TU WIEN Informatics

Visualisierungsgestützte Konfliktwahrnehmung in der verzweigten und projektübergreifenden Softwareentwicklung

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Christina Greil, BSc
Matrikelnummer 01426862

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Thomas Grechenig

Wien, 30. November 2021

Unterschrift Verfasserin

Unterschrift Betreuung

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

TU WIEN Informatics

Conflict Awareness by Visualisation in Multi-Branch and Multi-Project Software Development

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Software Engineering & Internet Computing

by

Christina Greil, BSc
Registration Number 01426862

to the Faculty of Informatics

at the TU Wien

Advisor: Thomas Grechenig

Vienna, 30th November, 2021

Signature Author

Signature Advisor

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

Visualisierungsgestützte Konfliktwahrnehmung in der verzweigten und projektübergreifenden Softwareentwicklung

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Christina Greil, BSc
Matrikelnummer 01426862

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, 30. November 2021

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

Erklärung zur Verfassung der Arbeit

Christina Greil, 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, 30. November 2021

Christina Greil

Kurzfassung

Das Zusammenführen von gleichzeitig durchgeführten Softwareänderungen in gleichen oder verschiedenen Projekten kann zu Konflikten führen, die von den Entwicklern gelöst werden müssen. Dies kann fehlerhaft und zeitaufwendig sein und die Entwickler bei ihrer Arbeit stören. Das frühzeitige Auffinden der Konflikte kann diese Probleme minimieren. Existierende Tools stellen nur Informationen zum Konfliktbewusstsein für die Softwareentwicklung innerhalb eines Projektes bereit. Daher wurde eine prototypische Visualisierung zur verbesserten Wahrnehmung von Konflikten für die Softwareentwicklung innerhalb eines Projektes, aber auch für die projektübergreifende Softwareentwicklung, erstellt.

Diese Diplomarbeit erforscht, welcher Informationsbedarf bezüglich Konfliktbewusstsein in Softwareprojekten für Entwickler besteht, wie Entwickler die vorgesehenen Funktionen priorisieren, wie zielführend Entwickler die vorgeschlagene Visualisierung des Konfliktbewusstseins bewerten, wie sinnvoll die vorgeschlagene Visualisierung des Awareness-Tools ist und wie effizient das Tool im Vergleich zu modernen Methoden ist.

Zuerst wurde eine Literatur- und Toolrecherche durchgeführt, um den bestehenden Informationsbedarf der Entwickler festzustellen. Basierend auf diesen Erkenntnissen und den eigenen Erfahrungen in der Entwicklung wurden einige erste Funktionen und Mock-ups des Prototyps vorgeschlagen. Diese wurden in semi-strukturieren Interviews von Entwicklerinnen und Entwickler priorisiert. Anhand dieser Funktionen wurde der Prototyp entwickelt. Szenariobasierte Expertenbewertungen wurden durchgeführt, um die Zweckmäßigkeit der Idee selbst, die Zweckmäßigkeit der gewählten Visualisierung und die Zweckmäßigkeit des Prototypen zu evaluieren.

Die befragten Entwickler sahen Potenzial in der Idee und auch in der gewählten Visualisierung. Allerdings müssen Performance- und Usability-Probleme behoben werden. Die Entwickler bewerteten die Funktionen des Prototyps ebenfalls als zweckdienlich, wünschten sich jedoch eine mehr passive und automatische Informationsbeschaffung und Möglichkeiten, sie in bestehende Tools zu integrieren.

Schlüsselwörter

Merge Konflikte, Softwarevisualisierung, Verzweigte Softwareentwicklung, Projektübergreifende Softwareentwicklung

Abstract

Merging simultaneously implemented software changes in the same project or in different projects together can create conflicts that must be resolved by the developers by hand. The changes can be made within a project or in a parent and one of its fork. This can be erroneous, time-consuming and disrupt the work of software engineers. Finding such conflicts in an early stage can minimise these merging problems. Although some tools exist that provide conflict awareness information, they are restricted to multi-branch software development only. Therefore, a prototypical conflict awareness visualisation for multi-branch but also multi-project software development was created.

This thesis investigates the conflict awareness information needs that exist in software projects for developers, how developers prioritise the envisioned features, how purposeful developers rate the proposed conflict awareness visualisation, how purposeful the proposed visualisation of the awareness tool is and how efficient the tool is compared to state-of-the-art methods.

First, a literature and tool research was carried out in order to determine the existing information needs of the developers. Based on these findings and the own experience in development, some initial features and mock-ups of the prototype were suggested. These functionalities were prioritised in semi-structured interviews by developers. Based on these prioritised features, the prototype was developed. The purposefulness of the idea itself, the chosen visualisation and the prototype itself were evaluated in scenario-based expert evaluations.

The interviewed developers saw potential in the idea and in the chosen visualisation. Performance and usability issues need to be addressed. The developers also rated the features of the prototype as purposeful, but a more passive and automatic information gathering and ways to integrate it into existing tools were requested.

Keywords

Merge Conflicts, Software Visualisation, Multi-Branch Software Development, Multi-Project Software Development

Contents

CHAPTER 1

Introduction

This chapter provides an overview of the problem tackled by this thesis and the motivation why this problem is addressed. Additionally, this chapter shows the aim of the work including the research questions that are answered, the methodology and the structure of the following chapters.

1.1 Problem Description

Nelson et al. [46] stated that in collaborative software development, merge conflicts can arise when developers change the same code in isolation without being aware of it. The authors noted when merging the changes together, the underlying Version Control System (VCS) may be able to resolve the conflicting parts on its own, although this is not the case every time. In such cases, the developers need to resolve the created conflict on their own, which is not always a trivial task. The merge can be time-consuming, can disrupt the workflow of the developers and can be erroneous.

Merge Conflicts may be still small if detected in an early stage. This reduces the amount of work that may have to be redone and decreases the time and effort in resolving the conflicts [53]. Another benefit of finding merge conflicts early is that developers may still have a better knowledge of why and how the previous changes were made [45, 46]. This knowledge can help in resolving the conflict faster and with less side effects. If this knowledge is fading away, it may also be harder to find out which impacts these previous changes had on other code fragments in the project [45].

Current tools only tackle these conflicts by providing an awareness visualisation for multi-branch environments (see Chapter 3). Multi-project environments are not included in these visualisations, although fork-based development is often used for contribution within the GitHub[1] community [38, 44]. Nowadays, forks also have a better reputation

[1] https://github.com/

than in the past [51, 64]. On the contrary, Robles and González-Barahona [52] stated that merging back into the original project is low. The authors found out that one reason for this is the complexity of this task. The mentioned tools also do not show the conflict awareness restricted to specific commits or restricted to commits of a specific feature or a specific incident. This information can come in handy for cherry picking commits from another branch within a project or from a fork.

That being said, the increased positive view on forks and the used contribution style in GitHub makes the conflict awareness interesting research for both multi-branch and multi-project software development. Further research can help in reducing the above-mentioned problems of merge conflicts for both environments.

1.2 Aim of the Work

The aim of this work is to develop a prototype for a conflict awareness visualisation for multi-branch and multi-project software development. The prototype should increase the efficiency of merges, rebases and cherry picks and provide divergence visibility (for example when did branches drift apart that much such that they can no longer be merged without conflicts). The tool should also be able to support non-developers, for instance release managers should be able to check if a feature is already integrated in a specific branch (see proposed features below).

The proposed main features of this prototype are:

- Visualisation showing the divergence between two forks
  - How much does the fork diverge from its parent?
- Conflict visualisation of branch integration of the same project or of a fork:
  - Is it possible to merge or rebase a branch of the same project or from a fork without possible conflicts?
  - Which commits may cause conflicts and what is the cause?
- Conflict visualisation of a cherry pick of a specific commit:
  - Is it possible to cherry pick a specific commit (like a bugfix or a feature integration) from another branch or fork, or is it dependent on other commits?
  - Which other commits must also be cherry picked?
- Conflict visualisation of a feature or bugfix integration:
  - Which commits must be cherry picked to integrate a feature or a bugfix?

Overall, this thesis will answer the following research questions:

RQ1: What conflict awareness information needs exist in software projects for developers?

RQ2: How do developers prioritise the envisioned features?

RQ3: a) How purposeful do developers rate the proposed conflict awareness visualisation?

b) How purposeful is the proposed visualisation of the awareness tool?

c) How efficient is the tool compared to state-of-the-art methods?

1.3 Methodology

The following section describes the methods that were used for this thesis. Based on the engineering cycle, described by Wieringa in [60], five methods were selected. For the problem investigation a literature and tool research (Section 1.3.1) was conducted. The treatment design phase consisted of a conceptual design phase (Section 1.3.2) and the treatment validation was done with semi-structured expert interviews (Section 1.3.3). The treatment implementation consisted of the interactive prototype development (Section 1.3.4) and the scenario-based expert evaluations (Section 1.3.5) were chosen for the implementation evaluation.

1.3.1 Literature and Tool Research

The literature and tool research provided insights of different information needs of developers and which studies and tools already exists in this area. For the research mostly the databases of ACM[2] and IEEE Xplore[3] was used. Additional papers were also found by using the cited sources of the found research papers. Afterwards the relevant research papers were identified and sorted after their subject areas like „tools“ or „information needs“. The results were then be used to answer RQ1.

1.3.2 Conceptual Design

Based on the findings of the literature and tool research (Section 1.3.1) and the own experiences in software engineering some initial features of the prototype were proposed. For these features a concept of the prototype was designed by creating mock-ups of the potential visualisation. These mock-ups can be seen in Chapter 4.

1.3.3 Semi-Structured Expert Interviews

In order to get an impression on the usefulness of these proposed features and to answer RQ1 and RQ2, semi-structured expert interviews were conducted. As a guideline and support, a questionnaire was created via Google Forms[4] which was filled during the

[2]https://dl.acm.org/

[3]https://ieeexplore.ieee.org/Xplore/home.jsp

[4]https://www.google.com/intl/de_at/forms/about/

interview. This qualitative method was chosen to combine the simplicity of questionnaires with the ability to get an insight into the thoughts of the interview participants. This would not be possible with a quantitative research method like a questionnaire alone. The results of the interviews should also act as a roadmap for the implementation of the features and to provide an answer to RQ2. Features which are rated more important should be implemented earlier than lower rated features. More details about these interviews, including the results and threats to validity can be found in Chapter 5.

1.3.4 Iterative Prototype Development contributing to Binocular

To provide a possibility to evaluate the predefined ideas to tackle some of the current information needs of software engineers a prototype with the identified features was created. The prototype was integrated into the Binocular[5] project which already provides time-oriented information for developers in different visualisations. The implementation process was structured into multiple iterations (see Chapter 6). In each iteration additional functionality or improvements of existing features were added.

1.3.5 Scenario-based Expert Evaluations

After completing the implementation phase, the visualisation and the implementation of the tool were evaluated for their suitability using scenario-based expert evaluations. During the evaluations, developers had to solve exercises using the prototype. The results (see Chapter 7) of the evaluations were used to answer RQ3. After each scenario, the participants were asked about their experiences with the tool (RQ3 b)), about the purposefulness of the visualisation (RQ3 a)) and how they would have solved the scenario without the prototype (RQ3 c)). To minimise incorrect results due to incorrect handling of the prototype, the participants are familiarised with the tool before executing the scenarios.

1.4 Structure of the Work

Chapter 2 provides an overview of important terms and concepts used for this thesis. In Chapter 3 existing awareness tools and differences to the implemented prototype are introduced. The results of the conceptual design can be found in Chapter 4 and the semi-structured expert interviews that followed afterwards are presented in Chapter 5. Chapter 6 describes the implementation phase in more detail. The structure and results of the evaluations can be found in Chapter 7 and the defined research questions are answered in Chapter 8. In the last chapter the conclusion and the future work are presented.

[5] https://github.com/INSO-TUWien/Binocular

CHAPTER 2

Fundamentals

This chapter gives an overview of important terms and concepts that are used in this thesis. At first key information about VCSs and elements around them are introduced and explained. After that details about the merge concept and merge conflicts are covered, followed by information about forks. Lastly a short overview of issues is provided.

2.1 Version Control System

In [18] VCSs (also called revision control systems [2]) are described as systems which enable versioning for a file or a set of files. Using such systems, people can keep track of changes over time, compare these changes and inspect the file's history, for example to find out who adapted what. If changes went wrong, it is easy to revert these modifications and jump back to a previous version of a file or the whole project.

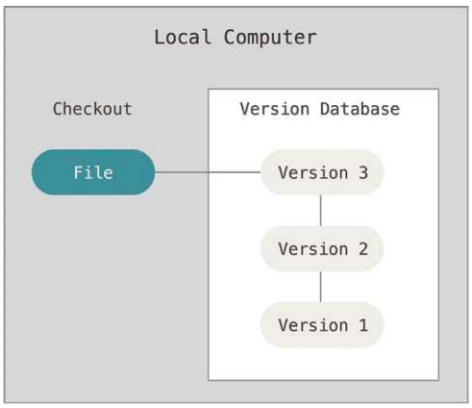
2.1.1 Types

In the book *Pro Git* [18], VCSs are categorised into three types: Local VCSs, Centralised Version Control Systems (CVCSs) and Distributed Version Control Systems (DVCSs). In this thesis the main focus lies on the DVCS Git[1].

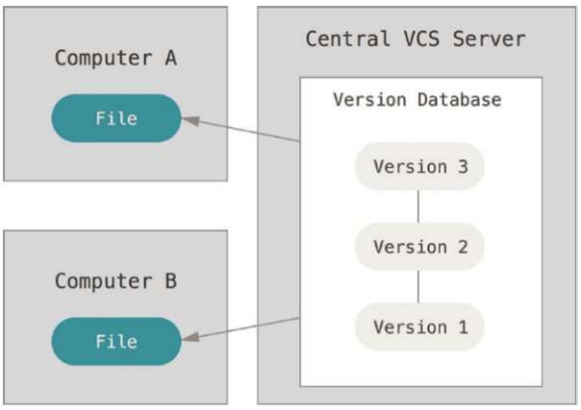
Local Version Control System

Chacon and Straub [18] described that an easy and common way to control the version of a project is to just copy the files into another directory. But the authors also noted that this approach is extremely error prone. If someone accidentally works in the wrong folder, versions can be overwritten easily. To mitigate this disadvantage local VCSs like RCS [58] were introduced in the past. The authors describe a local VCS as system with

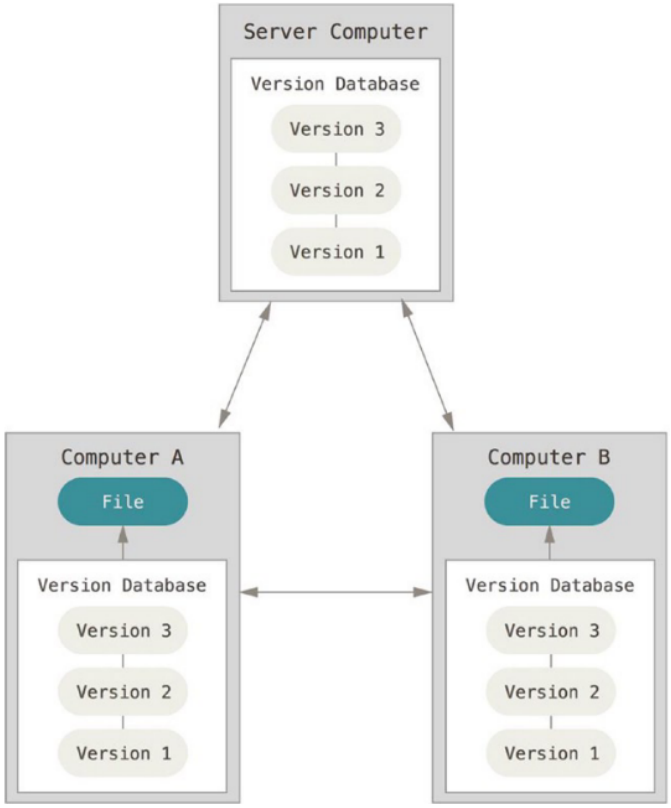
[1] https://git-scm.com/



(a) Local version control [18]



(b) Centralised version control [18]



(c) Distributed version control [18]

Figure 2.1: Version control types

a simple database which revisions all the file changes that are made. The basic principle of this VCSs type can be seen in Figure 2.1a.

Centralised Version Control System

A problem of local VCSs is that they are not useful in collaborative development [18]. Therefore, CVCSs like Subversion[2] and CVS [4, 17] were developed. Several authors ([5, 18, 24]) describe CVCSs as systems that consist of one central server and multiple clients (see Figure 2.1b). The server stores all the versioned files, and the clients check out these files from the server which can be changed afterwards. The files represent a snapshot of the moment on which the checkout was made.

Distributed Version Control System

In [5, 18, 24, 57] it is mentioned that DVCSs like Git and Mercurial[3] also have a server and multiple clients. But the difference to CVCSs is that the whole repository is mirrored on the client, and not only the latest file snapshots are checked out. Therefore, the complete history (see Section 2.1.4) will also be copied to the client machine. This concept is shown in Figure 2.1c. The local repository allows developers to work locally and in isolation [5, 12, 18, 24].

2.1.2 Commit

A commit bundles changes into one entity [12, 18, 31]. In [18, 25, 31] a commit in Git is refered to a snapshot of the whole repository, as a particular point in the history. Therefore, not only the changed files are saved like in Subversion. Additionally, commits contain metadata like a commit message which describes the changes [18, 31, 37]. Each commit is linked with its parent commits, which constructs a graph structure, the commit graph [25, 31].

2.1.3 Hunk

Hunks, as described in [18], are adjacent line changes and are bundled in a commit. If, for example, two changes are made to a file at different positions, two hunks exist. In Git it is not only possible to add all hunks in a commit, but also only selected ones.

2.1.4 History

The history of a software is described as the combination of all commits [19, 31]. This also includes non-committed changes [19]. So basically, the history is the commit graph (see Section 2.1.2) inclusive all current changes which have not yet been added.

[2] https://subversion.apache.org/
[3] https://www.mercurial-scm.org/

2.1.5 Branch

In [18, 31] a branch is characterised as a conceptual line of development. It is also stated that a repository can have multiple branches and that checked out branches are then called working trees. Additionally, the working tree can contain local changes which are not committed yet. The head of a branch is the most recent commit of this line [18, 31]. De Rosso and Jackson [25] used a different description of a branch. Technically, a branch is only a pointer to this specific commit, called the head. Switching branches enables to work on other tasks. If it is possible, uncommitted changes will be taken to the other branch. When the changes are conflicting with the new code base, the switch can be prevented. In order to keep the changes, the developer must stash them which means saving the current versions to another storage area.

2.1.6 Fetch, Pull and Push

Synchronising repositories is done with fetches, pulls and pushes [18, 25, 31]. A fetch checks whether the remote repository has commits, which are missing in the local repository. A pull performs a fetch and, if the local repository has to be updated, subsequently performs a merge to introduce the updates. A push on the other hand is used to bring local commits to the remote repository.

2.1.7 Cherry Pick

With a Cherry Pick a specific subset of changes or specific commits are put on top of the current branch [18, 25, 31]. These changes are introduced as a new commit and will be the new head of the branch.

2.1.8 Rebase

In a rebase all commits of one branch are re-applied to another branch [18, 25, 31]. It can be seen as a series of cherry picks of all commits of the selected branch. When rebasing a branch, the commits can be squashed, meaning that all commits will be combined to only one commit, or a commit can be split into multiple ones [25].

2.1.9 Pull Request

Pull Requests (PRs) are mechanisms that allow the synchronisation between a fork and its original project [18, 33, 39, 44]. A PR notifies the maintainers that changes should be merged into the main line of development.

2.2 Merge

As stated in [18, 25, 31, 45], in collaborative software development developers usually work in isolation within their own code base. Periodically, these isolated changes are synchronised into the main line of development or another branch by applying the changes

Revision SIZE

```
1  import java.util.LinkedList;
2  public class Stack<T>
       implements Cloneable {
3    private LinkedList<T> items =
         new LinkedList<T>();
4    public void push(T item) {
5      items.addFirst(item);
6    }
7    public int size() {
8      return items.size();
9    }
10   public T pop() {
11     if(items.size() > 0) return
           items.removeFirst();
12     else return null;
13   }
14 }
```

Revision TOP

```
1  import java.util.LinkedList;
2  public class Stack<T>
       implements Cloneable {
3    private LinkedList<T> items =
         new LinkedList<T>();
4    public void push(T item) {
5      items.addFirst(item);
6    }
7    public T top() {
8      return items.getFirst();
9    }
10   public T pop() {
11     if(items.size() > 0) return
           items.removeFirst();
12     else return null;
13   }
14 }
```

(a) Java Method Additions on same Code Lines, but on different Branches [2]

$merge_{unstructured}$(TOP, STACK, SIZE)

```
1  import java.util.LinkedList;
2  public class Stack<T> implements Cloneable {
3    private LinkedList<T> items = new LinkedList<T>();
4    public void push(T item) {
5      items.addFirst(item);
6    }
7  <<<<<<< Top/Stack.java
8    public T top() {
9      return items.getFirst();
10   }
11 =======
12   public int size() {
13     return items.size();
14   }
15 >>>>>>> Size/Stack.java
16   public T pop() {
17     if(items.size() > 0) return items.removeFirst();
18     else return null;
19   }
20 }
```

(b) Resulting Conflict in an unstructured Merge [2]

Figure 2.2: Unstructured Merge Conflict Example [2]

into it, resulting in a new merge commit. A merge commit has two or more parents instead of only one like a normal commit [25]. This procedure is called merging. If the destination branch contains no other commits, Git only needs to update the pointer without the need of a merge commit which is then called a fast-forward [18, 31].

2.2.1 Types of Merging

Merging can be divided into three types: unstructured merging, structured merging and semi-structured merging [2]. In this thesis the focus lies in unstructured merging.

Unstructured Merging

Unstructured merging is solely done with plain text and is used by systems like Git, Subversion and CVS [2]. An advantage, mentioned in [2], of this approach is that it is typically language independent. However, the authors also state that a disadvantage is the missing knowledge of the software's structure. An example can be found in [2]: If two developers add a Java method on the same code lines (like seen in Figure 2.2a) in different branches, the merge of these will then result in a conflict (shown in Figure 2.2b). This is due to the missing knowledge of the language's structure. For a developer the merge is quite easy because he/she knows that both elements are Java methods and that the code fragments just need to be concatenated.

Structured Merging

The authors in [2, 15, 16] state that structured merge tools gain knowledge by Abstract Syntax Trees (ASTs) or other similar representations. So, they do not rely on the code lines for the merge. A merge of the modifications in Figure 2.2a would be resolved by a

structured merge tool, which is built for Java code. Like the developer the tool knows that the entities are methods and that they can be merged without a conflict. A downside of this approach is that the language independence from unstructural merging is lost.

Semi-Structured Merging

Semi-structured merging is a combination of unstructured merging and structured merging [2, 15, 16]. The authors mentioned that language grammars are used to transform software artefacts into parse trees, and conflict handlers are used to resolve specific conflict types. Usually, these grammars do not describe the whole semantics and syntax of the language, but only partially. For example, method bodies are only represented as plain text. Non-exploited elements are then merged using unstructured merging. Using this approach, it is possible to easily add grammars for new languages and handlers for new conflict types.

2.2.2 Merge Conflicts

As in Section 2.2 explained, merging is used to synchronise the individual work of collaborating developers. The commits often can be cleanly merged, but sometimes changes can overlap, causing a merge conflict, leading to broken code [45, 57]. Zimmermann [66] found out in a study that about 23% to 46% of the merges results in a merge conflict. These merge conflicts must be resolved by the developers. Estler et al. [27] stated that there is no significant difference in the number of conflicts occurring whether the contributors worked co-located or remote.

Merge Conflict Indicators

Studies [30, 43] show that merge conflicts can be caused by different changes. These changes involve changes on both code-level and project-level. Code-level changes include changes of method calls or object creations like parameter value or type adaptations, changes of assert statement expressions, additions of statements in the same area, modifications in adjacent, but different statements in the same area and the modification and removal of statements. Furthermore, changes to IF statement conditions, imports and comments may cause conflicts. Project-level changes refer to refactorings, feature introduction and enhancements, test improvements, bug fixes, framework and library removals and breaking change fixes [43].

There are also studies trying to discover some indicators of merge conflicts. The goal was to find out, if

- the number of active, diverged branches (instead of inactive, close remaining branches) [41, 50],
- a high density of commits in a short period of time (instead of a lower commit density in a larger period) [41, 50],
- the number of files changed simultaneously in both branches [41, 50],

- the number of changed files in a branch (must not be simultaneously) [50],
- the number of changed lines of code [41, 50],
- the size of the code fragments [41],
- the scattering of the changes across classes or methods [41],
- the number of changes above class level declaration (instead of changes inside a class declaration) [41],
- the number of commits in one branch [50],
- the frequency of predefined words in commit messages, like fix, bug, feature [50],
- the length of the commit messages [50],
- the development duration of a branch in hours [50]

influence the number of occurring merge conflicts. The results of the studies [41, 50] indicate that none of the proposed indicators provide an insight of the number of resulting merge conflicts. However, Owhadi-Kareshk, Nadi, and Rubin [50] also stated that the machine learning technique Random Forest [11] can be used to predict merge conflicts for different programming languages.

Accioly et al. [1] also conducted a study for conflict predictors. They wanted to find out if edits to the same method and changes to directly dependent methods can act as indicators for merge conflicts (which also includes build and test breaks in this study). The results of this study show that edits on the same methods indicate merge conflicts at a precision about 55.51% and that about 82.45% of the conflicts can be found. The changes of directly dependent methods only have a precision of 8.85% and can identify about 13.15% of the merge conflicts. Using these results, Accioly et al. suggest that both indicators may be used, if it is no problem dealing with some false positives. But if the goal is precision, then only the indicator of the same method editions should be considered.

Merge Conflict Life Cycle

Nelson et al. [46] proposed a life cycle model of a merge conflict as seen in Figure 2.3. On the left side the phases of clean code states are shown, and the right side contains the phases where the state of the code is conflicting. This proposed model consists of five phases:

- **Development phase:** The developer maintains the code by adding features, implementing bug fixes and so on.

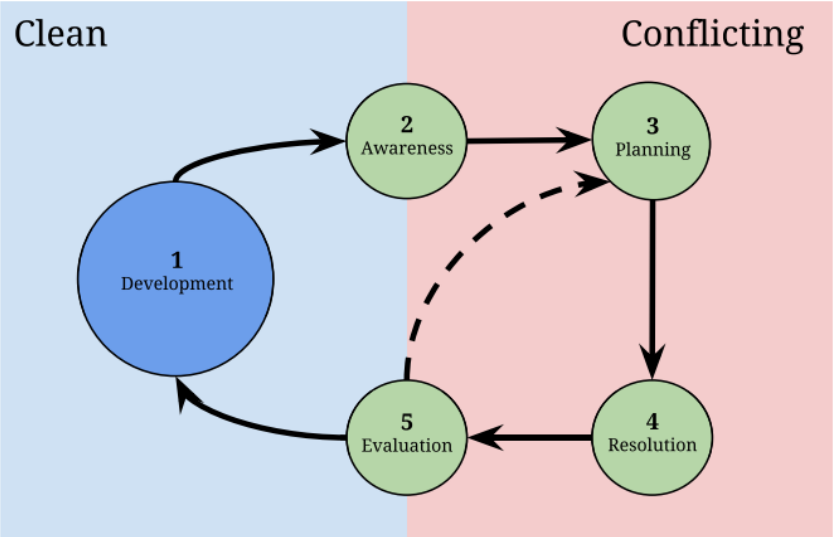


Figure 2.3: Merge Conflict Life Cycle Model [46]

- **Awareness phase:** The developer notices the merge conflict. This can either be active while, for example, monitoring the code manually or using specific tools, or passive during a pull or while merging changes. Manual steps can include holding stand-ups such that everyone knows about what the others are working on or emailing the colleagues before making breaking changes. For tool-supported awareness, developers rely on VCSs, Continuous Integration (CI) systems and code analysis tools.

- **Planning phase:** The developer became aware of the conflict and plans how to resolve it. This includes decisions like postponing the resolution or to do it immediately. It must also be decided to do the merge alone or with other developers together. The strategy may also be changed due to time limitations, deadlines or the availability of resources. Reasons for the resolution delay can be the complexity, size or ownership of the conflicting code, the number of conflicting code lines, approaching deadlines or work schedule constraints.

- **Resolution phase:** The developer planned the conflict resolution and is implementing the plan.

- **Evaluation phase:** The developer resolved the conflict and must evaluate the result. In order to proof the correctness of the merged code, steps like compiling the source code or running tests will be performed. Some developers also check if the code looks correct and if all VCS warnings are gone. Additionally, reviews can be done before the merged code is accepted. Another success criteria can be the acceptance into the production code base.

Merge Conflict Complexity

Studies [45, 46] show that the complexity of a merge conflict is estimated by the developers by their own experience. Some factors include the complexity and the number of conflicting code lines, as well as the files in the conflict, the atomicity of hunks in the conflict, the dependencies of conflicting code and non-functional changes. But also, the expertise which the developer has in the area of the conflict and the time play a role in the assessment. If the time period between the changes and the resolution of the conflict increases, the developer may no longer know exactly why and how the changes were made which can also increase the complexity of the merge [46]. But it is also worth mentioning that a merge conflict is not necessarily minor only because it is small [43].

Brindescu et al. [13] studied, if it is possible to rate the complexity of merge conflicts using machine learning. The authors found out that the Bagging (Bootstrap aggregating) [10] technique is the best one to do this job. The algorithm classified with a precision of 80%, 21% of the conflicts as severe and the remaining 79% as trivial, although the precision drops to 60% of cross-project training. The authors also looked for characteristics of a merge conflict that defines its difficulty using the feature subset selection technique with RIPPER [20] as a predictive model. They found out that the algorithm also considers the number and complexity of lines of code, the conflicting code dependencies and the atomicity of conflicting hunks as metrics for the merge conflict complexity. Additionally, the algorithm selected the number of developers and development patterns of a developer or in a branch as suitable metrics.

Nelson et al. [46] found out that the assessment of the complexity of a merge conflict can have an impact on the resolution strategy of this conflict. If it seems too hard to solve, then this may lead to a reimplementation of the changes with the current code base. However, a study in 2020 [30] indicates that mostly one version of conflicting hunks is used for the conflict resolution, and the other one is thrown away. At the other times, the two versions must be concatenated, combined, or as already mentioned above, implemented all over again.

2.3 Fork

A fork is a copy of a repository which belongs entirely to the person who forked it [18, 33, 39, 44, 63, 64]. Using forks, a fork-based development is possible.

„Fork-based development is a lightweight mechanism that allows developers to collaborate with or without explicit coordination.“ [63] The idea of fork-based development (or also called pull-based development [33]) is, that contributors simply fork an existing project and are therefore able to make changes independently [33, 63]. The authors state that after the work is finished the made changes are merged back into the original project. Gousios, Pinzger, and Deursen [33] explained that the advantage of this approach is that each potential contributor can make modifications without the need to gain write permissions on the original project. This is because the core team, which manages the

original, can then decide which changes should be integrated and which should be rejected. Nowadays, Zhou, Vasilescu, and Kästner [64, 65] state that two types of forking exist, a hard fork and a social fork.

Hard forks are referred to traditional forks in the past [64, 65]. This means a project was copied in order to start a new line of development, which often competes with the original project. A study [56] mentioned that although hard forks may never be integrated back to the original fully, the two projects may still be exchanging information. For example, it is possible that the fork needs to integrate important security patches from the original or vice versa. Several studies [29, 49, 52, 59] indicate multiple reasons for the creation of hard forks:

- **Technical (addition of functionality):** There are differences between the main developer(s) and some other developers whether a new functionality should be integrated into the project or not. [29, 49, 52, 59]
- **More community-driven development:** If the community is not considered enough by the original leaders of the project, the community may open a more community centred fork. [49, 52, 59]
- **Discontinuation of the original project:** The end of the maintenance of an old project may lead to a takeover of new developers. [49, 52, 59]
- **Commercial strategy forks:** Commercial strategy forks include making a proprietary software version out of an open source version or vice versa. [49, 52, 59]
- **Legal issues:** Some forks are created due to disagreements on the license, trademarks or changes to conform laws like encryption. [29, 49, 52, 59]
- **Differences among developer teams:** The project can also split apart, if developers have other disagreements than technical issues. [29, 49, 52, 59]

Other studies [52, 59, 64] found four possible outcomes involving hard forks:

- **The discontinuation of the fork:** The fork will not be maintained anymore over a short period of time.
- **A re-merging of the fork:** The changes of the fork are merged back into the original project. This may be the case, if the dispute, leading to the hard fork, could be resolved.
- **The discontinuation of the original:** The fork is more popular than the original. This may lead to the end of the maintenance of the original project.

- **Successful branching, typically with differentiation:** The original and the fork are active for a longer period. The community is fragmented over the two repositories.

Similar findings could also be observed by Duc et al. [26] who researched forking in multi-platform software. The authors explained that a multi-platform software operates on multiple platforms with identical or similar functionalities. The term platform can refer for example to types of operating systems, processors, communication protocols and hardware systems.

On contrary to hard forks, social forks as described in [38, 64] have no intention to compete with the original project. They are often created to contribute back to the original, for example by adding new features. In [38, 39] another reason for social forks is mentioned. Such a fork can be used to keep a repository without the intention for contributing changes. For example, it is possible that a fork is only created in order to implement personal customisations, which should never be merged back to the original.

Zhou, Vasilescu, and Kästner [64] found out that hard forks are sometimes not intentionally made but evolved out of a social fork. The reasons for such an evolution can be facing obstacles like unresponsive maintainers or rejected PRs. It is also possible that a social fork itself gains more popularity before merging back to the original project. In this case, more people may contribute to this fork, and it may become incompatible with the original.

In the past forking had a rather negative association [18, 49, 51, 64]. The reasons for this are the risk of community fragmentation and reduction of communication, the chance to confuse users, the duplication of effort, and splitting up repositories in competing and incompatible versions [51, 64]. These days not only social forks but also hard forks have mostly positive associations [51, 64]. Transparent tooling and a decreased concern that a hard fork does not have a concrete reason, were likely to boost that shift [64]. Also, Gamalielsson and Lundell [28] in 2012, and Nyman et al. [49] in 2012 noted that the view of forks is changing.

Constantino et al. found in a study [21] two main motivations why developers contribute in fork-based development. The first motivation was that this form of development enables working collaboratively. This includes knowledge sharing, learning from others, strengthened teamwork, improved reviewing and increased productivity and code quality. On the other hand, the second motivation is the independent working style. Reasons for this can be that developers are able to choose the work on personal interests, there is no pressure, and it is easier to work in different time zones. Other developers in the study stated that coding is an independent task and that working collaboratively is time consuming.

Although forks nowadays have a better reputation, fork-based development also has its downsides [21, 33, 63, 65]:

- **Lost contributions [63, 65]:** Forks which have no connection to the original project are hard to find and often lost for the larger community. This leads to the problem that developers, who are interested in other developers' activities like new feature implementation or bug fixes may not be able to follow them proactively, especially if the number of forks increases.

- **Redundant development [63, 65]:** Having a decreased overview of existing forks may lead to redundant development of similar functionalities. Contributors may start working on a new feature or a bug fix, unaware that the implementation of this has already started in another fork.

- **Rejected PRs [33, 63, 65]:** Redundant implementations can lead to rejected PRs. Gousios, Pinzger, and Deursen stated in a study from 2014 that 27% of PRs are not merged due to this reason. Another cause for rejection can be that the expectation of the maintainer's vision of the project is not met with the changes done in the PR, which is also shown in the study of Gousios, Pinzger, and Deursen. A possible reaction of rejected PRs is that developers may become demotivated in contributing to this project anymore.

- **Fragmented communities [21, 63, 65]:** Even in social forks community fragmentation can be observed. One reason is that secondary forks (forks of forks) may contribute back to its parent, but no contribution back to the original project takes place. This often leads to forked projects drifting further and further apart.

- **Failing to comply with the project's contribution guidelines [21]:** Projects usually have a code of conduct and guidelines for contributors in order to keep them active. But sometimes contributors do not meet these rules. For example, when implementing a new feature, the tests are missing.

- **Knowledge and Time [21]:** Knowledge and time are important for repository maintainers and contributors. Maintainers, for example, need to know how they prefer to manage their repository and how the knowledge is made available in the project. Then they also need the time to apply this knowledge. Contributors also need the technical knowledge and the time to help.

- **Documentation [21]:** Documentation is needed for new contributors to understand the project. The main problems with the documentation are if it is out of date or it is lacking in general.

- **Work overload [21]:** Sometimes contribution to a project includes also a work overload for the contributors and the maintainers. For instance, if a new feature should be implemented, but the project does not have any test suites. Then the contributor must create an issue which requests the test suite creation. This also involves the maintainers.

- **Collaborators [21]:** All the above mentions points may lead to a loss of developers as contributors due to decreasing motivation. This poses a big problem, because contributors are the key for a successful open source project.

Beside forking projects within a specific platform, it is also possible to fork projects to another platform, like forking from GitHub to GitLab[4] [7]. Bhattacharjee et al. [7] found five possible motives for such a fork:

- **Forks created using mirroring feature:** GitLab provides a mirroring feature for forking repositories from GitHub. Using this feature the original repository will be synced every five minutes, keeping it up to date.
- **Forks owner are also a contributor of the original project:** Bhattacharjee et al. observed a repository, where the fork in GitLab was the same person as the contributor in the original. The last commit of the forked project was done by this contributor. The user did not exist in GitHub any more. The identity was checked by the username in GitLab and GitHub. The authors suggested that such forks are created because contributors want to switch platforms, but also want to preserve their contributions.
- **Forks with changed title after forking:** Also cross-platform forks exist, where the name of the repository was changed. The particular reasons for this were not found out by the authors.
- **Forks intended for an individual copy:** Another motive for cross-platform forks may be only for keeping a copy on a different platform without any change.
- **Forks intended to continue development with another social coding platform:** Cross-platform forks may also occur due to other features in a different platform. An example is the Continuous Integration/Continuous Delivery (CI/CD) support provided by GitLab. The contributors fork the repository with the intent on continuing the development in addition with using the other features.

