solve this scenario.
- The evaluation of the performance in the scenario and evaluation of the scenario.

The participants had to solve the following scenarios with the help of the implemented visualizations:

1. Scenario: Detailed software issue information
2. Scenario: Overview of software issue distribution
3. Scenario: Overview of software issue contributions
4. Scenario: Overview of software issue status changes
5. Scenario: Software issue comparison

As in the validation phase, the interviews were conducted using the conferencing tool Zoom[2]. This approach allowed for more flexibility in scheduling the meetings, and the meetings could be documented using its built-in recording feature. The benefit of this method was that it preserved the natural flow of conversation without the need for pauses to write down answers to open-ended questions, ensuring the exact wording could be reproduced during the analysis.

Each interview started with an introduction to the topic and the problem that the visualizations aimed to solve. The interviewees were then shown how the prototype works, how the software issue illustrations are structured, and how they can be interacted with. During the interviews, the prototype ran locally on the interviewer's computer. The web application ran in a browser window controlled by the interviewer and transmitted to the participants via screen sharing. The questionnaire was opened in a separate tab, enabling quick transitions between questions and the visualizations to answer the questions. Since usability of the implementation was out of scope of this master thesis, the participants were advised to consider this when evaluating the visualizations for use in each scenario.

A total of five people participated in visualization evaluation. The interviews lasted an average of 50 minutes, with the first session serving as a pilot evaluation. This pilot was used to identify any inconsistencies in the prototype and the questionnaire. Three issues emerged from the initial interview session, which were addressed in a subsequent

[2] `https://zoom.us/`, Accessed: *23.08.2024*

iteration. First, the interviewee noted that some questions were unclear and only became understandable after further explanation by the interviewer. Second, it was observed that certain scenarios were not adequately aligned with the objectives necessary to demonstrate that the requirements were met. Third, going through the questions showed that the illustrations still revealed some bugs that needed to be fixed.

6.1.2 Example Project

The software project used as the basis for answering the questions in the scenarios was a student project developed as part of SE PR in a preceding semester. The project consists of two parts: a backend, developed in Java[3], which uses REST endpoints to perform Create, Read, Update and Delete (CRUD) operations on a database, and a frontend, developed with Angular[4], which communicates with the backend via REST commands. This project was provided as a compressed file with the structure outlined in Section 5.2.1 using EL. A total of:

- six students worked on this project,
- creating 151 issues, most of which resulted in changes to the project's source code,
- with the majority of the issues passing through the states *Opened*, *In Analysis*, *In Progress*, *In Testing*, *Tested*, and *Closed*.

6.2 Results

The following section presents the results of the interviews conducted to evaluate the developed visualization concepts. As with the interviews used to validate the mock-ups described in Chapter 4, demographic data and experience in the field of software engineering, as well as experience with software maintenance tools, were gathered first. These details are provided in Section 6.2.2. The scenarios, the questions asked in those scenarios, and the interviewees' responses are discussed in Sections 6.2.3 - 6.2.7.

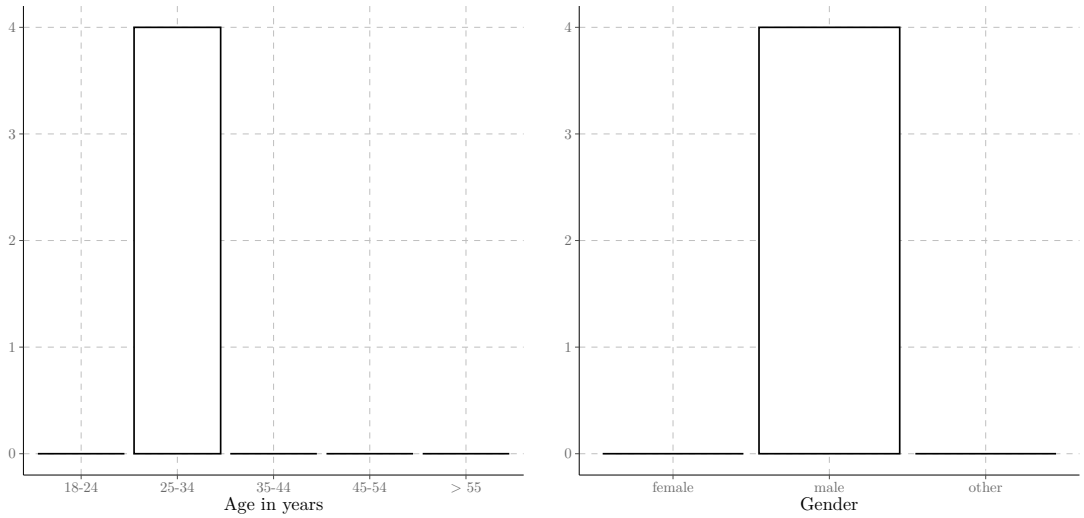
6.2.1 Demographics

The first two questions dealt with the participants' age and gender. All respondents indicated that they were between *25-34* years old and identified as *male*, as shown in Figure 6.1.

6.2.2 Experiences

The following four questions explored the participants' experience in the areas of software engineering, software engineering education, as well as with ITSs and VCSs. Figure

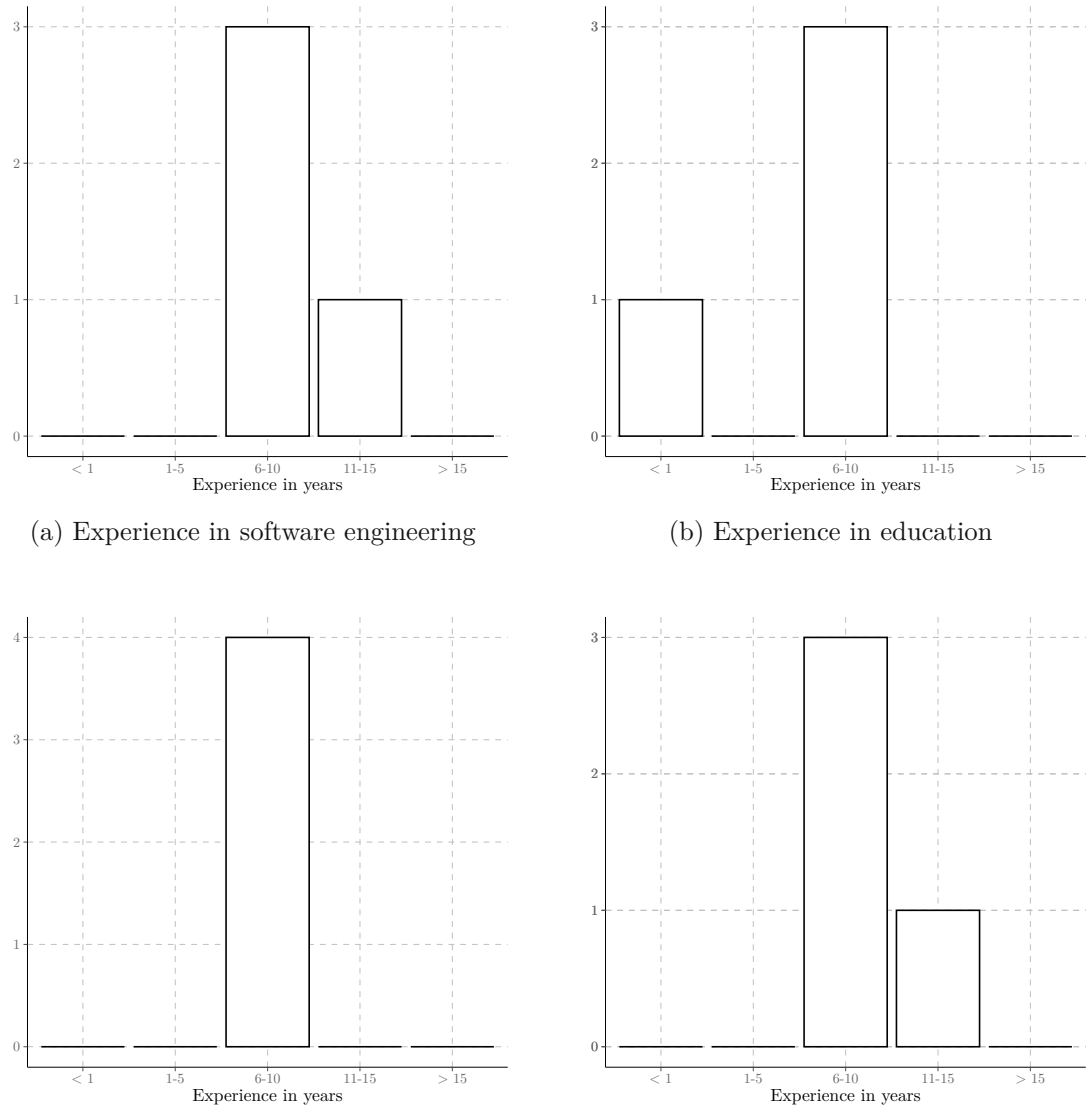
[3] `https://www.java.com/en/`, Accessed: *23.08.2024*
[4] `https://angular.dev/`, Accessed: *23.08.2024*



(a) Age of participants (b) Gender of participants

Figure 6.1: Demographics of participants

6.2 visualizes the participants' responses, with experience represented in years. The interviewees could choose from the following options: *less than one year*, *1-5 years*, *6-10 years*, *11-15 years*, and *more than 15 years*. Three out of four software engineers reported having *6-10 years* of background in software engineering, while one participant reported *11-15 years* of experience in this area. In contrast, one person stated having *less than one year* of experience in software engineering education, while the remaining participants had at least *6-10 years* of experience. Additionally, all developers answered having worked *6-10 years* with ITSs, and except one person (*11-15 years*), all respondents had used VCSs for *6-10 years*.



(a) Experience in software engineering

(b) Experience in education

(c) Experience with ITSs

(d) Experience with VCSs

Figure 6.2: Experiences of participants

6.2.3 Scenario: Detailed software issue information *(R-4, R-5, R-7)*

In order to demonstrate that the visualization fulfills *R-4*, detailed information had to be obtained from the example project. Therefore, participants were asked to analyze a specific software issue and investigate where changes related to this issue were made, the issue's current status, and which individuals were involved in addressing it. Using all available visualizations, the participants had to answer the following three questions:

1. a) During the implementation of *Issue-3*, how many commits were created and referenced?
   b) Which files were changed by *Issue-3*?
2. a) What issue status did *Issue-3* have?
   b) How long did *Issue-3* remain in status *In Testing*?
3. a) Which developers contributed changes to *Issue-3*?
   b) Is there a developer who has only tracked time, but has not made any other contribution to *Issue-3*?