2.4 Issue

An issue is a request to make improvements to a software system. This can include fixing bugs, adding new features or improving documentation [6, 9]. Bissyande et al. [9] stated software programs can contain bugs and may have incomplete features. The authors wrote that issues are a valuable feedback to the developers in order to improve the quality of the program. The authors also stated that fork-based development had „a positive impact on on issue reporting“[9]. They found out that the number of watchers and forks correlated strongly with the number of reported issues.

[4] https://about.gitlab.com/

Such issues can be maintained with issue trackers [5, 6, 9]. The authors stated that an issue tracker can be used by managers, developers and other users to manage the interactions for creating, monitoring and fixing software. Furthermore, the system also provides a detailed history of these interactions, and an overview of what tasks need to be done. Common issue tracker software are Jira[5] and the integrated issue tracker from GitHub[6].

[5] https://www.atlassian.com/de/software/jira
[6] https://guides.github.com/features/issues/

CHAPTER 3

State-of-the-Art

This section provides an overview of existing awareness tools and how the prototype covered by this thesis differs from them. Afterwards current information needs and how the prototype addresses these are pointed out.

3.1 Awareness Tools

Palantír [53], an Eclipse[1] plugin, monitors ongoing local changes of a repository. When a conflict arises, it shares the changes with developers to whom it is relevant. Each change contains the information which developers changed which artefact (for example a file) and to which extend. The conflict notifications are always at the level of a whole artefact. Palantír can analyse direct and indirect conflicts. An example can be seen in Figure 3.1. The plugin shows that at least one indirect conflict occurred in the classes DebitCard.java and Payment.java.

Another Eclipse plugin is called Syde [36]. Syde uses an AST as the basis of the analysis. Changes in the repository are saved to the ASTs. The plugin can detect merge conflicts by comparing those ASTs. The tool is able to provide the change detection in real time.

Like Syde, Lighthouse [23] is an Eclipse plugin which uses an AST to provide awareness to developers. Changes made by developers are shown in an Unified Modeling Language (UML)-like visualisation of changes done by the developers. These changes will be visible in all contributing workspaces and will be shown without the need to save those changes. Therefore, all developers have a common view on how the project changes over time and are able to detect arising conflicts.

Another plugin for Eclipse is WeCode [35] which detects conflicts by comparing trees of typed and attributed nodes. A node is either a file, a folder or a program element.

[1] https://www.eclipse.org/

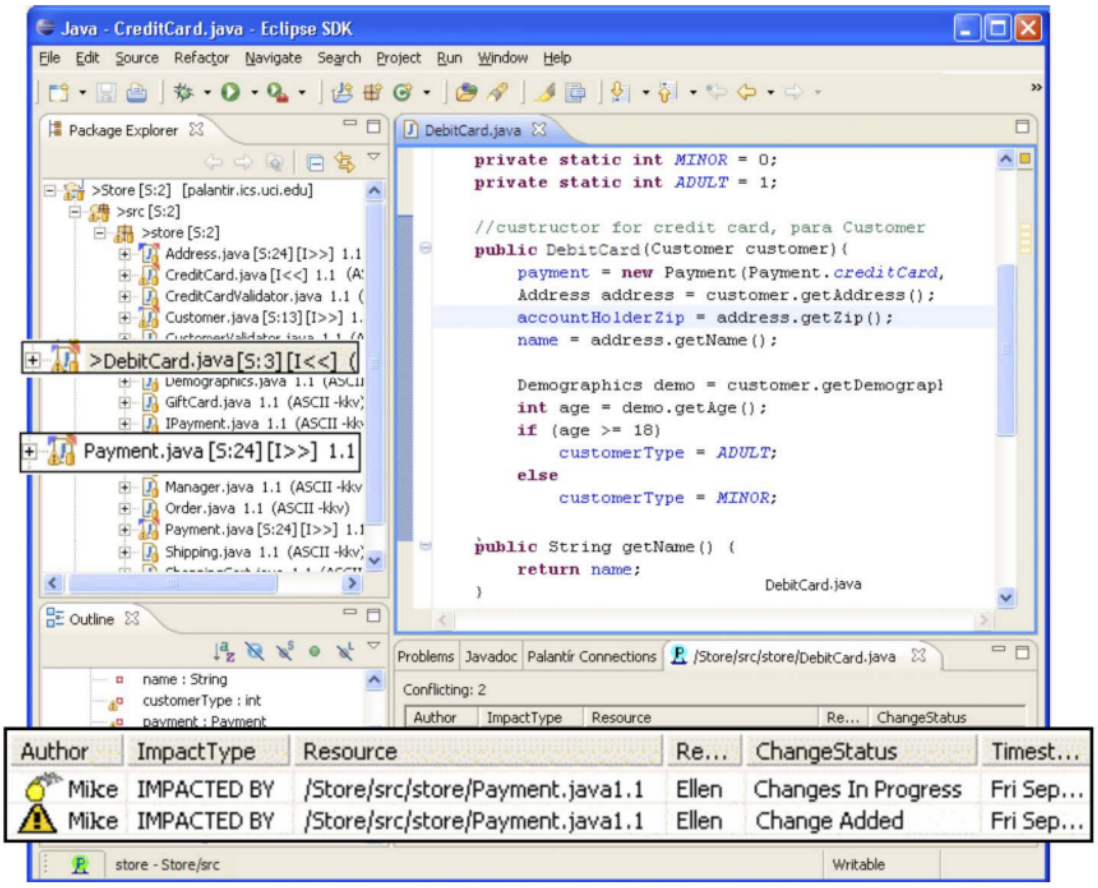


Figure 3.1: Conflict detection example of Palantír [53]

WeCode can also detect language conflict by compiling the merged system. This analysis is done every time the repository is updated. Figure 3.2 shows the User Interface (UI) of WeCode with the extended package explorer (Figure 3.2 1)), the extended editor (Figure 3.2 2)), the team view (Figure 3.2 3)) and the team merge view (Figure 3.2 4)).

FASTDash [8] (Figure 3.3), a plugin for Visual Studio[2], also provides information in real time. This information can be divided into file information and repository information. File information gives insights on open files, files which are being edited, a focused file of a developer and on classes or methods which are currently viewed, edited or debugged. The repository information gives information about checked out files, checked out files which are different from the current repository version and potential checkout conflicts (for example if two developers have the same file checked out). Developers can attach comments to a file to communicate together.

[2]https://visualstudio.microsoft.com/de/

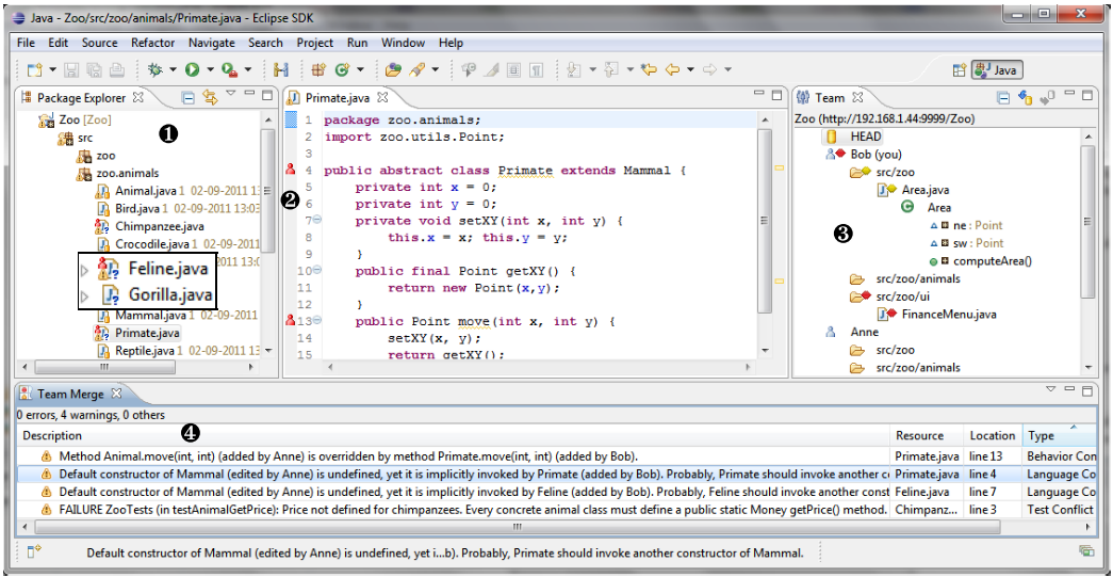


Figure 3.2: User Interface of WeCode [35]

Levin and Yehudai [42] developed in 2015 a plugin for IntelliJ IDE[3] which tries to detect potential merge conflicts (Figure 3.4) by using semantic path ids. A semantic path id uniquely identifies elements of the project and is constructed as follows: „/project/fileName/className/methodName/paramName/“. During the analysis, this prototype checks local and remote changes of the same semantic path ids. Using this approach, the computation time only depends on the number of team members (N) and the average number of changes per team member (M). Therefore, the computation time is near real-time ($O(N \cdot M)$).

In contrast to the tools described above, Crystal [14] is a standalone application (Figure 3.5). The tool checks for merge conflicts, but also for compiling and testing conflicts. Crystal tries to merge the local state of a project and reports back encountered conflicts. If no such conflicts arose, it tries to compile the merged projects. After a successful compilation Crystal runs the test suite and checks if the tests fail. The tool shows the conflict state with relationships and colours (Figure 3.6).

With CloudStudio [48] developers do not work on a personal workspace, but on a shared one over the web. Changes done by the developers themselves or by others are tracked in real time (Figure 3.7). Developers can decide on their own if they want to see the changes from a specific colleague or not and can define the granularity. For example, it is possible to choose that only changes of a specific colleague should be shown that generate conflicts. The ownership of uncommitted changes lies with the developers themselves. This means that the others can see these changes but are not able to edit them. Additionally,

[3]https://www.jetbrains.com/idea/

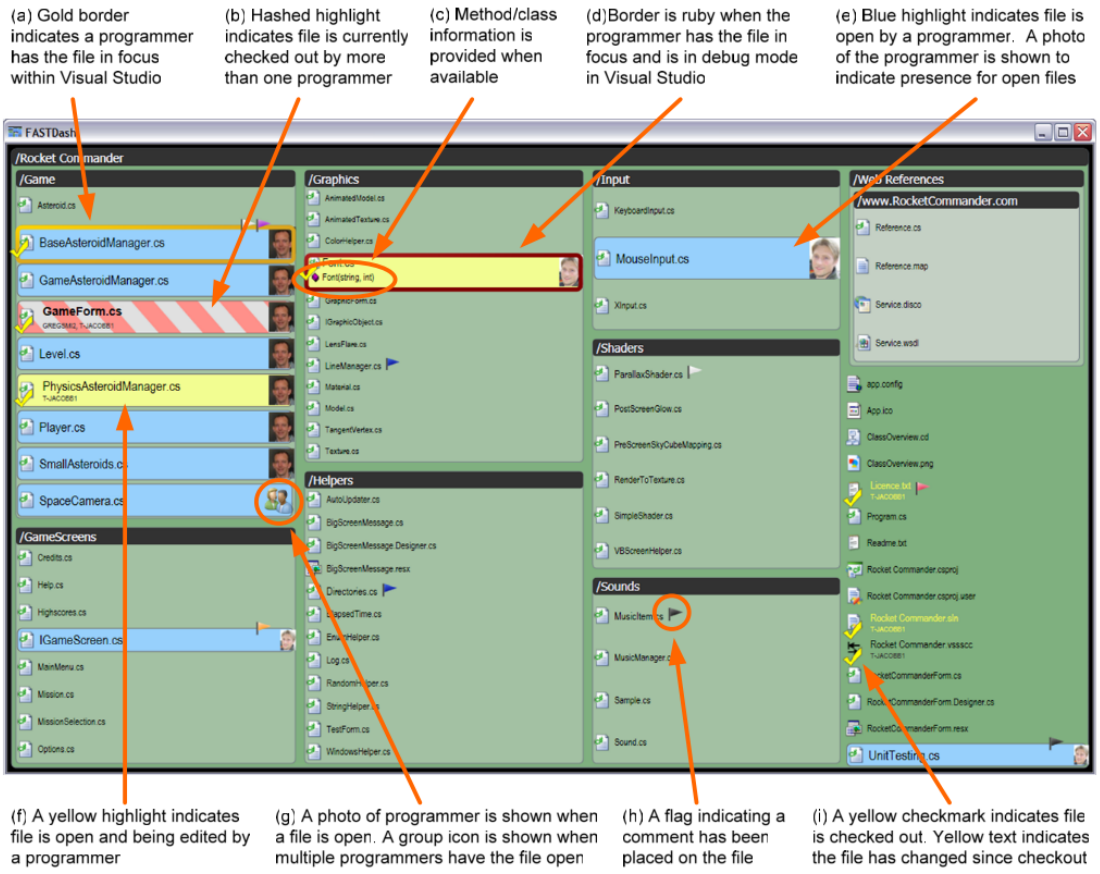


Figure 3.3: User Interface of FastDash [8]

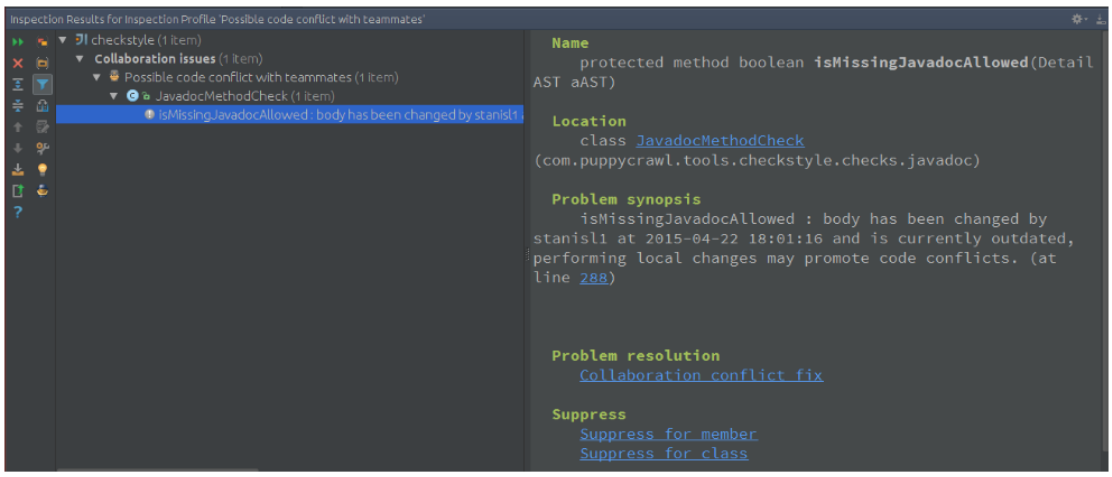


Figure 3.4: Potential conflict provided by awareness tool of [42]

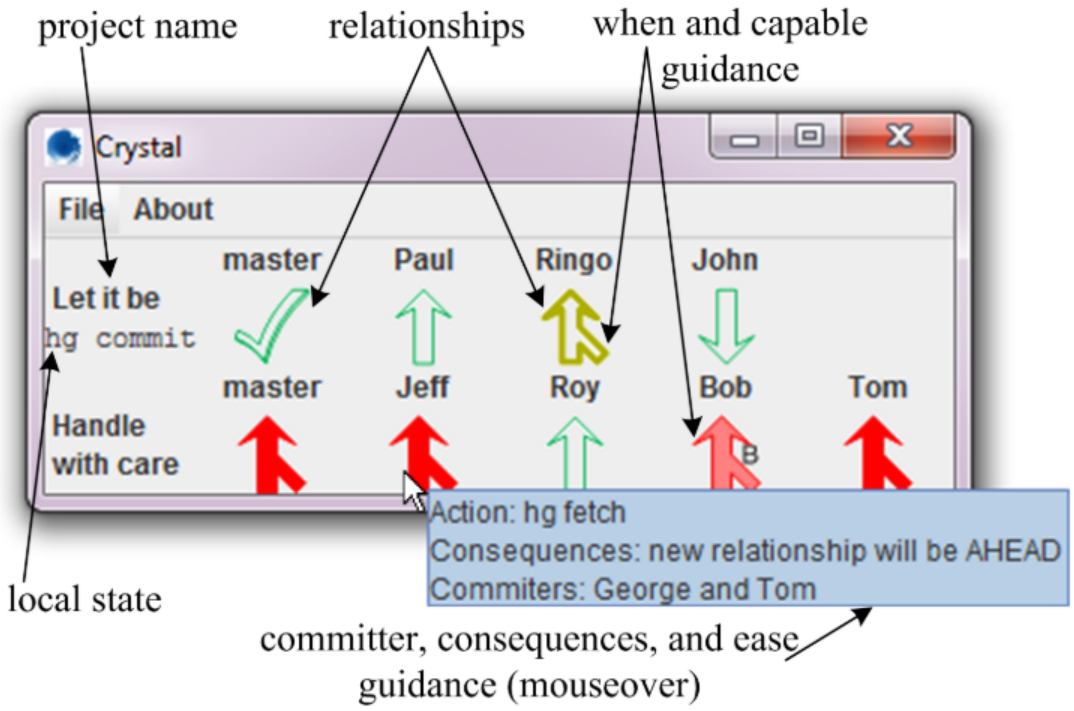


Figure 3.5: User Interface of Crystal [14]

Figure 3.6: Relationships of Crystal [14]

CloudStudio provides a built-in chat for communication, the possibility to call colleagues via Skype and includes verification tools for automatic testing and static verification.

Another web-based approach for providing awareness is Collabode [32]. Collabode is a web application which enables collaborative development for Java applications. Its frontend is implemented with JavaScript and HTML. The server uses Eclipse for the project management and for providing standard IDE services like code formatting, refactoring and continuous compilation. Code will only be shared if it is not broken. To check for broken code, signals like compilation errors or failing test cases (Figure 3.8) are used.

TIPMerge [22] is a tool which suggests the best suited developers for a merge. First, a project must be selected where a merge should take place. After that, two branches and optionally a hash of the commit are selected which should be merged. If no hash is

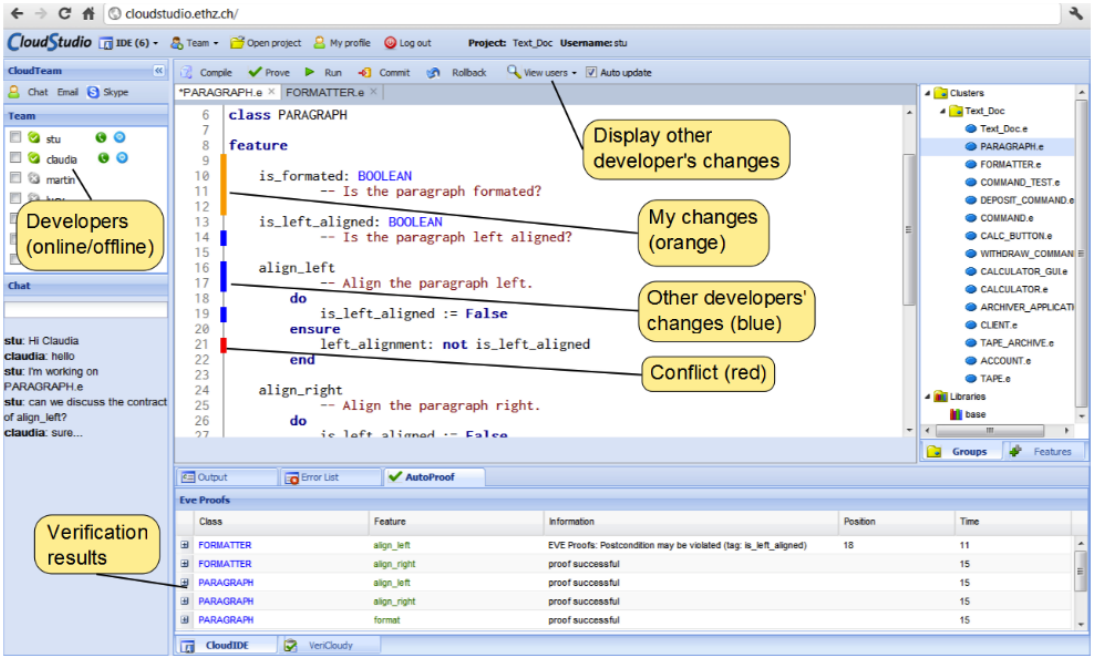


Figure 3.7: Web overview of CloudStudio [48]

provided, the last commit of the branch is used automatically. TIPMerge selects the best suitable developers with the following steps: First, the data is extracted to the selected commits of both branches. After that step, frequently co-committed files are detected and afterwards the developers who edited these files are identified. Finally, the system provides a ranked list of suitable developers who can perform the merge. Such a result can be seen in Figure 3.9. It shows the authors (Figure 3.9 a)), the files (Figure 3.9 b)) and the branches in which the changes were made (Figure 3.9 c)).

All the above mentioned tools [8, 14, 22, 23, 32, 35, 42, 48, 53] only focus on intra-project conflict awareness visualisations. The visualisation prototype of this thesis not only covers merge conflict visualisations within one project, but also for forked projects.

3.2 Information Needs

Although, as seen in Section 3.1, there are several awareness tools available, developers are still missing some functionalities [45, 46]. For example, the studies found out that developers find, it should be able to filter relevant information and that tools have a better support for exploring a project's history. Another problem, for example for automatic merge tools is, that their functionalities are not transparent enough. This may lead to a mistrust of developers in this tool because they cannot understand how a specific outcome was achieved. Additionally, a further issue pointed out by some developers in

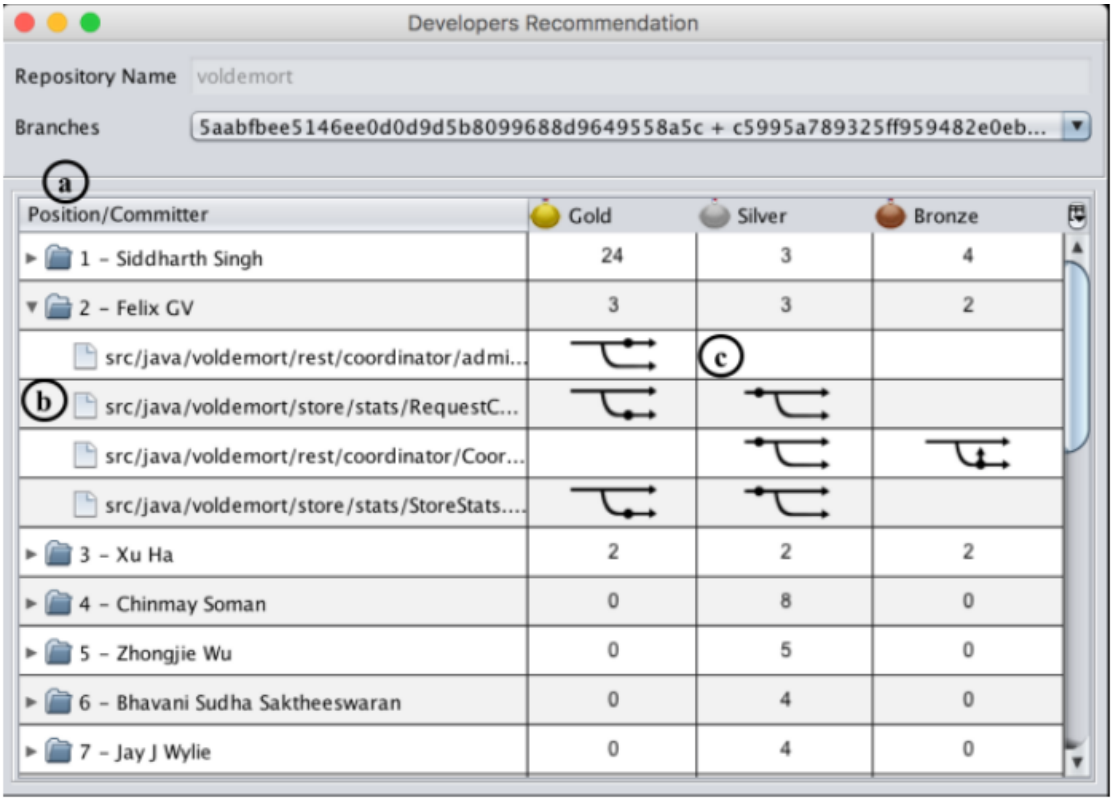


Figure 3.8: Collabode showing failed test cases [32]

the studies is that tools may have inconsistent terminologies and visual metaphors like colours or notifications.

The authors in [45] also learned, that in order to resolve merge conflicts tools can help the developer by providing enough information about the conflict and by presenting this information in an understandable way. It also helps if the tool is trustworthy, and it provides a possibility to examine the development history as also stated above. But the authors stated that there are additional information needs necessary for a conflict resolution. It is, for example, dependent on the expertise of the developer in the conflicting code area and how easy the person is able to understand the conflicting code. Also, the complexity of the project's structure, the project culture, changing assumptions and the informativeness of commit messages play a role in merge conflict resolution.

The authors in [56, 64] also stated that resolving merge conflicts while synchronising the fork with the original or vice versa is harder than fixing conflicts within a project, especially if the two projects have diverged substantially. Sung et al. [56] found three reasons for this: 1. The developers from the fork or the original often are not aware of the changes made by the counterpart. 2. When the fork or the original has a lot of commits, it is not easy to pinpoint the conflicting commit. 3. The root cause of a conflict may also

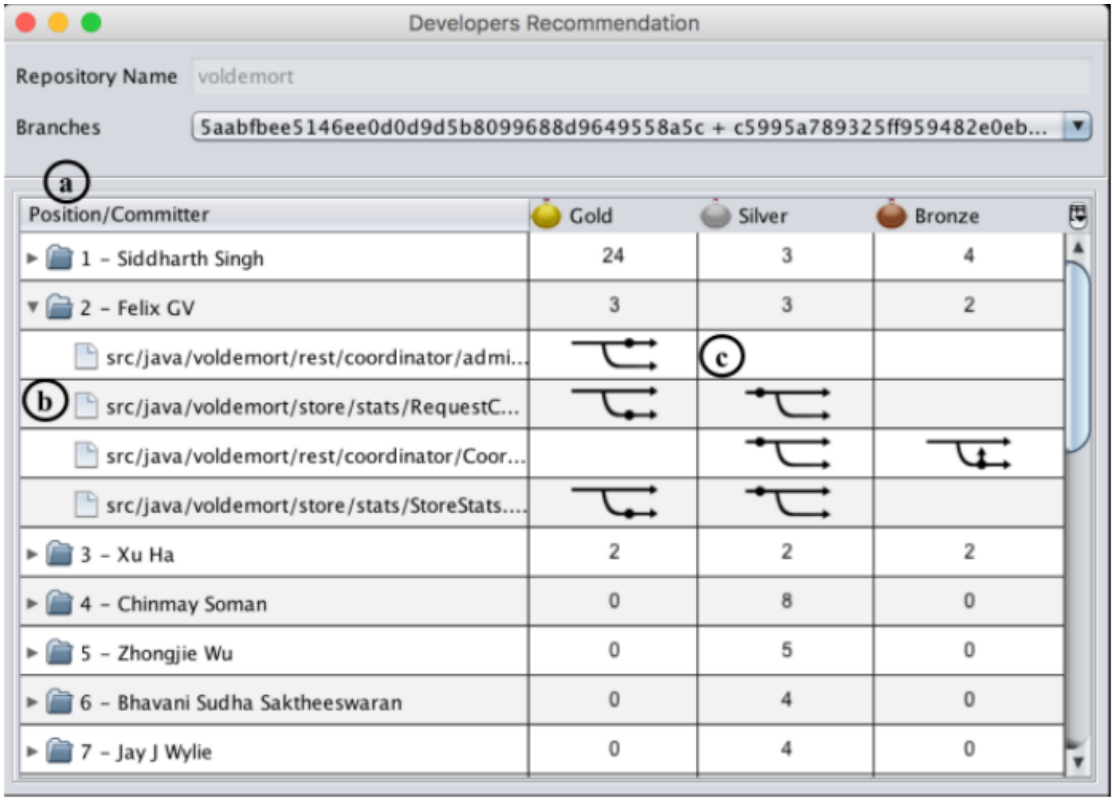


Figure 3.9: Ranked list of developers provided by TIPMerge [22]

be a commit in the fork or the original itself, made a long time ago. In this situation it can be difficult to find the right person for the fix, for example if the person already left the project, especially in open source development. This additional complexity might be the reason, why such synchronisations are still rare in hard forks. The monitoring of the projects can be overwhelming, such that also small changes can lead to a lot of work [64].

The tools described in Section 3.1 provide only intra-project awareness information which are only partially useful for fork-based development or for syncing hard forks with their original or vice versa. As Estler et al. stated [27], this awareness information need not to be provided in real time by most developers. Developers also have different exceptions on how much detail this information should provide. The authors recommend providing as much information as possible, but with a possibility to customise this level of detail. This enables everyone to decide for themselves, what should be shown and what not.

As mentioned in [45, 46], the software's history is an important information need for developers. Codoban et al. [19] also conducted a study in 2015 in order to find out why and how developers examine it. Figure 3.10 shows the findings on why developer examine old and new history. The authors also found out that developers mostly look at the commit messages, the difference between a commit and its parent in order to

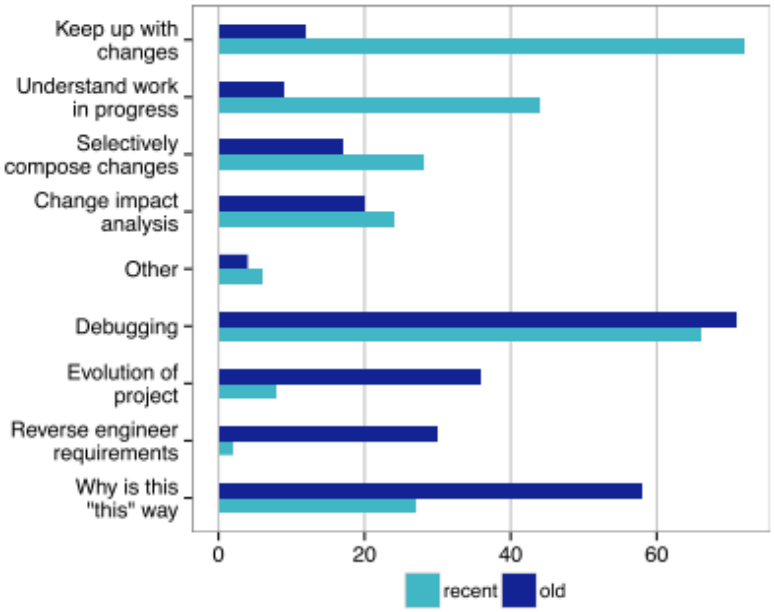


Figure 3.10: Developer's motivation for examining the software history (%) [19]

Traceability to versions	58%
Informative commit messages	48%
Aggregate commits into groups	46%
Ability to filter changes	40%
Support for managing uncommitted changes	28%
Traceability to architecture	26%
Selective change notifications	20%
Other	3%

Table 3.1: Developer's tool desires [19]

understand this commit. But also, the author and the knowledge of other colleagues are important for the understanding of commits. Additionally, the study revealed some challenges developers have to work with. This includes that the commits' differences are sometimes hard to understand, for example if white space and line-ending changes are included. Developers wish for tools to have a better version traceability of code snippets, especially if files are moved or renamed. Another desire which was found out in the study is a better traceability to requirements, meaning that it is helpful to find all commits which are linked to a specific requirement. A full list can be found in Table 3.1.

With the visualisation introduced in this thesis the above-mentioned information needs are addressed in various ways. Since the project history is a much-used way to collect information, its representation resemble the characterisation of a Git graph, because

most developers are familiar with this form of representation. The graph allows to easily navigate through the history by hovering over commits and by selecting commits which arouse the interest of the developer. The selection provides useful information of the commit like its metadata, other commits it depends on and its changes. This representation not only shows the history of one project, but also the history of a selected fork or its parent in a way that the common grounds and divergences are easily spotted. Developers are able to filter this information by providing a compact view of the history. Interesting commit clusters can be expanded by the user, explored and afterwards minimised again. Additionally, branches of a project can be faded in and faded out at will. This provides possibilities to only focus on specific areas in the graph. Not only the representation of the projects' history is familiar to most developers who worked with Git in the past, it is also possible to select the base colours of the visualisation such that it can be adapted to personal preferences. In order to assess the complexity of merge conflicts, the prototype shows when a merge, rebase or cherry pick within a project or from a fork result in a conflict. If a conflict occurs the visualisation provides some data about the merge conflict like conflicting code sections or authors which are involved in the changes. This allows the developer to assess, for example, if the merge could take longer and if it could be wise to invite other developers to the merge session. The prototype also provides a functionality, which marks the commits of a selected requirement.

CHAPTER 4

Conceptual Design

The next step after the literature research was to set up a plan with which features Binocular [34] (see Section 6.1) should be extended. Based on the literature research for the state-of-the-art and the own experiences as software developer, features were planned which the prototype should possibly contain. The proposed features can be found in Table 4.1. In the following phase, these suggestions were rated and reviewed by experts in semi-structured interviews on how important they would find such features in a visualisation. The outcome of these ratings had an influence on the priority of the implementation order of the features and if they all find enough support by the experts

F1 Show the divergence between two forks
F2 Show if branches within a project can be rebased without conflicts
F3 Show if branches of forks can be rebased without conflicts
F4 Show if branches within a project can be merged without conflicts
F5 Show if branches of forks can be merged without conflicts
F6 Show if commits within a project can be cherry picked without conflicts
F7 Show if commits of forks can be cherry picked without conflicts
F8 Show the code sections of the conflict if one would occur at the rebase, merge or cherry pick
F9 Show the commits which the selected one depends on
F10 Show the commits of a selected issue
F11 Show the metadata of a selected commit
F12 Provide a compact view of the visualisation
F13 Hide commits of selected branches
F14 Filter commits after a specific timestamp, within a specific time frame, of a specific author or committer and after a selected commit (subtree)

Table 4.1: Proposed features of the prototype

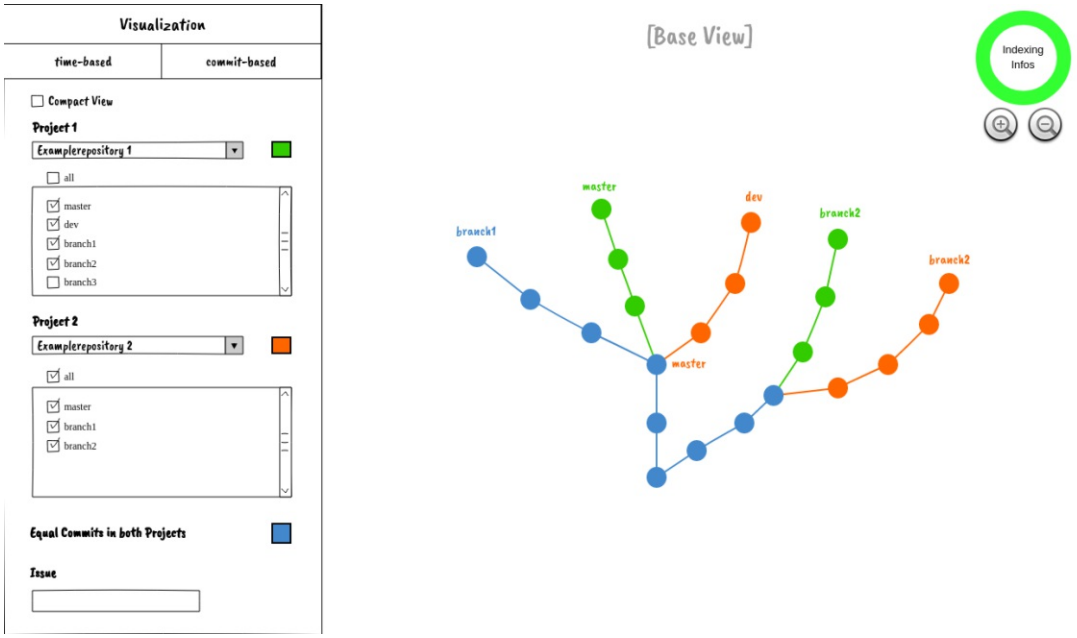


Figure 4.1: Mock-up: base view of the visualisation (F1)

to be included in the visualisation. The results of these interviews can be seen in Section 5.2. In order to further concretise these ideas, mock-ups were created.

For these mock-ups two main iterations were needed. The first mock-up versions were too focused on the individual features. Basically, the idea was to provide an own visualisation for each proposed feature. But this approach did not allow interactions between the features within the prototype. For the second version an approach was used to think more out of the box. The prototype should be able to combine the features within one visualisation in order to provide a better and more intuitive user experience. This brings more flexibility and a broader coverage of use cases that can be covered with the prototype extension.

This visualisation should extend the current development state of the Binocular project. Therefore, the structure of the prototype should be similar than the existing visualisations within the project. Because of this, the mock-ups were designed with two sections. On the left side will be the config section containing all configuration settings the user can adjust. The data visualisation is shown on the right side of the view. This section will change according to the set settings.

The basic structure of the data in the visualisation should be intuitive for developers using Git. Therefore, the basic view of the data should resemble a Git graph. The mock-up of the visualisations base view (Figure 4.1) displays such a possibility. The nodes in the graph represent the commits. The edges are indicators for parent-child relationships showing the overall Git history of the repository. Nodes that have labels on

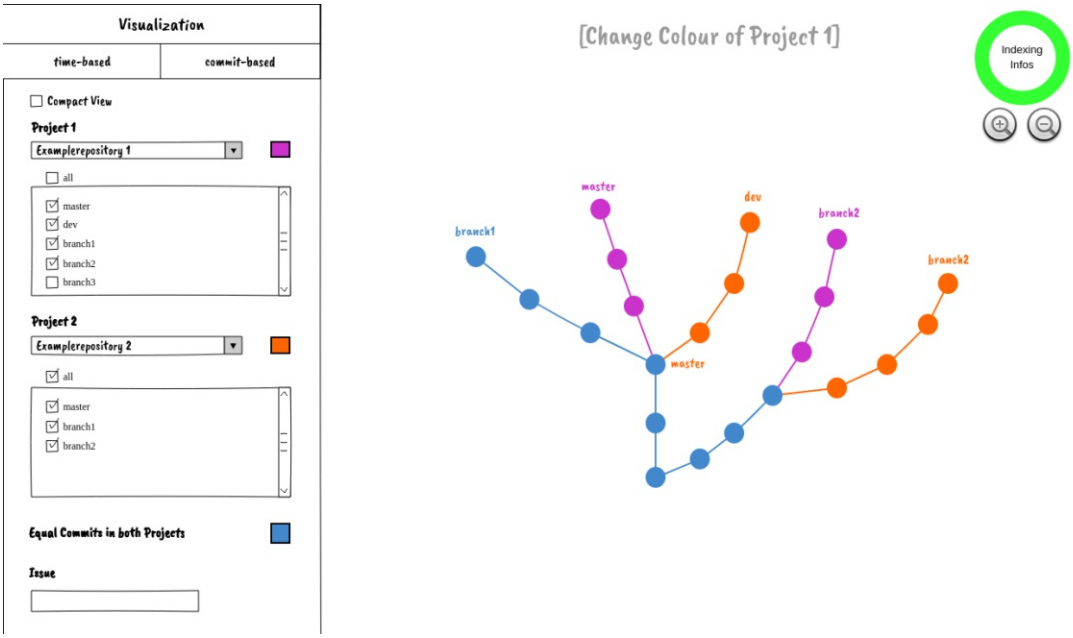


Figure 4.2: Mock-up: change the colour of a repository (F1)

top are branch heads with the branch names as label content. The colours in the graph show in which project the commit is available. In the mock-up, all commits coloured in green can only be found in the base repository, the orange nodes are only in the parent or fork of the base repository. All commits which are marked blue can be found in both projects. This indicates the state from which a project was forked. The colours of the labels also show the membership in the repositories. In Figure 4.1 branch1 was not changed by any of the projects and is equal in both repositories. On the contrary, the members of the base repository worked on the master branch and have pushed three additional commits which are represented with three green nodes on top of the common state, the blue nodes. In the other project nobody adapted the master branch. Therefore, both master labels can be found in different locations with the corresponding colour. This representation also shows how much forks were diverged over time (F1). This can provide hints on how easy or complicated a reintegration can get. The parent or a fork, if existing, can be selected over the drop-down menu in the config section.

Developers found fault with the lack of customisability of used tools. Therefore, it should be possible to adapt the used colours at will. This allows the users to adapt the appearance of the prototype more to their likings, preferences and habits. An example can be seen in Figure 4.2. In the contrary to Figure 4.1, the colour of the base project is now purple instead of green.

When clicking on a commit, the developer can see metadata of the selected commit (F11, see Figure 4.3). This includes the hash, the author of the commit and the corresponding time, the committer of the commit and the corresponding time, as well as the commit

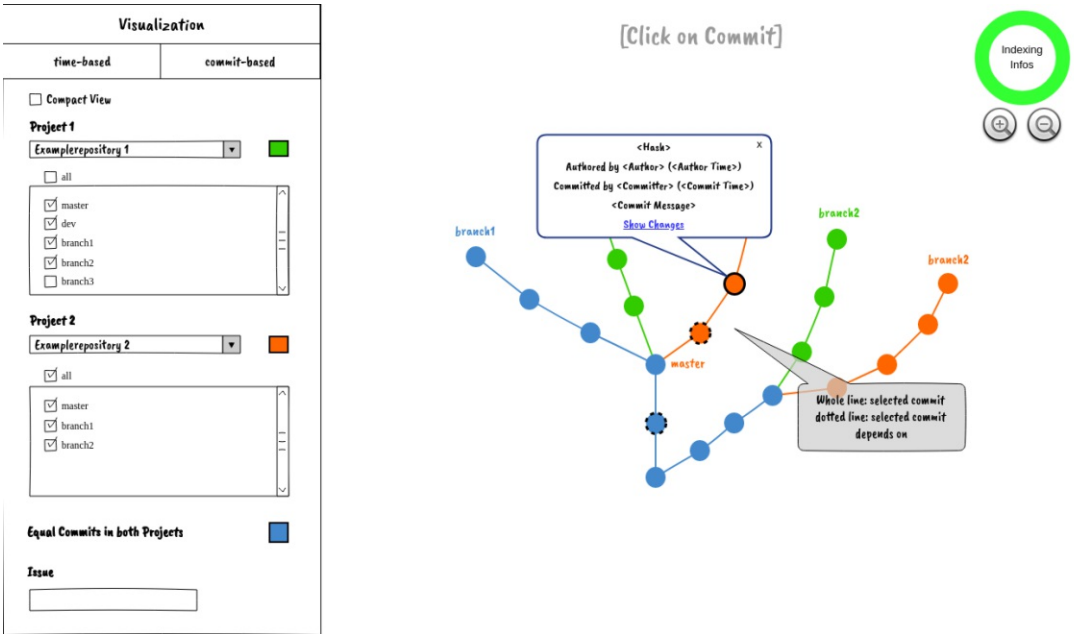


Figure 4.3: Mock-up: click on commit showing metadata (F9, F11)

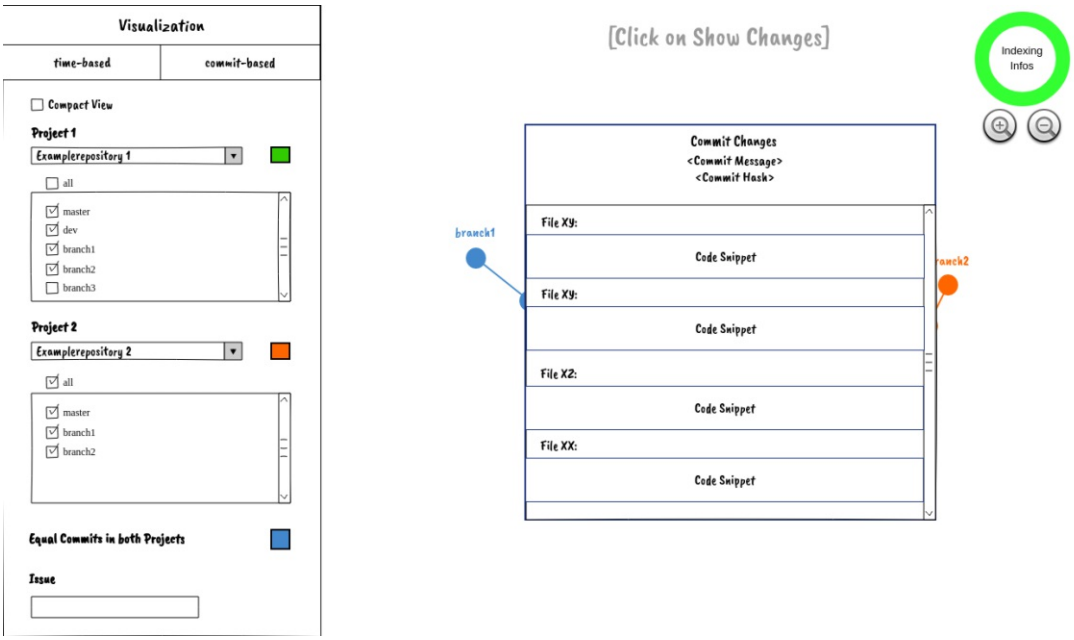


Figure 4.4: Mock-up: show changes of a commit (F11)

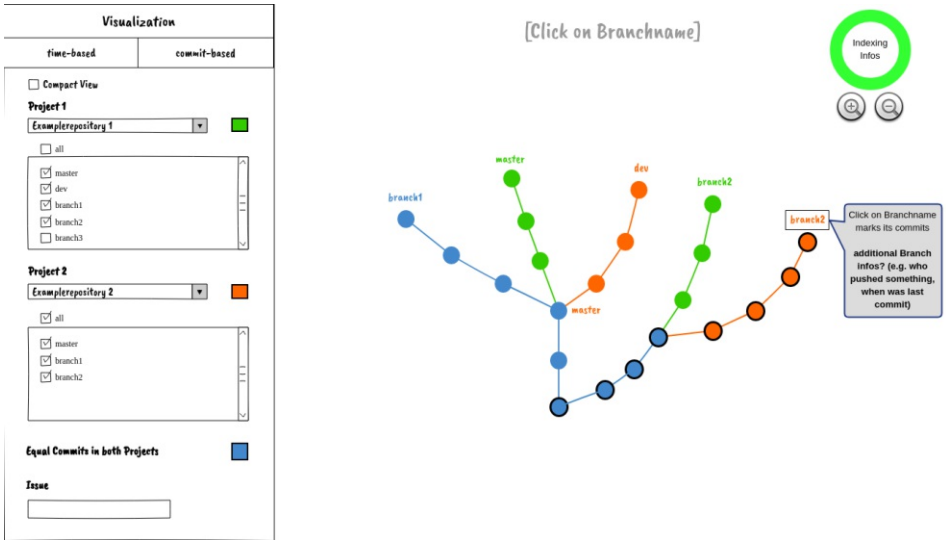


Figure 4.5: Mock-up: branch highlighting

message. The visualisation should also provide information on which previous commits the selected one depends on (F9). For this the functionality of git-deps[1] should be used. This information can come in handy especially when commits should be cherry picked. The information of a commit's dependencies can make it easier to identify earlier commits that have to be picked along with the selected one. This may decrease the conflict potential of cherry picks. Usually, software engineers are also interested in the changes which are introduced in the project with a commit. Therefore, the visualisation should also provide a view of its diffs (F11, Figure 4.4).

When clicking on a branch label, the commits of this branch should be highlighted. An example can be seen in Figure 4.5. This may help developers in identifying the path and the commits of a branch faster.

Over time repositories can become large and can hold a lot of information. This can result in an information overflow for the engineers, and it can get hard to find its way. One way to mitigate this threat is to allow the user to hide or show specific branches of a repository at will (F13). The branches of the base project and the selected fork or parent will be shown in scrollable checkbox lists within the config section of the visualisation. The user is able to select or deselect branches by checking or unchecking the corresponding checkboxes in the list. Figure 4.6 shows the commit graph with the branch „branch2“ of the fork „Examplerepository 2“ deselected. Deselecting this branch removes all its commits that can only be found in the corresponding repository. The other commits will be recoloured if the deselected branch was the only branch reference of this project for the commits. When selecting the branch again, the changes will be reverted again. This means that all the commits which were recoloured would get the „combined“ colour and

[1] https://github.com/aspiers/git-deps

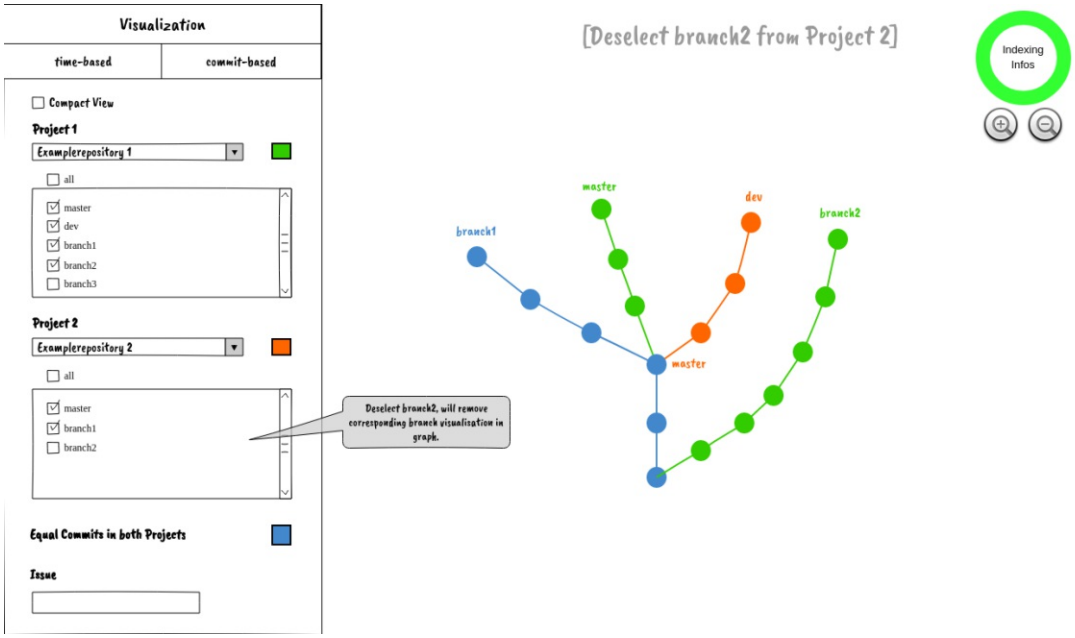


Figure 4.6: Mock-up: deselect branches of a repository (F13)

the removed commits will be shown again with their original colour. If a repository has many branches and the user wants to show only one or a few branches of the project, the prototype should provide a function to select or deselect all branches of the base project or the other repository. This should enable the user to first deselect the branches and then select the wanted ones. On default all the branches of the repository should be visible. This feature will be triggered by the „all“ checkbox above the branches list as shown in Figure 4.6. If all branches of the repository are selected, this checkbox will also be checked. Otherwise, if only one or more branches are deselected, the „all“ checkbox will also get deselected. Then the user must first check this checkbox again or select all branches individually before deselecting them again at once.

Another way dealing with a big amount of data should provide the feature of a compacted view (F12). This view should cluster non-branching commits to nodes that only hold information about how many commits are grouped together. The compacted version of the example graph can be seen in Figure 4.7. Branching nodes should be commits which have either multiple parents or multiple children. Those will not be clustered. Branching nodes can hold valuable information about the structure of the repository, like when a feature branch was started or when changes were merged. Furthermore, a single commit between two branching nodes should not be compacted, because this would lead to an information loss without gaining a more compacted graph. The whole graph should be able to be compacted or expanded completely using the „Compact View“ checkbox in the top of the config section on the left.

The feature of compacting or expanding only the whole graph alone would not be as

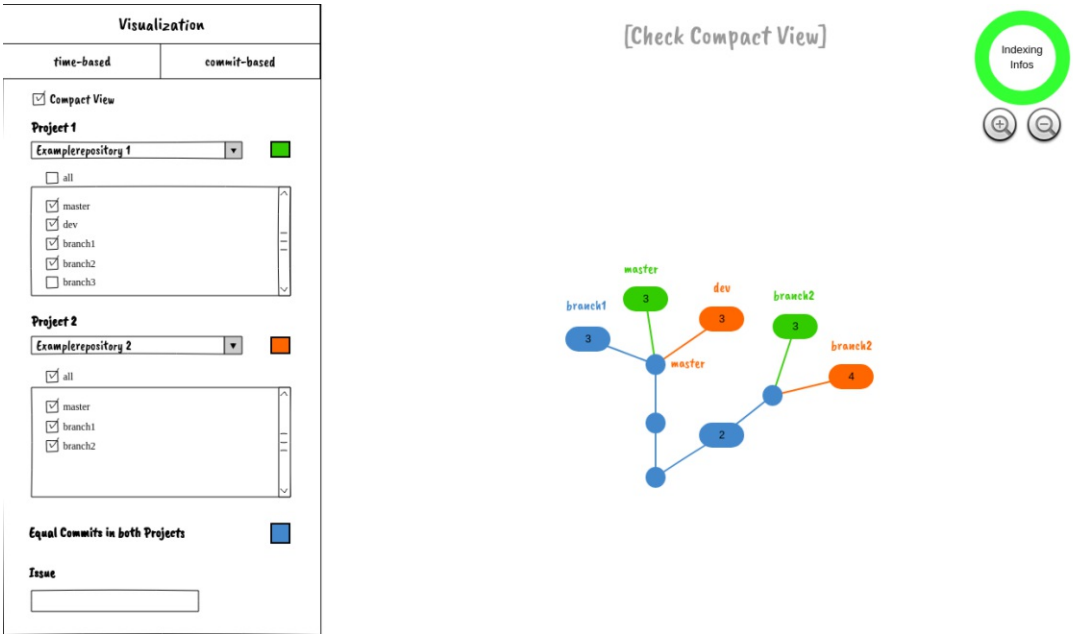


Figure 4.7: Mock-up: compacted view (F12)

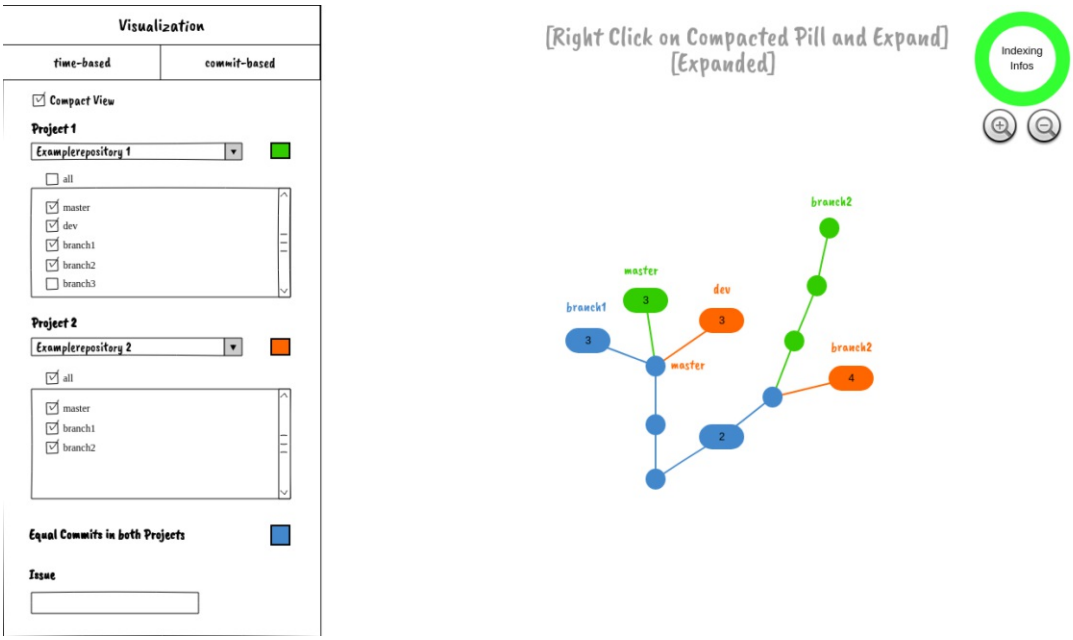


Figure 4.8: Mock-up: compacted craph with one expanded section (F12)

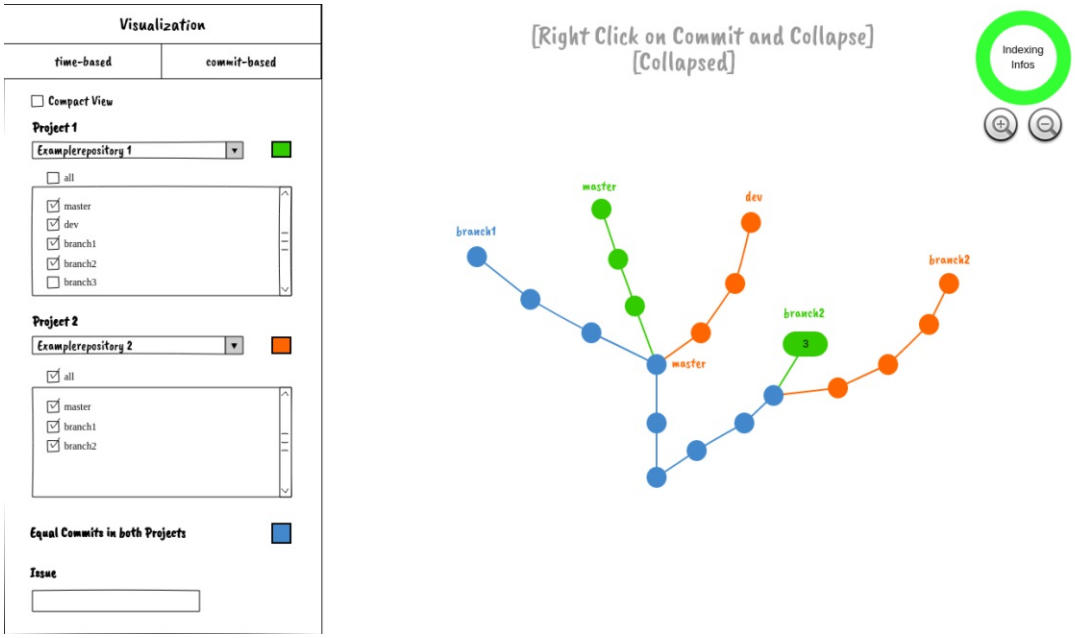


Figure 4.9: Mock-up: expanded graph with one collapsed section (F12)

helpful as intended, because then for the developers the situation could occur that they either have too much information or too little. Therefore, it should be able to expand collapsed clusters individually and to cluster specific sections which are not of interest of the software engineer. In order to expand clustered nodes again the user has to right click on a cluster and select the „Expand“ context menu item. The result of the fully compacted example graph with one expanded section can be seen in Figure 4.8. It should also be possible to collapse expanded sections separately. With this the software engineer can minimise the available information of the specific section without impacting all the other sections of the graph. Figure 4.9 shows the fully expanded example graph with one collapsed section. In order to get to this result, the user should select one node within the section that should be compacted, open the context menu with a right click and has to select the „Compact“ context menu item.

Other main features of the prototype should be the checks if merges, rebases and cherry picks will be successful or will result in a conflict. Those checks should be available for branches or commits of a single project but also across the repositories (F2 - F7). To check if branches can be merged successfully the idea was, that the user just drags the branch label over another label and drops it there (F4, F5). This would mean that the merge of the dragged branched into the branch where it was dropped onto branch will be checked. Afterwards, the prototype shows a message if a conflict would occur or not. A possible success message can be seen in Figure 4.10. This would be a simple message without much additional information other than a reminder, which branches were used for the merge. If a conflict is detected more information will be shown. A

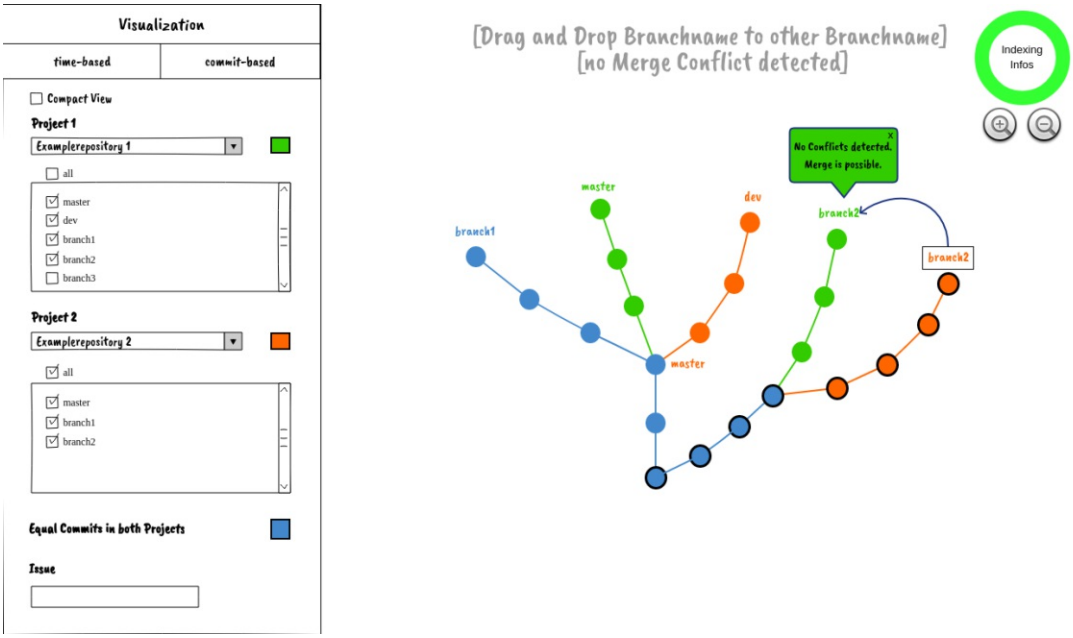


Figure 4.10: Mock-up: check merge - no conflict found (F4, F5)

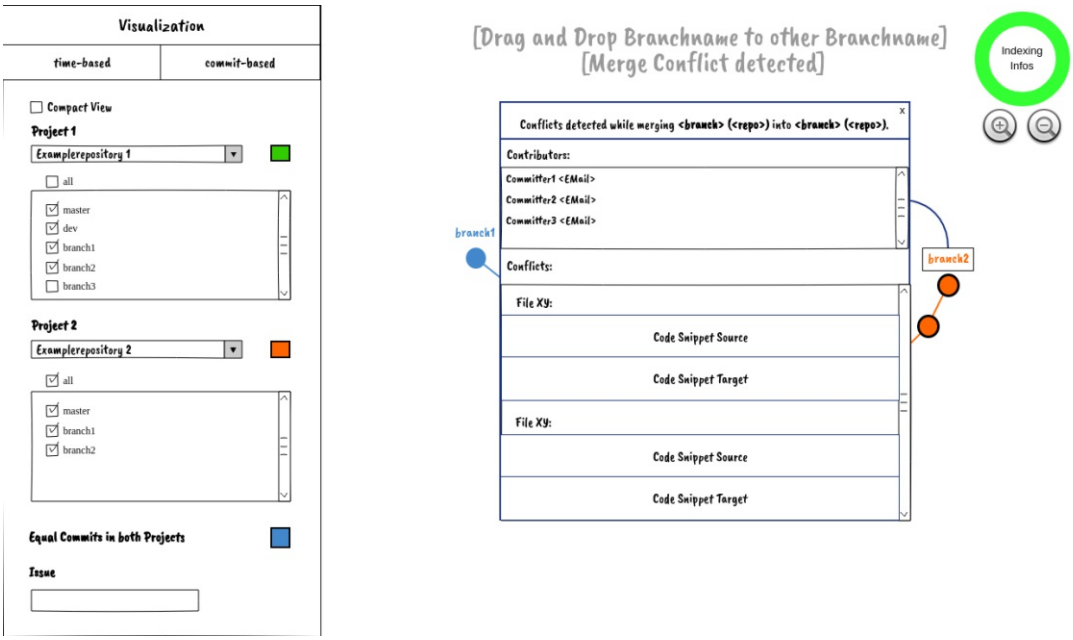


Figure 4.11: Mock-up: check merge - conflict found (F4, F5, F8)

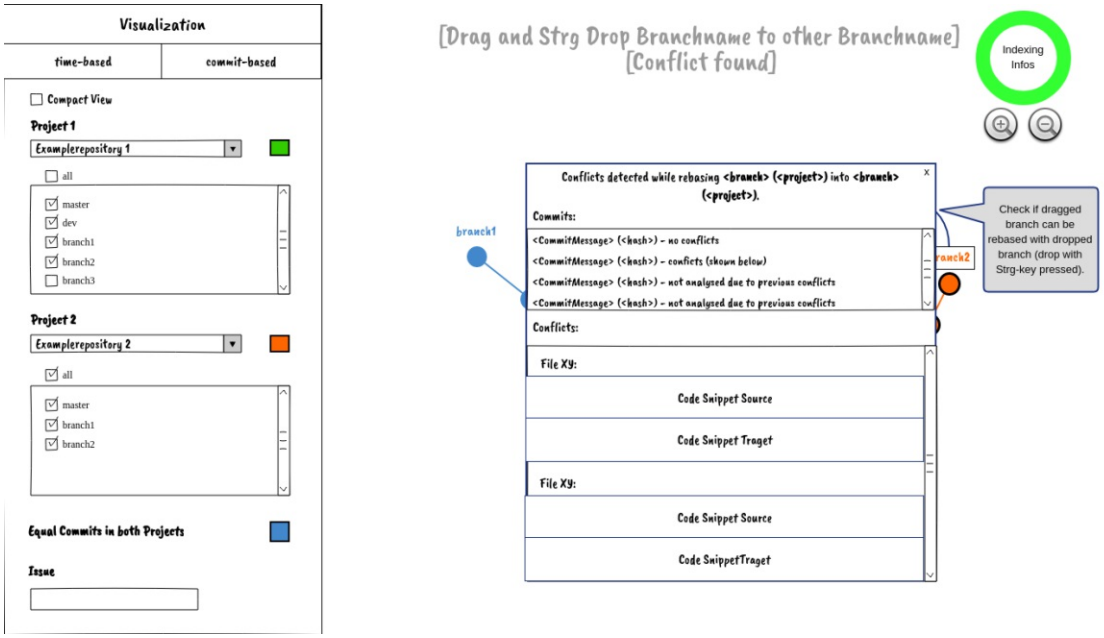


Figure 4.12: Mock-up: check rebase - conflict found (F2, F3, F8)

modal should be opened with a message displaying which branches were involved in the merge and the authors of the commits that were involved in the merge. To know who introduced changes into the repository can help in finding suitable developers who can provide help with solving the conflict. Additionally, the code snippets of the conflict should be shown in order to assist in the planning of the conflict resolution (F8). If the conflict seems big and complex, the merger may decide to postpone the resolution to gather more information or to look for help for the upcoming task. The information could also be interpreted as a warning sign that branches will diverge and such that they can be merged in an early stage before the conflict gets out of hand.

Similar data should be shown for checking a rebase for conflicts (F2, F3). This functionality should be called in a similar way as the check for merge conflicts. The idea was that the rebase check will be performed when the user drag and drops a branch label onto another one while holding the ctrl key. The check itself should also be possible for branches within a single project but also over the boundaries with branches of different repositories. If the rebase will be successful a similar success message should be shown as in Figure 4.10, but with the information which branch can be successfully rebased onto which other branch. When a conflict is detected, again, the information which branches are involved, and which code sections are conflicting should be displayed to the user (F8). Additionally, the developer should get information on which commit the conflict occurred, which commits could have been rebased successfully and which commits were not analysed due to the found conflict (Figure 4.12).

The third conflict check, the prototype should provide, is the cherry pick check (F6, F7).

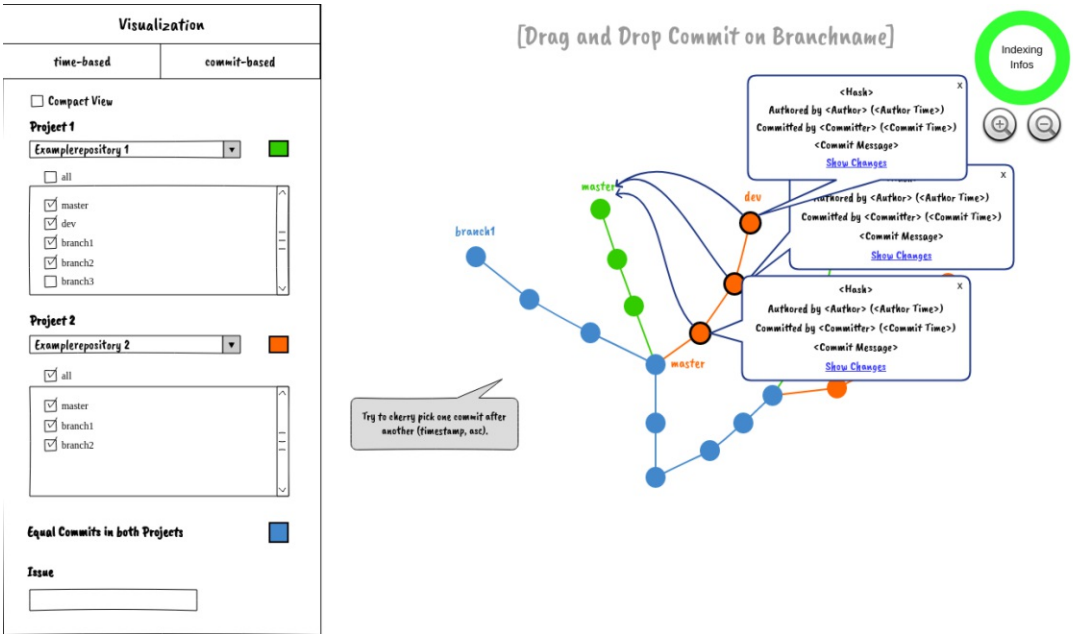


Figure 4.13: Mock-up: check cherry picks of conflicts (F6, F7)

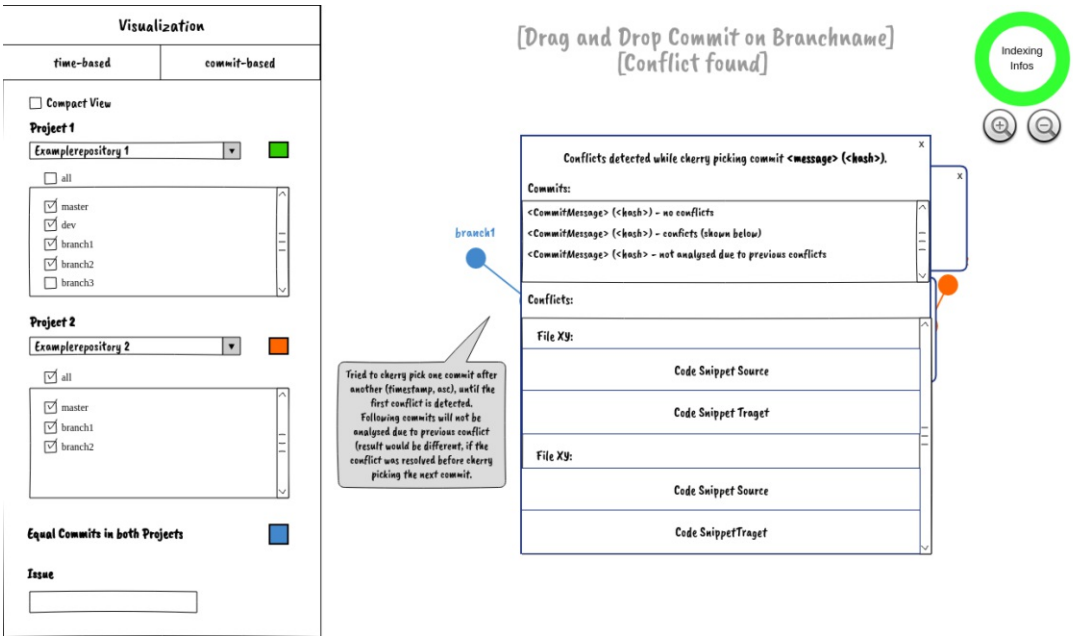


Figure 4.14: Mock-up: check cherry picks - conflict found (F6, F7, F8)

For this functionality the software engineers should not only be able to inspect if one commit can be cherry picked successfully but also if the cherry pick of multiple commits will be successful (see Figure 4.13). Therefore, the user should be able to choose multiple commits. Then the selected commits should be sorted after the commit date in ascending order and cherry picked one after another. Similar as the conflict modal from the rebase check this modal should also provide the information up to which commit the commits could have been successfully picked, when the conflict occurred, and which commits were not analysed because of the conflict (Figure 4.14). Additionally, the conflicting code sections should be shown (F8).

Often, not all information of the Git history is of interest for system engineers. Therefore, developers should be able to filter commits (F14) in an additional way as the previous represented information limitations like compacting the graph or specific sections and to fade out commits of a specific graph. For these supplement filter methods, the user should be able to choose if the filtered commits should be highlighted, or if only they should be visible in the graph. Additionally, the plan was that these filters can be combined with one another. Two ways of combining the filters came to mind. First, combination results in a wider search area, which means they will be combined with an „or“ link. Second, the combination of filters will restrict the search area further, therefore, the filters will be combined with an „and“ link. Based on the own experiences in the field most use cases would have required to further restrict the search than the other way around. For the filters the developers should be able to limit the commits in a timely manner. This functionality should include to filter commits after a specific date, to filter commits within a specific time frame but also to filter a specific subtree, which means showing all children of a specific commit inclusive. In addition, it can also be helpful to see which commits were made by a specific committer or were integrated from a specific author.

Developers also stated in the literature that it would be nice to see the commits of a specific issue (F10). Therefore, the visualisation should provide such a functionality. The user should be able to select an issue which will then result in the commits of this issue being marked within the graph (Figure 4.15).

Initially, it was also planned to show the mergeability and rebaseability of the branches and the cherry pickability of the commits in a passive way (Figure 4.16). However, due to the increased expenditure of time and complexity of this feature, this idea was not included in the scope of the thesis. While hovering over a commit the user should see the cherry pickability of the commit. A cherry pickability of 100% would mean that the commit can be cherry picked into each branch without conflicts. Otherwise, the branches would be displayed where a conflict would occur. The mergeability and rebaseability of branches should be visualised in a similar way. While hovering over a branch label, a tooltip should show the percentages of the mergeability and the rebaseability as well as the branches where the merge or result would result in a conflict.