Overall, all interviewed software engineers were able to answer the questions using the developed dashboard, with minor assistance from the interviewer. In order to answer Question 1.b, Participant 1 needed a reminder that a tooltip containing detailed information about a commit, including the files changed, was available. Similarly, Participants 3 and 4 required the same reminded when answering Question 3.b, as the needed information was also accessible through the tooltip. However, the interviewees noted that in a different setting, where they were using the visualization independently, they might have intuitively hovered over the chart entries to trigger the tooltip. All other questions, where the information was directly readable from the visualizations, were answered correctly by all participants.

Figure 6.3 visualizes the usefulness of the graph and the scenario, rated by the interview participants. All developers agreed, on a score of 4 out of 5 for the illustration's ability to provide necessary information. The reasons for the point deduction varied among the participants. Due to changes that were committed in a short period of time, it is possible that multiple commits are represented by a single entry in the visualization. Participant 1 suggested a visual indication, making it more likely for users to click on such points. Two commits with similar timestamps but occurring in different locations are not grouped together. Both Participant 2 and Participant 4 initially thought these were the same commit, as they appeared really close on the x-axis. Upon clicking, they realized these were separate commits. They recommended an indicator in the visualization to make this distinction clearer.

Regarding the visualization of files changed by a software issue, Participant 1 mentioned, modifying the bar chart to allow switching between displaying the number of lines of code

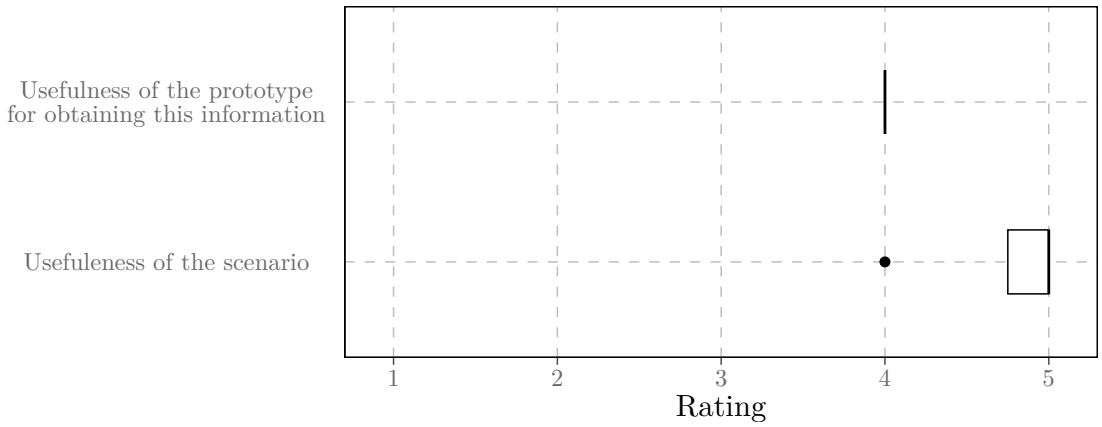


Figure 6.3: Usefulness of scenario: "Detailed information about a software issue"

added/ removed and the number of files changed, which would make it easier to answer Question 1.b. All participants agreed that different types of contributions (e.g., commits or changes to issue descriptions) should be encoded differently in the visualization to aid in easier distinction. Furthermore, three out of four participants rated the scenario as *Very Useful*. They agreed that this information could be very valuable in an educational context, such as evaluating student projects, as it quickly reveals which student was involved in which software issue. Additionally, Participant 2 mentioned that this could also be effectively used in management.

When discussing state-of-the-art methods for obtaining the information described in this scenario, the participants mentioned different approaches. Developer 1 mentioned that he would have searched for the issue platforms like GitLab[5] or GitHub[6] but noted that a comparable time-based representation would not be available. Developer 2 stated that he would have exported the data, loaded it into a database, and then queried it, preferring raw data due to difficulties with the user interface of the ITS. Participants 3 and 4 stated they would have used tools like Sourcetree[7], Gitinspector[8] to analyze Git history, review the change history of software issue statuses, and search for the issue in GitLab's[5] spent time listings. However, all participants agreed that the developed visualization simplified the process of obtaining this information.

6.2.4 Scenario: Overview of software issue distribution *(R-1, R-7)*

In this scenario, the interviewees were tasked with gaining an overview of where, in the source code, changes were made due to software issues. They needed to identify which files underwent the most modifications, which issues were responsible for these

[5] `https://about.gitlab.com/`, Accessed: *23.08.2024*
[6] `https://github.com/`, Accessed: *23.08.2024*
[7] `https://www.sourcetreeapp.com/` and *Accessed: 23.08.2024*
[8] `https://github.com/ejwa/gitinspector`, *Accessed: 23.08.2024*

modifications, and how much time these modifications took. To narrow down the number of issues, a period of one month was selected, and one package in the backend of the project was chosen. Using the issue distribution visualizations, the participants had to answer the following questions:

1. a) Based on the number of lines of code added and removed, which of the files has changed the most?
   b) Excluding the files that had no changes, which of the files has changed the least?
2. a) Which file was changed by the largest number of issues?
   b) Regarding the previous file, which issues caused its changes?
3. Which issue has changed the most files?
4. How much time has passed between the first commit and the last commit of this issue?

All questions were correctly answered by the interview participants, and most of them could be quickly resolved with a simple glance at the illustration. Unlike the first scenario, the participants were already more familiar with the visualization and were able to find detailed information, such as the required one for Question 2.b, without assistance from the interviewer. However, Question 3.b posed more of a challenge, as it required identifying the issue that caused changes in the most areas. This information could only be determined by counting the number of colors in the visualization, which was not immediately intuitive. To estimate the working time of a software issue (Question 4), Participants 1, 3, and 4 read the timestamps on the x-axis and approximated the duration based on this. Participant 2, on the other hand, relied on the tooltip to answer the question, as it provided exact dates for the entries. He noted that this approach was effective in this particular scenario, where a manageable number of locations were being accessed. However, he pointed out that this method might be less practical if an issue ticket affected a larger number of files.

The results of the usefulness of the illustration and the scenario can be seen in Figure 6.4. Participant 1 rated the visualization as *Very useful*, as he had no problems retrieving the required information and thinks that the dashboard effectively visualizes it. However, Participants 2, 3, and 4 had several complaints to deny a full rating for the graph. First, they noted that the choice of colors used to differentiate the software issues was too random. They struggled to distinguish between two similar colors, visualized next to each other - a problem that may have been strengthened by the reduced quality of screen sharing. They suggested that the colors for issues visualized next to each other should be more distinct to avoid confusion. Another point they raised was that, unlike in Scenario 1, more locations were displayed on the y-axis, making it difficult for them to match the entries in the bar chart with the y-axis. They felt that a grid could have helped make

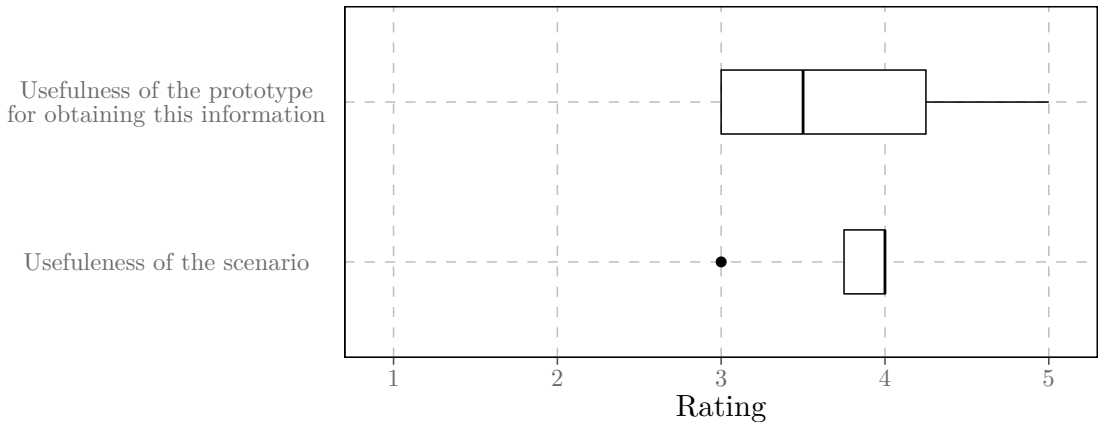


Figure 6.4: Usefulness of scenario: "Overview of software issue distribution"

these connections easier. Additionally, they found that the illustration was not ideal for answering Question 3. Developers 2 and 4 would have preferred a reversed view that lists files for each issue. Participant 1 suggested that including relevant information in the legend for each issue entry would also help for gathering this information.

This scenario received a mean rating of 3.75 points from the respondents. Participant 1 mentioned that he wasn't entirely sure how he would use this information, noting that this might be due to the fact that such data isn't typically accessible with other tools, resulting in a lack of experience with it. Developer 2, however, could see this information being useful for issue analysis, for example, to determine if a software issue was too broadly defined and should have been separated into smaller issues, particularly if it impacted too many files or if the resolution time was too long. Regarding the current methods available for obtaining this information, the participants provided similar answers to those given in Scenario 1. They would have either worked directly with the raw data or combined issues in an ITS with the Git history. However, they all agreed that there is no way to achieve this in the manner the visualization does.