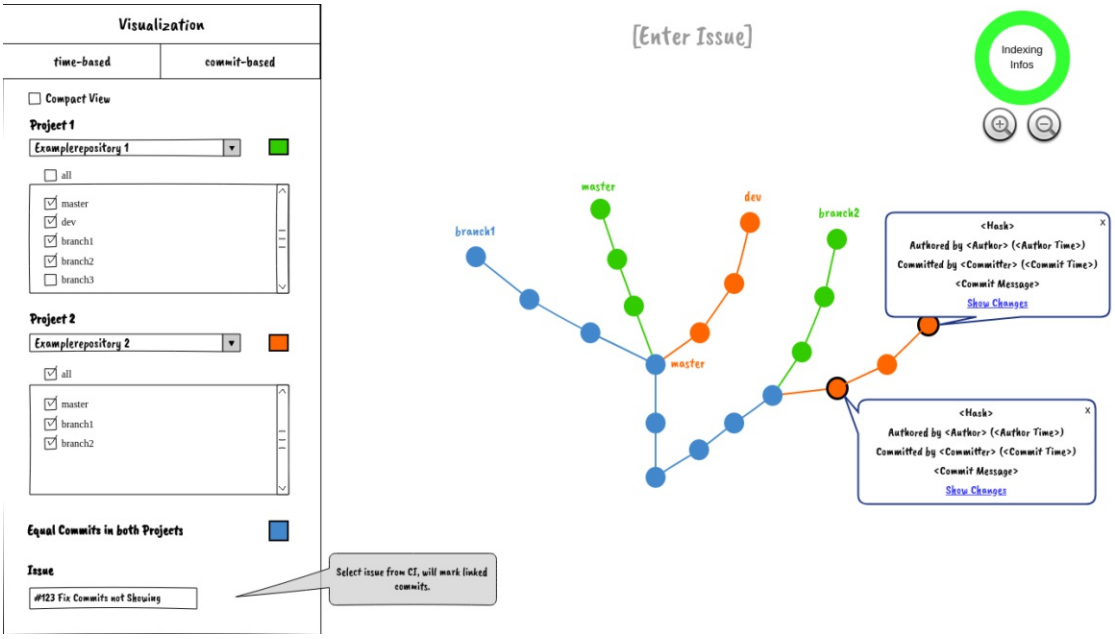


Figure 4.15: Mock-up: highlight commits of a specific issue (F10)

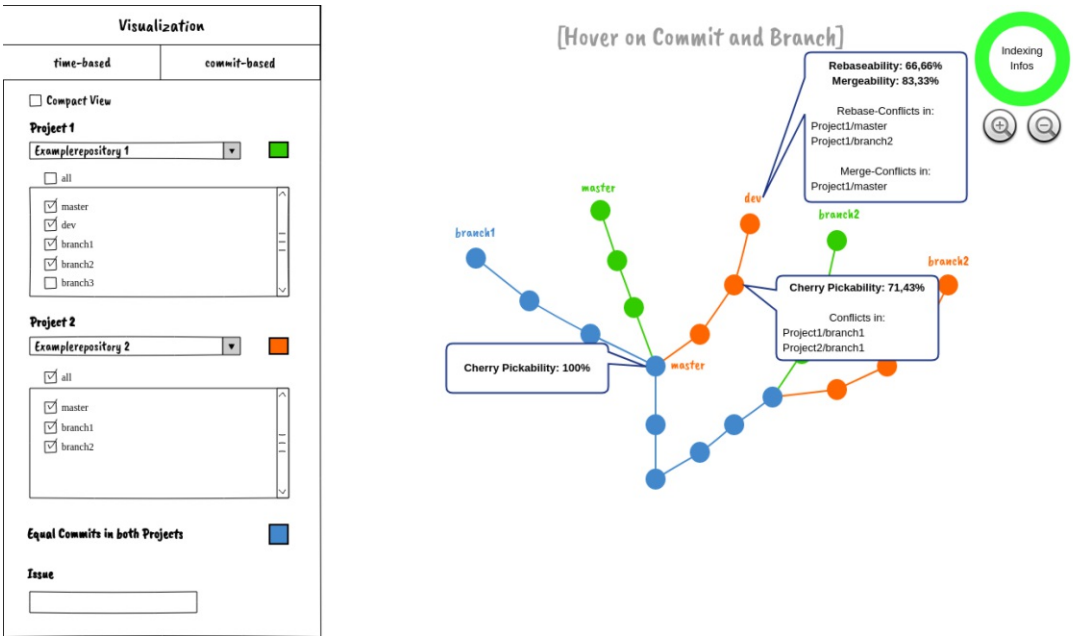


Figure 4.16: Mock-up: passive view for cherry pickability, mergeability and rebaseability

CHAPTER 5

Semi-Structured Expert Interviews

This section provides insights of the semi-structured expert interviews for rating and ranking the proposed features of the conflict awareness visualisation. First the plan of these interviews will be discussed and afterwards the result of them. Lastly, the threats of validity will be presented.

5.1 Plan

In order to get an insight of how important software engineers find the proposed features of the conflict awareness visualisation (Table 4.1) semi-structured expert interviews were conducted. The results of these interviews were also used to create a ranking in which order the proposed features were implemented in the prototype. Therefore, three software engineers were selected as participants for the interviews.

As support and guideline for these interviews a questionnaire was created. The questionnaire was divided into three sections. The first section contained questions about the person itself. In the second section questions about the experiences of the participant could be found. The last section contained the questions about the importance rating of the proposed features. The whole questionnaire can be found in the appendix (9).

All interviews were carried out remotely via Zoom[1] due to the COVID-19 pandemic. It was estimated that each interview will last about an hour. During the interview two interviewers and the interviewee were present. One interviewer took over the moderation, and the second interviewer was responsible for taking notes. At the beginning, the participants received a short overview on what the interview is about. Afterwards, the

[1] https://zoom.us/

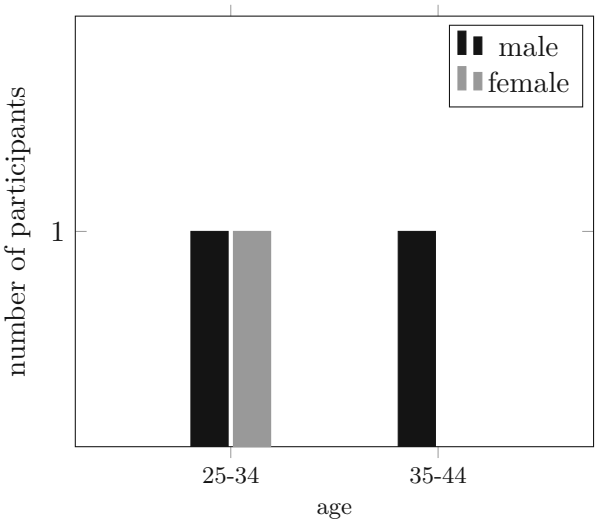


Figure 5.1: Demographics of the participants

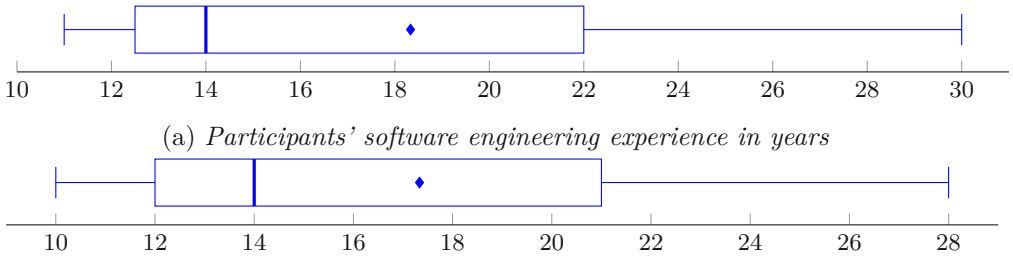
outlet of the interview was presented. If the interviewee had no further question, the display of an interviewer was shared with the attendants showing the open questionnaire. Then the participant had time to read the questions of the questionnaire, to ask questions if something was unclear and to explain why he or she chose a specific answer. The final answer to a question was filled in by an interviewer. This approach was chosen to mitigate potential technical errors because of different system setups of the participants.

5.2 Results

The results of these expert interviews were used to get an impression on how useful experts see the proposed features and if the visualisation should contain not mentioned features. The results also provided an approximate roadmap on what features should have been implemented first according to their rated importance.

5.2.1 Demographics and Experiences

All participants were asked to provide some information about themselves and their experiences in the field. The distribution of the participants' age and gender can be seen in Figure 5.1. The experiences of the attendees can be found in Figure 5.2 and 5.3. Participant 1 had about 30 years of experiences in software engineering and about 28 years with VCSs. He used different VCSs before, namely Git, Subversion, Mercurial, CVS, PVCS, RCVS, TFS, SourceSafe and Patches. Participant 2 had around 14 years of experiences in both software engineering and working with VCSs. During this time, he used Git and Subversion as VCSs. Participant 3 had approximately 11 years of experiences as a software engineer and worked for about 10 years with the VCSs Git



(a) *Participants' software engineering experience in years*

(b) *Participants' experience with Version Control System*

Figure 5.2: Experiences of the participants

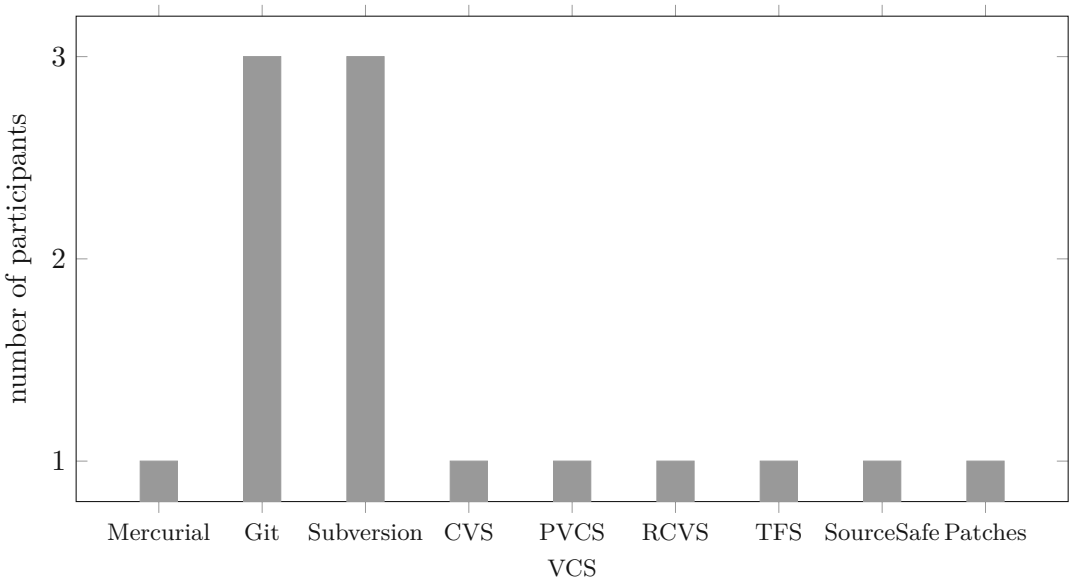


Figure 5.3: Version Control Systems used by the participants

and Subversion. All three respondents knew the principles of forking and PRs and each participant had also forked a project and merged a PR in the past (Figure 5.4.

5.2.2 Preference between Merging and Rebasing

Each participant had different preferences between merging and rebasing (Figure 5.5). Participant 1 was neutral towards rebasing and merging and stated that the choice between those two depends on the context of the task. When integrating features, he usually merges these changes back. For the integration of bug fixes, a rebase is preferred. It was also stated, that the commit tree can become confusing if the changes are only integrated using merges because each merge will create a separate merge commit.

The problematic of additional merge conflicts is also known by participant 2. His preferred method is the rebase because he tends to commit rather small changes instead

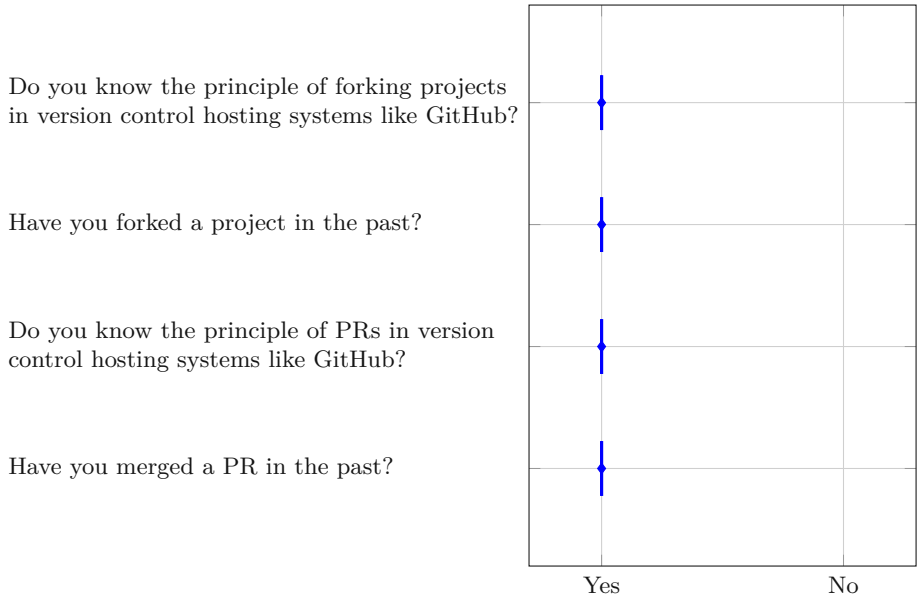


Figure 5.4: Forking and Pull Request experiences of the participants

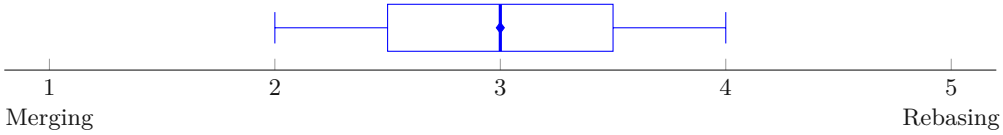


Figure 5.5: Participants' preferences between merging and rebasing

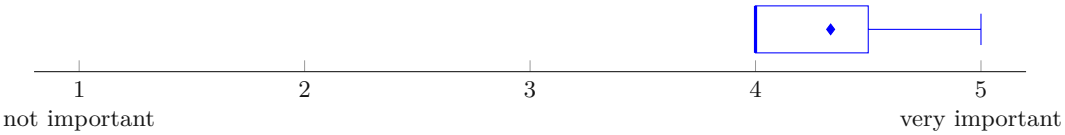


Figure 5.6: Importance for showing the divergence between forks

of introducing commits with a lot of changes. But if larger commits have to be integrated a merge will usually be used because the merge commit will document these changes.

On the contrary, the third respondent prefers merging over rebasing. For her, she usually only rebases when changes have to be integrated into the current branch she is working on.

5.2.3 Showing the Divergence between Forks (F1)

Participants 1 and 3 find it rather important for the visualisation to show the divergence between forks and participant 2 finds such feature in general very important (Figure 5.6). Participant 1 stated that he prefers to work with a shell, but he knows that such a

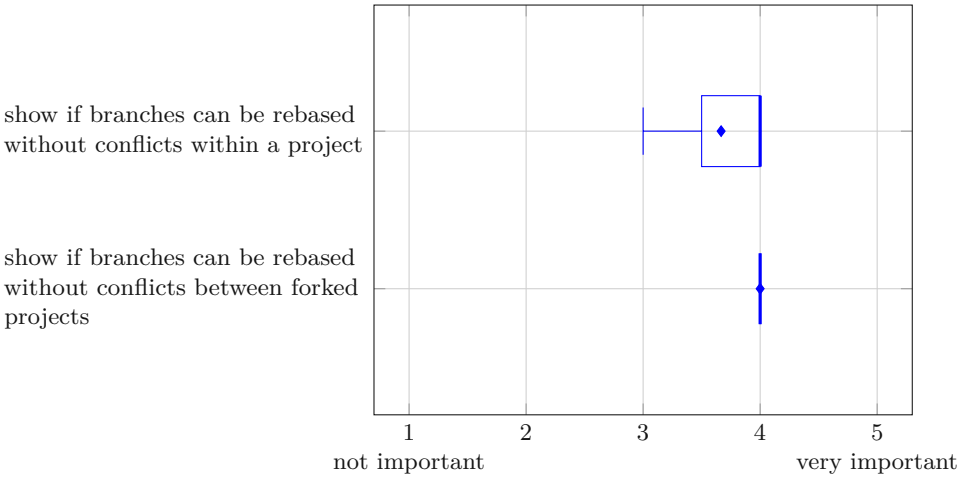


Figure 5.7: Importance for showing the rebaseability

feature would be very important by others. For himself, he is normally not interested in specific commits, only if he wants to know who did what. In general, this respondent looks at end to end diffs on the file level.

5.2.4 Showing if Branches can be rebased without Conflicts (F2, F3)

For participant 1 and 2 it was rather important that the conflict awareness visualisation shows if branches within a project but also from a fork can be rebased without a conflict. On the other hand, for participant 3 it is more important that the visualisation focuses on forks rather than on intra project rebases. Therefore, the awareness visualisation for rebases within a project was rated neutrally. The results can be seen in Figure 5.7. Participant 1 stated that this feature would be interesting, especially with large community projects if branches from a fork are still integrable or not. Participant 2 noted that such a check would be important, especially working with forks, but due to his working style, this feature was rated as rather important in both cases.

5.2.5 Showing if Branches can be merged without Conflicts (F4, F5)

For the proposed feature, if the prototype should show whether branches can be merged without conflicts or not, all three participants rated the interproject check equal to the check between forks (Figure 5.8). Participants 1 and 3 found these proposed features rather important and participant 2 very important. For participant 2 the check for merge conflicts is more important than the check for rebasing due to his working style. He usually has larger commits to merge than to rebase.

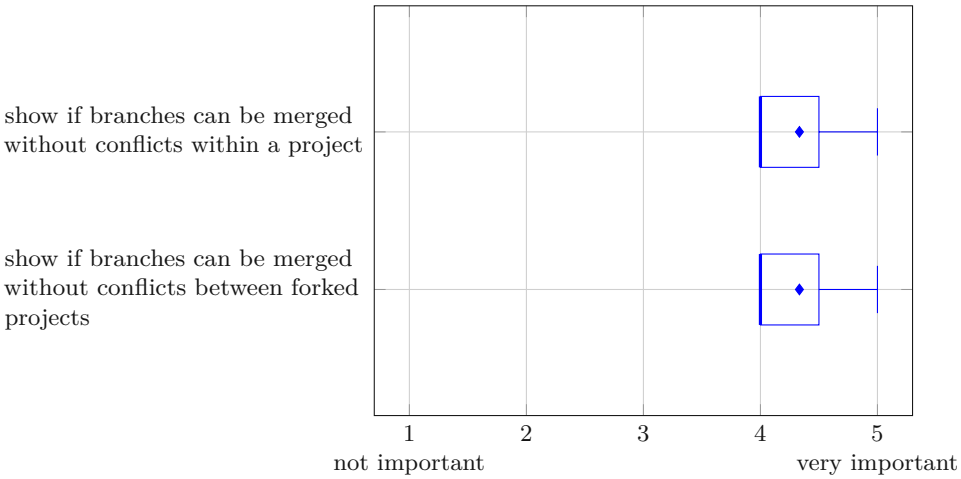


Figure 5.8: Importance for showing the mergeability

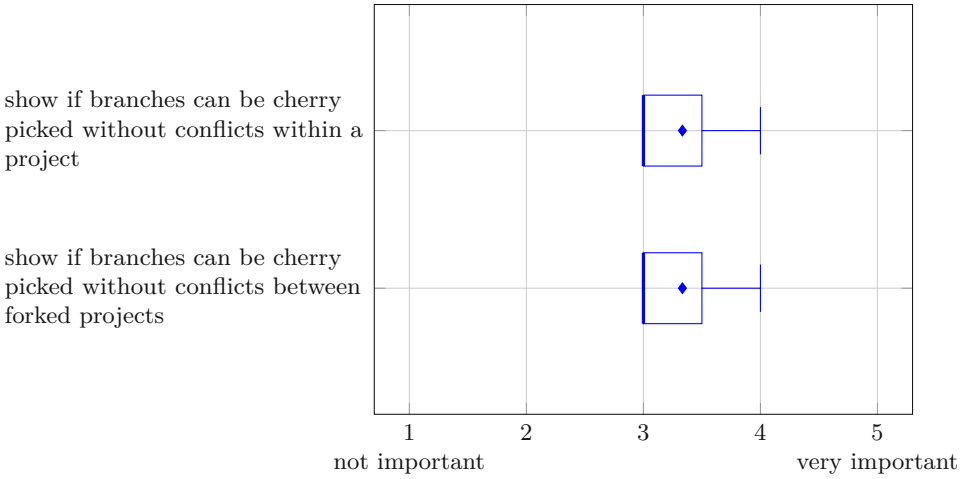


Figure 5.9: Importance for showing the cherry pickability

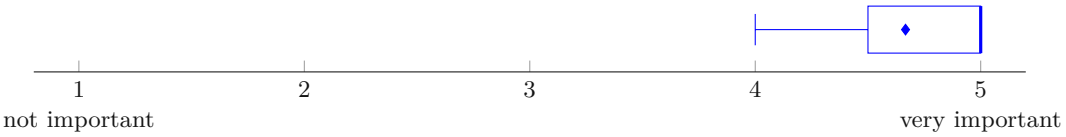


Figure 5.10: Importance of showing the code sections of conflicts

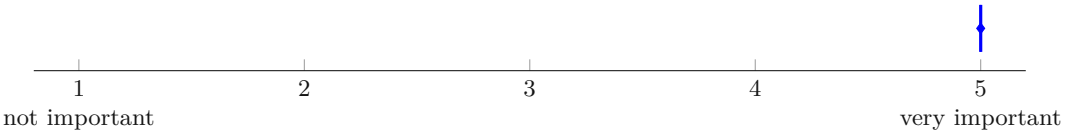


Figure 5.11: Importance of showing dependent commits

5.2.6 Showing if Commits can be cherry picked without Conflicts (F6, F7)

In case of the cherry pickability of commits, participant 1 rated such a visualisation as rather important. This interviewee stated that cherry picks are almost every time possible, except, for example, if a file is missing because it was created in a previous and not integrated commit. The other two participants were neutral about this feature and rated it as a nice to have (Figure 5.9). Participant 3 sees cherry picks as antipattern and that such commits should have been integrated with a merge or a rebase before. Participant 2 noted that he usually only cherry picks commits with just a few changes. Therefore, the chance of a conflict would be rather low.

5.2.7 Showing the Code Sections of a found Conflict (F8)

If a conflict in the check for the rebaseability, mergeability or cherry pickablility occurs, participant 1 and 3 rated it as very important, that the conflict awareness visualisation displays the code sections of the conflict and for participant 2 this feature is rather important (Figure 5.10). During the interview participant 1 also stated that it would be useful to not only see the conflicts, but to also edit these such that this conflict can be resolved at the time. Participant 2 noted that the code snippets of the conflict are sometimes too small because it might be possible that the context needed to understand the conflicting code is missing. For participant 3 it would be interesting to see at which commit the conflict occurs.

5.2.8 Showing the Commits a selected Commit depends on (F9)

The visualisation of dependent commits is rated by all participants as very important (Figure 5.11). Participant 3 noted that not only the syntactic code dependencies of the commits should be shown, but also the semantic dependencies, like missing functions in another class which was used within the commit. If the semantic dependencies are

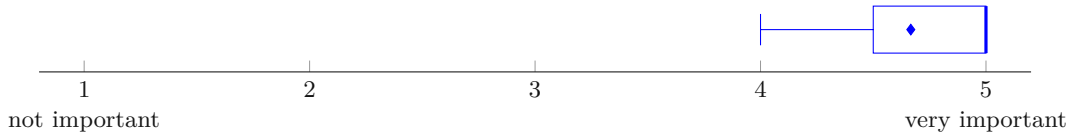


Figure 5.12: Importance of showing commits of a selected issue

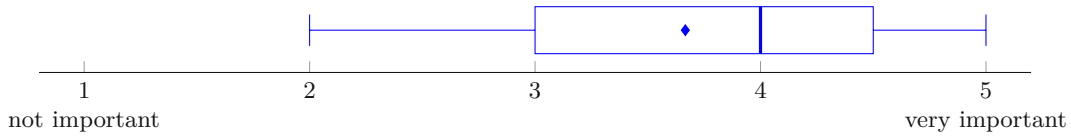


Figure 5.13: Importance of showing the metadata of a commit

missing, this may raise false hopes in software engineers and he or she thinks that all is working fine if no conflict was detected.

5.2.9 Showing Commits of a selected Issue (F10)

Participant 1 and 2 stated that a feature that marks the commits of a specific issue is very important and the other participant rated it as rather important (Figure 5.12). Participant 1 justifies his choice because he often looks at nearby commits of a specific one to see, if the selected commit was the last commit which referenced a specific issue or not. This feature would be helpful in such situations. This participant also stated that for him it would not be as useful if only issues from GitHub can be selected.

5.2.10 Showing the Metadata of a Commit (F11)

If the conflict awareness visualisation should show the metadata of the commits across the selected project, the respondents all have a different rating. For participant 2 this feature is not really significant, while participant 3 rated it as rather important and participant 1 very important (Figure 5.13). Participant 1 stated that especially the commit message and the author are significant in such a visualisation, because it can be the case that changes of a specific author should be reviewed more accurately. Participant 2 justifies the decision that for him the most interesting information would be the code, and not who is the author of a commit for example. Participant 3 stated that it would be nice to have such metadata within the tool because then a switch to another tool which provides this information is not necessary.

5.2.11 Making the Visualisation compact (F12)

All attendees of the interviews marked it as rather important, that the visualisation can be made compact at will (Figure 5.14). Participant 1 stated that it would be useful when it is possible to, for example, select an issue and that all commits are shown in a compact view, except the commits of that specific issue. Participant 3 noted that especially in

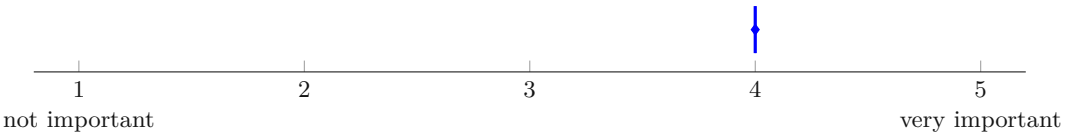


Figure 5.14: Importance of making the visualisation compact

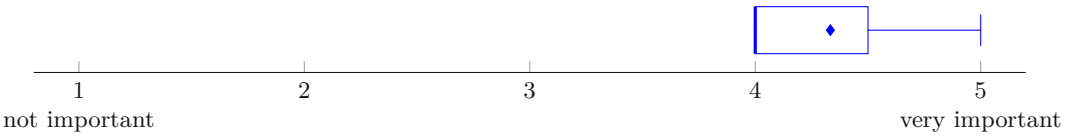


Figure 5.15: Importance of hiding commits of selected branches

large projects such visualisations can overstimulate the viewers. But she also pointed out that this feature would only be useful if someone can decide on what should be compacted and what not. An all or nothing choice would not really make sense.

5.2.12 Hide Commits of selected Branches (F13)

The possibility of hiding commits of selected branches is rated as very important by participant 3 and as rather important by participant 1 and 2 (Figure 5.15). Participant 2 noted that this feature can be helpful if someone wants to compare more than two branches.

5.2.13 Provide Possibilities for filtering (F14)

The interviewees were asked about different possible filter scenarios of the prototype. For the filtering option, two suggested methods were suggested. A filter which only shows the filtered commits and one which shows all commits but highlights the filtered ones. The participants were asked how important they would find such filter for commits:

- after a specific timestamp
- within a specific time frame
- of a specific author
- of a specific committer
- after a selected commit (subtree)

The results can be seen in Figure 5.16.

For participant 1 it is interesting to provide such filter for authors because then it is possible, for example, to get a brief overview on how much a person worked on the project.

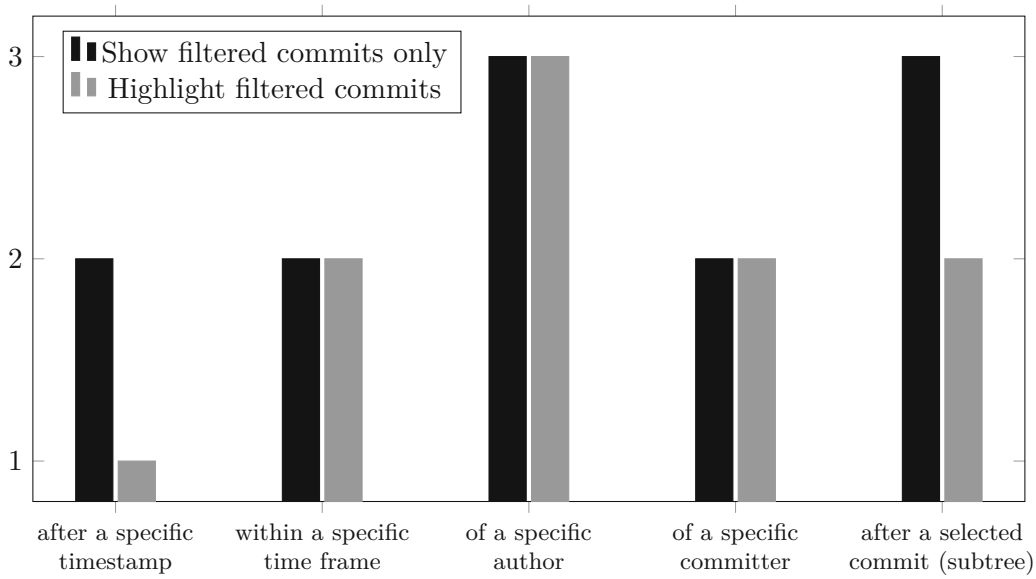


Figure 5.16: Importance of providing various filter options

On the other hand, a filter for the committer is not important for him. He stated that the committer often is the same person as the author, or the committer will have many commits due to a limited number of persons with write permissions on the project for example. This interviewee also noted that it may be important to see coherences between commits when filtering for a specific time frame. Therefore, it is not only important to show only these commits, but also to highlight them and display the others as well. For the filter after a specific timestamp, such coherences are usually not important.

The participants were also asked if they find other filter options important. New suggestions included a *git bisect* like intersection between two commits, a filter for commit messages and a filter that shows in which commits a specific file was changed.

5.2.14 Roadmap for the iterative Prototype Development

Table 5.1 shows the ranked proposed features of the prototype based on the ratings from the interviews. This ranking shows the basic roadmap for the implementation. Features with a higher rating were added sooner than features with a lower rating if reasonable. For example, it would have not been reasonable to implement the feature „Showing the divergence between two forks“ in a later step, because the other features were built on top on it.

The table excludes the filter options (F14) due to a different question style. Based on their ranking, first, a filter for commits of a specific author was planned. This filter included an option for highlighting and for showing only the filtered commits. Next, the implementation of the filters after a specific commit were implemented. Afterwards, the

Proposed Feature	Mean
F9 Show the commits which the selected one depends on	5,00
F8 Show the code sections of the conflict if one would occur at the rebase, merge or cherry pick	4,67
F10 Show the commits of a selected issue	4,67
F1 Show the divergence between two forks	4,33
F4 Show if branches within a project can be merged without conflicts	4,33
F5 Show if branches of forks can be merged without conflicts	4,33
F13 Hide commits of selected branches	4,33
F3 Show if branches of forks can be rebased without conflicts	4,00
F2 Show if branches within a project can be rebased without conflicts	3,67
F11 Show the metadata of a selected commit	3,67
F6 Show if commits within a project can be cherry picked without conflicts	3,33
F7 Show if commits of forks can be cherry picked without conflicts	3,33
F12 Provide a compact view of the visualisation	3,00

Table 5.1: Proposed features sorted after the importance rating

filters for commits of a specific committer and for commits within a specific timeframe were added. Lastly, the filters for commits after a specific timeframe were implemented.

5.3 Threats to Validity

In this section threats to validity of the conducted expert interviews are presented.

5.3.1 Number of Interview Participants

For the semi-structured interviews three experts were selected. This number of participants could not provide sufficient insights of the importance of the proposed features over the wide mass. The goal of these interviews was to get a small insight of the opinions of software engineers. A more detailed analysis would have gone beyond the scope of a master's thesis.

5.3.2 Mock-ups as Visualisation Examples

Some questions included mock-ups from the requirements analysis to provide a visual description as well as a textual description of the feature. This could have led to a rating of the provided mock-up and not to a rating of the feature directly. For example, it was possible that a participant would have rated the feature as not important because he or she disliked the provided image. In order to mitigate this behaviour, the participants received an explanation that the mock-ups should only provide additional context to the proposed feature and that the idea itself should be evaluated.

5.3.3 Wording in the Questionnaire

During the interview some questions were not immediately understood by the participants. For example, one participant could not answer question 6, if he knows the principle of forking projects in version control hosting systems. For this attendee it was not clear if the term forking also includes for example downloading a project from a platform, or only using forking mechanisms provided by version control hosting systems. Due to the structure of these interviews the participants were able to ask questions and to clear up ambiguities.

5.3.4 Preference between Merging and Rebasing

Participants of the survey may have a preference between merging and rebasing. This could have led to a higher or lower rate of the proposed features related to these two techniques. To mitigate this threat and to get better insight of the context of their choices, the attendees were asked about their fondness.

CHAPTER 6

Implementation

The following chapter describes the implementation of the prototype, the encountered problems during this phase and differences between the mock-ups from the conceptual design (Chapter 4) and the actual implementation. Before going into the implementation in more detail, Section 6.1 provides a brief overview of Binocular, the program that is expanded in this thesis. The prototyping was split up in seven iterations. In iteration 0 (Section 6.2) the existing data mining algorithm of Binocular was extended to the needs of the new visualisation. After iteration 1 (Section 6.3) the basic visualisation was set up and the commits of a selected issue could have been highlighted. The conflict checks were implemented in iteration 2 (Section 6.4). In iteration 3 (Section 6.5) the features to show or hide specific branches and to highlight the commits a selected one depends on were added to the prototype. Iteration 4 (Section 6.6) included the implementation of the filters and after iteration 5 (Section 6.7) the user was able to compact or expand the whole graph or sections of the graph. In the sixth and last iteration, bugfixes and improvements to the visualisation were made. These improvements are not discussed in a separate section, but rather in the corresponding sections to provide the complete picture of the implemented features.

6.1 Binocular as Base

Grabner et al. [34] designed a tool to visualise time-oriented data from different software engineering tools. The program combines data from VCSs, CI systems and issue tracker into different visualisation types. The program described in [34] provides three different visualisations: Change Impact Wheel, Code Ownership River and Activity Peak Dial. The basis for the new prototype extension has changed slightly since the paper was published. A new visualisation, a „Dashboard“, was added, the „Activity Peak Dial“ visualisation was renamed to „HotspotDials“ and the „Change Impact Wheel“ visualisation was renamed

to „Issue Impact“. The names of the current implementations are used for the following paragraphs.

The authors [34] chose a simple client-server based architecture. For storing data an ArangoDB database is used. The backend, a Node.js application, is responsible for the data gathering using different indexers for hosting the frontend and for installing the GraphQL-service which allows the frontend to query data from the database. The frontend is a single-page web-application that uses React[1], D3.js[2] and Redux[3].

As described above, three of the four visualisations are introduced in [34]. The „Hotspot-Dials“ visualisation should provide an overview of when how much work happened. The „Issue Impact“ visualisation shows impacts of issues on the code base and the „Code Ownership River“ visualisation summarises how much code a contributor owns. The „Dashboard“ provides information about issues, changes and continuous integration for selectable time buckets.

6.2 Iteration 0: Data Mining

Prior to the implementation of the new features discussed in Section 5.2 the mining algorithm for collecting the needed repository information had to be extended. The existing implementation allowed the data mining of only one base project. But this prototype needs information about the base project and additionally data about its forks and its parent project. This information includes the metadata of commits, information about branches of a project and the connections between branches and commits.

There were primarily two choices for the data storage of the other repositories. One was to store the metadata of the other projects in a separate database. The second one was to store all the data in one database and to put an identifier to the needed collections. For this prototype the second variant was selected in order to have all the needed data collected in one spot.

Three additional fields were added to the commit collection: the projects, the author and the corresponding date. The projects field is an array of strings. It contains all the projects the commit can be found. The project itself is a unique combination of the repository owner and the repository name. For example, the key for the Binocular repository of the GitHub user INSO-TUWien would be „INSO-TUWien/Binocular“. This combination allows the prototype to uniquely distinguish between forks. The repository name alone exists multiple times, but in combination with the repositories owner it will become unique. The proposed features of this prototype also include the author of the commit and the corresponding time. Like the committer, the author consists of the username and the e-mail address of the GitHub user. For the data mining of the commits

[1] https://reactjs.org/
[2] https://d3js.org/
[3] https://redux.js.org/

the current implementation was used and extended such that this additional information is also saved in the database.

Additionally, branch information is needed for this visualisation. Therefore, a new collection „branches“ was added to the database. This collection contains the following information: a branch key, the name of the branch and a list of the projects the branch can be found inclusive the corresponding sha of their head. In order to save or update the branch data the implementation of the Git indexer was extended. After indexing the commits, the indexing of the branch data starts. For the index process a bulk creation, update and deletion was chosen because its more performant than saving, updating and deleting all entries after another via the database API and because it makes the stopping and restarting of this algorithm easier to understand. The algorithm searches all branches that were newly created and each branch which was updated after the last indexing. A branch will be updated if the sha from the head retrieved from Git is a different one than the stored sha in the database. Furthermore, the algorithm also gets all branches that were deleted in Git but are still saved within the database. Afterwards, all new and updated branches are stored and updated using the „UPSERT“ statement within one database transaction. Additionally, the connections between the branches and their commits will also be added to the collection if they do not already exist. At last, for each deleted branch the entry and their corresponding commit connections are removed from the database. The deletion consists of two steps. First, all the connections between the branches and their commits will be deleted and the corresponding entry from the list of projects and head shas will be removed, all in one transaction. Second, each branch entry that has no projects and head shas in the list anymore is deleted from the database.

6.3 Iteration 1: Setting up basic visualisation

Before the features of the prototype can be implemented, the basic visualisation of the projects' Git histories must be created. The first step was to add a new tab „Conflict Awareness“ for the visualisation. The configuration section of the visualisation will be extended for each feature if necessary. For the representation of the Git commits and the history the plan was to only use the D3.js library for this representation. Unfortunately, no suitable predefined data structure could have been found. The tree structure for example lacked in being able for nodes to have multiple parents and the force-directed graph was not shown in a structured way, which made it impossible to understand the history in an early time of manner. After another search, the dagre-d3[4] library was found. This library uses the directed graph layouts of the dagre[5] library and renders them using D3.js. For the prototype the „tight-tree“ layout fitted the best. Using this, it was possible to use the functionalities of D3.js in order to manipulate the rendered graph.

At first, only a horizontal view of the graph was implemented from the earliest commit at the bottom from the latest commits at the top. In the last iteration, other layout

[4]https://github.com/dagrejs/dagre-d3
[5]https://github.com/dagrejs/dagre

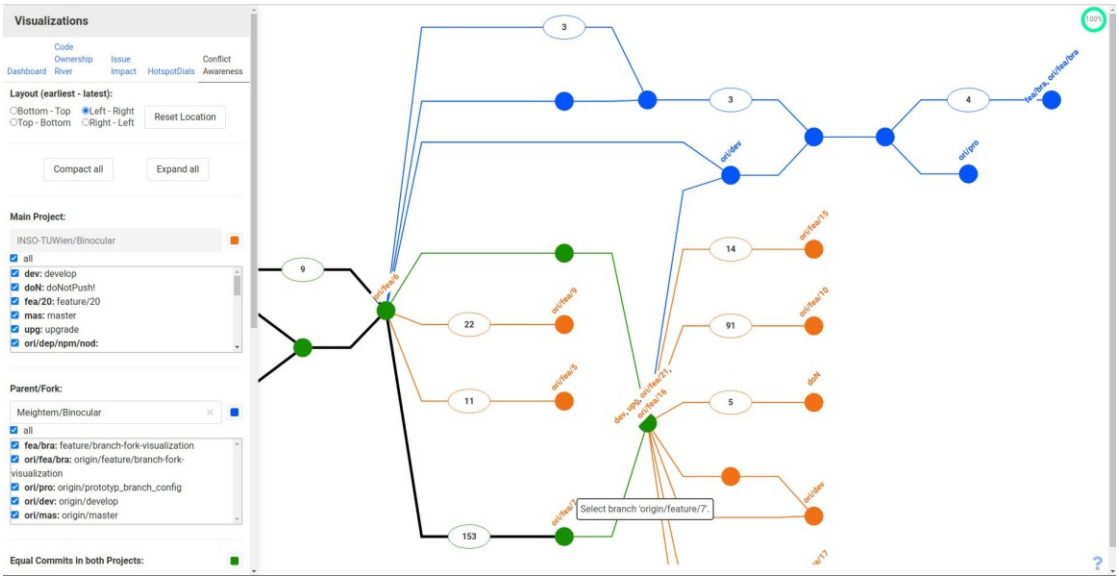


Figure 6.1: Base view of the new visualisation with vertical layout (earliest to latest)

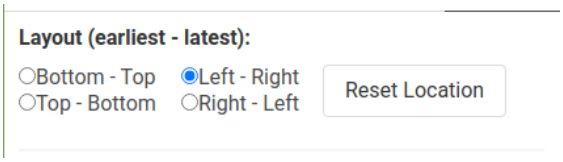


Figure 6.2: Possible layout selections

possibilities were added to the prototype. On the top of the configuration section the user can select between four different base layout styles (Figure 6.2). The selected style can be horizontal where the earliest commit is on the bottom and the latest commits are on top or vice versa. Another possibility selectable by the user is a vertical representation where the earliest commit is either on the left or on the right side. An example of a vertical view can be seen in Figure 6.1.

When the graph is rendered, the user can zoom in and out using the mouse wheel. The graph can be moved by grabbing the background with the left mouse button and then moving it to the desired position. After re-rendering the graph, the previous location and zoom level of the user will be preserved. Especially when switching the layout between the horizontal and vertical view it is possible that the engineer will not see the graph anymore. Therefore, a button for resetting the position and zoom level was added to the layout configuration setting (see Figure 6.2).

As already mentioned in Chapter 4 commits are represented with filled nodes, collapsed commit sections are shown with ellipses with the number of commits it holds and the edges describe the parent-child relationships between commits. The graph does not directly show the timeline of the commits, but rather the relationships between them.

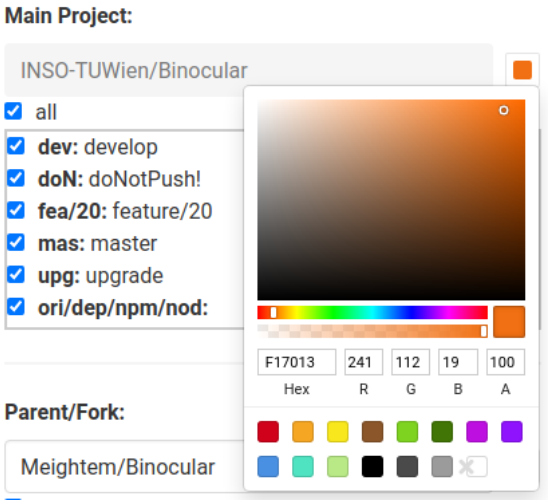


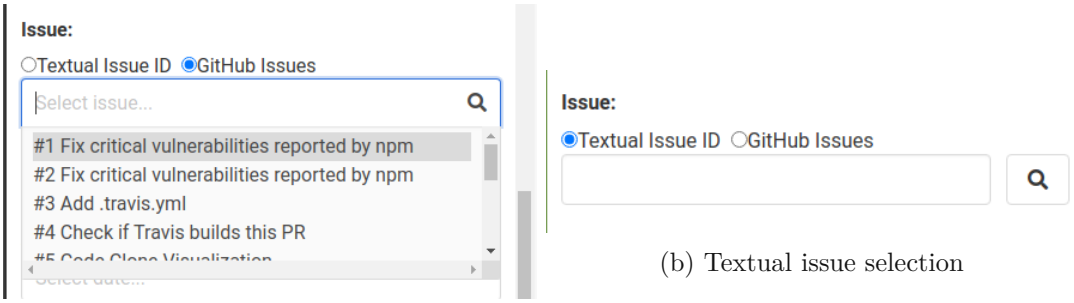
Figure 6.3: Selecting the colour for the main project

This means that it is possible for a newer branch to be behind a branch with older commits if it has more commits than the newer branch.

The graph also includes the branch references over their head commits which can be seen in Figure 6.1. At first, the whole branch name was shown in the visualisation. The problem with this was that this became confusing if more branches shared the same head. The branch reference consists of the first three letters of the branch name sections split by „/“. When hovering over a reference the whole branch name is shown in a tooltip. Additionally, the branch history is highlighted. For this the edges are marked because there are already different highlighting styles for the commit nodes. Both can also be seen in Figure 6.1.

Colours are used in order to show the divergence between two repositories (F1). When the user selects the parent or a fork of the base project, the commits of both repositories are fetched, and the graph will be re-rendered. During this process, the parent or fork will be cloned in a preconfigured projects folder in order to prevent unwanted changes in the main project while checking for conflicts (see Section 6.4). The path to the projects will be defined in the „.pupilrc“ configuration file of the Binocular project. All commits, edges and branch references will then be coloured according to the repository they belong to. There are three types of colours for project differentiations. The colour of branches, edges and branch references is dependent if the data can only be found in the main project, the chosen other project or if it exists in both projects. In the example (Figure 6.1) the colour orange belongs to the main project, blue to the chosen other repository and green to commits, edges and branches that can be found in both projects. Basically, the colour of the combined view shows the forking point of the repositories. As already mentioned in Chapter 4 the user should be able to choose the preferred colours. In order to change a specific colour, the software engineer just needs to click on the corresponding

Figure 6.4: Detailed commit metadata



(a) Issue selection of GitHub

(b) Textual issue selection

coloured square on the right part of the configuration section and then select the colour using the colour picker (Figure 6.3). The react-color[6] library was used for this.

The visualisation also shows metadata of the commits (F11). When hovering over a commit node its basic metadata is shown using a tooltip. This information includes the sha of the commit, its author and the timestamp, its committer and the timestamp and the shortened commit message. Such a tooltip can be seen in Figure 6.6. By double clicking on the node more detailed information are loaded (see Figure 6.4). This includes again the sha of the commit, the whole commit message in a shrinkable card and the changes introduced with this commit.

Another feature implemented in this iteration is the commit highlighting of a specific issue (F10). A participant of the semi-structured expert interviews stated (see Section

[6]https://casesandberg.github.io/react-color/

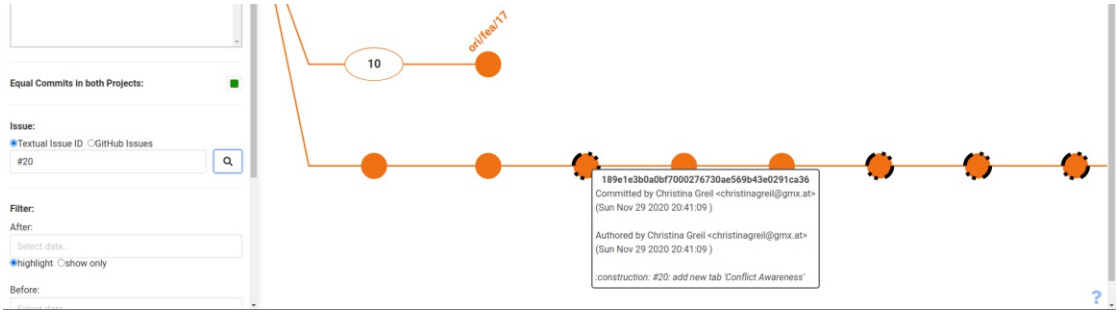


Figure 6.6: Highlighted commits of a selected issue

5.2.9) that this feature would be more useful if not only commits from GitHub can be selected. Therefore, the prototype includes two ways to select an issue. The first one is that the user can select the issue of the base project ticketed on GitHub (Figure 6.5a). The issue id will then be searched in all commit messages and the corresponding commits are highlighted with a dot-dashed border around them. When the graph is compacted, the number of found commits are shown in the label of the collapsed node as used for the filters (see Section 6.6). This feature goes hand in hand with the other filters resulting in a more restrictive filter result. The second option for the user is to provide a textual representation of the issue selector (Figure 6.5b). With a click on the search button the input is then again searched within the messages of all commits. Figure 6.6 shows a textual issue search with the id „#20“. Actually, all commits of this history line should be marked but for some commits the issue number was forgotten in the commit message. Therefore, these commits will not be shown by the filter although they are also part of the issue. Such a finding, the missing link between the commits and the issues, was also documented in different studies ([3, 12]). Because this is a common problem, several studies ([40, 47, 54, 55, 61, 62]) have looked at the recovery of missing issue-commit links.

6.4 Iteration 2: Checking for Conflicts

Having set up the basic structure of the visualisation the next chosen feature to implement was to show the code sections of the found conflicts (F8). To show these, some lower ranked features are required, namely the merge checks (F4, F5), the rebase checks (F2, F3) and the cherry pick checks (F6, F7). Because these features combined can be seen as one larger feature of the visualisation, these were picked for an earlier iteration than the highest rated feature.

All three types of conflict checks were implemented one after another, but the basic algorithm was reused. In order to provide an accurate result whether the operations can be successfully performed without creating a conflict or which conflict will occur, each check will perform the actual Git operations. The result will then be presented to the user. As already mentioned in Section 6.3 all needed repositories will be cloned in a

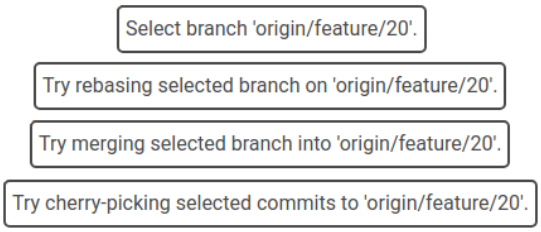


Figure 6.7: Action tooltips shown when hovering over a branch reference

preconfigured folder to prevent potential side effects within the base project.

A difference between the previously created mock-ups and the actual implementation was the starting event the engineer must do in order to trigger the check. In the mock-ups the start was described with a drag and drop event of the commits or branch labels onto another branch label. In the prototype the user first must select either a branch or one to multiple commits. If commits are selected and the user clicks on a branch reference, the software engineer triggers a cherry pick check. If a branch is selected there are three possibilities. At first, the user can left click onto another branch which will switch the current selection to this branch. The second possibility is to click on a different branch with the shift key pressed. This will trigger a check for conflicts of a merge of the previously selected branch into the currently clicked branch. When the ctrl key is pressed instead of the shift key a check will be performed if the already selected branch can be rebased onto the clicked branch without conflicts. It is not possible to have a branch and commits selected simultaneously. The escape key can be used to reset the current selection. The reason for this indifference was a better user experience. On the one hand, the position of the graph can be changed by dragging it which may lead to not intended repositioning or dragging branches and commits. On the other hand, the graph can get large over time. If the user must drag and drop the elements for triggering the checks it can be possible that the destination is not visible on the screen especially if the graph is fully expanded. The user would have to zoom out to the point where the destination points are all visible, which can make the graph elements extremely small, or filters must be used to shrink the visible number of commits and branches. Using the implemented approach, the software engineer can first choose a selection and then position the graph easily to the destination point and trigger the desired check. To further improve the user experience, a tooltip showing the action which is going to be triggered will be shown when hovering over the branch reference in the visualisation. All possible tooltips showing the current action can be seen in Figure 6.7.

After the trigger from the frontend, the backend will perform the chosen action and report the result back to the frontend. The algorithm works as follows: If the branch that should be checked out is a local one of the base project will be renamed to „root/[name]“. This is necessary because in the next step the local base repository will be cloned again into the configured projects folder, the origin remote will be changed to point to the

Figure 6.8: Success message for the cherry pick check

GitHub URL of the repository and a new remote „root“ will be created that references the local repository. This allows to get the latest version of the local remote branches by pulling from all remotes. In order to quickly reset the done changes after the conflict check the repository of the branch that should be checked out will be backed up by creating a local copy. If the repository of the other branch is a different one, another remote will be created if it does not already exist. A pull from all remotes will update the references again. The next step is to check out the branch. For the merge and cherry pick check that branch in which to merge or cherry pick is chosen and for the rebase the branch that should be rebased onto another is checked out. Subsequently, the merge, the rebase or the cherry picks are performed. When cherry picking, the commits will first be sorted after their commit date ascending. If a conflict is found the conflicting data will be retrieved. The result will then be returned to the frontend.

At first, the entire logic of this algorithm should be made with the nodegit[7] library. In the last iteration, the checks were refactored such that nodegit was only used to retrieve the conflict data. Instead, scripts are used to prepare the repositories and to carry out the checks. The reasons for this decision were problems with missing or inadequate documentation and incomprehensible behaviour of the functions, especially after repeated checks.

After the check the user would get information if a conflict occurred or not. If the action could have been done successfully a small green banner will appear. Figure 6.8 shows an example of successful cherry picks. If a conflict was detected a modal will be shown. In Figure 6.9 an example for a detected merge conflict can be seen. On the top the modal shows which action were checked and which branches were used. The modal also includes a collapsible list of authors of the involved commits. Additionally, a list of the conflicting files is visible. Each file is displayed as a shrinkable card which contains the filename as card header and the content including the coloured conflicts as content. The conflict visualisation for rebase conflicts and cherry pick conflicts are similar. The difference is that not only the authors of the involved commits are shown but also their shas and an information if the commits could have been successfully cherry picked or rebased, if the commit caused the conflict or if the commit was not analysed due to the previously found conflict. An example can be found in Figure 6.10.

[7] https://www.nodegit.org/

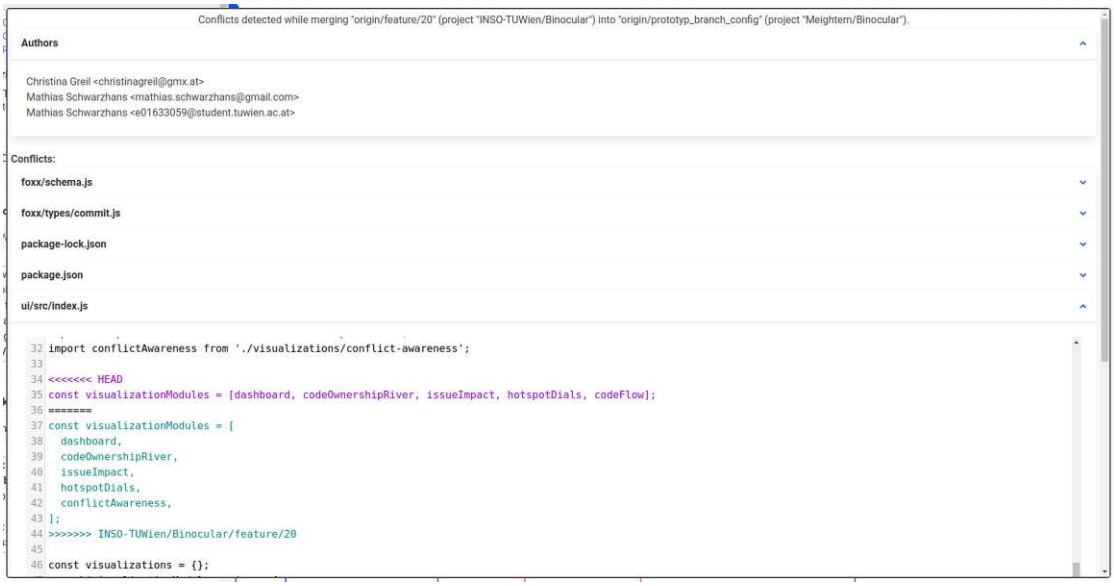


Figure 6.9: Merge conflict information

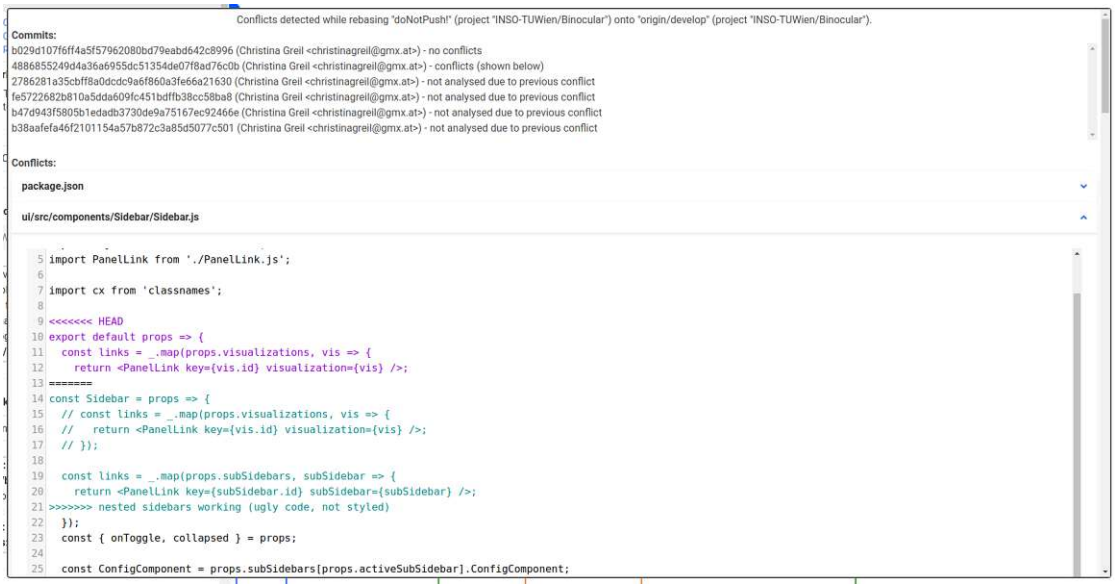


Figure 6.10: Rebase conflict information

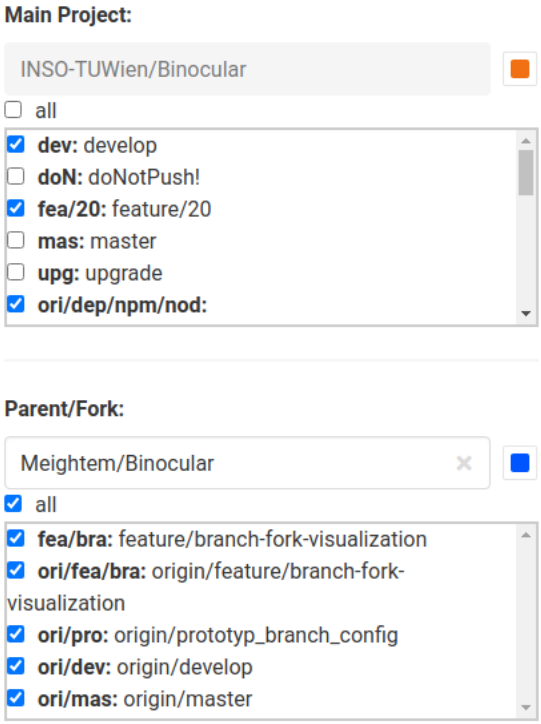


Figure 6.11: Examples of branch selections

6.5 Iteration 3: Branch Selection and Dependency Highlighting

In this iteration the two remaining highest rated features were implemented. This included the hiding and showing the commits of a selected branch (F13) and the highlighting of commit dependencies (F9).

When the base project and a selected other project is loaded their branches will be added to the corresponding checkbox list. Other than in the mock-ups the checkbox labels contain the branch reference shown in the graph and additionally the full name of the branch. At the beginning all branches are selected and therefore visible in the visualisation. At this point the user can either deselect single branches or all branches at once using the „all“ checkbox above the branches list. Deselected branches will be removed from the visualisation. These branches can be inserted again by selecting the corresponding checkbox. Multiple faded out branches can be faded in again by checking the „all“ checkbox again. Examples of different branch selections are available in Figure 6.11. The recolouring of the node and edges was implemented the same way as described in Chapter 4.

The second feature in this iteration was the highlighting of commit dependencies. When a commit is selected, the commits on which the selected one directly depends are retrieved

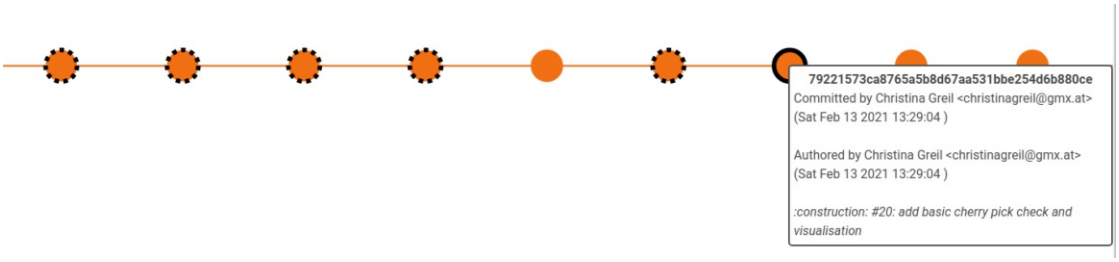


Figure 6.12: Highlighted commits the selected commit depends on

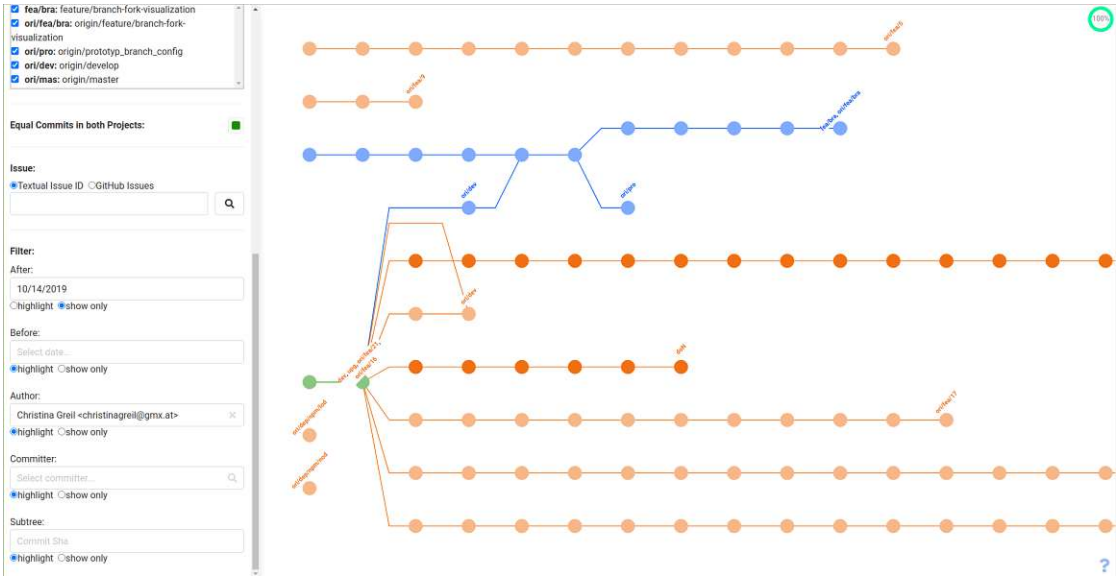


Figure 6.13: Example of filters with an expanded graph

from the backend and highlighted in the visualisation. Such highlighted commits can be seen in Figure 6.12. The selected commit has a continuous black frame, and the dependencies have a dashed black frame. For the calculation of the dependencies the python library git-deps is used. The calculation will only be done for the first level. For other levels the user must also select the dependencies. This means that the developer can decide for himself or herself how far the dependencies should be displayed. This restriction on mock-ups was implemented because the calculation of all levels in large graphs would take too long. Additionally, developers may only be interested in this information up to a certain level because earlier commits have already been integrated into the branches of interest.

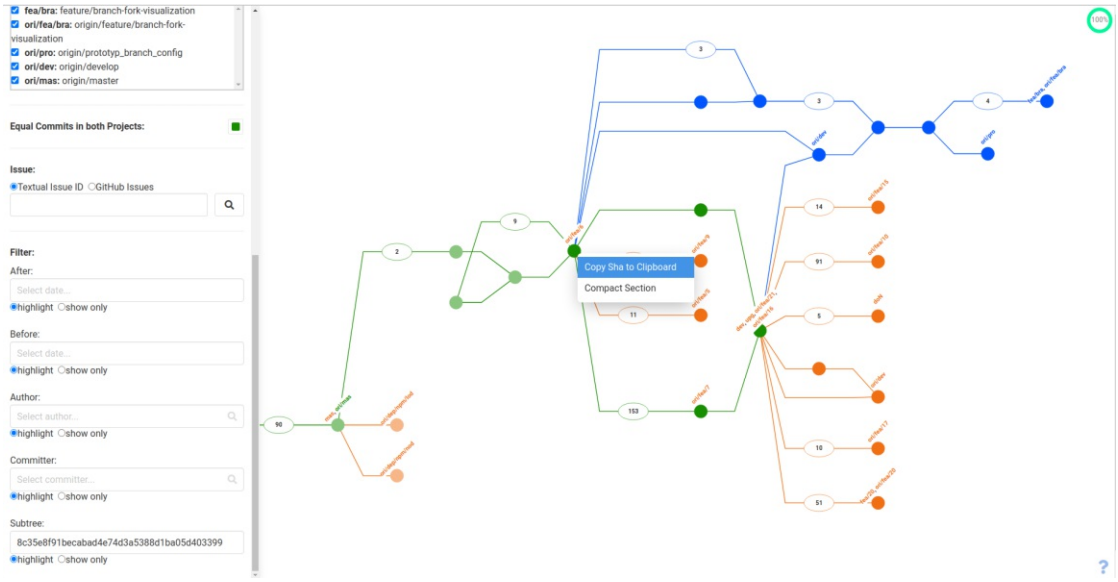


Figure 6.14: Set subtree filter

6.6 Iteration 4: Filtering

The filters were added in the fourth iteration of the prototype implementation (F14). As mentioned in Chapter 4 the filters can be combined which will make the filter more restrictive. Filters with the show only option set have a higher priority than filters that should only highlight the commits. For example, a filter that should only show commits after a specific date will stay faded out even if the commit would have been highlighted because of another set filter. Figure 6.13 shows such a filter combination.

The prototype has five filters. The „after“ filter shows or highlights all commits that are committed after a specific date. The „before“ filter works in a similar way but shows or highlights commits committed before a specific date. The date can either be typed in or can be selected via a datepicker. The react-datepicker[8] library was used for the datepicker. By combining both filters, commits can be filtered that were created within a certain period. The subtree filter shows or highlights a specific commit including all its children (see Figure 6.13). The filter is activated by inserting the sha of the selected commit into the textbox. For this an additional feature was implemented. This feature allows the developer to copy the sha of a commit via the context menu opened with a right click on the node (Figure 6.14). Then the user can easily paste the sha into the text field of the filter and does not have to type it in manually.

The author and committer filter will display commits of a specific author and committer. The highlighting option will work the same way as for the other filters described above. But the show only function of these two filters works differently. With the same functionality,

[8] https://reactdatepicker.com/

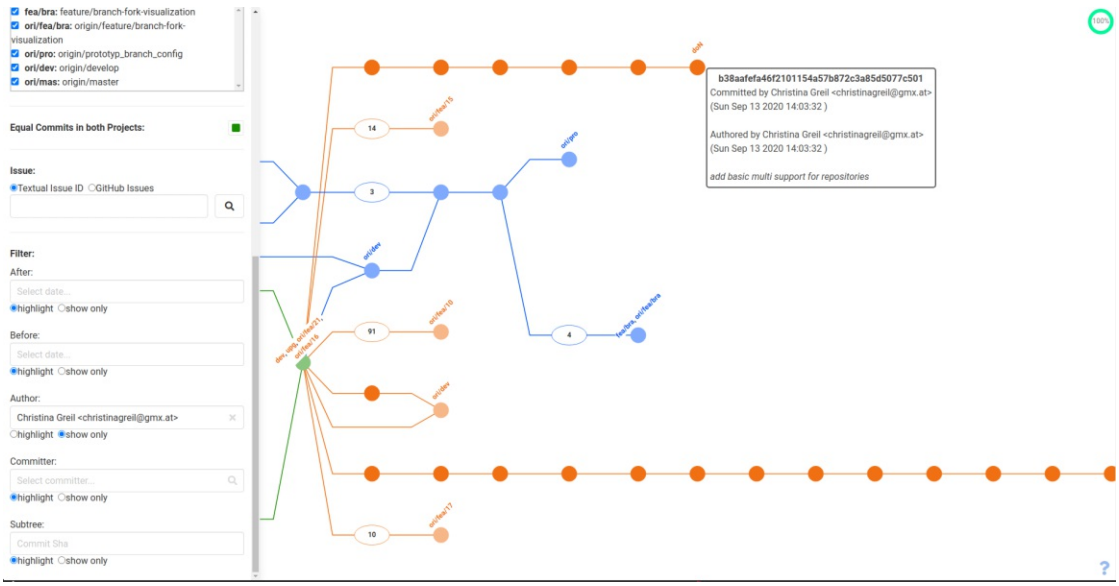


Figure 6.15: Example of an author filter with the show only option set

it would make the history of the commits disappear. Therefore, an improvement was added in the last implementation iteration. Now the show only option of these filters is combined with the compact view functionality (see Section 6.7). This means that the commits of certain authors and committers are treated like branching nodes and are not clustered together with other nodes. All branching nodes that do not come from the selected author or committer are treated the same way as normal nodes. Their colour saturation is reduced. An example of this behaviour can be seen in Figure 6.15. Due to this special treatment, there are further restrictions when expanding the entire graph or parts of it. If the author or committer filter is set with the show only option, the expanding feature of the entire graph and of single sections is disabled and the user will see an error message when trying.

After adding the compact view feature to the prototype, a problem occurred with the implemented filters. The filtering did not work for collapsed nodes and for a single commit node between two branching nodes. The filtering logic had to be adapted in the improvement iteration later on. Because collapsed nodes should provide minimal information of the commits they hold, the number of commits that passes the set filter options are shown within the node labels. Figure 6.16 shows a filter to get all commits committed after the year 2019. This example shows that the compacted section of branch „origin/feature/15“ has 13 of its 14 nodes were committed after the set date. When collapsed nodes only have one number as a label, this means that all its commits meet the requirements of the filter. If the filters will be set with the show only option, then the collapsed sections are visible in the graph as long as at least one of its commits to meet the filter criteria.

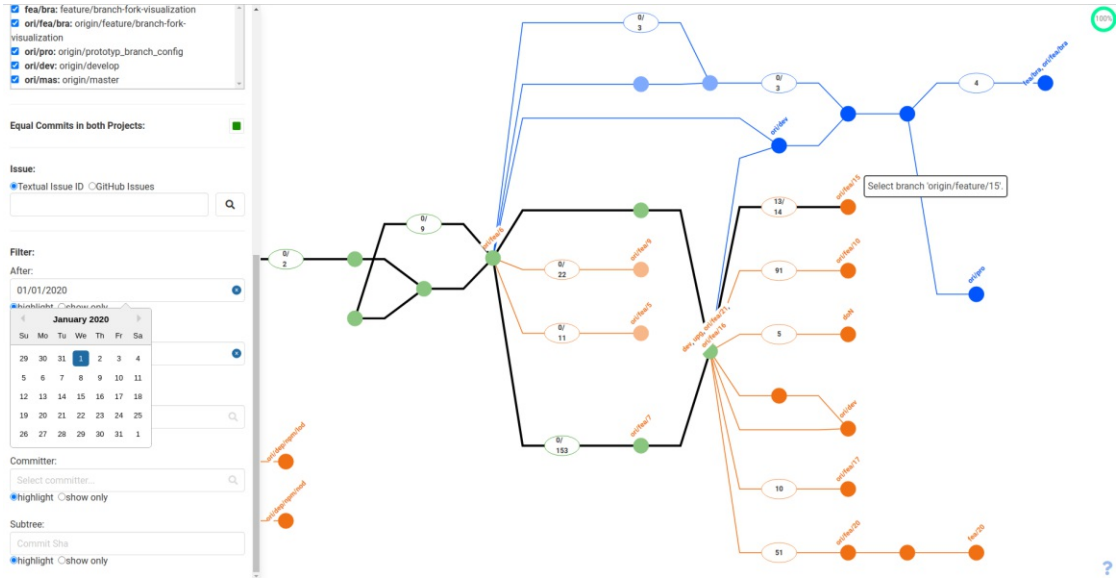


Figure 6.16: Example of filters with a compacted graph

6.7 Iteration 5: Compacting the View

In this iteration the last feature of the prioritised feature list, the compact view (F12), was implemented. The integration of this feature at the end of the implementation phase turned out to be a bit problematic because the basic data structure that was used for the nodes to be displayed in a graph had to be changed. This resulted in destroying working functionalities like the filtering as described at the end of the previous section.

As already seen in the mock-ups the compacted view of the graph clusters multiple commits between branching nodes. In difference to the definition of branching nodes in Chapter 4, not only commit nodes with multiple parents or children fall within this definition, but also the heads of branches. This decision was made because the heads of the branches also provide useful information, such as the last person who committed on this branch and when the last commit was made. Additionally, the design of the clustered node was changed a bit. To improve the readability, these nodes are not filled with the chosen project colour. There is also a difference in expanding or compacting the entire graph. These functionalities are made available by means of two buttons („Compact all“ and „Expand all“) in the upper half of the configuration section. This variant is more intuitive than the checkbox defined in the design phase. By default, the graph is displayed completely compacted when loaded.

Not only the entire Graph can be expanded or collapsed but also single sections. To compact the commits between branching nodes the user has to right click on any commit between these branching nodes and choose the „Compact Section“ context menu item. When the user chooses this option on a branching node itself, all its child path sections will be collapsed. In order to expand a compacted section the developer again needs to

right click on the specific clustered node and choose the context menu item „Expand“. This will show all clustered commits of this section within the graph again as single nodes.

CHAPTER 7

Scenario-based Expert Evaluation

This chapter addresses the evaluation of the conflict awareness idea, the chosen visualisation and the implemented prototype. This includes the planning phase of the evaluation, the results and the threats to validity.

7.1 Plan

For answering the defined sub questions of RQ3 scenario-based expert evaluations were conducted. Due to the COVID-19 pandemic, the scenario-based evaluations were carried out with the experts via Zoom accompanied by two interviewers. The tool was started on an interviewer's computer. The participant was able to operate the prototype by means of screen transmission and handover of the remote control. This also allowed the attendee to switch between the visualisation, the questionnaire (appendix 9), the quick start guide (appendix 9) and the screenshots for the first scenario.

The participants had to solve six scenarios with the implemented visualisation.

1. Scenario: Which forked project diverged at most from the base project?
2. Scenario: On which branch (project INSO-TUWien/Binocular) was feature #20 implemented?
3. Scenario: How many commits committed Maximilian Zenz in 2020 to the repository INSO-TUWien/Binocular?
4. Scenario: Which commits must be cherry picked as well such that no conflict occurs when cherry picking commit d249bb8ff7b5904a179006726181160f4bd2ef62 (project INSO-TUWien/Binocular) into the branch origin/develop (project Meightem/Binocular)?

5. Scenario: Who may be able to help solving the conflict when merging branch origin/feature/branch-fork-visualization (project Meightem/Binocular) into branch origin/master (project INSO-TUWien/Binocular)?

6. Scenario: A conflict occurs when rebasing branch origin/feature/5 (project INSO-TUWien/Binocular) onto origin/feature/9 (project INSO-TUWien/Binocular). Which commit introduced the conflict?

The evaluation sessions were accompanied by a questionnaire in order to keep a common thread during the conversation. The questionnaire contained questions about the person, their experiences in the development area and with VCSs, the descriptions of the scenarios to be solved including follow-up questions and finally questions about the purposefulness of the conflict awareness, the visualisation and the prototype.

During the pilot evaluation three main problems occurred. One problem was performance issues of the Zoom session. It was hard for the user to navigate within the prototype due to screen transfer time delays. Such navigations included scrolling and zooming. The other problems were performance and usability issues of the tool itself. Because of the prototypical style of the implemented visualisation, the loading times can be long. These include for example for the conflict checks and the selection of a fork or parent. Therefore, screenshots of the divergences between the base projects and its forks were added to the first scenario. The attendee could then decide if he or she uses the provided screenshots or the prototype itself for solving this scenario.

At the beginning of the evaluation session, the project was briefly presented, and the meaning and purpose of this evaluation were explained. Afterwards, the participant was able to have a look at the quick start guide (appendix 9) and became a short demonstration of the prototype. The guide contained a brief description of the visualisation's functionality explained mostly with graphics and additionally a more detailed textual operation manual.

Because the usability of the tool was out of scope in the sessions, the interviewees were informed about the usability issues and got help when they were stuck during the execution of the scenarios. Various pieces of information could have been collected with this approach. On the one hand, the participant was able to complete the scenario and provide information about the meaningfulness of the scenario without being blocked by the prototype. The lack of knowledge about how to use the tool should have influenced the evaluation of the basic idea as little as possible. On the other hand, possibilities for improvement of the prototype could have been shown by asking the participant about their expectations.