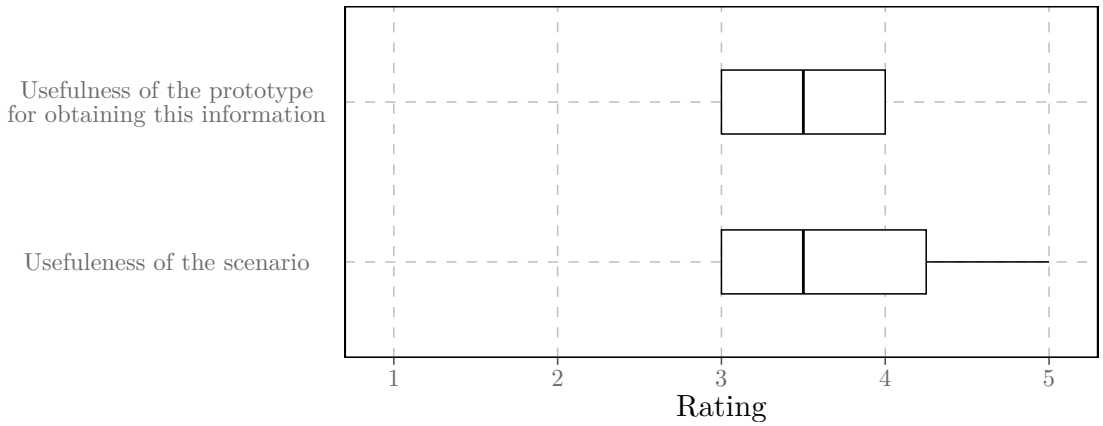


Figure 6.5: Usefulness of scenario: "Overview of software issue contributions"

6.2.5 Scenario: Overview of software issue contributions *(R-3, R-7)*

To show that the visualization implementation can be used to get an overview of the issue contribution, the participants had to answer questions about which group member contributed to which software issue and how the work was distributed within the team. As with Scenario 2, a period covering two project management iterations was selected. The following questions had to be answered using the visualization results:

1. a) Which team member was involved in the most issues?
   b) Which team member was involved in the least issues?
2. a) Which team member has tracked the most time?
   b) Which team member has tracked the least time?
3. a) Which issue had the most time tracked?
   b) Which person tracked the most time on the previous issue?
   c) Regarding the previous person, did that person also contribute the most changes?

For this scenario, most of the questions were answered without much effort by the interviewees, even though they had chosen different approaches in some cases. However, Questions 1.a and 3.a posed more of a challenge, taking the software engineers more time to resolve, though in the end they succeeded. This is reflected by the rating, which can be seen in Figure 6.5. With a mean score of 3.5 points, the dashboard performed worse for the information retrieval in this scenario compared to the previous scenarios. The participants noted that the visualizations were not specifically designed to address some questions in this scenario. For Question 1.a they mentioned that the possibility of

two consecutive issues being represented by similar colors affected their initial judgment of which entry had the biggest width. Developer 1 suggested an additional metric to make the distinction clearer, or a hover feature that would quickly reveal whether the entries belong to the same software issue or not. In his opinion, the tooltip provides the desired information too slowly. For Question 3.a the same issue regarding color selection was applied. Another point of critique was that the software issue with the most time spent on it could only be found by repeatedly using a filter option. One software engineer suggested adding a functionality to sort issues by tracked time. Participant 2 also noted that the illustration was well-suited for the remaining questions.

Two participants rated the usefulness of the scenario as neutral, as they were unsure of its practical application. Participant 3 said it is *Very Useful*, while Participant 1, similar to Scenario 2, could not assess how useful the scenario was due to his lack of experience with such metrics. Nevertheless, when discussing traditional methods for solving such questions, the developers agreed that they would have typically searched for the software issues using the user interface of an ITS. They also acknowledged that a summary view like the one provided by the visualization would be a better approach solving it.

6.2.6 Scenario: Overview of software issue status changes *(R-2, R-7)*

In this scenario, the interviewees were required to analyze the lifecycle of software issues. The goal was to identify when software issues transitioned to a specific status and how long they remained in that status. Participants were asked to perform this analysis for a project management iteration and then analyze issues of the types *Defect* and *Issue* (i.e., all issues that are not classified as software defects). The following questions had to be answered:

1. How long did it take for issues of type *Defect* to move from status *Open* to status *Closed*?
2. How long did it take for issues of type *Issue* to move from status *Open* to status *Closed*?

In this scenario, the participants had no trouble answering the questions. Although none of the participants felt confident in providing an exact result, they made estimates and concluded that, for the analyzed project management iteration, on average, issues of the type *Defect* transitioned more quickly from the status *Opened* to *Closed*. Looking at the evaluation of the visualization in Figure 6.6, it can be seen that the opinion of the usefulness of the dashboard for obtaining this information varied. Unlike Participant 1 (Score: 5) and Participant 4 (Score: 4), Participants 2 and 3 rated the visualization with 2 points out of 5. Developer 2 pointed out that the total transition time of an issue had to be calculated by summing the individual time spans in each status. Additionally, Developer 3 added that the color gradient intended to clarify status transitions was more confusing than helpful.

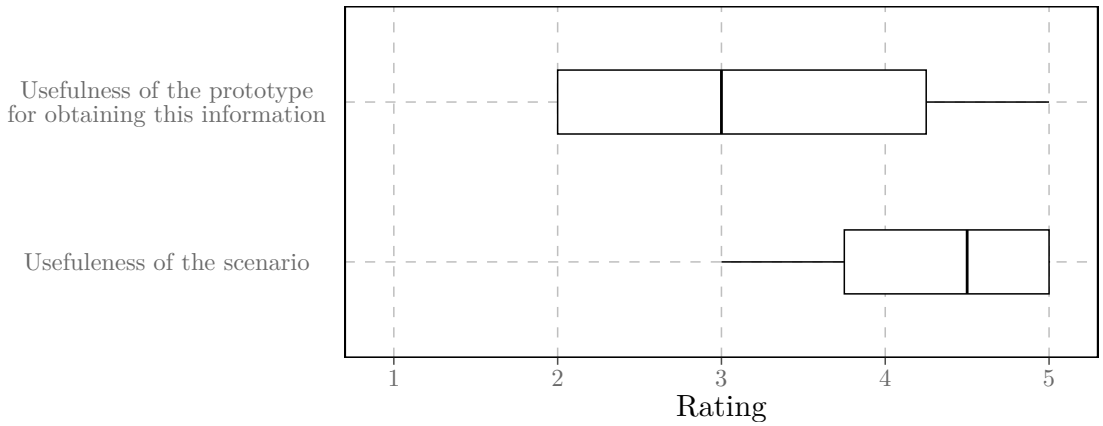


Figure 6.6: Usefulness of scenario: "Overview of software issue status changes"

The scenario itself received a mean usefulness rating of 4.25 from the developers. Only Participant 1 was unsure about what he would do with the provided information of the visualizations. When asked about alternative methods to obtain the same information without the visualization, most participants stated that they would have used a project management tool to filter the relevant software issues and calculate the average time to close them. Participant 2 added that he would have preferred this method over the implemented visualization.

6.2.7 Scenario: Software issue comparison *(R-6)*

In the final scenario, respondents were asked to compare two filtered versions and answer questions about the distribution of issues and work among the team members. For the first version, *Spring-1* and *Sprint-2* were selected, and for the second version, *Sprint-3* and *Sprint-4* were chosen. The following questions had to be answered:

1. a) Which component received the most changes in *Sprint-1* and *Sprint-2*?
   b) Which component received the most changes in *Sprint-3* and *Sprint-4*?
2. a) How was the distribution of work between the team members in *Sprint-1* and *Sprint-2*?
   b) How was the distribution of work between the team members in *Sprint-3* and *Sprint-4*?

As for the previous scenarios, the interview participants were able to compare different versions of a project and had no difficulty in answering the questions. For instance, they noticed that in the first four project management iterations, the work was not evenly distributed, as one group member worked on significantly more issues and tracked

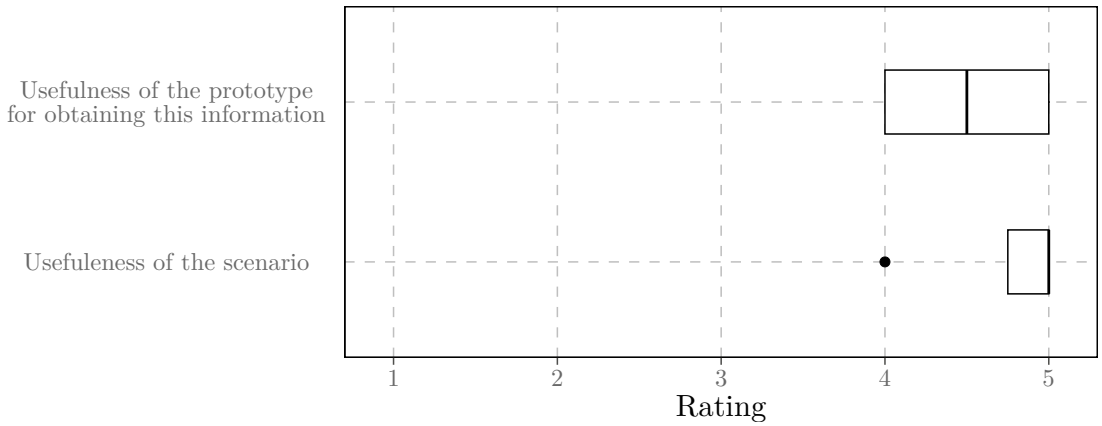


Figure 6.7: Usefulness of scenario: "Issue comparison"

almost twice as much time as the other members. They also observed that one student invested very little time in the project during the first iterations. The comparison feature of the visualization received an average rating of 4.5 from all the interviewed software engineers, reflecting its strong performance, as shown in Figure 6.7. Participant 3 noted the visualization's usefulness in showing who worked on which issues and how contributions evolved over time. Developer 2 added that while this approach works well for a student project conducted over a short period, comparisons might be more difficult in larger projects.

Also illustrated in Figure 6.7, most of the interviewees rated such a scenario to be useful. Participant 2 mentioned that these illustrations are particularly valuable when examining the evolution of a project, such as determining whether all group members contributed from the beginning. Compared to traditional tools, they preferred the visualization. Participant 1 noted that, without the illustrations, he would have filtered the software issues by iterations and kept each iteration open in separate browser tabs for comparison. Developer 2 would have searched the git logs for the corresponding issue identifiers and then grouped them to find out how many lines of code were changed. Interviewee 4 would have used a table, provided by SE PR and ASE that shows the tracked time per week for each student but noted that it wouldn't provide any information about the code changes.

6.3 Threats to Validity

This section describes potential threats that could impact the validity of the scenario-based evaluation.