7.2 Results

Like in the previous interviews, the participants were asked to provide information about themselves and their experiences in the field. The participants' demographics can be found in Figure 7.1. Most developers had about 10 years of experiences with software

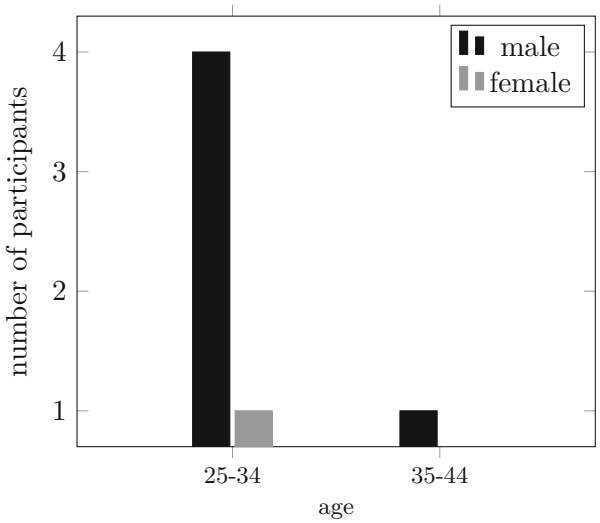
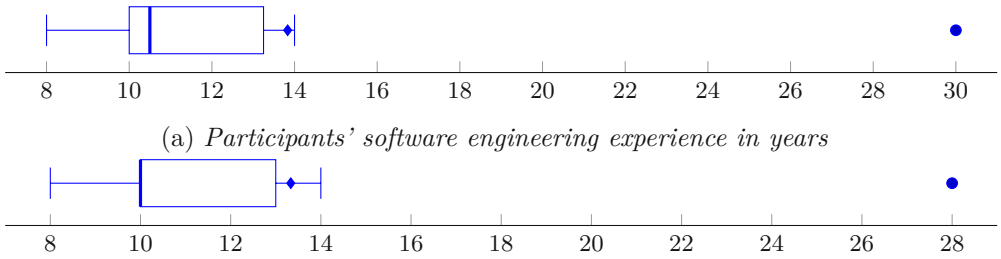


Figure 7.1: Demographics of the participants



(a) *Participants' software engineering experience in years*

(b) *Participants' experience with Version Control System*

Figure 7.2: Experiences of the participants

engineering and with VCSs (Figure 7.2) and the VCSs Git and Subversions are most common among them (Figure 7.3). Furthermore, none of the interviewees had a strong preference for either merging or rebasing only (Figure 7.5).

7.2.1 Scenario 1: Finding out the most diverged Fork (F1)

In the first scenario the attendees had to find the fork which diverged at most from the main project. Divergences between projects can provide information about possible conflicts. The further projects diverge, the higher is the chance of conflicts to occur when integrating back the changes into the other project. The colour-coded affiliation of the commits in either one or both projects is intended to enable the user to identify potential conflicting areas more easily.

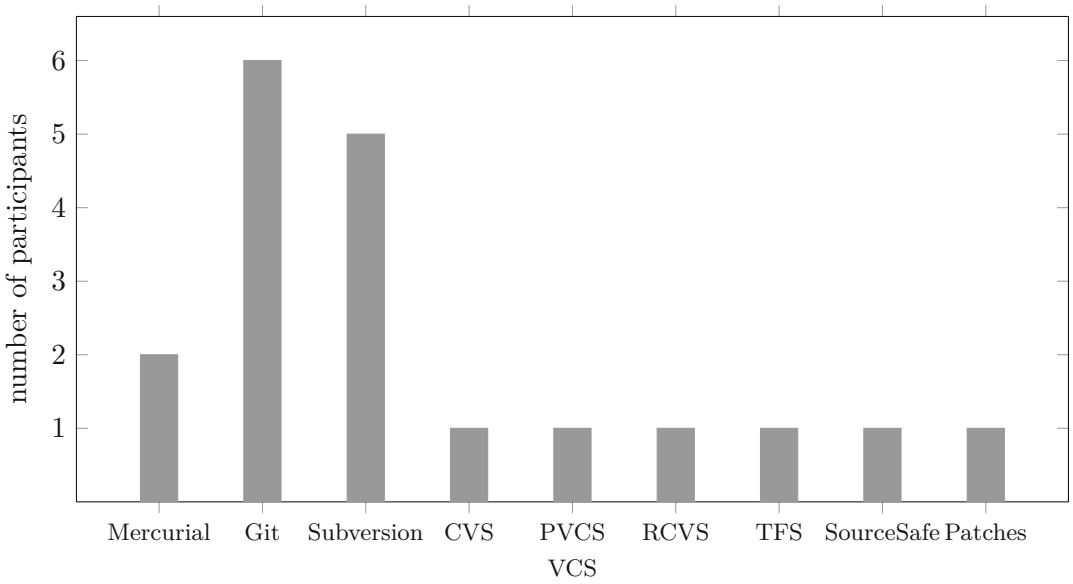


Figure 7.3: Version Control Systems used by the participants

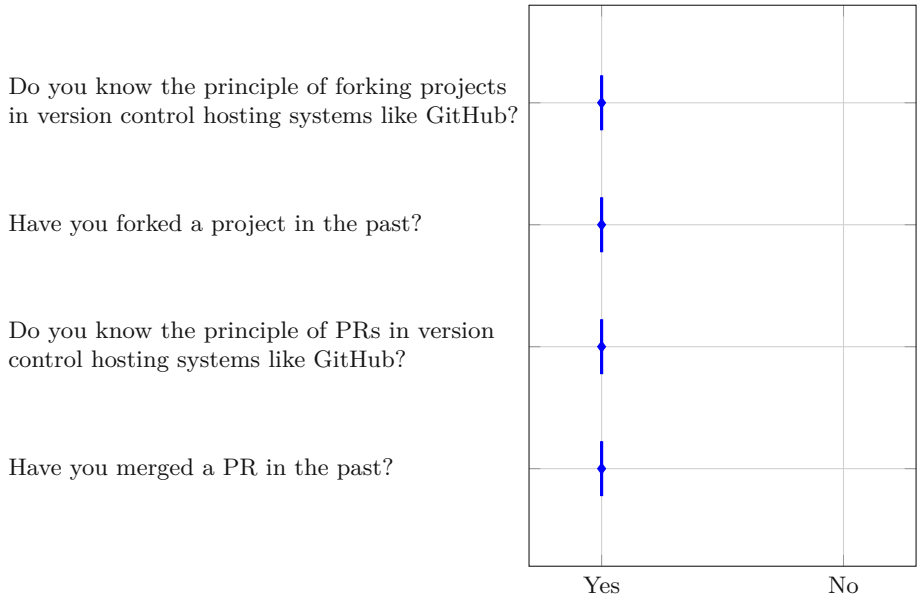


Figure 7.4: Forking and Pull Request experiences of the participants

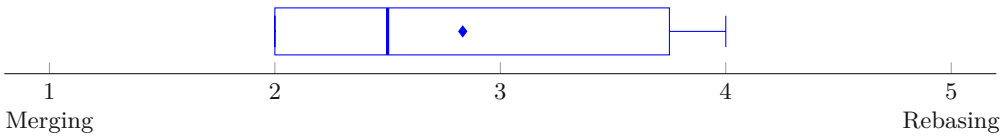


Figure 7.5: Participants' preferences between merging and rebasing

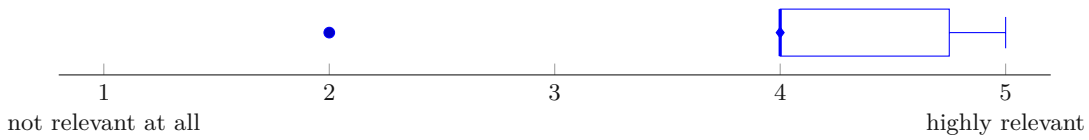


Figure 7.6: Participants' rating of the purposefulness of scenario 1

Purposefulness of the Scenario

Most of the attendees rated the purposefulness of this scenario with a four or five and one participant with a score of two (Figure 7.6). Participant 1 stated that he liked the fact that the colours and the project structure is still visible when zooming out. This allows to view big parts of large projects. He also noted that GitHub and GitLab also provide a similar, but more rudimentary approach than the prototype. For participant 2 the feature can also come in handy for open source projects. The view provides information on who contributes to the project with what. Manually searching the forks is often considered difficult and catastrophic by him. Participant 3 and 4 liked the intuitive representation of the project structures. For participant 5 this scenario was not quite relevant. The divergences between the projects do not play a major role for the conflict awareness for him. On the contrary, participant 6 found this feature especially purposeful if the forks diverge further apart.

Traditional Solutions to solve the Scenario

Figure 7.7 shows with which other tools the participants usually had solved the scenario. Participant 1 normally uses the tool gitk[1] to see differences between projects. The tool allows him to see the graph of the repository, but not as clear as the prototype. For him, the coloured affiliation marks and the possibility to construct a compacted view (F12) help to understand the project structures in an easier and faster way. Participant 2 usually uses a more social approach to find out the divergences between projects. He makes sure that the developers actively create a PR and analyses the changes manually. But this approach is not always applicable. The solution is usually more promising in large projects that are well established than in smaller projects. For small repositories it is more often the case that software engineers fork the project, try out some code changes, but throw away the work or leave it be without finishing it. The participant noted that in these cases the maintainer of the main project can see the time of the last commit, but he cannot really see if some useful changes were made. Participant 3 would have used the command line, a graphical UI or the Integrated Development Environment (IDE). Like this selection participant 5 also chooses the command line or a graphical UI for solving such scenarios. Participant 4 would have used the GitLab interface and participant 6 the preferred the IDE.

[1] https://git-scm.com/docs/gitk/

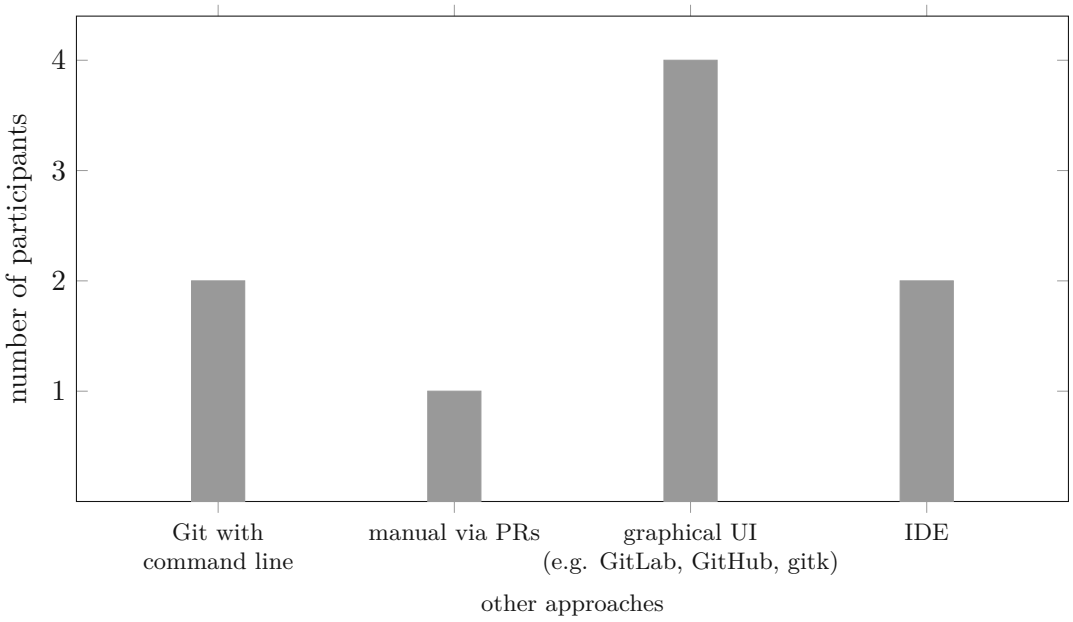


Figure 7.7: Participants' solutions to solve scenario 1

Preferred Way to solve the Scenario

Most attendees would prefer the tool over their traditional toolset. Participant 5 does not have a preference because for him the divergence between forks with their parent does not really provide the necessary information for conflict awareness. Participant 2 stated that he would prefer the prototype if the tool can be automated such that, for example, the tool actively makes a scan of the different divergences and provides this information. Another possibility would be the integration in GitHub. Participant 3, participant 4 and participant 6 stated that the graphical representation with the colours is helpful because you get a quicker overview of the project structure than with the other tools.

Found Limitations

One found limitation during the sessions was the performance of the tool. The graph took some time to be loaded and be visible for the user. Another finding was that it would be more useful if the prototype can show the different divergences all at once for example one after another. Currently, only one graph is visible, and the user has to remember it or take a screenshot if the state should be compared with another graph. Participant 3 and 6 stated that it would also be interesting if the tool can compare local repositories and not only forks or the parent. For participant 2 a local filter is missing in the prototype. He would like to have a filter of elements in the project structure, like a specific package. With this the software archaeology would be easier and a developer could see, for example, in which commits the specific package was changed, which changes were made, and which part of the package was already in the project since the initial commit and which parts

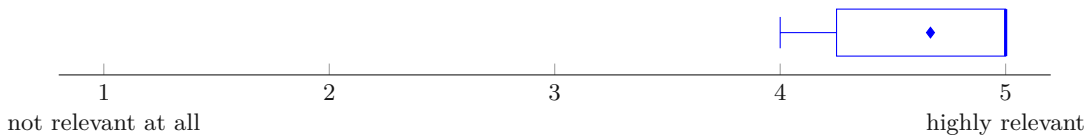


Figure 7.8: Participants' rating of the purposefulness of scenario 2

were introduced later by a feature. Participant 5 noted that for him it would be better if the layout of the main project would stay the same when changing the selected fork or parent. This would improve the comparability of the divergences.

7.2.2 Scenario 2: Finding where a Feature was implemented (F10)

A graphical localisation of the commits based on their messages makes it possible to determine whether, for example, features or bug fixes have already been integrated into certain branches or projects. Large changes with many code adjustments and longer implementation times can have a higher potential for conflicts than small code changes. This information can help developers to better plan the upcoming integrations. Another advantage of the graphical representation is that the search takes place across projects and branches. This means that the developer does not have to check out multiple branches first or clone forks when performing a manual search via the console.

Purposefulness of the Scenario

All of the participants rated the purposefulness of this scenario with 4 or 5 (Figure 7.8). Participant 3 stated that branch names sometimes are not named after the ticket number of the bug or feature which will be fixed or implemented in this branch. If that is the case, she finds it hard to find where the tickets are being processed. For participant 5 this scenario is highly relevant because it is good to know where changes were made, especially if the commits were created a longer time ago. Another use case where the graphical localisation of commits can be important is in projects that work release dependent. Sometimes the developer does not know or loses the overview when features are reintegrated into a specific branch, or it is not known if the feature is already integrated into a release branch or the main branch. Additionally, in this scenario the user can check who implemented such a feature if questions about it occur.

Traditional Solutions to solve the Scenario

The participants have similar approaches to solve such a task (7.9). Nearly all attendees stated to search commits of a specific feature or bug fix with the command line, especially with the git log command. Another widely used tool is the graphical UIs provided by GitHub, GitLab or gitk. For example, participant 1 usually only uses the issue linking provided by GitLab to find the commits. Participants 3 and 4 also use the functionalities provided by the IDE for such a task.

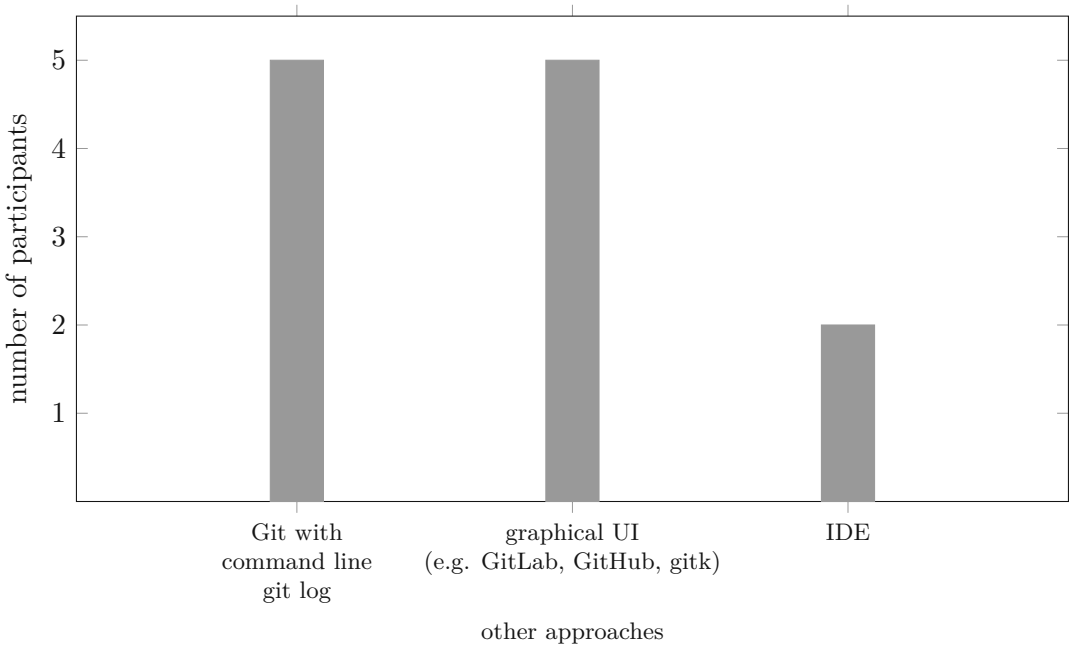


Figure 7.9: Participants' solutions to solve scenario 2

Preferred Way to solve the Scenario

Although participant 1 rated the purposefulness of this scenario as high, he usually do not often need to know what and where the commits of a specific issue are. If this functionality is needed, the GitLab issue linking is sufficient for him. The preferred solution for participant 2 depends on the use case. If the tool is already in use, he would use the search in the prototype right away. Otherwise, the possibilities of the other tools are his preference. Participant 3 would use the prototype additionally as help. If the project has strict rules with the naming of the branches, such that the branches must be named after the issue ticket number, then the preference will lie with the other tools. Otherwise, the graphical search is preferred because of the easier and more visual localisation of the commits she is looking for. Participant 4 prefers the prototype over the usually used tools, because the visual approach is clearer for him and provides a better understanding of the commit locations. This attendee also stated that he liked the compact view (F12) because the overall structure of the projects becomes clearer that way. For participant 5 the integration of the GitHub issue list was a plus point for the conflict awareness visualisation in addition to the graphical clarity. Like participant 2, the preference of participant 6 is use case dependent.

Found Limitations

One common issue was the highlighting of the found commits. All participants needed help in finding the selected commits. A better highlighting and an autofocus functionality

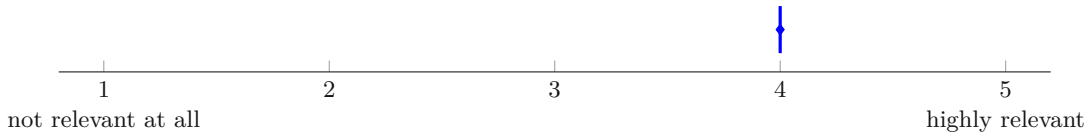


Figure 7.10: Participants' rating of the purposefulness of scenario 3

would have helped solving this issue. Especially the highlighting of the clustered commits was not noticeable for the developers. Participant 2 also suggested the possibility to jump between commits of interest for example with a previous and next button. When the filter was selected, the tool should first automatically focus on the first or last selected commit in the history and then the user would be able to navigate through the commit selection with the two buttons. For participant 3 it would have been helpful if the clustered commits were automatically expanded if such a node contains a commit of interest.

7.2.3 Scenario 3: Filtering the Project Structure (F14)

Like the scenarios 1 and 2 the visual filtering provides passive conflict awareness information. Developers can easily determine which changes their colleagues or developers of a fork or parent project are working on and where in the project structure these code adjustments are located. This gives an overview of which project sections are currently being worked on and by whom, and which developers are working on similar codes. With this information potential conflicts can be identified early, and developers can be made aware of them.

Purposefulness of the Scenario

All participants rated this scenario with a score of 4 out of 5 as purposeful (Figure 7.10). Participant 2 stated that this information is particularly interesting in large projects to get an overview what a person has done in the project. For participant 3 this scenario can be useful when a project member leaves the project. If that iss the case the graphical filter can help in getting information about where in the project structure this developer made code changes and in identifying if knowledge exists that only this person knows about. Participant 6 noted that he faces such scenario daily.

Traditional Solutions to solve the Scenario

For all participants Git commands play a major role in solving such scenarios (7.11). Participant 1 additionally uses Gitinspector[2] to filter the commits. The GitLab UI is used by participant 3, 4 and 5 and the functionalities of the IDE are being used by participant 3 and 6.

[2]https://github.com/ejwa/gitinspector

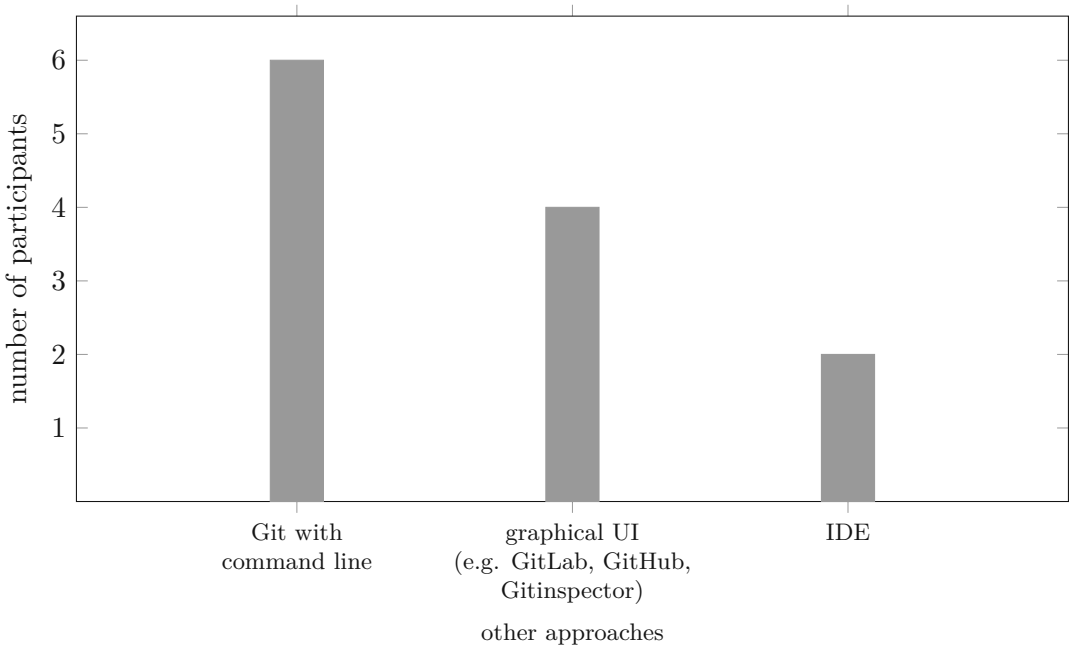


Figure 7.11: Participants' solutions to solve scenario 3

Preferred Way to solve the Scenario

Participants 1, 2, 3 and 6 would continue to use their usual methods to solve such scenarios. For the search after the number of commits, the usual tools already provide enough functionality, participant 1 stated. But if the scenario would cover the question on where the filtered commits are in the history, then the prototype would be more helpful. Participant 2 stated that he can get a sufficient overview with the git log command and a word count and see how much code lines a person added or deleted. For participant 6 a tabular view of the commits provides enough information. Participant 5 would prefer the prototype and participant 4 noted that for him the preference would be use case dependent. For simple searches he would use the usual tools, but for searches with multiple filter criteria the prototype would be preferred.

Found Limitations

As in scenario 2 the insufficient highlighting of the filtered commits and the missing autofocus function were problematic. Like participant 2 in the second scenario, participant 1 suggested that the user should be able to traverse between the filtered commits. Another noticed problem was that the attendees did not make a difference between a committer and an author. Participant 2 noted that the distinction between committer and author should not be made. Most developers do not differentiate between those two, although from the technical point of view this distinction is important. During the interviews similar findings were recorded. Almost every participant asked what the difference

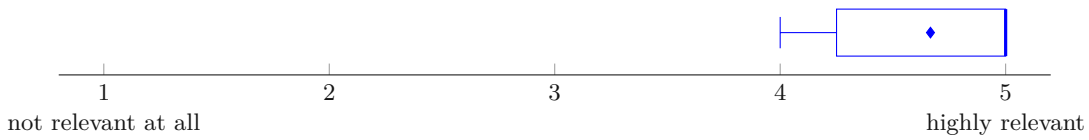


Figure 7.12: Participants' rating of the purposefulness of scenario 4

between a committer and an author was. For this reason, participant 2 would find it better if the filter would be combined and finds all commits that have the selected user as committer or author.

7.2.4 Scenario 4: Finding Commit Dependencies for a Cherry Pick (F6, F7, F8, F9)

If commits are cherry picked conflicts can arise not only from changes made to the selected commit, but also if previous changes, on which the commit to be cherry picked logically depends, are not included in it. Scenario 4 of the evaluation deals with such a case. The prototype shows direct and logical predecessor commits on which the commit that should be cherry-picked is based on. This should help to make developers aware that previous commits must be integrated first.

Purposefulness of the Scenario

Most of the participants rated the purposefulness of this scenario as highly relevant and therefore with 5 of 5 points. Participant 2 and 3 rated this scenario as relevant (Figure 7.12). Participant 1 often must tackle such a scenario at releases. For participant 2 the visualisation of depending commits can provide hints of missing commits in a cherry pick but he would not trust the algorithm alone.

Traditional Solutions to solve the Scenario

There are two ways the participants usually take to find out necessary commits that need to be cherry picked previously to the desired commit (Figure 7.13). First is the manual analysis of the commit and its changes to find out possible dependencies to prior commits. Second is the trial-and-error strategy. The participants try to cherry pick the selected commit and if it results in a conflict the previous commits in the history are analysed or also tried to be cherry picked.

Preferred Way to solve the Scenario

Most of the participants would use the prototype in the future to check the dependencies of commits. But participant 1 and 6 also stated that writing actions should also be allowed and not only readable actions. With this, the preference would also be the conflict awareness tool. For participant 2 the tool must provide a diff of the commits that he would use the prototype for such a scenario. Reason for this is that he checks each

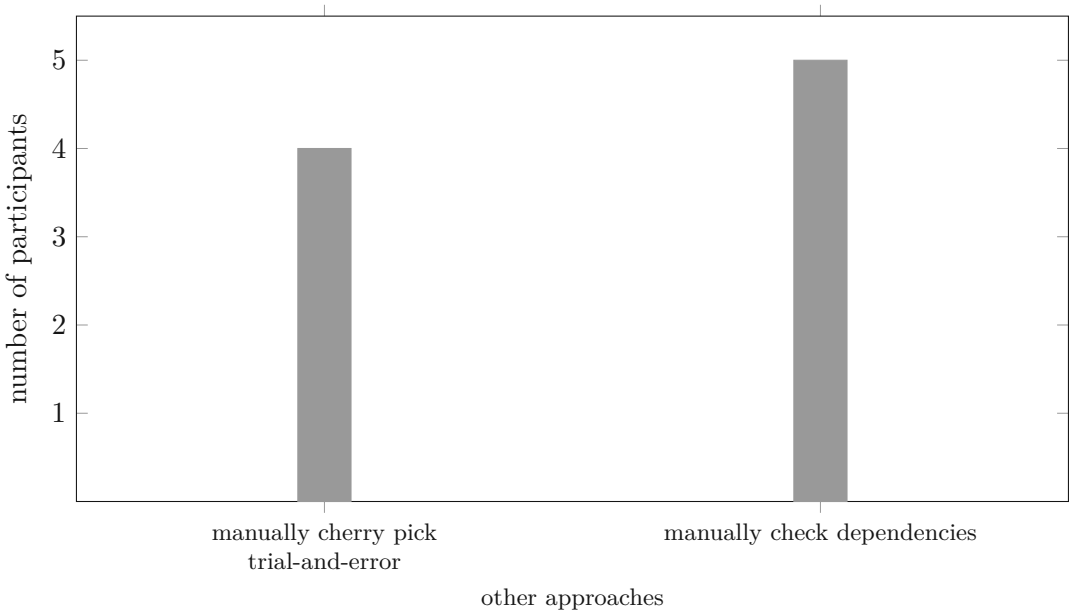


Figure 7.13: Participants' solutions to solve scenario 4

changed code line before integrating the changes. Participant 4 stated that he prefers the visualisation because this way is much faster than the manual analysis.

Found Limitations

Like in the previous two scenarios the highlighting of the necessary commits is too poor. If the dependencies are far apart some needed commits for the cherry pick may be overlooked. Therefore, participant 3 suggested to additionally show the number of the dependencies. Multiple participants stated that they would like to not only check if the cherry pick is possible but also to perform it within the tool if no conflict occurs. If a conflict occurs it would be helpful when it can be resolved directly in the prototype. Switching to another tool to carry out the activity being tested reduces the acceptance of the prototype. The performance of the dependency check was also a noted issue. It would be better if the dependencies of commits will be indexed and saved into the database for a faster information retrieval when needed. Additionally, the usability should be improved. The mark and click procedure to check a cherry pick is not intuitive. Another problem is that the user does not get any information that the cherry pick is currently checked for conflicts.

7.2.5 Scenario 5: Finding Developers who may help in solving a Merge Conflict (F3, F4, F8)

Scenario 5 includes an active conflict awareness which shows additional metadata about a conflicting merge. In this scenario the developer is actively advised that the cross-project

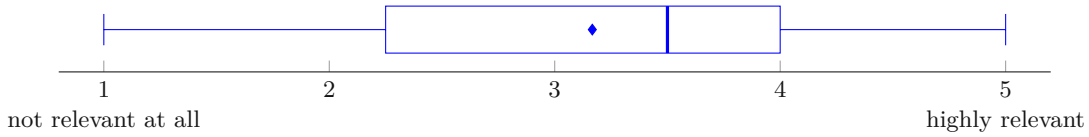


Figure 7.14: Participants' rating of the purposefulness of scenario 5

merge of two branches would cause a merge conflict. In addition to the files that contain a conflict, the authors who have worked on the two branches from the point in time of the branching to the point of merging are also displayed. This provides the developer with a more limited list of developers who can help in resolving the conflict.

Purposefulness of the Scenario

The ratings about this scenario are distributed over the scale (Figure 7.14). Participant 5 rated this scenario as highly relevant. For participant 1 and 5 the scenario has a score of 4 and is seen as relevant. Participant 3 stated that the developer who merges the changes usually knows who made these. Its rather seldom for him that someone merges changes and does not know who was involved. Participant 3 rated the scenario as not relevant and participant 6 rated it as not relevant at all.

Traditional Solutions to solve the Scenario

Most of the time the participants check the involved commits with the command line or with the IDE to find out who to ask for help when getting stuck with a conflict to resolve. As seen in Figure 7.15, 5 out of 6 the participants would use the command line for such an analysis and three of the attendees would look, also additionally, at the information provided by the IDE. Participant 1 usually first checks with the IDE which files are conflicting. Afterwards, he uses the git blame command to get further information. Participants 3 and 6 also make use of both variants. As for participant 2, 4 and 5 the command line is way to retrieve the needed information.

Preferred Way to solve the Scenario

For participant 1, 3 and 6 the usual way of solving such a scenario is preferred. Participant 2 does not have a strong preference. If he would use the prototype at that moment, he would have used its functionality for the conflict information, but otherwise the manual search of the authors is preferred. Participant 4 noted that he would prefer the prototype if it would support writeable Git actions. In this case the developer does not have to commit or stash the current local changes for the merge. For participant 5 the prototype provides the information in a more structured way.

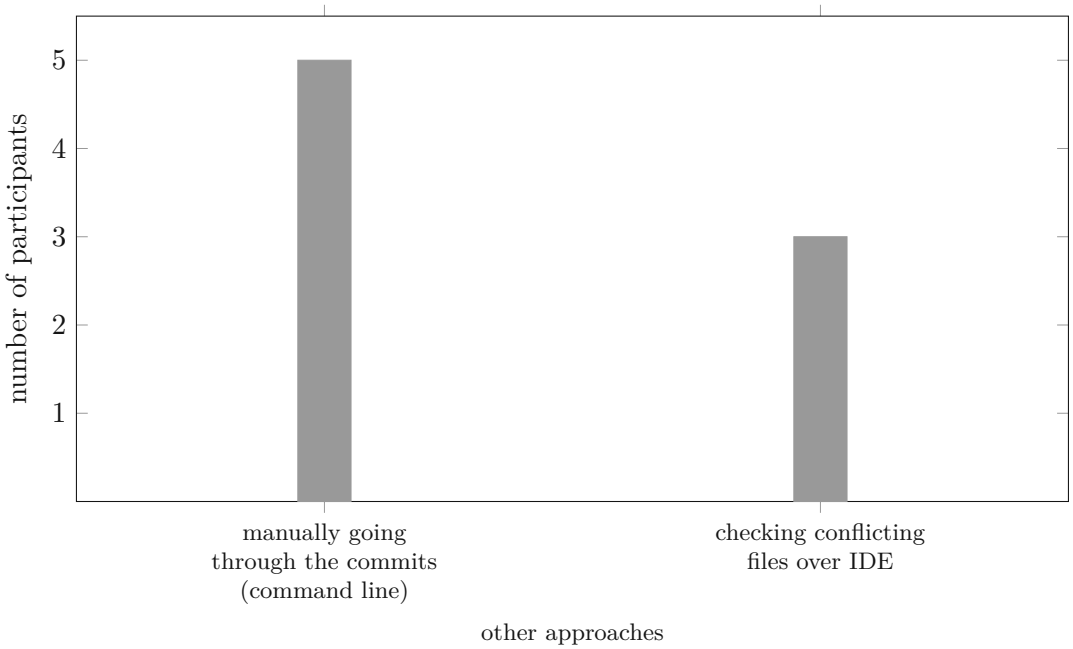


Figure 7.15: Participants' solutions to solve scenario 5

Found Limitations

As in scenario 4 some attendees commented that it would be helpful if not only readable Git actions can be performed, but also writing actions. Also, the interaction to check if a merge will result in a conflict is not intuitive enough for the participants and a loading indicator is missing. Participant 4 and 5 noted that the usability may be improved if the check can be started using drag and drop instead of a mouse and key combination. Participant 3 whished for a more visual representation of the merge check. This way, the visual representation does not have an advantage over the other possibilities to solve such a scenario. Participant 4 also stated that a passive representation of the mergeability of branches would be particularly helpful with an early conflict detection. In this case the developers do not need to take an action themselves to check for conflicts. Additionally, the integration of the tool into existing IDEs was requested. Another problem that was found is the granularity of the author list the merge conflict information holds. The list should only include authors of the conflict directly and not the authors of all commits of the merge. With the current granularity, the list also includes authors that did not contribute directly to the conflicting code sections.

7.2.6 Scenario 6: Finding the Commit that causes a Conflict in a Rebase (F2, F3, F8)

As with scenario 5, scenario 6 also provides active conflict awareness information in addition to further metadata. This scenario was about a rebase, which resulted in a

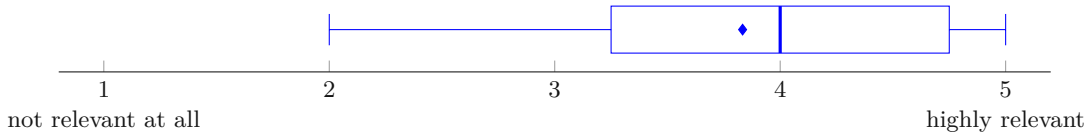


Figure 7.16: Participants' rating of the purposefulness of scenario 6

conflict. The developers had to find out which of the commits caused the conflict.

Purposefulness of the Scenario

The distribution of the purposefulness of this scenario can be seen in Figure 7.16. Participant 2 rated it as not relevant, because he does not see the benefit of this information. When the developer wants to rebase a branch then he will just do it and the rebase will stop at the conflict which has to be resolved before continuing. If afterwards another conflict occurs, he still has to resolve it and so on. The only reason why he may see this scenario as relevant in the field is that someone can quickly judge what is rebaseable and what not. Participant 3 rated the purposefulness with a 3 and participant 1 and 5 with a 4. For participants 4 and 6 such scenarios are highly relevant.

Traditional Solutions to solve the Scenario

Similar to the other solving ways of the last scenario, the developers would perform the rebase manually over the command line, the IDE or some graphical UIs (Figure 7.17). The most preferred tool is the IDE, followed by the command line.

Preferred Way to solve the Scenario

Participants 4 and 5 prefer the prototype because the tool makes the information gathering more easily with less clicks needed and because the author of the conflicting commit is visible right away in case help is needed. The other participants prefer the usual way for checking a rebase of conflicts and retrieving the conflict information. For participant 3 the current implementation is too text based, like in scenario 5. She would like to have a stronger visual representation of the rebase and its conflict information. Participant 6 stated that the IDE provides a similar representation and with the prototype the user still must trigger the conflict checks.

Found Limitations

The found limitations are like the limitations of the fourth and fifth scenario. Again, the usability to carry out the conflict check needs improvement and a more passive approach would be helpful. Also, the loading indicator for the rebase check and writable actions are missing. Participant 3 would like to have a more visual representation of the check and the processing of the information for this scenario as well. Participant 4 would have expected more metadata of the commits. For example, the commit message may be

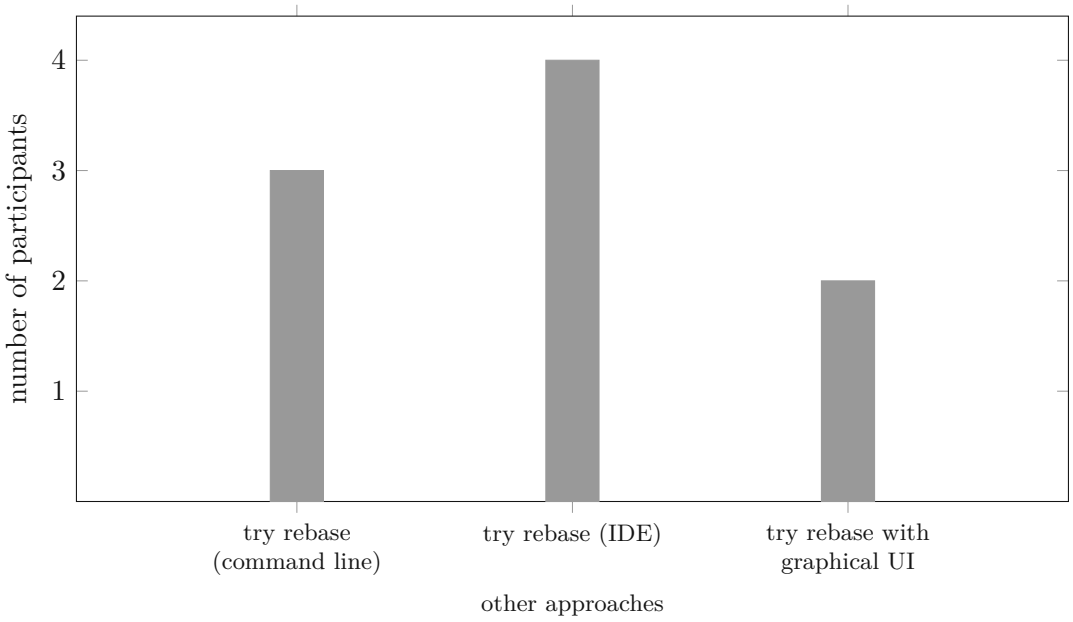


Figure 7.17: Participants' solutions to solve scenario 6

helpful to get the context of the commit such that other developers can be asked if the author is not available at the moment. For Participant 6 an indication is missing if other conflicts would occur when the current conflict is resolved.

7.2.7 Purposefulness

After finishing, the scenarios the participants were asked about their perceived purposefulness of the idea, of the chosen visualisation and of the tool itself. (7.18). For the rating of the purposefulness of the idea the participants were asked to not include the usability issues into the evaluation. Solely the potential of the conflict awareness idea should be rated. Based on the provided rating and the given feedback during the execution of the scenarios, the participants see a lot of potential in this idea. Participant 1 and 2 rated the idea as purposeful and the other participants as highly purposeful.

For the purposefulness of the chosen visualisation participant 1 and 2 provided a full score. Participant 4, 5 and 6 assessed the chosen visualisation with a rating of 4 out of 5 and participant 3 with a 3. Participant 1, for example, stated that the fundamental graph of the visualisation is known from Git and that he liked the compact view (F12). The main pain points of the visualisation are the current highlighting and the usability issues.

For the rating of the prototype the participants were asked to assess the functional scope and not the current implementation with the already noted usability and performance issues. The functional scope of the tool was evaluated as purposeful. Participant 5 and 6

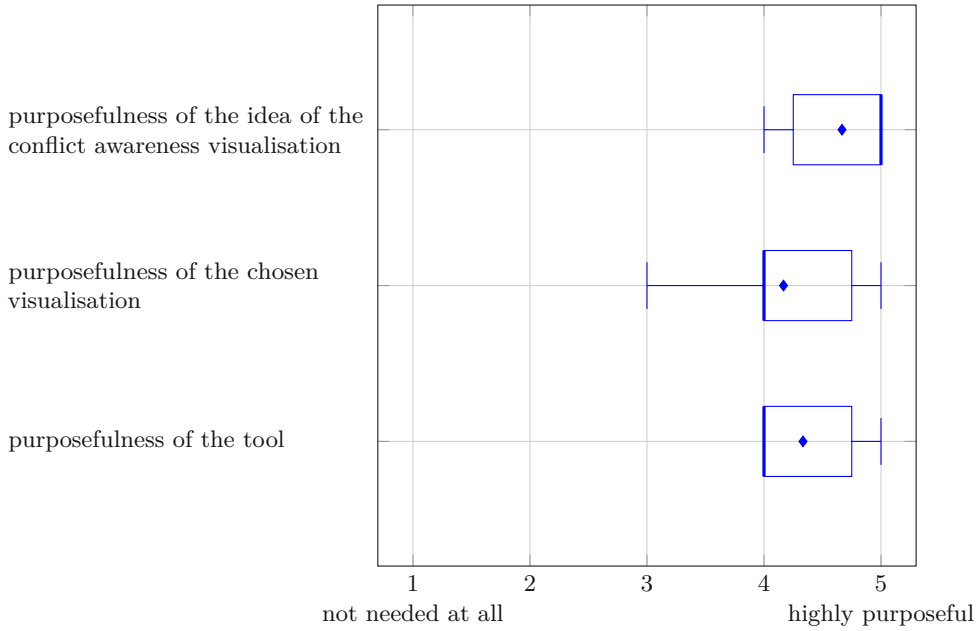


Figure 7.18: Participants' rating of the purposefulness of the idea, of the visualisation and of the tool

provided a rating of 5 and the other participants a score of 4. The participants missed the more passive view of the active conflict checks, an integration in existing IDEs and an integration possibility into CIs pipelines.

7.3 Threats to Validity

This section describes the threats to validity of the scenario-based expert evaluations.

7.3.1 Number of Interview Participants

Six developers participated in the evaluation sessions. This number cannot provide a full evaluation of the conflict awareness visualisation concept. The goal of the evaluation was to provide an overview of the relevance of the idea of a conflict awareness visualization not only at branch level, but also at fork level. Additionally, the prototype can act as a base for improvements of the found limitations.

7.3.2 Performance and Usability Issues

The performance and usability issues may have a negative impact on the assessments of the general idea and the visualisation itself. Because the usability and the performance were not in scope of the evaluation sessions, the participants were made aware of the

problems if they were already known. Additionally, the participants got help during the tool interaction if needed.

7.3.3 Wording of the Scenarios and Questions

The wording of the scenarios and the post questions may be misleading for some participants. To counteract this, the attendees were able to ask questions and the misunderstanding was cleared up if the interviewers noted such a misunderstanding.

7.3.4 Purposefulness of the chosen Scenarios

The overall purposefulness of the conflict awareness visualisation may be rated lower if the chosen scenarios themselves were not relevant at all in the field. The participants had to additionally rate the purposefulness of each single scenario to mitigate this threat.

CHAPTER 8

Findings

RQ1: What conflict awareness information needs exist in software projects for developers? The findings of the literature research and the provided expert opinions from the semi-structured expert interviews gave insight on existing conflict information needs. One existing information need mentioned was better filtering possibilities in tools. The participants of the interviews provided recommendations for additional filters for the tool. The findings of the literature research suggest that a tool should provide as much awareness information to the user as possible and ensure different ways to filter this information such that all the various preferences of developers can be addressed. Additionally, a better support for exploring the project's history was labelled as a current information need in this subject area. Examples for this were for example to mark commits of a specific feature, a better version traceability of code snippets or of renamed or moved files. Furthermore, conflict information should be delivered in an understandable way. The findings also suggest that these information needs exist not only for multi-branch but also for multi-project environments. Resolving conflicts in forks is often harder because the projects may have diverged substantially.

RQ2: How do developers prioritise the envisioned features? Based on the findings of the literature and tool research features of the prototype were proposed. During the semi-structured expert interviews these suggestions were evaluated and prioritised. The experts found all listed features useful. The top three rated proposed functionalities of the prototype were showing commit dependencies, showing commits of specific issues and showing the code sections of the conflict introduced by a merge, rebase or cherry pick. The complete sorted list of the suggested prototype features can be found in Table 5.1.

RQ3a: How purposeful do developers rate the proposed conflict awareness visualisation? The purpose of the idea of a conflict awareness visualisation for the

multi-branch and multi-project software development was rated high during the scenario-based expert evaluations. The participants noted that there was a high probability that they would use this type of visualisation in their everyday work once a certain market maturity had been achieved. Some participants already suggested that such a visualisation can include additional active or passive conflict information or that an integration, for example, in different IDEs and CI pipelines would increase the purposefulness. This indicates an interest in such awareness information. Another finding was that such a conflict awareness visualisation should provide the conflict information more passively as currently implemented. The user should be able to see the states of integrability of commits and branches of one or more projects without actively checking them one after another.

RQ3b: How purposeful is the proposed visualisation of the awareness tool? The chosen visualisation was generally seen as purposeful by the participants. But they also noted that the usability and performance issues need to be fixed before such visualisation is widely accepted. The biggest issues were the inadequate highlighting of commits the user filtered for, the missing autofocus on points of interest and the unintuitive checks for conflicts in cherry picks, merges and rebases. To improve the usefulness of the divergence visibility between forks and their parent, the user should be able to see different graphs at once such that he or she can compare the structures without the need to switch between them. But not only the usability issues need to be addressed, also the performance problems. For example, an indexing of the commit dependencies can help in such matter.

RQ3c: How efficient is the tool compared to state-of-the-art methods? For showing divergences of the project structures, the prototype was rated better than the functionalities of the existing tools. The combined view of the project structures with the different colours provides a quick overview of similarities and differences. In addition, the compact view was well received by the participants, as it makes the general project structure clearer. The prototype was also able to convince the developers in terms of the representation of commit dependencies in comparison to the methods previously used. In comparison, the respondents rated the presentation of metadata of a merge or rebase as neutral. Above all, the possibility to also carry out writing actions and a more passive display of conflicts would encourage the participants to use the prototype rather than the tools previously used.

CHAPTER 9

Conclusion

This thesis examined the information needs of developers for conflict awareness in multi-branch and multi-project software development to prevent or minimise the costs and time to resolve merge conflicts and to make merges, rebases and cherry picks more efficient. A literature research and semi-structured expert interviews were conducted in order to get insight of the current information needs of the software engineers. Existing conflict awareness tools provide conflict information for multi-branch environments but for the multi-project software development, although the fork-based development is becoming increasingly popular, especially in the GitHub community.

The results of the literature research were used to create a conceptual design of a possible awareness visualisation. The mock-ups were then used for the semi-structured interviews as possible examples. During the interviews the participants were asked to rate the envisioned features. The received feedback showed an information deficit in the area.

In order to be able to evaluate the purposefulness of the envisioned idea of the conflict awareness visualisation, a prototype was developed. The prototype visualises the project structure of the main project, its forks and its parent and provides an overview of divergences between them. Additionally, the visualisation provides a compacted view of the project structures, as well as different filters and conflict checks.

Afterwards, scenario-based expert interviews were conducted. Six software engineers were asked to solve predefined scenarios with the prototype. The purposefulness of the idea, the chosen visualisation and the current features of the prototype were recognised by the interviewees. The participants suggested various improvement possibilities and feature extensions for the prototype which can be used as a base for further research.

One of them is to improve the performance of the prototype. The loading times for the initial graph to show up and for the graph to be constructed when selecting another project in the view are quite long and can be improved. Also, performance issues with the commit dependency calculation were identified. Indexing the dependencies in the

start-up phase can help with this issue. When the information is needed, it only needs to be retrieved from the database.

Additionally, usability issues were found. Filtered commits and clustered nodes need to be highlighted in a more prominent way. An additional help would be an autofocus functionality such that the tool automatically focuses on some points of interest like the first filtered commit. A possibility for the user to travel back and forth between filtered nodes was seen as additional improvement for this issue. For the commit dependencies, not only the highlighting should be improved, but also the number of how many commits the selected one depends on should be shown. This helps to prevent marked commits to be overlooked, especially if they are far apart. It was also desired that the tool provides an option that clustered node automatically expand if they hold filtered commits in them. Most of the developers also had problems with the difference between a committer and an author. For this reason, the explicit distinction between those filters should be removed. When interacting with the visualisation, for example by expanding clustered commits or loading the view with a fork of the main project, the graph should be more stable. Currently, the user gets lost on the last focus point he set. The colour-based divergence between the main project and one of its forks or its parent was rated as good, but in order to compare different projects the user has to manually save the view with a screenshot for example or the divergences have to be remembered. A possibility to show more projects at once will improve the opportunities to compare the project structures. An often-requested improvement was to allow also writing actions within the tool and not only read actions. Currently, the user must switch tools and can use the prototype only for the purpose of conflict checking. The possibility to perform cherry picks, merges and rebases and to solve conflicts when they are detected right away would improve the general usability and acceptance. During the checks the missing loading indications were misleading to the users. In this state they did not know if the tool is currently working or not. Additionally, the checks should be made available with a more intuitive approach like a drag and drop feature. Another noted issue was the granularity of the author list which is provided when a merge conflict is detected. This list contains too many authors and should only include those that change the conflicting code sections.

The functionality of the prototype should be adapted so that the merge and rebase checks are more passive. The participants wished for a passive overview which branches are still integrable and which not. Without the need of an active action to see this information, a greater benefit can be drawn from it. It would be more likely to detect conflicts earlier when they are still small to resolve. The integration in existing IDEs and possibilities to integrate automate the visualisations and to integrate the tool into a CI pipeline was a frequently requested feature. Furthermore, the conflict awareness visualisation is currently only compatible with GitHub projects. But many participants during the evaluation session stated that their current projects are hosted in GitLab. Therefore, the prototype should be extended such that also GitLab projects are supported. Additionally, not only forks or parents should be supported. The tool should also be able to show the divergences between local repositories.

List of Figures

List of Tables

Acronyms

AST Abstract Syntax Tree. 11, 21

CI Continuous Integration. 14, 57, 89, 92, 94

CI/CD Continuous Integration/Continuous Delivery. 20

CVCS Centralised Version Control System. 7, 9

DVCS Distributed Version Control System. 7, 9

IDE Integrated Development Environment. 77–82, 85–89, 92, 94

PR Pull Request. 10, 17, 18, 47, 48, 76–78, 96

UI User Interface. 22–25, 77–82, 87, 88, 95

UML Unified Modeling Language. 21

VCS Version Control System. xiii, 1, 7, 9, 14, 15, 46, 47, 57, 74–76, 96

Bibliography

[1] Paola Accioly et al. „Analyzing conflict predictors in open-source Java projects“. In: *Proceedings - International Conference on Software Engineering*. Vol. 11. ACM, 2018, pp. 576–586. ISBN: 9781450357166. DOI: 10.1145/3196398.3196437. URL: https://doi.org/10.1145/3196398.3196437.

[2] Sven Apel et al. „Semistructured merge: Rethinking merge in revision control systems“. In: *SIGSOFT/FSE 2011 - Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering*. 2011, pp. 190–200. ISBN: 9781450304436. DOI: 10.1145/2025113.2025141.

[3] Adrian Bachmann and Abraham Bernstein. „Software process data quality and characteristics - A historical view on open and closed source projects“. In: *International Workshop on Principles of Software Evolution (IWPSE)*. New York, New York, USA: ACM Press, 2009, pp. 119–128. ISBN: 9781605586786. DOI: 10.1145/1595808.1595830. URL: http://www.eclipse.org/.

[4] Brian Berliner. „CVS II : Parallelizing Software Development“. In: *Proceedings of the Winter 1990 USENIX Conference* (1990), pp. 341–352. URL: http://www.tiffe.de/Robotron/PDP-VAX/rtVAX300/NetBSD6.0/usr/src/external/gpl2/xcvs/dist/doc/cvs-paper.pdf.

[5] Nic Bertino. „Modern Version Control: Creating an efficient development ecosystem“. In: *SIGUCCS'12 - ACM Proceedings of the SIGUCCS Annual Conference* (2012), pp. 219–222. DOI: 10.1145/2382456.2382510.

[6] Dane Bertram et al. „Communication, collaboration, and bugs: The social nature of issue tracking in small, collocated teams“. In: *Proceedings of the ACM Conference on Computer Supported Cooperative Work, CSCW* (2010), pp. 291–300. DOI: 10.1145/1718918.1718972.

[7] Avijit Bhattacharjee et al. „An Exploratory Study to Find Motives Behind Cross-platform Forks from Software Heritage Dataset“. In: *Proceedings of the 17th International Conference on Mining Software Repositories*. ACM, 2020, pp. 11–15. ISBN: 9781450375177. DOI: 10.1145/3379597.3387512. arXiv: 2003.07970. URL: https://doi.org/10.1145/3379597.3387512.

[8] Jacob T. Biehl et al. „FASTDash: A visual dashboard for fostering awareness in software teams“. In: *Conference on Human Factors in Computing Systems - Proceedings* (2007), pp. 1313–1322. DOI: 10.1145/1240624.1240823.

[9] Tegawende F. Bissyande et al. „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 2013* (2013), pp. 188–197. DOI: 10.1109/ISSRE.2013.6698918.

[10] Leo Breiman. „Bagging predictors“. In: *Machine learning* 24.2 (1996), pp. 123–140.

[11] Leo Breiman. „Random Forests“. In: *Machine learning* 45.1 (2001), pp. 5–45.

[12] Caius Brindescu et al. „How do centralized and distributed version control systems impact software changes?“ In: *Proceedings - International Conference on Software Engineering*. CONFCODENUMBER. 2014, pp. 322–333. ISBN: 9781450327565. DOI: 10.1145/2568225.2568322. URL: http://dx.doi.org/10.1145/2568225.2568322.

[13] Caius Brindescu et al. „Planning for untangling: predicting the difficulty of merge conflicts“. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. Vol. 11. 20. ACM, 2020, pp. 801–811. ISBN: 9781450371216. DOI: 10.1145/3377811.3380344. URL: https://doi.org/10.1145/3377811.3380344.

[14] Yuriy Brun et al. „Crystal: Precise and unobtrusive conflict warnings“. In: *SIGSOFT/FSE 2011 - Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering* (2011), pp. 444–447. DOI: 10.1145/2025113.2025187.

[15] Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. „Evaluating and improving semistructured merge“. In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (2017), pp. 1–27. ISSN: 2475-1421. DOI: 10.1145/3133883. URL: https://doi.org/10.1145/3133883.

[16] Guilherme Cavalcanti et al. „The impact of structure on software merging: Semistructured versus structured merge“. In: *Proceedings - 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*. 2019, pp. 1002–1013. ISBN: 9781728125084. DOI: 10.1109/ASE.2019.00097. URL: https://git.io/fjneH.

[17] Per. Cederqvist. *Version management with CVS : for CVS 1.11*. 2002, p. 204. ISBN: 0954161718.

[18] Scott Chacon and Ben Straub. *Pro Git*. Vol. 2. Springer Nature, 2014, p. 441. URL: https://git-scm.com/book/en/v2.

[19] M Codoban et al. „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)*. 2015, pp. 1–10. DOI: 10.1109/ICSM.2015.7332446.

[20] William W Cohen. „Fast Effective Rule Induction“. In: *Machine Learning Proceedings 1995*. 1995, pp. 115–123. DOI: 10.1016/b978-1-55860-377-6.50023-2.

[21] Kattiana Constantino et al. „Understanding collaborative software development“. In: *Proceedings of the 15th International Conference on Global Software Engineering*. ACM, 2020, pp. 55–65. ISBN: 9781450370936. DOI: 10.1145/3372787.3390442. URL: https://doi.org/10.1145/3372787.3390442.

[22] Catarina Costa et al. „TIPMerge: Recommending developers for merging branches“. In: *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Vol. 13-18-Nove. 2016, pp. 998–1002. ISBN: 9781450342186. DOI: 10.1145/2950290.2983936. URL: http://dx.doi.org/10.1145/2950290.2983936.

[23] Isabella A Da Silva et al. „Lighthouse: Coordination through emerging design“. In: *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange, ETX 2006*. 2006, pp. 11–15. ISBN: 1595936211. DOI: 10.1145/1188835.1188838.

[24] 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. ISBN: 9781424437122. DOI: 10.1109/CHASE.2009.5071408. URL: www.selenic.com/mercurial/;.

[25] Santiago Perez De Rosso and Daniel Jackson. „Purposes, concepts, misfits, and a redesign of git“. In: *ACM SIGPLAN Notices* 51.10 (2016), pp. 292–310. ISSN: 15232867. DOI: 10.1145/2983990.2984018. URL: http://dx.doi.org/10.1145/2983990.2984018.

[26] Anh Nguyen Duc et al. „Forking and coordination in multi-platform development: A case study“. In: *International Symposium on Empirical Software Engineering and Measurement*. 2014. ISBN: 9781450327749. DOI: 10.1145/2652524.2652546. URL: http://dx.doi.org/10.1145/2652524.2652546.

[27] H. Christian Estler et al. „Awareness and merge conflicts in distributed software development“. In: *Proceedings - 2014 IEEE 9th International Conference on Global Software Engineering, ICGSE 2014* (2014), pp. 26–35. DOI: 10.1109/ICGSE.2014.17.

[28] Jonas Gamalielsson and Björn Lundell. „Long-term sustainability of open source software communities beyond a fork: A case study of LibreOffice“. In: *IFIP Advances in Information and Communication Technology* 378 AICT (2012), pp. 29–47. ISSN: 18684238. DOI: 10.1007/978-3-642-33442-9_3.

[29] Mehmet Gençer and Bülent Özel. „Forking the commons: Developmental tensions and evolutionary patterns in open source software“. In: *IFIP Advances in Information and Communication Technology* 378 AICT (2012), pp. 310–315. ISSN: 18684238. DOI: 10.1007/978-3-642-33442-9_27.

[30] Gleiph Ghiotto et al. „On the Nature of Merge Conflicts: A Study of 2,731 Open Source Java Projects Hosted by GitHub“. In: *IEEE Transactions on Software Engineering* 46.8 (2020), pp. 892–915. ISSN: 19393520. DOI: 10.1109/TSE.2018.2871083.

[32] Max Goldman, Greg Little, and Robert C Miller. „Collabode: Collaborative coding in the browser“. In: *Proceedings - International Conference on Software Engineering*. 2011, pp. 65–68. ISBN: 9781450305761. DOI: 10.1145/1984642.1984658. URL: http://dx.doi.org/10.1145/1984642.1984658http://hdl.handle.net/1721.1/79662http://creativecommons.org/licenses/by-nc-sa/3.0/.

[33] Georgios Gousios, Martin Pinzger, and Arie Van Deursen. „An exploratory study of the pull-based software development model“. In: *Proceedings - International Conference on Software Engineering*. 1. 2014, pp. 345–355. ISBN: 9781450327565. DOI: 10.1145/2568225.2568260. URL: http://dx.doi.org/10.1145/2568225.2568260.

[34] Johann Grabner et al. „Combining and Visualizing Time-Oriented Data from the Software Engineering Toolset“. In: *Proceedings - 6th IEEE Working Conference on Software Visualization, VISSOFT 2018*. 2018, pp. 76–86. ISBN: 9781538682920. DOI: 10.1109/VISSOFT.2018.00016.

[35] Mário Luís Guimarães and António Rito Silva. „Improving early detection of software merge conflicts“. In: *Proceedings - International Conference on Software Engineering* (2012), pp. 342–352. ISSN: 02705257. DOI: 10.1109/ICSE.2012.6227180.

[36] Lile Hattori and Michele Lanza. „Syde: A tool for collaborative software development“. In: *Proceedings - International Conference on Software Engineering* 2 (2010), pp. 235–238. ISSN: 02705257. DOI: 10.1145/1810295.1810339.

[37] Yuan Huang et al. „Mining Version Control System for Automatically Generating Commit Comment“. In: *International Symposium on Empirical Software Engineering and Measurement* 2017-November (2017), pp. 414–423. ISSN: 19493789. DOI: 10.1109/ESEM.2017.56.

[38] Jing Jiang et al. „Why and how developers fork what from whom in GitHub“. In: *Empirical Software Engineering* 22.1 (2017), pp. 547–578. ISSN: 15737616. DOI: 10.1007/s10664-016-9436-6.

[39] Eirini Kalliamvakou et al. „An in-depth study of the promises and perils of mining GitHub“. In: *Empirical Software Engineering* 21.5 (2016), pp. 2035–2071. ISSN: 15737616. DOI: 10.1007/s10664-015-9393-5. URL: https://github.com/features.

[40] Tien Duy B. Le et al. „RCLinker: Automated Linking of Issue Reports and Commits Leveraging Rich Contextual Information“. In: *IEEE International Conference on Program Comprehension* 2015-Augus (2015), pp. 36–47. DOI: 10.1109/ICPC.2015.13.

[41] Olaf Leßenich et al. „Indicators for merge conflicts in the wild: survey and empirical study“. In: *Automated Software Engineering* 25.2 (2018), pp. 279–313. ISSN: 15737535. DOI: 10.1007/s10515-017-0227-0.

[42] Stanislav Levin and Amiram Yehudai. „Alleviating Merge Conflicts with Fine-grained Visual Awareness“. In: (2015). arXiv: 1508.01872. URL: http://arxiv.org/abs/1508.01872.

[43] Wardah Mahmood et al. „Causes of merge conflicts: A case study of elasticsearch“. In: *ACM International Conference Proceeding Series* (2020). DOI: 10.1145/3377024.3377047.

[44] Nora McDonald and Sean Goggins. „Performance and Participation in Open Source Software on GitHub“. In: *Conference on Human Factors in Computing Systems - Proceedings* 2013-April (2013), pp. 139–144. DOI: 10.1145/2468356.2468382.

[45] Shane McKee et al. „Software practitioner perspectives on merge conflicts and resolutions“. In: *Proceedings - 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017* (2017), pp. 467–478. DOI: 10.1109/ICSME.2017.53.

[46] Nicholas Nelson et al. *The life-cycle of merge conflicts: processes, barriers, and strategies*. Vol. 24. 5. Empirical Software Engineering, 2019, pp. 2863–2906. ISBN: 1066401896. DOI: 10.1007/s10664-018-9674-x.

[47] Anh Tuan Nguyen et al. „Multi-layered approach for recovering links between bug reports and fixes“. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE 2012*. New York, New York, USA: ACM Press, 2012. ISBN: 9781450316149. DOI: 10.1145/2393596.2393671. URL: http://www.prismaindustriale.com.

[48] Martin Nordio et al. „Collaborative Software Development on the Web“. In: (2011). arXiv: 1105.0768. URL: http://arxiv.org/abs/1105.0768.

[49] Linus Nyman et al. „Perspectives on code forking and sustainability in open source software“. In: *IFIP Advances in Information and Communication Technology* 378 AICT (2012), pp. 274–279. ISSN: 18684238. DOI: 10.1007/978-3-642-33442-9_21.

[50] Moein Owhadi-Kareshk, Sarah Nadi, and Julia Rubin. „Predicting Merge Conflicts in Collaborative Software Development“. In: *International Symposium on Empirical Software Engineering and Measurement* 2019-Septe (2019). ISSN: 19493789. DOI: 10.1109/ESEM.2019.8870173. arXiv: 1907.06274.

[51] Antoine Pietri, Guillaume Rousseau, and Stefano Zacchiroli. „Forking Without Clicking“. In: *Proceedings of the 17th International Conference on Mining Software Repositories*. Vol. 11. 2020. ACM, 2020, pp. 277–287. ISBN: 9781450375177. DOI: 10.1145/3379597.3387450. URL: https://doi.org/10.1145/3379597.3387450.

[52] Gregorio Robles and Jesús M. González-Barahona. „A comprehensive study of software forks: Dates, reasons and outcomes“. In: *IFIP Advances in Information and Communication Technology* 378 AICT (2012), pp. 1–14. ISSN: 18684238. DOI: 10.1007/978-3-642-33442-9_1.

[53] Anita Sarma, David F. Redmiles, and André Van Der Hoek. „Palantír: Early detection of development conflicts arising from parallel code changes“. In: *IEEE Transactions on Software Engineering* 38.4 (2012), pp. 889–908. ISSN: 00985589. DOI: 10.1109/TSE.2011.64.

[54] 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 (2017), pp. 33–47. ISSN: 09505849. DOI: 10.1016/j.infsof.2016.11.010.

[55] Yan Sun et al. „Improving missing issue-commit link recovery using positive and unlabeled data“. In: *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (2017), pp. 147–152. DOI: 10.1109/ASE.2017.8115627.

[56] Chungha Sung et al. „Towards understanding and fixing upstream merge induced conflicts in divergent forks“. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. Vol. 20. 2020, pp. 172–181. ISBN: 9781450371230. DOI: 10.1145/3377813.3381362. URL: https://doi.org/10.1145/3377813.3381362.

[57] Wouter Swierstra and Andres Löh. „The semantics of version control“. In: *Onward! 2014 - Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Part of SPLASH 2014* (2014), pp. 43–54. DOI: 10.1145/2661136.2661137.

[58] Walter F. Tichy. „Rcs — a system for version control“. In: *Software: Practice and Experience* 15.7 (1985), pp. 637–654. ISSN: 1097024X. DOI: 10.1002/spe.4380150703.

[59] R Viseur. „Forks impacts and motivations in free and open source projects“. In: *International Journal of Advanced Computer Science and Applications* 3.2 (2012), pp. 1–6. ISSN: 2158107X. DOI: 10.14569/ijacsa.2012.030221.

[60] Roel Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Jan. 2014, pp. 1–332. ISBN: 978-3-662-43838-1. DOI: 10.1007/978-3-662-43839-8.

[61] Rongxin Wu et al. „ReLink: Recovering links between bugs and changes“. In: *SIGSOFT/FSE 2011 - Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering*. New York, New York, USA: ACM Press, 2011, pp. 15–25. ISBN: 9781450304436. DOI: 10.1145/2025113.2025120. URL: http://code.google.com/p/zxing/issues/detail?id=18.

[62] Rui Xie et al. „DeepLink: A Code Knowledge Graph Based Deep Learning Approach for Issue-Commit Link Recovery“. In: *SANER 2019 - Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering* (2019), pp. 434–444. DOI: 10.1109/SANER.2019.8667969.

[63] Shurui Zhou. „Improving collaboration efficiency in fork-based development“. In: *Proceedings - 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019* (2019), pp. 1218–1221. DOI: 10.1109/ASE.2019.00144.

[64] Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. „How Has Forking Changed in the Last 20 Years? A Study of Hard Forks on GitHub“. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* 12 (2020), pp. 445–456. DOI: 10.1145/3377811.3380412. URL: https://doi.org/10.1145/3377811.3380412.

[65] Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. „What the fork: A study of inefficient and efficient forking practices in social coding“. In: *ESEC/FSE 2019 - Proceedings of the 2019 27th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, pp. 350–361. ISBN: 9781450355728. DOI: 10.1145/3338906.3338918. URL: https://doi.org/10.1145/3338906.3338918.

[66] Thomas Zimmermann. „Mining workspace updates in CVS“. In: *Proceedings - ICSE 2007 Workshops: Fourth International Workshop on Mining Software Repositories, MSR 2007* (2007), pp. 7–10. DOI: 10.1109/MSR.2007.22.

Online

[31] *Git - gitglossary Documentation*. URL: https://git-scm.com/docs/gitglossary (visited on 10/15/2020).

Appendix

Conflict Awareness Visualisation - Ranking

This questionaire is used to evaluate the importance of proposed features of a conflict awareness visualisation by importance.
The questionnaire has three sections: General, Experiences and Importance Evaluation
*Required

General

Please provide some information about your person.

1. How old are you (in years)? *

_____

2. What is your gender? *

Mark only one oval.

- ◯ Male
- ◯ Female
- ◯ Other: _____

Experiences

Please provide some information about your experiences in the field.

3. How much experience do you have in software engineering (in years)? *

_____

4. How much experiences do you have with Version Control Systems (in years)? *

_____

5. Which Version Control Systems have you used before? *

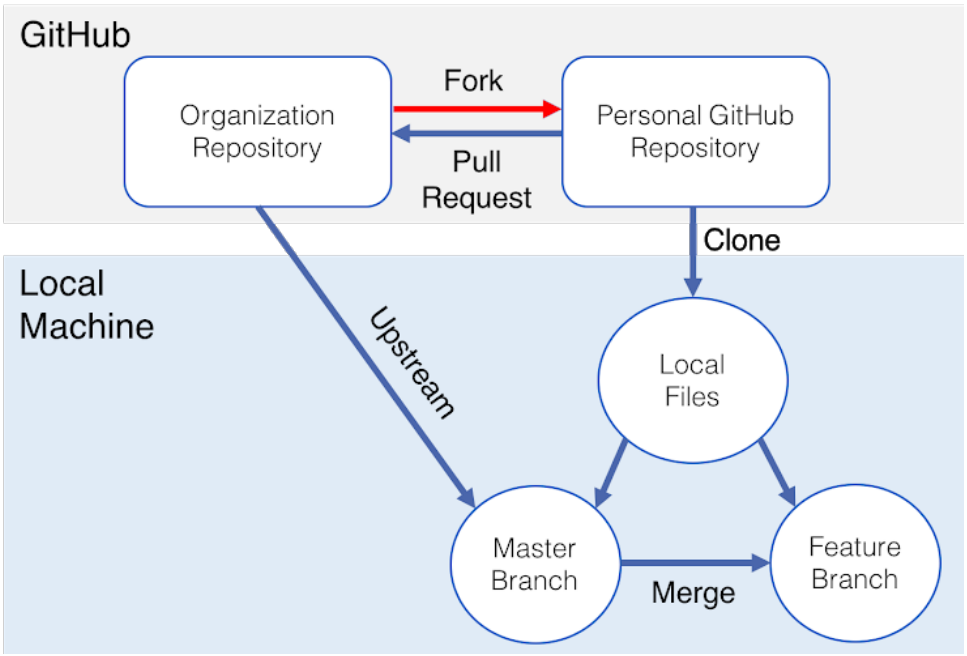
Tick all that apply.

- [ ] None
- [ ] Git
- [ ] Mercurial
- [ ] Subversion
- [ ] CVS

Other: [ ] _____

6. Do you know the principle of forking projects in version control hosting systems like GitHub? *

Example of a Fork



Mark only one oval.

- ◯ Yes
- ◯ No

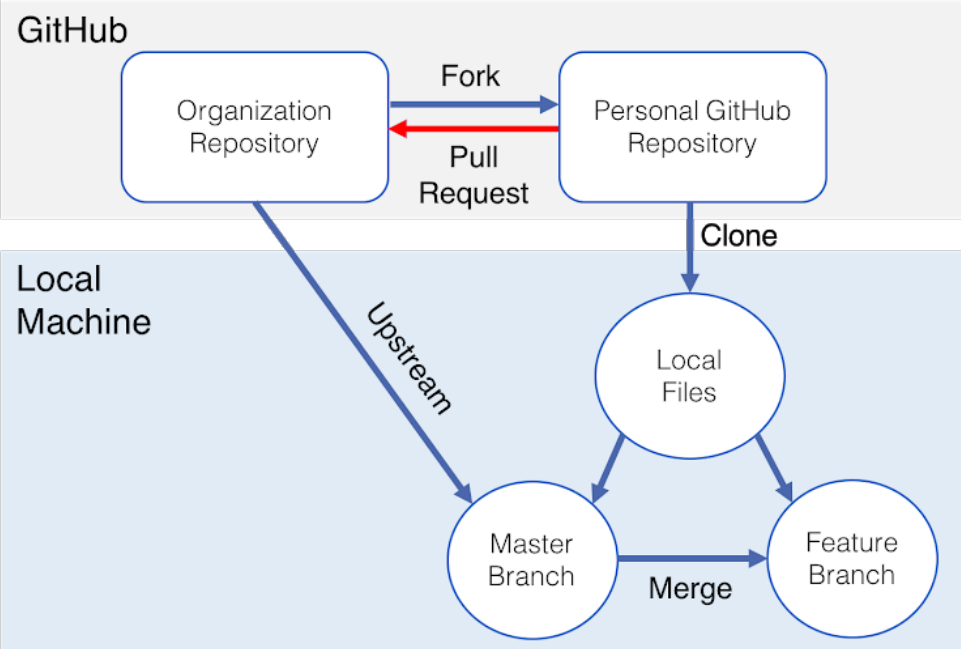
7. Have you forked a project in the past? *

Mark only one oval.

- Yes
- No

8. Do you know the principle of pull requests in version control hosting systems like GitHub? *

Example of a Pull Request



Mark only one oval.

- Yes
- No

9. Have you merged a pull request in the past? *

Mark only one oval.

- Yes
- No

10. Do you have a preference between Merging and Rebasing?

Mark only one oval.

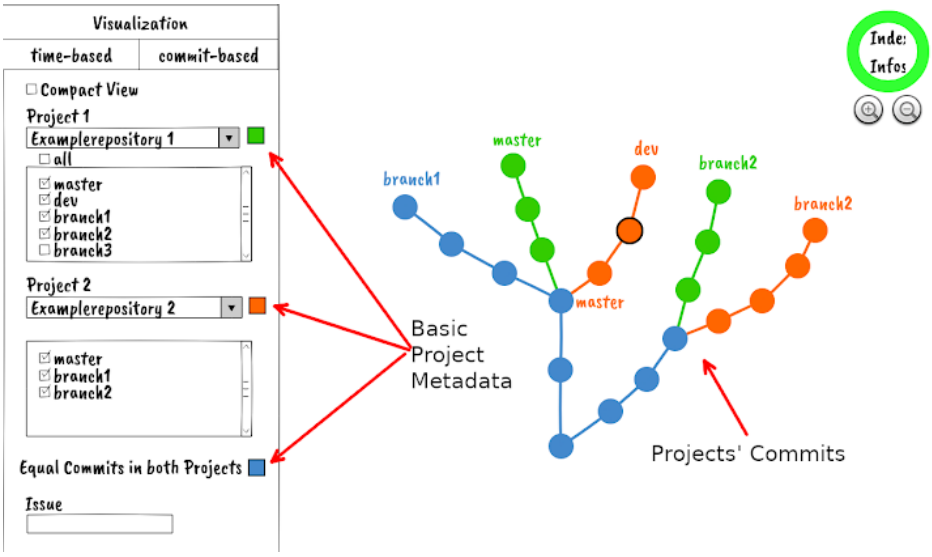
	1	2	3	4	5	
Preference Merging	◯	◯	◯	◯	◯	Preference Rebasing

Importance Evaluation

Please rate, how important you consider the proposed features of the visualisation.