6.3.1 Number and Demographics of Participants

One threat to validity is the number of participants selected for the scenario-based evaluation. Similar to the semi-structured expert interviews for the validation of the visualization concepts, four people participated in the interviews. The size of the sample may be too small to generalize the results of this evaluation, as normally a larger number of selected participants is required. Additionally, the diversity of the selected participants could be a threat, as all interviewees identified as *male* and reported being between *25 and 34* years old.

6.3.2 Scenarios

The scenarios used to evaluate the visualization pose another potential threat. First, the scenario explained, and the questions asked during the interview may have been misunderstood. Although participants were given the opportunity to ask clarifying questions, misunderstandings could still lead to random or inaccurate ratings. Furthermore, the choice of unsuitable scenarios could influence the purposefulness of the developed dashboard. For this reason, questions were introduced at the end of each scenario to evaluate the usefulness of the scenario.

CHAPTER 7

Discussion

This chapter reflects the results of this master thesis by answering the research questions from Section 3.1.

RQ1a: What information needs exists for a detailed analysis of software issues that can be used for software engineering education? The results of the scientific literature review and the analysis of the results of the semi-structured expert interviews were used to identify existing information needs related to ITSs. First, software engineers prefer to obtain historical and evolutionary information about a software project [12, 13, 17] since it can be used either to recover lost knowledge or to track work in progress [17]. It is preferable due to the lower effort compared to tools that try to make predictions about the future [13]. There is also a need to identify potential trends, which can be accomplished by comparing multiple versions of the history [17]. Furthermore, it has been explored that both hotspots for frequent changes and the duration of processing are of great importance [69]. This information can be used, for example, to determine whether parts of the source code are prone to software defects. Particularly in the area of software engineering education, it has become apparent that there is a need for information about the distribution of work among students in a project group [16, 28, 55]. Finally, the interviews confirmed that issue status lifecycle can provide a good overview of a project, although such a feature is not currently available in tools like GitLab[1].

RQ1b: What visual aspects are necessary to support detailed analysis of software issues in software engineering education? In order to create comprehensive visualizations for the detailed analysis of software issues, visualization concepts were developed based on the information needs explored in RQ1a. In order to present these as simply as possible, only graph-based techniques were used. Scatter plots, stacked bar

[1] `https://about.gitlab.com/`, Accessed: *23.08.2024*

charts, and radar charts were used to illustrate the information and multiple views were chosen to avoid overloading the visualizations with information. Entries in the graphs can be distinguished by both, their color and their spatial position. Colors were used to differentiate between software issues or their status, while spatial position is used to differentiate between source code areas or project members. Additionally, a number of interaction mechanisms have been integrated to improve the analysis of the data. These include a filter to adapt the results to the user's needs, on-demand details in the form of a legend or tooltip to present additional information, scrolling to navigate through the data, and a collapsible folder tree to show files and subfolders in a selected folder.

RQ2: How do experts rate the proposed visualization concepts for use in software engineering education? The concepts developed were then validated through semi-structured expert interviews with experts experienced in software engineering and software engineering education. All except one visualization concept received a mean rating of 3 or higher. The visualization concepts that show the distribution of issues (Figure 4.2 and Figure 4.3c), their life cycle based on status changes (Figure 4.5a), and student contributions to software issues in the context of the project timeline were rated positively. Similarly, the visualization that lists the issues each student tracked time for was also well-rated (Figure 4.7a). Less well received were illustrations based on relative values and general information. The majority of the suggested filters were rated by respondents as helpful and necessary for a large amount of data.

RQ3a: How do the experts evaluate the visualization developed with regard to its hypothesized benefits in software engineering education? In general, the functionality of the implemented visualizations was rated useful with regard to the hypothetical benefits of identification of code hotspots, bottlenecks, distribution of work and trends. The majority of the questions in the defined scenarios could be answered with the help of the implemented illustrations and the filters. In some cases, however, the participants of the interviews had to be supported with small hints. A significant issue with the illustrations was the use of colors to differentiate between software issues or their status. The chosen color palette sometimes led to confusion, particularly when similar colors were used for issues that were displayed next to each other. With a mean rating of 3.25 points, the visualizations received the lowest rating in the scenario for software issue lifecycle inspection. The highest mean score of 4.75 was given to the visualizations for the version comparison scenario to show the ability to analyze trends.

RQ3b: How effective is the developed visualization compared to traditional methods in software engineering education? For the majority of the questions asked in the scenarios, the participants of the interviews preferred the visualizations over the conventional method of obtaining the information. In the majority of cases, the interviewees would have tried different approaches of extracting the information from the repositories of ITSs and VCSs and linking them together. This would involve querying raw data and utilizing the user interfaces of these systems. The participants agreed that

the implemented visualizations offered historical insights not easily obtainable from other tools. The comparison functionality that enables the comparison of, for example, two different periods of a project was also rated as an effective method for analyzing the evolution of a project.

CHAPTER 8

Conclusion

In this thesis, visualizations were developed that provide users with a decision-making basis with data from ITS software repositories. This approach was designed specifically for the field of software engineering education. As part of this work, a scientific literature review was conducted, which revealed that the visualization tools in both scientific and educational contexts in previous work have mainly specified data from VCS repositories. ITS repositories have received less attention. Furthermore, it was explored that scientific visualizations often rely on simple representations that are sufficient for overviews, but when detailed historical information needs to be explored, these are no longer suitable. In addition, the research and semi-structured expert interviews revealed that software engineers have a need for both historical information and data on the distribution and participation of various data sources. Therefore, an approach has been developed that focuses on visualizing the distribution of software issues, contributions to these issues, and their status changes in an attempt to fill this knowledge gap.

Based on the results of the literature research, concepts for possible visualizations were developed to illustrate the aforementioned metrics. These visualizations were then validated through five semi-structured expert interviews. The interviewees were software engineers with experience in software engineering education, and the feedback showed that most of the concepts were considered useful. This validation served as a foundation to decide which visualizations should be implemented for a proof of concept. The visualizations were developed in an iterative process, where improvements were discussed and incorporated at the end of each iteration. First, the data preparation components were developed. To allow the visualizations to interact with existing software repositories, whether they are actively hosted on a server or archived, the extraction and processing of the data was split into two layers, allowing the user to connect the visualizations directly to either an active or archived repository. The visualizations were then built on top of these two layers.

The final implementation was then evaluated using scenario-based expert interviews. A total of five participants took part in the interviews, whereby care was taken to ensure that they were software engineers with experience in the field of education. The usefulness of historical project analysis based on software issue data was confirmed in an educational context, as it was possible for the interview participants to gain a deeper understanding of the project's history and its development up to the current state. Additionally, the developers provided further suggestions for improving the visualizations.

8.1 Future Work

Due to the time constraints of this thesis, not all ideas and features could be realized. This section describes possible approaches for future work in this field.

The visualizations currently developed focus mainly on distribution, status changes and contributions to software issues. However, there is potential to explore and evaluate further visualizations that consider additional metrics. As mentioned in Chapter 4, data related to CI/CD in particular would be of great interest. For example, visualizations could show in which areas CI/CD pipelines were triggered more frequently, who initiated them and what status they had. Furthermore, the visualizations with lower ratings from the first round of interviews, as described in Chapter 4, could be revised and validated.

Additionally, it could be investigated how the visualization concepts perform in the context of a software engineering course and whether they can be used effectively both by teaching staff to assess groups and by students to support decision-making. The experts' improvement suggestions described in Chapter 6 should be taken into account and integrated.

As already described in Chapter 1, the focus of this thesis was on the development of visualizations that are specifically suitable for projects in the field of software engineering education. Since real-world projects differ in terms of the number of developers and the time scope, future work could investigate how the visualization concepts need to be adapted accordingly. Subsequent research could then investigate the use of these concepts in real-world software projects.

List of Figures

List of Tables

Listings

Acronyms

APM Agile Project Management. 1, 5, 6, 7, 95

ASE Advanced Software Engineering. 1, 23, 50, 87

CI/CD Continuous Integration/Continuous Deployment. 47, 94

CRUD Create, Read, Update and Delete. 77

CVCS Centralized Version Control System. 8, 9

DVCS Distributed Version Control System. 8, 9

EL Extraction Layer. 53, 55, 60, 61, 65, 66, 67, 69, 77

HTTP Hypertext Transfer Protocol. 66, 71

IoC Inversion of Control. 61

ITS Issue Tracking System. 2, 5, 9, 11, 16, 17, 21, 29, 38, 39, 40, 41, 42, 53, 66, 77, 78, 79, 81, 83, 85, 89, 90, 93, 95

JSON JavaScript Object Notation. 55, 57, 63, 66

LVCS Local Version Control System. 8

MR Merge Request. 10

NDJSON Newline Delimited JavaScript Object Notation. 56, 62, 63, 64, 99

PBL Project Based Learning. 1

PL Processing Layer. 53, 55, 66, 67, 69, 71, 96, 99

PM Project Management. 5

PMI Project Management Institute. 5

POJO Plain Old Java Object. 63, 66

PR Pull Request. 10

RCS Revision Control System. 7

REST Representational State Transfer. 17, 53, 54, 55, 61, 65, 66, 67, 69, 77

SE PR Software Engineering Project. 1, 23, 50, 56, 66, 77, 87

SQL Structured Query Language. 54

SVN Subversion. 21

TPM Traditional Project Management. 5, 6, 7, 95

VCS Version Control System. 2, 5, 7, 8, 9, 11, 16, 21, 31, 39, 40, 41, 42, 53, 77, 78, 79, 90, 93, 95

VL Visualization Layer. 53, 55, 66, 69, 71

XML Extensible Markup Language. 16

Literature References

[1] Basant Lal Agarwal. *Basic statistics*. New Age International, 2006.

[2] Adebayo Agbejule and Lassi Lehtineva. „The relationship between traditional project management, agile project management and teamwork quality on project success“. In: (). DOI: 10.1108/IJOA-02-2022-3149. URL: https://www.emerald.com/insight/1934-8835.htm.

[3] Abdulkareem Alali, Huzefa Kagdi, and Jonathan I. Maletic. „What's a typical commit? A characterization of open source software repositories“. In: *IEEE International Conference on Program Comprehension* (2008), pp. 182–191. DOI: 10.1109/ICPC.2008.24.