11. How important do you find a visualisation that shows the **divergence between forks**? *

Example of a possible visualisation showing divergences (**green**: commits of original project; **orange**: commits of a selected fork; **blue**: commits of both projects)

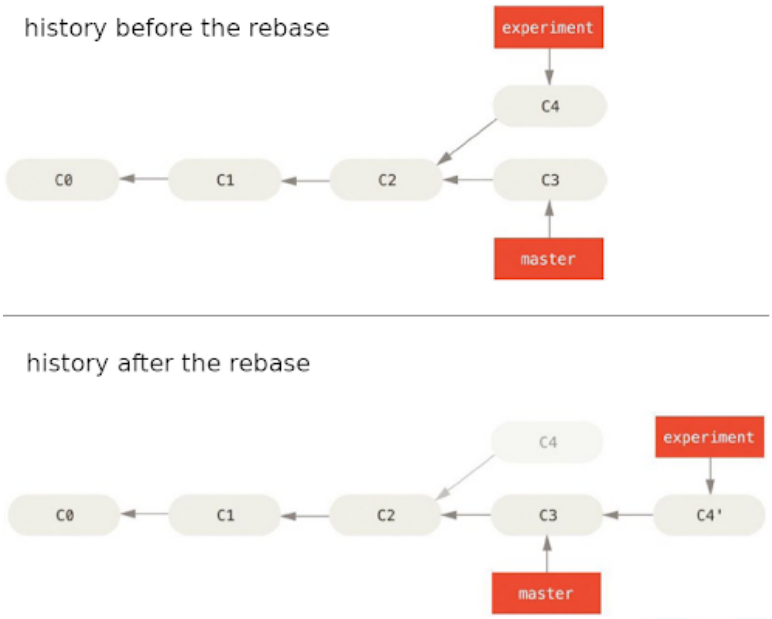


Mark only one oval.

	1	2	3	4	5	
not important	◯	◯	◯	◯	◯	very important

12. How important do you find a visualisation that shows, if branches can be **rebased** without conflicts **within a project**? *

Rebase example showing Git history **before** rebasing the **experiment** branch to the **master** branch, and **after** the rebase



Mark only one oval.

	1	2	3	4	5	
not impotant	◯	◯	◯	◯	◯	very important

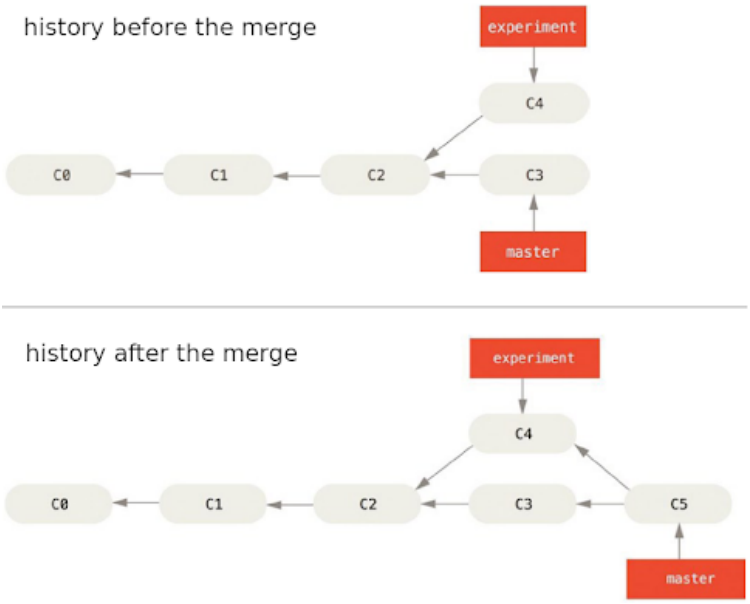
13. How important do you find a visualisation that shows, if branches can be **rebased** without conflicts **between forked projects**? *

Mark only one oval.

	1	2	3	4	5	
not important	◯	◯	◯	◯	◯	very important

14. How important do you find a visualisation that shows, if branches can be **merged** without conflicts **within a project**? *

Merge example showing Git history **before** merging the **experiment** branch into the **master** branch, and **after** the merge



Mark only one oval.

	1	2	3	4	5	
not important	◯	◯	◯	◯	◯	very important

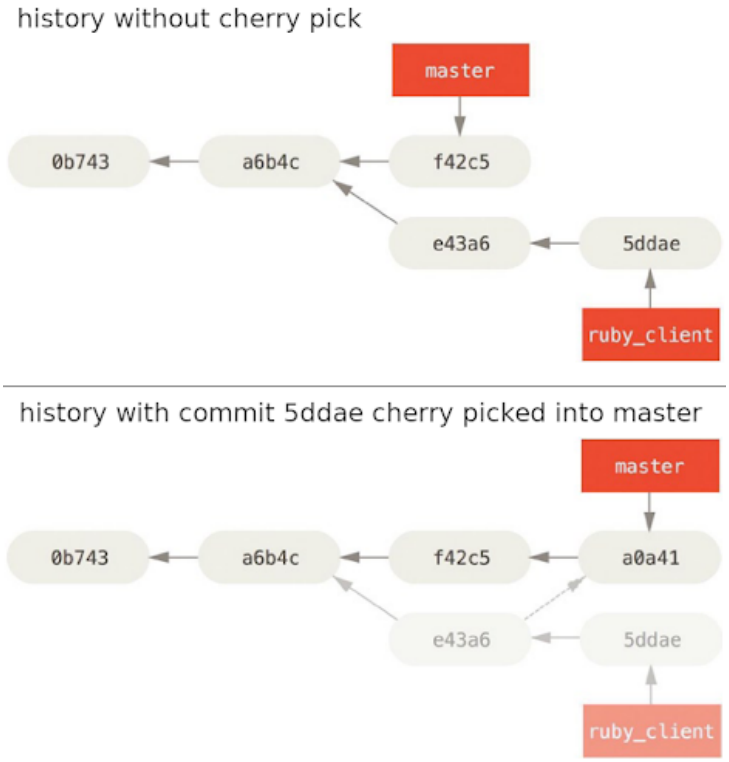
15. How important do you find a visualisation that shows, if branches can be **merged** without conflicts **between forked projects**? *

Mark only one oval.

	1	2	3	4	5	
not important	◯	◯	◯	◯	◯	very important

16. How important do you find a visualisation that shows, if commits can be **cherry picked** without conflicts in a branch **within a project**? *

Cherry Pick example showing Git history **before** cherry picking the **5ddae** commit to the **master** branch, and **after** the cherry pick



Mark only one oval.

	1	2	3	4	5	
not important	◯	◯	◯	◯	◯	very important

17. How important do you find a visualisation that shows, if commits can be **cherry picked** without conflicts **from one project into another**? *

Mark only one oval.

	1	2	3	4	5	
not important	◯	◯	◯	◯	◯	very important

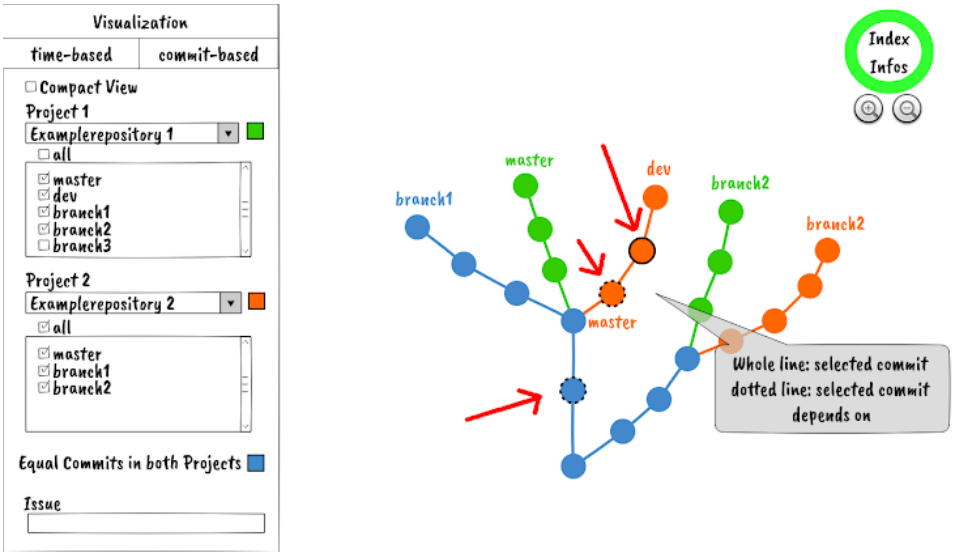
18. How important do you find a visualisation that shows the **code sections of a conflict**, if one was found? *

Mark only one oval.

	1	2	3	4	5	
not important	◯	◯	◯	◯	◯	very important

19. How important do you find a visualisation that shows the commits which the selected one **depends** on? *

Example of a possible visualisation showing dependent commits (**whole circled**: selected commit; **dotted circled**: previous commits which the selected one depends on (e.g. they have code snippets that are needed in the selected commit))

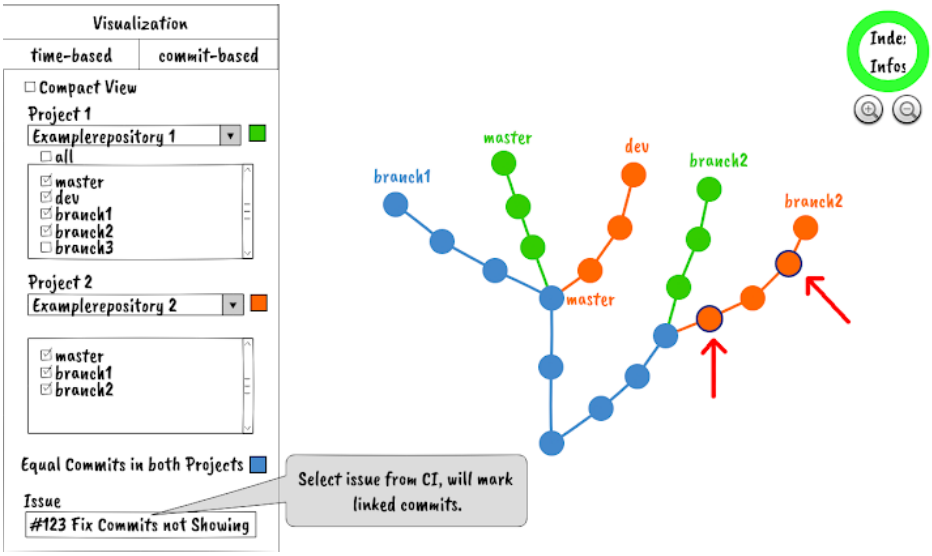


Mark only one oval.

	1	2	3	4	5	
not important	◯	◯	◯	◯	◯	very important

20. How important do you find a visualisation that shows **commits of a selected issue**? *

Example of a possible visualisation showing commits of a selected issue

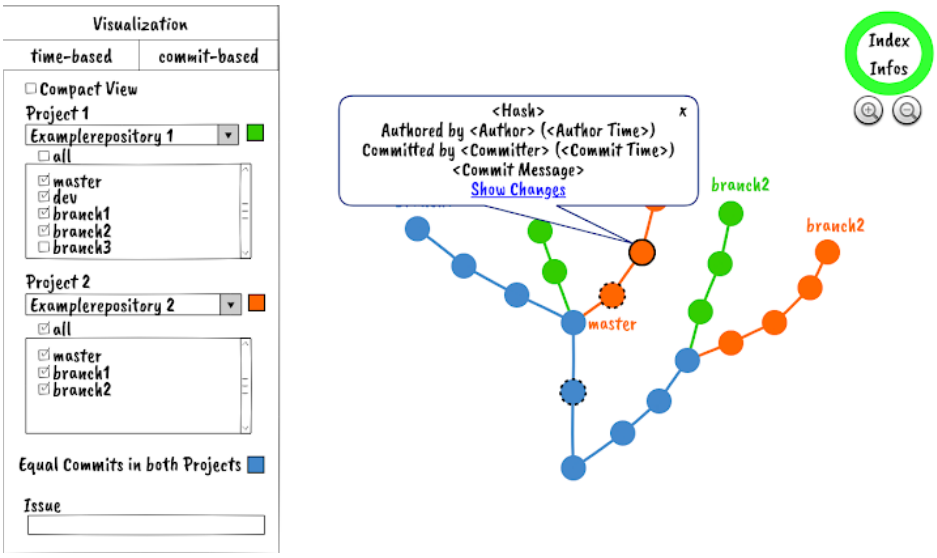


Mark only one oval.

	1	2	3	4	5	
not important	◯	◯	◯	◯	◯	very important

21. How important do you find a visualisation that shows **metadata** of a selected commit **across forks**? (commit hash, author, committer, commit message, changes) *

Example of a possible visualisation showing metadata of a specific commit across forks

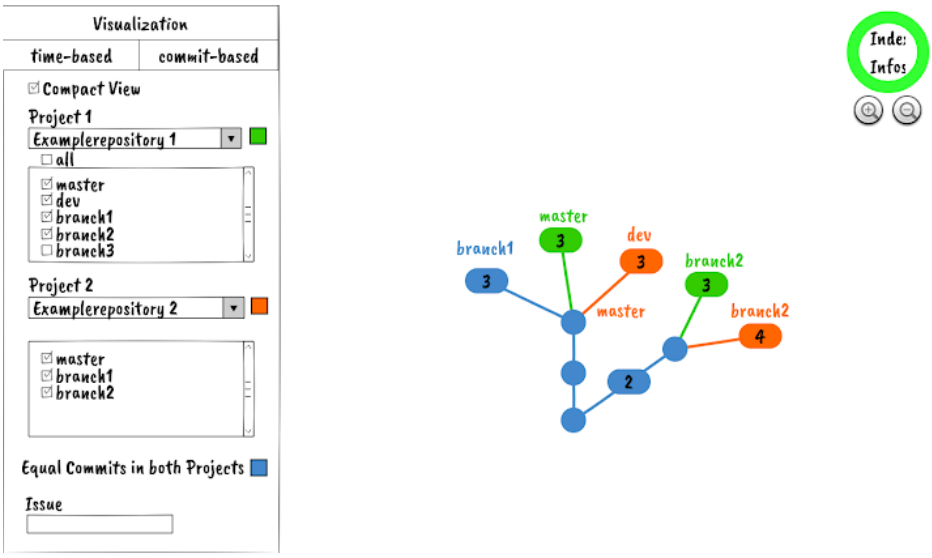


Mark only one oval.

	1	2	3	4	5	
not important	◯	◯	◯	◯	◯	very important

22. How important do you find a possibility to make the visualisation **compact**? *

Example of a possible visualisation showing a compact view of the projects representation



Mark only one oval.

	1	2	3	4	5	
not important	◯	◯	◯	◯	◯	very important

23. How important do you find a possibility to **hide commits of selected branches** in the visualisation? *

Mark only one oval.

	1	2	3	4	5	
not important	◯	◯	◯	◯	◯	very important

24. Which **filter options** do you find important in such a visualisation for **commits**?

Tick all that apply.

	Show filtered commits only	Highlight filtered commits
after a specific timestamp	☐	☐
within a specific time frame	☐	☐
of a specific author	☐	☐
of a specific committer	☐	☐
after a selected commits (subtree)	☐	☐

25. Do you find **other filter options** important in such a visualisation?

Scenario-based Expert Evaluations

*Required

General

Please provide some information about your person.

1. How old are you (in years)?

   Mark only one oval.

   - ◯ < 25
   - ◯ 25 - 34
   - ◯ 35 - 44
   - ◯ 45 - 54
   - ◯ 55 - 64
   - ◯ >= 65

2. What is your gender?

   Mark only one oval.

   - ◯ Male
   - ◯ Female
   - ◯ Other: _____

Experiences

Please provide some information about your experiences in the field.

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

   _____

4. How much experiences do you have with Version Control Systems (in years)?

   _____

5. Which Version Control Systems have you used before?

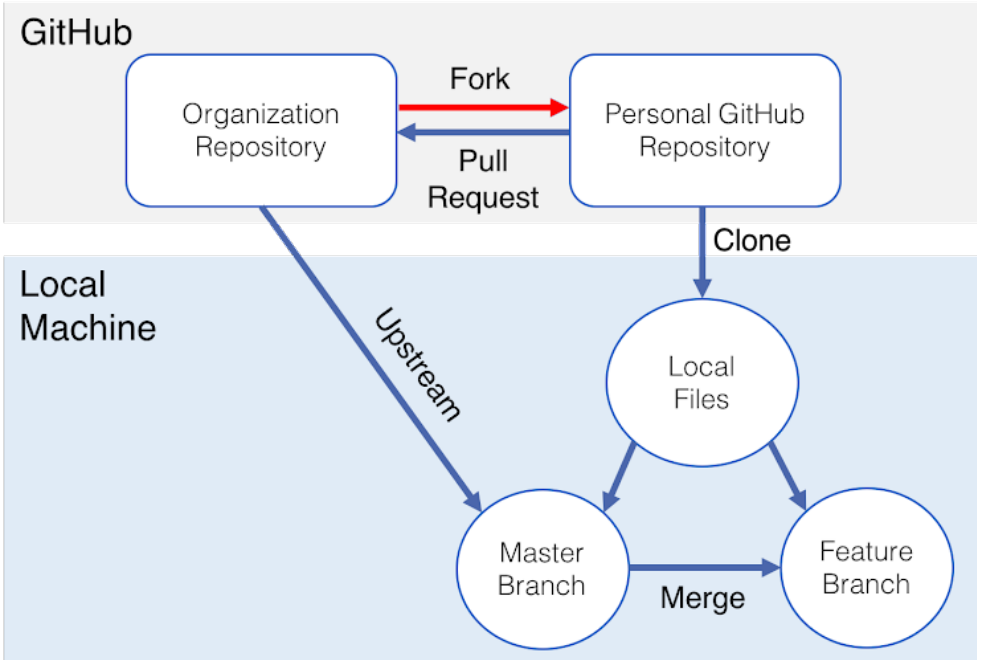
Tick all that apply.

- [ ] None
- [ ] Git
- [ ] Mercurial
- [ ] Subversion
- [ ] CVS

Other: [ ] _____

6. Do you know the principle of forking projects in version control hosting systems like GitHub?

Example of a Fork



Mark only one oval.

- ◯ Yes
- ◯ No

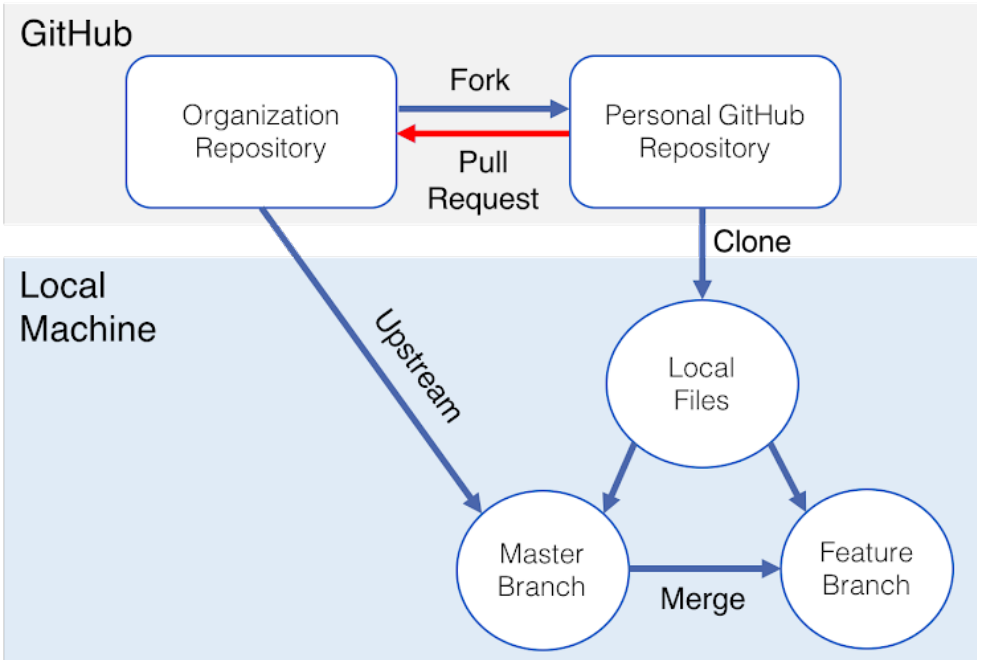
7. Have you forked a project in the past?

Mark only one oval.

- Yes
- No

8. Do you know the principle of pull requests in version control hosting systems like GitHub?

Example of a Pull Request



Mark only one oval.

- Yes
- No

9. Have you merged a pull request in the past?

Mark only one oval.

- Yes
- No

10. Do you have a preference between Merging and Rebasing?

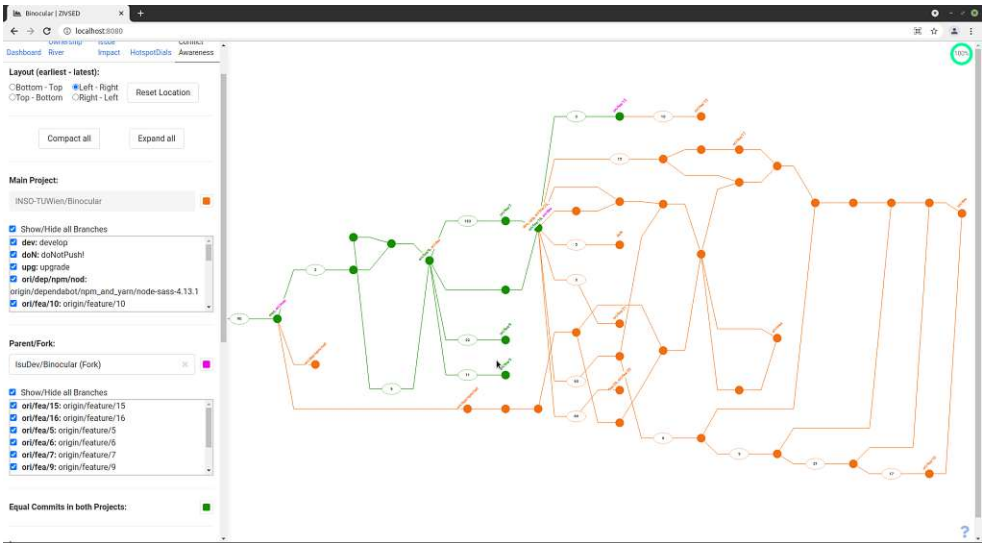
Mark only one oval.

	1	2	3	4	5	
Preference Merging	◯	◯	◯	◯	◯	Preference Rebasing

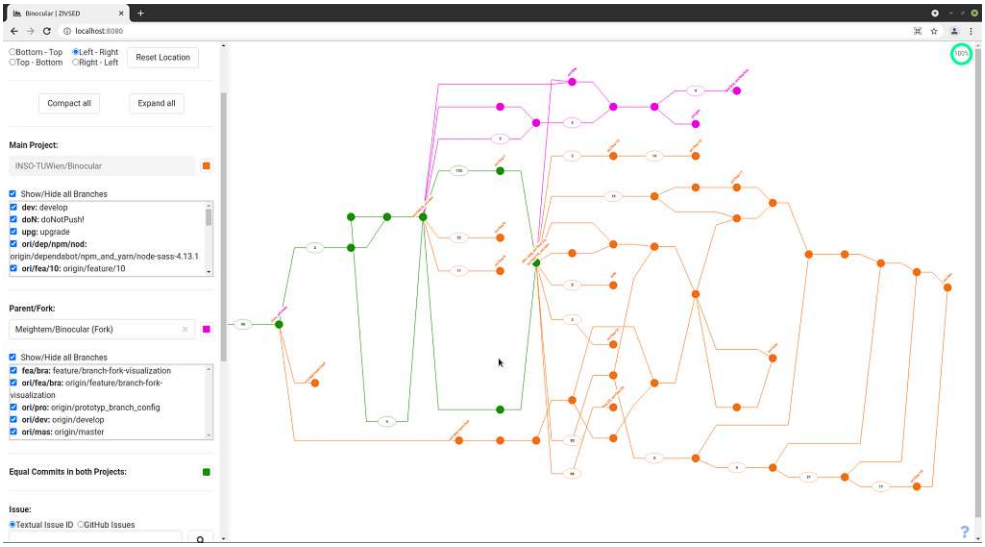
Scenario 1/6

11. Which **forked project diverged at most** from the base project?

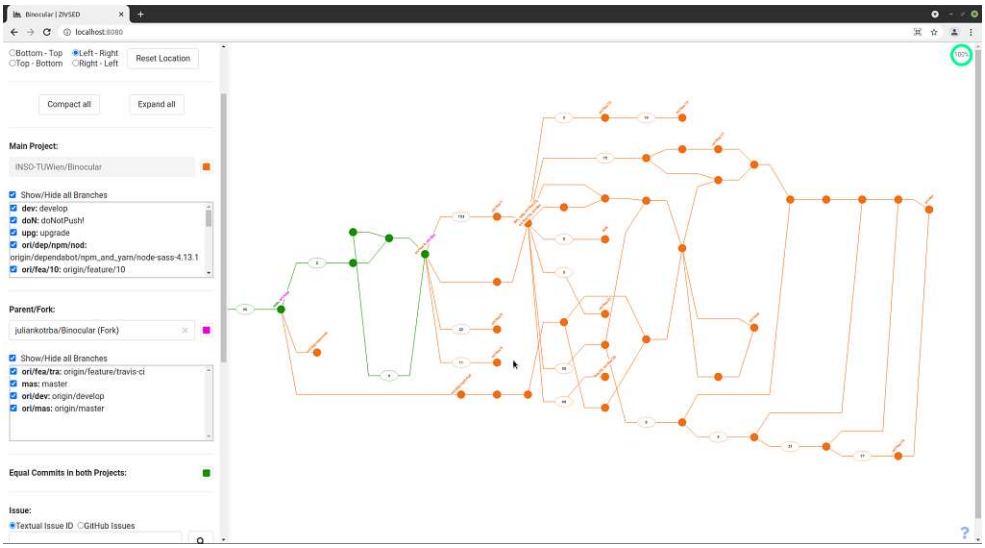
Divergence INSO-TUWien/Binocular - IsuDev/Binocular (Fork)



Divergence INSO-TUWien/Binocular - Meightem/Binocular (Fork)



Divergence INSO-TUWien/Binocular - juliankotrba/Binocular (Fork)



12. How relevant and practical did you find this scenario?

Mark only one oval.

	1	2	3	4	5	
not relevant at all	◯	◯	◯	◯	◯	highly relevant

13. Which tools would you normally have used and how would you have solved this scenario?

14. Which approach would you prefer in comparison and why?

Scenario 2/6

Hint: The project uses feature branches.

15. On which branch (project INSO-TUWien/Binocular) was **feature #20** implemented?

16. How relevant and practical did you find this scenario?

Mark only one oval.

	1	2	3	4	5	
not relevant at all	◯	◯	◯	◯	◯	highly relevant

17. Which tools would you normally have used and how would you have solved this scenario?

18. Which approach would you prefer in comparison and why?

Scenario 3/6

Committer: Maximilian Zenz (e1633058@student.tuwien.ac.at)
Year: 2020
Project: INSO-TUWien/Binocular

19. **How many** commits **committed Maximilian Zenz** in **2020** to the repository **INSO-TUWien/Binocular**?

20. How relevant and practical did you find this scenario?

Mark only one oval.

	1	2	3	4	5	
not relevant at all	◯	◯	◯	◯	◯	highly relevant

21. Which tools would you normally have used and how would you have solved this scenario?

22. Which approach would you prefer in comparison and why?

Scenario 4/6

Commit Sha: d249bb8ff7b5904a1790067261811160f4bd2ef62
Commit Message: visualize co-change direction with graph edges
Commit Project: INSO-TUWien/Binocular
Commit Branch: origin/feature/15
Commit Position: 3rd Commit

Cherry Pick Project: Meightem/Binocular
Cherry Pick Branch: origin/develop

Hint: use the subtree filter to find the required commit (copy the commit sha above)

How to check cherry pick:
1. select commits (right click on node, incl. strg-key for multi selection)
2. select (right-click) on branch-reference in the graph
3. wait for modal window

23. **Which commits** must be **cherry picked as well** such that no conflict occurs when cherry picking commit **d249bb8ff7b5904a179006726181160f4bd2ef62** (project **INSO-TUWien/Binocular**) into the branch origin/develop (project **Meightem/Binocular**)?

24. How relevant and practical did you find this scenario? *

Mark only one oval.

	1	2	3	4	5	
not relevant at all	◯	◯	◯	◯	◯	highly relevant

25. Which tools would you normally have used and how would you have solved this scenario?

26. Which approach would you prefer in comparison and why?

Scenario 5/6

Merge Branch: origin/feature/branch-fork-visualization

Into Project: INSO-TUWien/Binocular
Into Branch: origin/master

How to check merge:
1. select (right-click) branch reference in the graph that should be merged into another branch
2. select branch reference in the graph into which should be merged
3. wait for modal window

27. **Who** may be able to help **solving the conflict** when **merging** branch **origin/feature/branch-fork-visualization** (project **Meightem/Binocular**) into branch **origin/master** (project **INSO-TUWien/Binocular**)?

28. How relevant and practical did you find this scenario?

Mark only one oval.

	1	2	3	4	5	
not relevant at all	◯	◯	◯	◯	◯	highly relevant

29. Which tools would you normally have used and how would you have solved this scenario?

30. Which approach would you prefer in comparison and why?

Scenario 6/6

Rebase Project: INSO-TUWien/Binocular
Rebase Branch: origin/feature/5

Onto Project: INSO-TUWien/Binocular
Onto Branch: origin/feature/9

How to check rebase:
1. select (right-click) branch reference in the graph that should be rebased onto another branch
2. select branch reference in the graph onto which should be rebased
3. wait for modal window

31. A conflict occurs when **rebasing** branch **origin/feature/5** (project "**INSO-TUWien/Binocular**") onto **origin/feature/9** (project **INSO-TUWien/Binocular**). **Which commit** introduced the conflict?

32. How relevant and practical did you find this scenario?

Mark only one oval.

	1	2	3	4	5	
not relevant at all	◯	◯	◯	◯	◯	highly relevant

33. Which tools would you normally have used and how would you have solved this scenario?

34. Which approach would you prefer in comparison and why?

Evaluation of the Visualisations Purposefulness

35. How **purposeful** would you rate the **idea of the conflict awareness visualisation** in general?

Mark only one oval.

	1	2	3	4	5	
not needed at all	◯	◯	◯	◯	◯	highly purposeful

36. How **purposeful** would you rate the **chosen visualisation** in general?

Mark only one oval.

	1	2	3	4	5	
not needed at all	◯	◯	◯	◯	◯	highly purposeful

37. How **purposeful** would you rate the **tool** in general?

Mark only one oval.

	1	2	3	4	5	
not needed at all	◯	◯	◯	◯	◯	highly purposeful

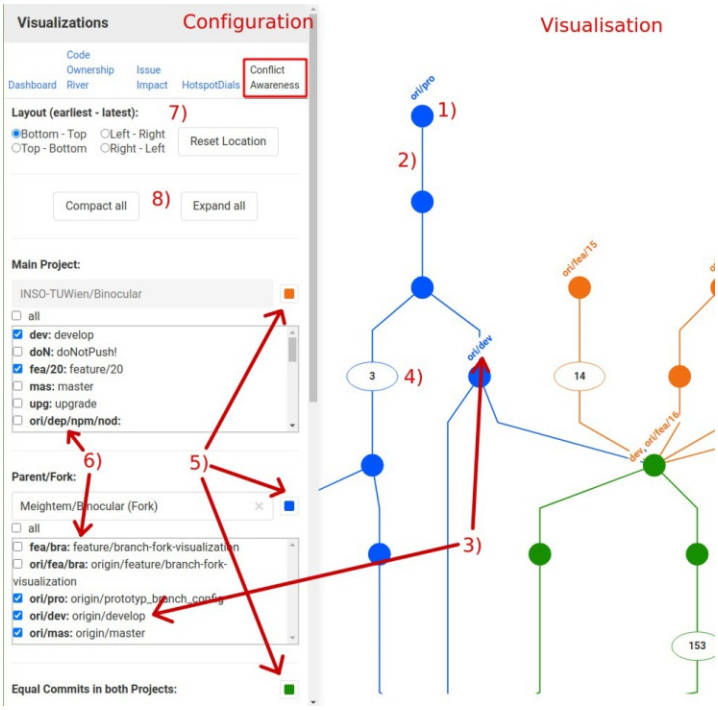
Scenario-based Expert Evaluation Quick Start Guide

Quick Start Guide

The visualisation shows divergences between parent repositories and their forks. It also provides conflict checks for merges, rebases and cherry picks within a repository or across forks. Additionally, different filter mechanisms are available across repositories.

General

1) **Commit**
2) **Parent-Child Relationship**
3) **Branch-Reference**
4) **Clustered Commits**: shows number of commits it holds
5) **Colours**: shows in which repository the element can be found
6) **Branches** of the Repository: checked ones shown in the graph
7) **Layout Options**:
   earliest = initial commit
   latest = branch heads
   Reset Location resets zoom and graph position
8) **Compact all**: compact whole graph(seen in the figure on the right)
   Expand all: expand the whole graph (clustered commits expanded)



General Visualisation Structure

Highlightings

Commit which meet all set Filters

Commit which not meet all set Filters

Dependency of a selected Commit

Commit of a selected Issue

selected Commit

Clustered Commits - all Commits within meet the set Filters

Clustered Commits - some Commits within meet the set Filters

Interactions

Hover over Commit:	show basic metadata (sha, author & time, committer & time, short message)
Double Click on Commit:	show detailed metadata (sha, commit message, diff)
Left Click on Commit:	select a commit
Left Click on Commit + Strg Key:	multi-select commits
Right Click on Commit:	copy sha
	collapse section
Right Click on Clustered Commits:	expand section
Hover over Branch-Reference:	show action + highlight history path
Click on Branch-Reference (nothing selected):	select branch
Click on Branch-Reference + Commit(s) selected:	start conflict check: cherry pick(s)
Click on Branch-Reference + Strg Key + Branch selected:	start conflict check: rebase
Click on Branch-Reference + Shift Key + Branch selected:	start conflict check: merge

Detailed Description

Graph

The graph consists of **nodes (= commits)** and **edges (= parent-child relationships of the commits)**. A commit which is a **head of at least one branch** has the corresponding **branch-references** floating around. The references including the full name of the branches can be found in the checkbox lists of the corresponding project sections in the configuration. **Checked** branch entries are shown in the visualisation, **unchecked** ones are not shown and not considered for the graph colouring.

The **colour** of these elements shows in which repository they can be found. There are three possibilities: 1) The element only exists in the **Main Project**. 2) The element only exists in the selected **Parent/Fork**. 3) The element can be found in **both**. This shows the forking point of the repositories.

The **layout** of the graph can be changed to four different settings. The **earliest commit** is the **initial commit**. The **latest commits** are the **branch heads**. The „**Reset Location**“ Button can help in finding the graph again when lost. This can happen especially when changing the layout.

Issue Selection

When selecting an issue the **linked commits** are **highlighted** with an dot-dashed black border. Issue commits in collapsed sections are handled like filtered commits. The compacted commits show the number of commits which meet the set filter criteria (including the selected issue) and the number of all commits it holds.

Selections

Branch-references and **nodes** are **selectable** with a left click. **Selected branches** are marked with a **black border** around its reference. **Hovering** over a branch will **highlight its path**. Only one branch can be selected at once.

Selected commits also get a **black border** as highlighting. When selecting a commit its **immediate dependencies** are **highlighted with a dashed black border**. Its possible to **select multiple commits** at once by **pressing the ctrl key**. Hovering over a commit will show its shortened commit message, its committer and the time of the commit and its author with the timestamp. By double clicking on the node the detailed commit message and its diff is loaded. To **reset all selections** press the escape key.

Conflict Checks

There are three types of conflict checks. The specific action that will be done is shown when hovering into the branch-reference.

1. Check if **cherry picks** can be done without conflicts:
   a) Select the commit(s) to cherry pick.
   b) Select the branch-reference onto which to cherry pick from.
   c) Wait for a success message (green modal on top) or a conflict message (modal over the screen with conflict details).
2. Check if a **merge** can be done without conflicts:
   a) Select the branch-reference which should be merged into another.
   b) Select the branch-reference that should be the base of the merge while pressing the **shift key**.
   c) Wait for a success message (green modal on top) or a conflict message (modal over the screen with conflict details).
3. Check if a **rebase** can be done without conflicts:
   a) Select the branch-reference which should be rebased onto another.
   b) Select the branch-reference that should be the base of the rebase while pressing the **ctrl key**.
   c) Wait for a success message (green modal on top) or a conflict message (modal over the screen with conflict details).

Compacting/Expanding

Compacting the graph can make the history structure clearer but this will lead to an information loss. The **whole graph** can be **compacted** by clicking the “**Compact all**” button and can be **expanded** by clicking the “**Expand all**” button on top of the configuration section.

Expanded sections can be **compacted** one by one by **right clicking on a commit** from its section and selecting the “**Compact Section**” menu item. When compacting from a branching node all its child paths will get compacted.

Collapsed sections can be **expanded** again by **right clicking on the collapsed commits** and by selecting the “**Expand**” menu item.

Filter

There are **five different filters** in the filter section of the configuration. For each filter two options can be chosen: “**highlight**” or “**show only**”. **Highlight filters** show all commits that are not covered by the set filters with less saturation.

Show only filters only show nodes that are covered by the set filters.

Exception: show only filter option with Author and Committer Filter:

In this case the whole graph will be compacted except those commits with the specific author or committer. The expansion feature does not work with both filter options.

For the subtree filter the sha of the commit is needed. It can be copied by right clicking on the commit and selecting the “**Copy Sha to Clipboard**” menu item.

- **After Filter**: commits after a selected date (inclusive)
- **Before Filter**: commits before a selected date (inclusive)
- **Author Filter**: commits with a selected author
- **Committer Filter**: commits with a selected committer
- **Subtree Filter**: commit children of a specific commit (inclusive)