[4] Jeroen van Baarsen. *GitLab Cookbook*. eng. 1st ed. Birmingham: Packt Publishing, 2014. ISBN: 9781783986842.

[5] T. Ball and S.G. Eick. „Software visualization in the large“. In: *Computer* 29.4 (Apr. 1996), pp. 33–43. ISSN: 00189162. DOI: 10.1109/2.488299. URL: http://ieeexplore.ieee.org/document/488299/.

[6] Sebastian Baltes and Paul Ralph. „Sampling in Software Engineering Research: A Critical Review and Guidelines“. In: (2021). DOI: 10.1007/s10664-021-10072-8.

[7] Andrew Begel and Thomas Zimmermann. „Analyze this! 145 questions for data scientists in software engineering“. In: *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014* (2014), pp. 12–23. ISSN: 02705257. DOI: 10.1145/2568225.2568233. URL: http://dl.acm.org/citation.cfm?doid=2568225.2568233.

[8] Thomas Bergmann and Waldemar Karwowski. „Agile project management and project success: A literature review“. In: *Advances in Intelligent Systems and Computing* 783 (2019), pp. 405–414. ISSN: 21945357. DOI: https://doi.org/10.1007/978-3-319-94709-9_39. URL: https://link.springer.com/chapter/10.1007/978-3-319-94709-9_39.

[9] Nikos Bikakis. „Big Data Visualization Tools“. In: *Encyclopedia of Big Data Technologies* (Jan. 2018), pp. 1–8. DOI: 10.1007/978-3-319-63962-8_109-2. URL: http://arxiv.org/abs/1801.08336%20http://dx.doi.org/10.1007/978-3-319-63962-8_109-2.

[10] Tegawende F. Bissyande, David Lo, Lingxiao Jiang, Laurent Reveillere, Jacques Klein, and Yves Le Traon. „Got issues? Who cares about it? A large scale investigation of issue trackers from GitHub“. In: *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Nov. 2013, pp. 188–197. ISBN: 978-1-4799-2366-3. DOI: 10.1109/ISSRE.2013.6698918. URL: http://ieeexplore.ieee.org/document/6698918/.

[11] John D Blischak, Emily R Davenport, and Greg Wilson. „A Quick Introduction to Version Control with Git and GitHub Introduction to Version Control“. In: (2016). DOI: 10.1371/journal.pcbi.1004668. URL: https://git-scm.com/downloads/guis.

[12] Henri Bomström, Markus Kelanti, Elina Annanperä, Kari Liukkunen, Terhi Kilamo, Outi Sievi-Korte, and Kari Systä. „Information needs and presentation in agile software development“. In: *Information and Software Technology* 162 (Oct. 2023), p. 107265. ISSN: 09505849. DOI: 10.1016/j.infsof.2023.107265. URL: https://linkinghub.elsevier.com/retrieve/pii/S0950584923001192.

[13] Raymond P L Buse and Thomas Zimmermann. „Information needs for software development analytics“. In: *Proceedings of the 2012 International Conference on Software Engineering*. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 987–996. ISBN: 978-1-4673-1067-3. URL: http://dl.acm.org/citation.cfm?id=2337223.2337343.

[14] Guilherme Cavalcanti, Paola Accioly, and Paulo Borba. „Assessing Semistructured Merge in Version Control Systems: A Replicated Experiment“. In: *International Symposium on Empirical Software Engineering and Measurement* 2015-November (Nov. 2015), pp. 267–276. ISSN: 19493789. DOI: 10.1109/ESEM.2015.7321191.

[15] Scott Chacon and Ben Straub. *Pro Git*. eng. 2nd ed. Berkeley, CA: Apress, 2014. ISBN: 978-1-4842-0077-3. DOI: 10.1007/978-1-4842-0076-6. URL: http://link.springer.com/10.1007/978-1-4842-0076-6.

[16] Hsi-Min Chen, Bao-An Nguyen, and Chyi-Ren Dow. „Code-quality evaluation scheme for assessment of student contributions to programming projects“. In: *Journal of Systems and Software* 188 (June 2022), p. 111273. ISSN: 01641212. DOI: 10.1016/j.jss.2022.111273. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121222000358.

[17] M Codoban, S S Ragavan, D Dig, and B Bailey. „Software history under the lens: A study on why and how developers examine it“. In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Sept. 2015, pp. 1–10. DOI: 10.1109/ICSM.2015.7332446.

[18] David Coppit. „Implementing large projects in software engineering courses“. In: *Computer Science Education* 16.1 (2006), pp. 53–73. ISSN: 17445175. DOI: 10.1080/08993400600600443. URL: https://www.tandfonline.com/doi/abs/10.1080/08993400600600443.

[19] Marco D'Ambros, Michele Lanza, and Martin Pinzger. „"A Bug's Life" Visualizing a Bug Database". In: *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, June 2007, pp. 113–120. ISBN: 1-4244-0599-8. DOI: 10.1109/VISSOF.2007.4290709. URL: http://ieeexplore.ieee.org/document/4290709/.

[20] Tommaso Dal Sassc and Michele Lanza. „A closer look at bugs". In: *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, Sept. 2013, pp. 1–4. ISBN: 978-1-4799-1457-9. DOI: 10.1109/VISSOFT.2013.6650542. URL: http://ieeexplore.ieee.org/document/6650542/.

[21] Tommaso Dal Sasso and Michele Lanza. „In bug: Visual analytics of bug repositories". In: *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, Feb. 2014, pp. 415–419. ISBN: 978-1-4799-3752-3. DOI: 10.1109/CSMR-WCRE.2014.6747208. URL: https://ieeexplore.ieee.org/document/6747208/.

[22] Brian De Alwis and Jonathan Sillito. „Why are software projects moving from centralized to decentralized version control systems?" In: *Proceedings of the 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering, CHASE 2009* (2009), pp. 36–39. DOI: 10.1109/CHASE.2009.5071408.

[23] Mário André F. De Farias, Methanias Colaço, Manoel Mendonça, Renato Novais, Luís Paulo Da Silva Carvalho, and Rodrigo Oliveira Spínola. „A systematic mapping study on mining software repositories". In: *Proceedings of the ACM Symposium on Applied Computing* 04-08-April-2016 (Apr. 2016), pp. 1472–1479. DOI: 10.1145/2851613.2851786. URL: http://dx.doi.org/10.1145/2851613.2851786.

[24] Daniel Dietsch, Andreas Podelski, Jaechang Nam, Pantelis M Papadopoulos, and Martin Schäf. „Monitoring Student Activity in Collaborative Software Development". In: (). DOI: https://doi.org/10.48550/arXiv.1305.0787.

[25] Dutt. *Software Project Management*. eng. 1st edition. Pearson India, 2015. ISBN: 9389552788.

[26] Tore Dybå, Torgeir Dingsoyr, and Nils Brede Moe. „Agile project management". In: *Software Project Management in a Changing World* 9783642550355 (Mar. 2014), pp. 277–300. DOI: 10.1007/978-3-642-55035-5_11. URL: https://link.springer.com/chapter/10.1007/978-3-642-55035-5_11.

[27] Keith Engwall and Mitchell Roe. „Git and GitLab in Library Website Change Management Workflows". In: *Code4Lib Journal* 48 (May 2020). ISSN: 1940-5758. URL: https://journal.code4lib.org/articles/15250.

[28] Sukru Eraslan, Kamilla Kopec-Harding, Caroline Jay, Suzanne M. Embury, Robert Haines, Julio César Cortés Ríos, and Peter Crowther. „Integrating GitLab metrics into coursework consultation sessions in a software engineering course". In: *Journal of Systems and Software* 167 (Sept. 2020), p. 110613. ISSN: 01641212. DOI: 10.

1016/j.jss.2020.110613. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121220300911.

[29] J. Favela and F. Pena-Mora. „An experience in collaborative software engineering education“. In: *IEEE Software* 18.2 (Mar. 2001), pp. 47–53. ISSN: 07407459. DOI: 10.1109/52.914742. URL: http://ieeexplore.ieee.org/document/914742/.

[30] Johannes Feiner and Keith Andrews. „RepoVis: Visual Overviews and Full-Text Search in Software Repositories“. In: *2018 IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, Sept. 2018, pp. 1–11. ISBN: 978-1-5386-8292-0. DOI: 10.1109/VISSOFT.2018.00009. URL: https://ieeexplore.ieee.org/document/8530126/.

[31] Aron Fiechter, Roberto Minelli, Csaba Nagy, and Michele Lanza. „Visualizing GitHub Issues“. In: *2021 Working Conference on Software Visualization (VISSOFT)*. IEEE, Sept. 2021, pp. 155–159. ISBN: 978-1-6654-3144-6. DOI: 10.1109/VISSOFT52517.2021.00030. URL: https://ieeexplore.ieee.org/document/9604892/.

[32] Maria Lydia Fioravanti, Bruno Sena, Leo Natan Paschoal, Laíza R. Silva, Ana P. Allian, Elisa Y. Nakagawa, Simone R.S. Souza, Seiji Isotani, and Ellen F. Barbosa. „Integrating Project Based Learning and Project Management for Software Engineering Teaching: An Experience Report“. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. SIGCSE '18. Baltimore, Maryland, USA: Association for Computing Machinery, 2018, pp. 806–811. ISBN: 9781450351034. DOI: 10.1145/3159450.3159599. URL: https://doi.org/10.1145/3159450.3159599.

[35] Denis Gračanin, Krešimir Matković, and Mohamed Eltoweissy. „Software visualization“. In: *Innovations in Systems and Software Engineering* 1.2 (Sept. 2005), pp. 221–230. ISSN: 16145046. DOI: 10.1007/S11334-005-0019-8. URL: https://link.springer.com/article/10.1007/s11334-005-0019-8.

[36] Andre Hora, Nicolas Anquetil, Stephane Ducasse, Muhammad Bhatti, Cesar Couto, Marco Tulio Valente, and Julio Martins. „Bug Maps: A Tool for the Visual Exploration and Analysis of Bugs“. In: *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, Mar. 2012, pp. 523–526. ISBN: 978-0-7695-4666-7. DOI: 10.1109/CSMR.2012.68. URL: http://ieeexplore.ieee.org/document/6178935/.

[37] Ryo Ishizuka, Hironori Washizaki, Yoshiaki Fukazawa, Shinobu Saito, and Saori Ouji. „Categorizing and Visualizing Issue Tickets to Better Understand the Features Implemented in Existing Software Systems“. In: *Proceedings - 2019 10th International Workshop on Empirical Software Engineering in Practice, IWESEP 2019* (Dec. 2019), pp. 25–30. DOI: 10.1109/IWESEP49350.2019.00013.

[38] Ryo Ishizuka, Hironori Washizaki, Naohiko Tsuda, Yoshiaki Fukazawa, Saori Ouji, Shinobu Saito, and Yukako Iimura. „Categorization and Visualization of Issue Tickets to Support Understanding of Implemented Features in Software Development Projects“. In: *Applied Sciences 2022, Vol. 12, Page 3222* 12.7 (Mar. 2022), p. 3222. ISSN: 2076-3417. DOI: 10.3390/APP12073222. URL: https://www.mdpi.com/2076-3417/12/7/3222/htm%20https://www.mdpi.com/2076-3417/12/7/3222.

[39] Curt Jones. „Using subversion as an aid in evaluating individuals working on a group coding project“. In: *J. Comput. Sci. Coll.* 25.3 (Jan. 2010), pp. 18–23. ISSN: 1937-4771.

[40] Stuart T Kard, Jock D Mackinlay, and Ben Scheiderman. *Readings in Information Visualization, Using vision to think*. 1999. ISBN: 1558605339.

[41] Harold Kerzner. *Project management : a systems approach to planning, scheduling, and controlling*. eng. 10th ed. Hoboken, N.J.: Wiley, 2009. ISBN: 0470503831.

[42] Ali Koc and Abdullah Uz Tansel. „A Survey of Version Control Systems“. In: 2011. URL: https://api.semanticscholar.org/CorpusID:41570936.

[43] Rainer Koschke. „Software visualization in software maintenance, reverse engineering, and re-engineering: A research survey“. In: *Journal of Software Maintenance and Evolution* 15.2 (Mar. 2003), pp. 87–109. ISSN: 1532060X. DOI: 10.1002/SMR.270.

[44] Qi Li. „Overview of Data Visualization“. In: *Embodying Data* (2020), pp. 17–47. DOI: 10.1007/978-981-15-5069-0_2. URL: https://link.springer.com/chapter/10.1007/978-981-15-5069-0_2.

[45] Chang Liu. „Using Issue Tracking Tools to Facilitate Student Learning of Communication Skills in Software Engineering Courses“. In: *18th Conference on Software Engineering Education & Training (CSEET'05)*. IEEE, Apr. 2005, pp. 61–68. ISBN: 0-7695-2324-2. DOI: 10.1109/CSEET.2005.40. URL: http://ieeexplore.ieee.org/document/4698909/.

[47] Thorsten Merten, Matus Falis, Paul Hubner, Thomas Quirchmayr, Simone Bursner, and Barbara Paech. „Software Feature Request Detection in Issue Tracking Systems“. In: *2016 IEEE 24th International Requirements Engineering Conference (RE)*. IEEE, Sept. 2016, pp. 166–175. ISBN: 978-1-5090-4121-3. DOI: 10.1109/RE.2016.8. URL: http://ieeexplore.ieee.org/document/7765522/.

[48] Thorsten Merten, Bastian Mager, Paul Hübner, Thomas Quirchmayr, Barbara Paech, and Simone Bürsner. „Requirements Communication in Issue Tracking Systems in Four Open-Source Projects“. In: 2015, pp. 114–125. URL: https://ceur-ws.org/Vol-1342/.

[49] Megha Mittal and Ashish Sureka. „Process mining software repositories from student projects in an undergraduate software engineering course“. In: *Companion Proceedings of the 36th International Conference on Software Engineering*. ICSE Companion 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 344–353. ISBN: 9781450327688. DOI: 10.1145/2591062.2591152. URL: https://doi.org/10.1145/2591062.2591152.

[50] Sarhan M Musa, Cajetan Akujuobi, Matthew N O Sadiku, Adebowale E Shadare, Cajetan M Akujuobi, and Roy G Perry. „Publication Impact Factor (PIF): 1.02 www.sretechjournal.org DATA VISUALIZATION“. In: *International Journal of Engineering Research And Advanced Technology* (2016). ISSN: 2454-6135. URL: https://www.researchgate.net/publication/311597028.

[51] Muskan, Gurpreet Singh, Jaspreet Singh, and Chander Prabha. „Data Visualization and its Key Fundamentals: A Comprehensive Survey“. In: *7th International Conference on Communication and Electronics Systems, ICCES 2022 - Proceedings* (2022), pp. 1710–1714. DOI: 10.1109/ICCES54183.2022.9835803.

[52] Giang Nguyen-Truong, Hong Jin Kang, David Lo, Abhishek Sharma, Andrew E. Santosa, Asankhaya Sharma, and Ming Yi Ang. „HERMES: Using Commit-Issue Linking to Detect Vulnerability-Fixing Commits“. In: *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Mar. 2022, pp. 51–62. ISBN: 978-1-6654-3786-8. DOI: 10.1109/SANER53432.2022.00018. URL: https://ieeexplore.ieee.org/document/9825835/.

[53] Adam O'Grady. „Issues to Merge Requests“. eng. In: *GitLab Quick Start Guide*. United Kingdom: Packt Publishing, Limited, 2018. ISBN: 9781789534344.

[54] Sofia Ouhbi and Nuno Pombo. „Software Engineering Education: Challenges and Perspectives“. In: *2020 IEEE Global Engineering Education Conference (EDUCON)*. Vol. 2020-April. IEEE, Apr. 2020, pp. 202–209. ISBN: 978-1-7281-0930-5. DOI: 10.1109/EDUCON45650.2020.9125353. URL: https://ieeexplore.ieee.org/document/9125353/.

[55] Reza M. Parizi, Paola Spoletini, and Amritraj Singh. „Measuring Team Members' Contributions in Software Engineering Projects using Git-driven Technology“. In: *2018 IEEE Frontiers in Education Conference (FIE)*. Vol. 2018-October. IEEE, Oct. 2018, pp. 1–5. ISBN: 978-1-5386-1174-6. DOI: 10.1109/FIE.2018.8658983. URL: https://ieeexplore.ieee.org/document/8658983/.

[56] Shaun Phillips, Jonathan Sillito, and Rob Walker. „Branching and merging: An investigation into current version control practices“. In: *Proceedings - International Conference on Software Engineering* (2011), pp. 9–15. ISSN: 02705257. DOI: 10.1145/1984642.1984645.

[57] Wouter Poncin, Alexander Serebrenik, and Mark Van Den Brand. „Process mining software repositories“. In: *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR* (2011), pp. 5–13. ISSN: 15345351. DOI: 10.1109/CSMR.2011.5.

[58] Project Management Institute Inc. (PMI). „A Guide to the Project Management Body of Knowledge“. eng. In: *A Guide to the Project Management Body of Knowledge (PMBOK ® Guide) – 7th Edition and The Standard for Project Management*. United States: Project Management Institute, Inc. (PMI), 2021, pp. 1–2. ISBN: 9781628256642.

[59] Christian Rodriguez-Bustos and Jairo Aponte. „How Distributed Version Control Systems impact open source software projects“. In: *IEEE International Working Conference on Mining Software Repositories* (2012), pp. 36–39. ISSN: 21601852. DOI: 10.1109/MSR.2012.6224297.

[60] Jihane Roudias. *Mastering principles and practices in PMBOK, PRINCE2, and Scrum : using essential project management methods to deliver effective and efficient projects*. eng. 1st edition. FT Press project management series. Upper Saddle River, New Jersey: Pearson Education/FT Press, 2015.

[61] Hang Ruan, Bihuan Chen, Xin Peng, and Wenyun Zhao. „DeepLink: Recovering issue-commit links based on deep learning“. In: *Journal of Systems and Software* 158 (Dec. 2019), p. 110406. ISSN: 01641212. DOI: 10.1016/j.jss.2019.110406. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121219301803.

[62] Nayan B. Ruparelia. „The history of version control“. In: *ACM SIGSOFT Software Engineering Notes* 35.1 (Jan. 2010), pp. 5–9. ISSN: 0163-5948. DOI: 10.1145/1668862.1668876. URL: http://doi.acm.org/10.1145/1668862.1668876.

[63] Hanadi Salameh. „What, When, Why, and How? A Comparison between Agile Project Management and Traditional Project Management Methods“. In: *International Journal of Management Reviews* Vol.2, (Oct. 2014). URL: https://www.eajournals.org/wp-content/uploads/What-When-Why-and-How-A-Comparison-between-Agile-Project-Management-and-Traditional-Project-Management-Methods.pdf.

[64] Andreas Schreiber, Lynn von Kurnatowski, Annika Meinecke, and Claas de Boer. „An Interactive Dashboard for Visualizing the Provenance of Software Development Processes“. In: *2021 Working Conference on Software Visualization (VISSOFT)*. IEEE, Sept. 2021, pp. 100–104. ISBN: 978-1-6654-3144-6. DOI: 10.1109/VISSOFT52517.2021.00019. URL: https://ieeexplore.ieee.org/document/9604814/.

[65] Sergey A Shershakov, Sergey S Shershakov, and Alexey A Mitsyuk. „Term Projects Workflow for Modern Software Engineering Education“. In: (2017). DOI: 10.13140/RG.2.2.23674.18889. URL: https://www.researchgate.net/publication/318494889.

[66] Maurício Souza, Renata Moreira, and Eduardo Figueiredo. „Students Perception on the use of Project-Based Learning in Software Engineering Education“. In: *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*. SBES '19. Salvador, Brazil: Association for Computing Machinery, 2019, pp. 537–546. ISBN: 9781450376518. DOI: 10.1145/3350768.3352457. URL: https://doi.org/10.1145/3350768.3352457.

[67] Diomidis Spinellis. „Git“. In: *IEEE Software* 29.3 (May 2012), pp. 100–101. ISSN: 0740-7459. DOI: 10.1109/MS.2012.61. URL: http://ieeexplore.ieee.org/document/6188603/.

[68] Yan Sun, Qing Wang, and Ye Yang. „FRLink: Improving the recovery of missing issue-commit links by revisiting file relevance“. In: *Information and Software Technology* 84 (Apr. 2017), pp. 33–47. ISSN: 09505849. DOI: 10.1016/j.infsof.2016.11.010. URL: https://linkinghub.elsevier.com/retrieve/pii/S0950584916303792.

[69] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. „How do software engineers understand code changes?: an exploratory study in industry“. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE '12. New York, NY, USA: ACM, 2012, 51:1–51:11. ISBN: 978-1-4503-1614-9. DOI: 10.1145/2393596.2393656. URL: http://doi.acm.org/10.1145/2393596.2393656.

[70] Alexander Tarvo, Thomas Zimmermann, and Jacek Czerwonka. „An integration resolution algorithm for mining multiple branches in version control systems“. In: *IEEE International Conference on Software Maintenance, ICSM* (2011), pp. 402–411. DOI: 10.1109/ICSM.2011.6080807.

[71] Mariot Tsitoara. *Beginning Git and GitHub*. eng. 1st ed. Berkeley, CA: Apress L. P, 2019. ISBN: 1484253124. DOI: 10.1007/978-1-4842-5313-7.

[72] Miroslav Tushev, Grant Williams, and Anas Mahmoud. „Using GitHub in large software engineering classes. An exploratory case study“. In: *Computer Science Education* 30.2 (Apr. 2020), pp. 155–186. ISSN: 17445175. DOI: 10.1080/08993408.2019.1696168. URL: https://www.tandfonline.com/doi/abs/10.1080/08993408.2019.1696168.

[73] Andric Valdez, Hanna Oktaba, Helena Gomez, and Aurora Vizcaino. „Sentiment Analysis in Jira Software Repositories“. In: *2020 8th International Conference in Software Engineering Research and Innovation (CONISOFT)*. IEEE, Nov. 2020, pp. 254–259. ISBN: 978-1-7281-8450-0. DOI: 10.1109/CONISOFT50191.2020.00043. URL: https://ieeexplore.ieee.org/document/9307811/.

[74] Joakim Verona. „Issue Tracking“. eng. In: *Practical DevOps*. United Kingdom: Packt Publishing, Limited, 2016. ISBN: 9781785882876.

[75] Roel J. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. eng. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. ISBN: 978-3-662-43838-1. DOI: 10.1007/978-3-662-43839-8. URL: https://link.springer.com/10.1007/978-3-662-43839-8.

[76] Haitham Yaish and Madhu Goyal. „A multi-tenant database architecture design for software applications“. In: *Proceedings - 16th IEEE International Conference on Computational Science and Engineering, CSE 2013* (2013), pp. 933–940. DOI: 10.1109/CSE.2013.139.

[77] Fiorella Zampetti, Salvatore Geremia, Gabriele Bavota, and Massimiliano Di Penta. „CI/CD Pipelines Evolution and Restructuring: A Qualitative and Quantitative Study“. In: *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2021, pp. 471–482. DOI: 10.1109/ICSME52107.2021.00048.

[78] Fengmin Zhu, Xingyu Xie, Dongyu Feng, Na Meng, and Fei He. „On the methodology of three-way structured merge in version control systems: Top-down, bottom-up, or both“. In: *Journal of Systems Architecture* 145 (Dec. 2023), p. 103011. ISSN: 1383-7621. DOI: 10.1016/J.SYSARC.2023.103011.

[79] Nazatul Nurlisa Zolkifli, Amir Ngah, and Aziz Deraman. „Version Control System: A Review“. In: *Procedia Computer Science* 135 (Jan. 2018), pp. 408–415. ISSN: 1877-0509. DOI: 10.1016/J.PROCS.2018.08.191.

Online References

[33] *Git Glossary*. Accessed: 23.08.2024. URL: https://git-scm.com/docs/gitglossary.

[34] *GitLab Merge Request*. Accessed: 23.08.2024. URL: https://docs.gitlab.com/ee/user/project/merge_requests/.

[46] *Manifesto for Agile Software Development*. Accessed: 23.08.2024. URL: https://agilemanifesto.org/.

Appendix

Semi-Structured Expert Interview Questionnaire

Semi-Structured Expert Interviews

Demographics/Experiences

1. How old are you?

   Mark only one oval.

   - ◯ 18-24 years
   - ◯ 25-34 years
   - ◯ 35-44 years
   - ◯ 45-54 years
   - ◯ 55 years or older

2. What is your gender?

   Mark only one oval.

   - ◯ Female
   - ◯ Male
   - ◯ Other: _____

3. How much experience do you have in software engineering?

   Mark only one oval.

   - ◯ < 1 year
   - ◯ 1-5 years
   - ◯ 6-10 years
   - ◯ 11-15 years
   - ◯ > 15 years

4. How much experience do you have in software engineering education?

   Mark only one oval.

   - ◯ < 1 year
   - ◯ 1-5 years
   - ◯ 6-10 years
   - ◯ 11-15 years
   - ◯ > 15 years

5. How much experience do you have with software issue tracking systems?

   Mark only one oval.

   - ◯ < 1 year
   - ◯ 1-5 years
   - ◯ 6-10 years
   - ◯ 11-15 years
   - ◯ > 15 years

6. How much experience do you have with version control systems?

Mark only one oval.

- ◯ < 1 year
- ◯ 1-5 years
- ◯ 6-10 years
- ◯ 11-15 years
- ◯ > 15 years

7. What tools have you used so far?

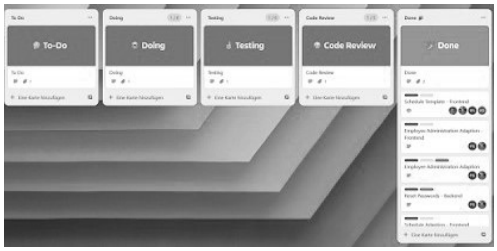
Tick all that apply.

- ☐ JIRA
- ☐ GitHub
- ☐ GitLab
- ☐ Redmine
- ☐ Bugzilla
- ☐ nTask
- ☐ YouTrack
- ☐ Other: _____

Importance Evaluation

8. State of the art

Consider the following issue board. This issue board was used for the development of a project for a university course. How useful is the presented information for you?

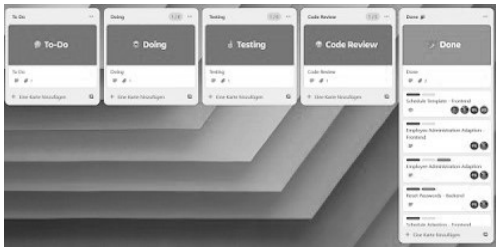


Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

9. State of the Art

Consider the same issue board as above. How useful is it to obtain the historical software issue data for a project?

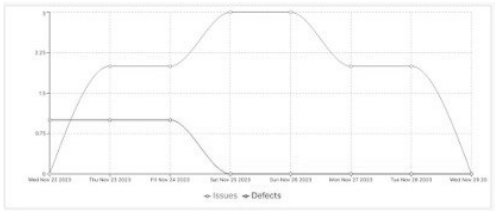


Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

10. Consider the following diagram. How useful is the information presented to you?

The following diagram illustrates the current number of software issues. This includes software issues which are in the following status: Open, In Progress, Test. The number of issues is shown on the y-axis and the time when the issue was created is shown on the x-axis

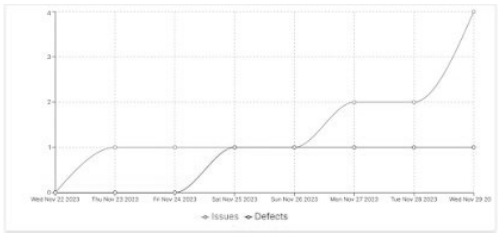


Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

11. Consider the following diagram. How useful is the information presented to you?

The following diagram illustrates the cumulative number of issues resolved in the course of a project. The number of issues is shown on the y-axis and the time when the issue was resolved is shown on the x-axis

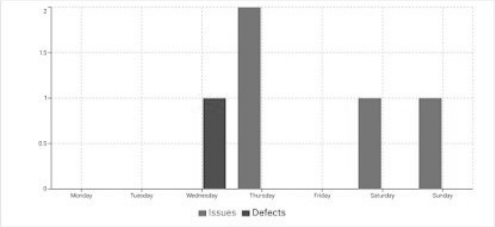


Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

12. Consider the following diagram. How useful is the information presented to you?

The following diagram illustrates the number of issues created per weekday.

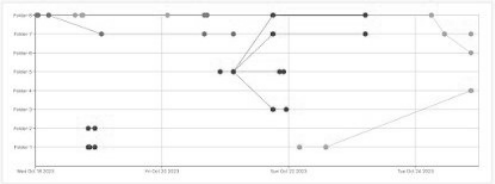


Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

13. Consider the following diagram. How useful is the information presented when it comes to finding out in which parts of the project many changes have occurred?

This diagram illustrates the local distribution of software issues in a project. The location where a change occurred while working on a software issue is represented by the y-axis, and the time at which this occurred is shown on the x-axis. Each issue has its own color.

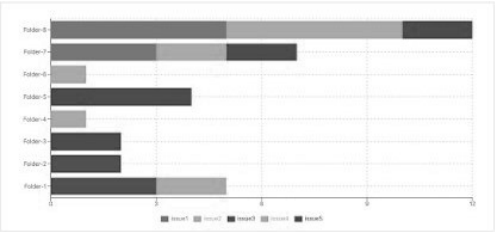


Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

14. Consider the following diagram. How useful is the information presented to you?

This diagram visualizes the distribution of issues within a project. The locations are shown on the y-axis and the number of commits on the x-axis. A bar represents the total number of commits at a location and consists of several stacked smaller bars that represent the number of commits for an issue, with each issue having a different color.

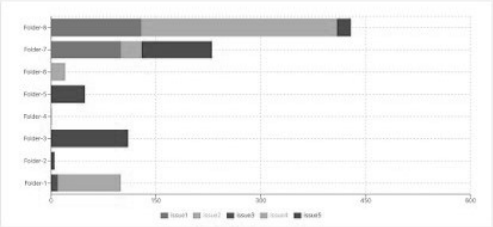


Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

15. Consider the following diagram. How useful is the information presented to you?

This diagram illustrates the distribution of issues within a project. The y-axis shows the locations, and the x-axis shows the time spent on the commits. A bar consists of several stacked smaller bars representing the time spent on one issue, with each issue having a different color.

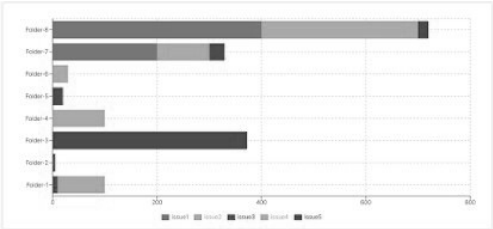


Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

16. Consider the following diagram. How useful is the information presented to you?

This diagram visualizes the distribution of issues within a project. The y-axis shows the locations, and the x-axis shows the number of line changes. A bar consists of several stacked smaller bars that represent the number of line changes for an issue, whereby each issue has a different color.



Mark only one oval.

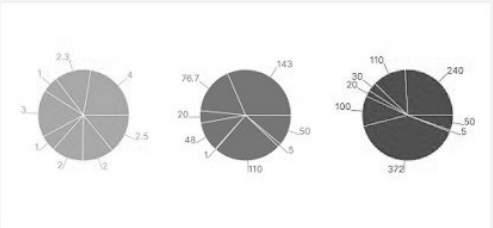
	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

17. Consider the following diagram. How useful is the information presented to you?

The first pie chart shows the average number of commits per location, where one location corresponds to one segment in the pie chart.

The second pie chart shows the average number of time units per location, where one location corresponds to one segment in the pie chart.

The third pie chart shows the average number of changes per location, where one location corresponds to one segment in the pie chart.

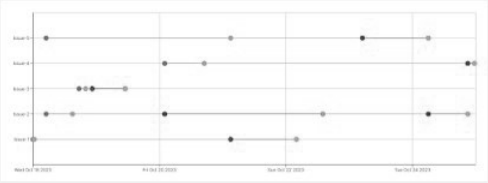


Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

18. Consider the following diagram. How useful is the information presented to you?

The following diagram visualizes the status changes of the software issues. The issues are shown on the y-axis, and the time when a status change occurred is shown on the x-axis. Each point represents a status change, and each status has an assigned color.



Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

19. Consider the following diagram. How useful is the information presented to you?

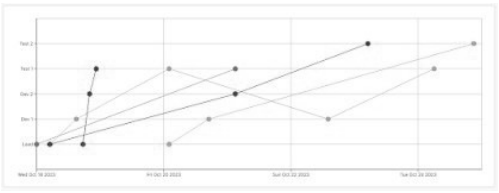
The following diagram illustrates the average duration for which a software issue is in a certain status. Furthermore, this view offers the possibility of comparing the duration of changes and defects.

Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

20. Consider the following diagram. How useful is the information presented to you?

The following diagram illustrates the contributors of a software issue. The y-axis shows the contributors, and the x-axis shows the time when another developer contributed to a software issue. Each issue is represented by a color.

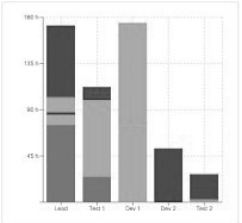


Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

21. Consider the following diagram. How useful is the information presented to you?

This diagram visualizes the time the team members have spent on the project. The bar represents the total time and consists of sub-bars that describe the time spent on an issue. It can be seen which issues a person was involved in.



Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

22. Consider the following diagram. How useful is the information presented to you?

This diagram illustrates the average time a contributor spends on a software issue. In addition, this view offers the possibility of comparing the time spent on changes and errors.

Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

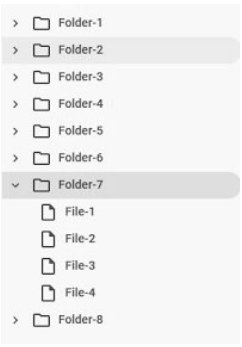
23. Do you have any suggestions for possible improvements to the previous graphs?

_____

24. Are there other statistical metrics that you consider necessary for the historical analysis of software issue?

_____

25. How useful do you find an option that allows you to display software issues for a specific folder, package or file?



Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

26. How useful is an option to filter by functional requests/software defects?

Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

27. How useful is an option to filter for a specific amount of software issues to only show information about them?

Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

28. How useful is an option to filter software issues by a specific date range?

Mark only one oval.

	1	2	3	4	5	
No ι	◯	◯	◯	◯	◯	Very useful

29. How useful is an option to filter for software issues that got reopened again?

Mark only one oval.

	1	2	3	4	5	
No ι	◯	◯	◯	◯	◯	Very useful

30. How useful is an option to filter software issues by the number of commits?

Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

31. How useful is an option to filter software issues by spent time?

Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

32. How useful is an option to filter software issues by changed code lines?

Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

33. How useful is an option that makes it possible to compare two filtered versions of the project?

Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

34. Do you think there should be additional filtering options available?

_____

Scenario-Based Evaluation Questionnaire

Scenario Based Evaluation

Demographics/Experiences

1. How old are you?

   Mark only one oval.

   - ◯ 18-24 years
   - ◯ 25-34 years
   - ◯ 35-44 years
   - ◯ 45-54 years
   - ◯ 55 years or older

2. What is your gender?

   Mark only one oval.

   - ◯ Female
   - ◯ Male
   - ◯ Other: _____

3. How much experience do you have in software engineering?

   Mark only one oval.

   - ◯ < 1 year
   - ◯ 1-5 years
   - ◯ 6-10 years
   - ◯ 11-15 years
   - ◯ > 15 years

4. How much experience do you have in software engineering education?

   Mark only one oval.

   - ◯ < 1 year
   - ◯ 1-5 years
   - ◯ 6-10 years
   - ◯ 11-15 years
   - ◯ > 15 years

5. How much experience do you have with software issue tracking systems?

   Mark only one oval.

   - ◯ < 1 year
   - ◯ 1-5 years
   - ◯ 6-10 years
   - ◯ 11-15 years
   - ◯ > 15 years

6. How much experience do you have with version control systems?

Mark only one oval.

- ◯ < 1 year
- ◯ 1-5 years
- ◯ 6-10 years
- ◯ 11-15 years
- ◯ > 15 years

Scenario: Detailed information about a software issue

In this scenario, we are looking at a single issue ticket within a project. We want to examine which files were added/modified by this issue, what issue status the issue was in, and who was involved in resolving the issue. To answer these questions, use the charts in the "Overview" section.

Project: SEPM_INSO_01
Issue: 3

7. During the implementation of Issue 3, how many commits were created and referenced?

_____

8. Which files were changed by Issue 3?

_____

9. What issue status did Issue 3 have?

_____

10. How long did Issue 3 remain in status "Testing"?

_____

11. Which developers contributed changes to Issue 3?

_____

12. Is there a developer who has only tracked time, but has not made any other contribution to Issue 3?

_____

13. How would you obtain this information without the developed prototype?

_____

14. How useful do you rate the prototype for the collection of this information?

Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

15. How useful do you rate such a scenario?

Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

Scenario: Overview of software issue distribution

In this scenario, we want to get an overview of the local distribution software issues in a project.The line and bar chart in the "Local Distribution" section are intended to answer the questions below.

Project: SEPM_INSO_01
Date Range: 05/01/2020 - 05/31/2020
Folder: backend/src/main/.../service/impl

16. Based on the number of lines of code added and removed, which of the files has changed the most?

_____

17. Excluding the files that had no changes, which of the files has changed the least?

_____

18. Which file was changed by the largest number of issues?

_____

19. Regarding the previous file, which issues caused its changes?

_____

20. Which issue has changed the most files?

_____

21. How much time has passed between the first commit and the last commit of this issue?

_____

22. How would you obtain this information without the developed prototype?

_____

23. How useful do you rate the prototype for the collection of this information?

_____

24. How useful do you rate such a scenario?

_____

Scenario: Overview of software issue contributions

In this scenario, we want to get an overview of the involvement of the project team members. We want to estimate how the work has been distributed among the team. To answer the following questions, use the charts in the "Contribution" section.

Project: SEPM_INSO_01
Date Range: 05/01/2020 - 05/31/2020

25. Which team member was involved in the most issues?

_____

26. Which team member was involved in the least issues?

_____

27. Which team member has tracked the most time?

_____

28. Which team member has tracked the least time?

_____

29. Which issue had the most time tracked? (Use the "Spent Time" filter)

_____

30. Which person tracked the most time on the previous issue?

_____

31. Regarding the previous person, did that person also contribute the most changes?

_____

32. How would you obtain this information without the developed prototype?

_____

33. How useful do you rate the prototype for the collection of this information?

Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

34. How useful do you rate such a scenario?

Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

Scenario: Overview of software issue status changes

Overview about the status changes of software issues

In this scenario, we want to look at the issue status changes of software issues in a project. We want to estimate how long a software issue has been in a particular state. To answer the questions below the status changes line chart should be used.

Project:
SEPM_INSO_01
Issue Types: ISSUE, DEFECTS
Milestones: Sprint#7

35. How long did it take for issues of type "Defect" to move from status "Open" to status "Closed"?

36. How long did it take for issues of type "Issue" to move from status "Open" to status "Closed"?

37. How would you obtain this information without the developed prototype?

38. How useful do you rate the prototype for the collection of this information?

Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

39. How useful do you rate such a scenario?

Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

Scenario: Software issue comparison

In this scenario, we want to compare two filtered versions of a project. We want to analyze the distribution of issues, status changes, and issue contribution. The isolated graphs from the tab above are intended to answer the following questions.

Project:
SEPM_INSO_01
Version 1 - Milestones: Sprint1, Sprint2

Version 2 - Milestones: Sprint3, Sprint4
Folder: frontend/src/app/components

40. Which component received the most changes in "Sprint1" and "Sprint2"?

41. Which component received the most changes in "Sprint3" and "Sprint4"?

42. How was the distribution of work between the team members in "Sprint1" and "Sprint2"?

43. How was the distribution of work between the team members in "Sprint3" and "Sprint4"?

44. How would you obtain this information without the developed prototype?

45. How useful do you rate the prototype for the collection of this information?

Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

46. How useful do you rate such a scenario?

Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful