Extending Git for Feature-Oriented Software Development

Tabea Viktoria Röthemeyer

Master's Thesis, 23.10.2024

Chair of Software Engineering

Supervisor: Prof. Thorsten Berger

Co-Supervisor: Associate Prof. Daniel Strüber

Abstract

Feature-oriented software development is an approach for managing configurable software products. However, managing features and distributing bug fixes across different variants present significant challenges, especially when aiming to unify version and variant management.

This thesis aims to investigate whether a Git extension can assist developers in finding and updating features across the project history. The proposed integration concept includes internal data structures stored in a separate branch and a command-line interface to customize and utilize this information.

To achieve this, the thesis investigates three primary research questions. First, it explores what commands a command-line interface (CLI) should provide to facilitate feature-oriented workflows that assist developers in feature management, while reflecting the syntax typically used in Git commands. Second, it examines how Git can be extended to support feature-oriented development, and what integration concept enables this while ensuring that the commands feel native to the user. Finally, it evaluates how the *Git with Features* prototype enhances the management and traceability of features in software development.

To answer these questions, three design research cycles were conducted, involving literature reviews, technical analyses, and iterative prototype developments to continuously improve the concept and the tool.

The evaluation was carried out through a user study using the developed prototype in a controlled setting. The results show that the tool enhances accuracy in feature-based tasks and can speed up more complex comparison-related ones. Although a learning effect was observed and some users initially had reservations, the overall feedback was positive. Also, the tool allows the migration of codebases to benefit from the new CLI commands.

I conclude that the Git extension contributes to improving feature management and traceability in software development. Future work should focus on optimizing the prototype based on the received feedback and extending the concept to other areas like bug tracking, where metadata can be linked to commits.

Acknowledgments

I would like to express my sincere gratitude to my supervisors, Thorsten Berger and Daniel Strüber, for their support and guidance throughout this project. Their insightful discussions about features, encouragement, and invaluable feedback greatly contributed to the completion of this work.

I also wish to thank the doctoral students and members of the Chair of Software Engineering for participating in the user study and for their assistance with organizational matters and camaraderie. Special thanks go to Sven Peldszus for his critical questions and discussions during the conceptual phase of my work, and to Kevin for answering all my questions and providing feedback on my user study design.

I am deeply grateful to Ansel Sermersheim for sharing his extensive knowledge of fundamental software concepts and for providing resources available to him through his vast experience. His deep understanding of Git allowed him to point me toward subtle ideas that aligned with the fundamental concepts I aimed to implement.

Finally, I am grateful to my colleagues at Auto-Intern GmbH for participating in the study and for their unwavering support, which allowed me to focus on writing my thesis.

Tabea Röthemeyer, Bochum, October 2024

Contents

List of Figures

Listings

1. Introduction

Software engineering has developed a wide range of advanced techniques to handle the growing complexity of software systems. Over the years, methods for handling versions, configurations, and changes have been refined, resulting in a variety of models and tools that provide structured management of these aspects across the software lifecycle [5].
One approach addressing the complexity in modern software systems is Feature-Oriented Software Development (FOSD), which organizes features as modular, reusable components within software. This approach is closely integrated with Software Product Lines (SPLs), where a common codebase generates multiple product variants. These variants are defined by their feature configurations. SPLs are widely used in industries such as automotive, telecommunications, and consumer electronics, where the ability to customize and vary features is crucial for product differentiation and market adaptability [1, 2].
Like any other codebase, every change in an SPL produces a new, distinct version of the software, forming a history of the project. Additionally, SPLs introduce a new dimension beyond the historical evolution in the time domain: the variant domain. This domain represents different configurations or products derived from the same codebase, existing alongside the version space. Managing both dimensions simultaneously presents unique challenges [2]. Developers often need to evolve features within the main codebase or for specific variants, and depending on the nature of the change, they might need to update other variants as well.
Despite the growing body of research on feature management within SPLs, identifying and localizing feature-related code remains a significant challenge. Although developers are aware of the correspondence between code and features during the development process, this information is often not recorded in the project's history [7]. Consequently, it becomes challenging to retrieve feature information later on, which can complicate maintenance and evolution tasks [10].
Existing tools, such as FeatureIDE and pure::variants, provide some level of feature management, but they operate separately from version control, requiring developers to synchronize data manually [15, 12]. Annotation systems that include comments in the code can further help locating features [14] but still require additional tooling to profit from the integration.

Motivation

Various f methods for annotating and localizing features within code have already been developed. The next step to improve tooling support might be integrating annotating systems for feature management into existing into a commonly used

version control system. As version control systems like Git already track all code changes, associating those changes with specific features could aid with the location, tracking and retrieval of features. Currently, this concept is not supported in version control systems [12] even though it would eliminate the need for developers to manually annotate code with feature information or rely on external tools.
My hypothesis is that by extending Git to store and manage associations between code changes and features, developers could not only track feature-specific changes but also gain insights into feature evolution, making it easier to update and maintain variants across SPLs. This could significantly reduce the time and effort spent on manual feature tracking and provide a seamless integration of variant management into the development workflow.

Problem Statement

However, it remains unclear how the extension of Git, called *Git with Features,* will be structured and whether it will offer measurable benefits to developers. As this idea has not been explored in depth, there is limited knowledge about the workflows and support developers would need to manage features in Git. Moreover, integrating feature management into Git poses technical challenges, such as how to store, retrieve, and associate feature information with code changes without affecting Git's performance or altering its core functionalities. Without a practical implementation and empirical evaluation, the potential advantages of this approach remain speculative.

Research Questions

To evaluate whether the presented approach of *Git with Features* is feasible and beneficial for developers, the following research questions arise. First, the structural requirements for an extension in terms of the user interface need to be known.

Research Question 1: What commands should a command-line interface provide to facilitate feature-oriented workflows that assist developers in feature management, while reflecting the syntax typically used in Git commands?

Then, the technical possibilities of how an extension can look like and what conceptual approach could be possible need to be explored.

Research Question 2: How can Git be extended to support feature-oriented development, and what integration concept enables this while ensuring that the commands feel native to the user?

Finally, the question arises of how this theoretical idea performs in practice. First, it must be demonstrated whether the proposed implementation concept can be feasibly translated into a functional and usable research prototype. Once the prototype is developed, its impact on developers' daily workflows needs to be evaluated to determine if it indeed simplifies feature-based development tasks.

Research Question 3: How does the *Git with Features* prototype enhance the management and traceability of features in software development?

Objectives

By answering these research questions, the following objectives are expected to be achieved: The primary objective of this thesis is to design, implement, and evaluate a Git extension that integrates feature-oriented principles into the version control process. The secondary objectives include:

- Defining workflows and data structures necessary for managing feature evolution within Git.
- Implementing a research prototype of the Git extension to demonstrate the feasibility of the solution.
- Designing and conducting an experiment, through a user study, to compare traditional Git workflows with feature-based workflows in terms of usability, learning curve, and efficiency.
- Developing an example project to demonstrate the practical applicability of the Git with Features tool.

Structure of the Thesis

Based on these ideas and the planned execution, the structure of the thesis is organized as follows: section 2 provides an overview of Git, variability management, and other relevant background information. Section 3 outlines the methodological approach used to address the research questions, detailing the steps and scientific methods applied throughout the thesis. The results of these methods, including the integration concept and evaluation outcomes, are presented in section 4 and section 5, with a thorough discussion of the findings and answers to the research questions provided in section 6. Finally, section 7 summarizes the key findings, addresses limitations, offers an outlook on future work, and concludes with a personal reflection on the overall process.

2. Background

This section provides background information on key concepts related to this thesis, including variability management, SPLE, and version control systems, with a particular emphasis on Git. Understanding these topics is critical for addressing the challenges discussed in later chapters, especially in relation to the proposed extension of Git for feature-based development.

2.1. Git as a Version Control System

Version control systems solve the problem of tracking and managing changes in software projects. As software evolves and bugs are introduced, it becomes crucial to keep track of different code versions to allow for easy comparison and reverting to previous states. Multiple versions of the software can exist simultaneously, with each change creating a new version that needs to be recorded and searchable. Git plays a significant role as a widely used distributed version control system that helps manage these versions efficiently.
One of the key challenges in using Git is dealing with merge conflicts, which can arise when multiple developers work on the same part of the code. Effective resolution strategies are necessary to ensure smooth collaboration. Git's approach to storing versions relies on a combination of snapshots and deltas, providing both flexibility and efficiency. It is the most prevalent version control system worldwide, primarily because of its distributed nature, allowing developers to work on local repositories and synchronize changes remotely. Core concepts in Git include the staging area, commit history, and branching, all of which are essential for effective version management. However, challenges like merge conflicts, branching strategies, and synchronizing local with remote repositories remain common issues that developers face.

2.1.1. Mental Model

The idea and development of Git originated from the need to manage the Linux Kernel's codebase effectively, as there was insufficient software to handle the project's scale and complexity. Linus Torvalds initiated the development of Git after announcing on the Linux Kernel Mailing List that he would be offline to create a new source control system[1]. Within a few days, he released a work-in-progress version of Git, with the first commits being made manually[2]. Torvalds emphasized that Git should be seen primarily as a distributed filesystem rather than a traditional

[1]https://lkml.iu.edu/hypermail/linux/kernel/0504.0/1540.html
[2]https://lkml.iu.edu/hypermail/linux/kernel/0504.0/2022.html

version control system[3]. This perspective underscores its design philosophy of efficient data storage, retrieval, and distribution, aligning with Unix principles.

2.1.2. Core Concepts and Operations in Git

Git is a distributed version control system that enables developers to efficiently track and manage changes in software projects. Unlike traditional version control systems, Git operates on a decentralized model, allowing each developer to have a complete copy of the repository, including its full history[4]. Core concepts in Git include the **staging area**, where changes are prepared before committing, and **commits**, which capture snapshots of the project's state along with metadata such as author and timestamp. Another fundamental concept is **branching**, which allows developers to work on multiple features or fixes simultaneously without affecting the main codebase[5].
Git's **merge** and **rebase** features provide strategies to integrate changes from different branches, ensuring that collaboration remains smooth and conflicts are minimized. Additionally, Git emphasizes fast, local operations by treating data as a series of **snapshots** rather than a set of changes (or deltas)[6]. This design choice, combined with robust branching and merging capabilities, allows teams to work efficiently even on large-scale projects. For a more detailed understanding, refer to the official Git documentation[7].

2.2. Feature-Oriented Software Development (FOSD)

Software Product Lines (SPLs) represent a strategy for creating multiple variants or configurations of a software product that are tailored to specific customer needs. SPLs can be managed through different approaches, including the clone-and-own method, where separate versions of the software are manually adapted, or integrated platform approaches that allow for shared development. A feature in this context is defined as a distinct block of functionality, often directly visible to the user. Managing features effectively involves tracing their presence across different variants and ensuring they are properly documented, sometimes through metadata embedded directly in the code, as discussed by Schwarz et al. [16].

[3] https://lkml.iu.edu/hypermail/linux/kernel/0504.1/0212.html, https://lkml.iu.edu/hypermail/linux/kernel/0504.1/0190.html

[4] https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

[5] https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell

[6] https://git-scm.com/book/en/v2/Git-Internals-Plumbing-and-Porcelain

[7] https://git-scm.com/doc

2.3. Variability Management and Variation Control Systems

Variability management refers to the techniques and processes used to handle and control the differences between product variants within a software product line. Various operators have been identified for managing these variations, such as those described by Linsbauer et al. [11]. However, variability management faces several challenges, including the need for efficient traceability, consistency, and integration with existing development tools. Current tool support for variability management focuses on tasks like feature selection, variant configuration, and automated builds, with tools like FeatureIDE and configuration management systems playing key roles.

2.4. System PATH in the Context of Installation

The installation of software adheres to established conventions for the storage of executable code and data. While discrepancies exist across operating systems, a unifying concept is the use of a PATH environment variable, which lists directories that the operating system scans to locate programs. In the event that a program is located within one of the previously mentioned directories, it is possible to access said program via the command line without the necessity of providing the complete file path.
This enables users to execute programs via straightforward commands, as the system searches directories for pertinent executables. It is necessary for the directory in which new software is to be installed to be either in the PATH or for the user to be informed of the procedure for adding it. In the absence of this configuration, the software will not function when executed via the command line, as the command will not be recognized.

3. Methods

In the introduction, the research questions that guide this thesis are presented. This section outlines the methods used to address those questions, explaining the rationale for each approach and detailing their implementation. It also discusses the expected outcomes of the selected methods. The results are presented in the following chapters: the first focuses on the conceptual work, and the second on the practical evaluation.

3.1. Design Science Cycles

Since the practical value of the proposed concept will be determined by the users themselves, this work is fundamentally based on Design Science Research (DSR). As outlined in *Constructive Master's Thesis Work in Industry: Guidelines for Applying Design Science Research*([9]), DSR follows an iterative approach for refining tools or systems through user interaction and feedback. The process begins with developing an initial prototype, followed by conducting user tests to gather both qualitative and quantitative data. These tests evaluate how participants engage with the system, measuring usability, task performance, and user satisfaction. Based on this feedback, the prototype is iteratively improved, enhancing its functionality and alignment with user needs and system requirements.
Below, it is explained how this method is used to develop the initial prototype for *Git with Features* and get the first iteration of user feedback.

Cycle 1 **Workflow and User Interface Description** The primary goal of this cycle is to define a workflow model and user interface that allow feature-oriented work within Git. By conducting a literature review and comparing with existing Git commands to maintain a consistent structure, potential feature-oriented workflows are proposed and a command-line based user interface is defined.

Cycle 2 **Integration Concept** Within this cycle, the technical integration of the developed user interface is evaluated. Also the internal data handling and storage model is mapped out. Based on this, an integration concept for the implementation of the *Git with Features* extension is described.

Cycle 3 **Prototype Implementation and User-Centered Evaluation** Finally, the proposed concept is assessed. To get structured feedback and insight into the improvements made compared to what is called here *traditional usage of Git*, a research prototype will be given to the users together with a test scenario and questions. The answers are used to derive potential improvements for the tool and quantitatively describe its impact.

3.2. Workflow and User Interface Design Approach

First, the methods used to define feature-oriented workflows and the corresponding user interface for the *Git with Features* extension are outlined. These workflows guide the design of the CLI, and the results are later used to derive an answer for research question 1.

3.2.1. Systematic Literature Analysis

As feature-oriented development has been a research focus for some time, there already exist various literature on feature management. Conducting a systematic literature review with a focus on common operations and tasks for developers can help to create a set of workflows that feature-oriented Git should support. To perform a systematic literature review, as explained by Kitchenham and Charters([8]), the key steps involve identifying relevant research, selecting databases, defining inclusion and exclusion criteria, and evaluating the selected papers [8]. The selection of relevant papers is made in collaboration with the supervising chair, based on their knowledge of the subject. The list of papers is included in the results. Based on the discussed challenges and recurring operations required when working on features, the scenarios of the workflows are selected.

3.2.2. Workflow Description

The workflow descriptions are developed by integrating both existing Git commands and new commands introduced by the *Git with Features* extension. Each workflow outlines how users interact with the command line to perform feature-related tasks, using scenarios identified through the literature review. Based on my prior Git experience, the workflows reflect feature management activities described in the selected papers. These workflows serve as a basis for designing the user interface and structuring the interactions between developers and the tool to ensure that the *Git with Features* extension supports feature management.

3.2.3. User Interface Description

The workflows derived from the analysis are translated into a set of specific command line interface (CLI) commands for the prototype implementation. Each command is defined with associated flags, a description of its functionality, and the expected output. The listing of these commands provides a specification of the user interface, which serves as the basis for the implementation of the *Git with Features* extension.

3.3. Development of the Integration Concept

The integration concept outlines how the derived user interface can be implemented through two key components. First, it explains the internal data models and handling mechanisms that make up information management. Second, it details the technique for integrating the new commands into Git's existing architecture. This concept provides a technical foundation for extending Git by addressing the structural requirements necessary for feature management and responding to research question 2. The design of the integration concept is done by evaluating the technical capabilities of Git and proposing data structures that build on top of Git's object model to store and retrieve all the information needed.

3.3.1. Technical Analysis of Git

The objective of this analysis is to evaluate Git's internal structure and identify extension points that allow the integration of feature-oriented functionality without modifying Git's source code. To achieve this, a review of Git's documentation is conducted, focusing on its storage model (commits, trees, and blobs) and extension mechanisms (hooks and subcommands). An existing Git extension, *git-annex*[8] is also analyzed to understand best practices for extending Git without breaking its core functionality. These tools demonstrate potential approaches to data storage and command integration, providing insights for the design of the *Git with Features* extension.
The result of this analysis is a structural understanding of Git's data model and an overview of extension mechanisms and their use cases. This lays the groundwork for the integration concept, ensuring that feature management can be implemented in a way that is consistent with Git's architecture.

3.3.2. Development of the Integration Concept

Based on the results of both the technical analysis of Git and the previously defined UI, an integration concept for *Git with Features* is derived. The integration concept describes, in a technology-agnostic way, how the *Git with Features* extension can be implemented by integrating the UI with Git's underlying architecture. It builds on both components by introducing a model for storing and accessing feature information within Git, which is necessary for the new subcommands.
The insights gained from the Git analysis inform the design of the internal data structures, while the results of the UI design ensure that these structures can be interacted with through subcommands. This concept outlines how feature-related

[8] `https://git-annex.branchable.com/`

information is stored, retrieved, and updated in Git, defining both the internal data handling mechanisms and the user-facing commands required for interaction.
As a result, the integration concept serves as an implementation guideline for the *Git with Features* prototype, including how feature information is stored, retrieved, modified, and displayed.

3.4. Evaluation Methods

Finally, it describes how the results for answering the last research question 3 can be provided. Previous research has focused on requirements. Now the actual impact of the tool on the developer workflow is evaluated.
This section describes the methodical approach for setting up the test environment. This includes the development of the prototype as well as all materials necessary to conduct the experiment. The objective of the evaluation is to test whether the proposed *Git with Features* extension leads to a measurable improvement in workflow efficiency compared to standard Git.

3.4.1. Research Prototype Implementation

To ensure the technical feasibility of implementing a prototype based on the integration concept from research question 2, a research prototype will be developed according to the proposed concept.
This section outlines the approach methodically during the implementation process, including technical and organizational decisions. These relate to the choice of programming language, development frameworks and libraries. The result is a research prototype that can be distributed as a single-file package containing feature-oriented Git commands without additional setup.
A prototype is needed to test the usability of the tool. The first version will be built and then evaluated to guide further technical requirements and improvements.

Selection of Programming Language One of the first choices was regarding the programming language to be used as it defines the ecosystem available during development. The decision was made to utilize Python as the programming language based on the following considerations.
First, Python was selected for its ability to simplify installation. As a platform-independent scripting language, it allows for easy distribution without requiring separate binaries for different operating systems.
Second, Python's extensive ecosystem offers numerous libraries that are well-suited for the project, including robust command-line and Git interaction tools.

Finally, prior familiarity with Python and its support for building command-line tools made it the most suitable choice for this prototype.

Selection of Libraries To develop the command-line interface and interact with Git repositories, `Typer`[9] and `GitPython`[10] are chosen as external libraries.
`Typer` was chosen for its decorator-based syntax for command creation and it's provision of automatic input validation, improving both usability and maintainability. It also supports nested commands, which is important for the structure of this project.
For Git interaction, `GitPython` was chosen because of its comprehensive functionality, allowing for the execution of all Git operations by design. Its flexibility in handling Git commands make it a candidate for this project.

Packaging and Distribution The project is packaged using a `pyproject.toml` configuration, based on PEP 518[11], to handle the build and distribution process. This setup allows the tool to be bundled with executable scripts, so it can be installed directly into the user's command-line environment without requiring additional steps. The tool is packaged into a `.whl` (wheel) file, which users can install via `pip`, making the process identical to installing any other Python library. The configuration also manages dependencies, ensuring that all necessary packages are automatically installed during the setup.

3.4.2. User-based Experiment Design

With the research prototype completed, the next step is to create an experiment to gather feedback on the *Git with Features* integration.
The purpose of the experiment is to evaluate the effectiveness of the Git extension in improving developer workflows. Following the guidelines outlined by [6], a user study was designed to gather both quantitative and qualitative data. The study compares key performance metrics such as task completion time, correctness of results, and user satisfaction between the standard Git workflow and the Git extension with feature support.
The experiment employs a within-subject design, as outlined by Charness, Gneezy, and Kuhn ([4]), which is particularly suitable for comparing conditions with the same group of participants. This approach reduces variability caused by individual differences, allowing for a direct comparison between the two workflows. Each participant completes the same set of tasks using both tools, with the order of

[9] `https://typer.tiangolo.com/`
[10] `https://gitpython.readthedocs.io/en/stable/`
[11] `https://peps.python.org/pep-0518/`

tool usage randomized to mitigate learning effects. This ensures that performance differences are attributable to the tools themselves rather than external factors. Data collected during the experiment includes task completion time, task accuracy, and user satisfaction. These metrics provide a basis for analyzing the impact of the Git extension on developer efficiency.

Hypotheses and Variables The following hypotheses are tested in the experiment:

1. Developers complete tasks faster using the Git extension (reduced task completion time).
2. Developers prefer the Git extension over the standard Git workflow while the usability of both systems is comparable (higher tool preference).
3. Developers make fewer mistakes with the Git extension (increased task accuracy).

The independent variables include the tool used, distinguishing between the standard Git and Git with feature support. Additionally, the developers' experience with Git and programming is assessed on a scale from 1 to 5 to account for varying levels of expertise as well as whether they already used other feature-oriented tools. Participants are also divided into two groups, which determines the order in which they use the tools, helping to reduce learning effects.
The dependent variables that reflect the outcomes of the experiment include the time it takes participants to complete each task, measured in seconds, and task accuracy, which assesses whether tasks were completed correctly. Task accuracy is manually evaluated after the experiment, with partial correctness also being noted. For example, if participants only partially complete a task such as identifying only some of the required files or providing only the name instead of both the name and email address a score of 1 is assigned to reflect partial correctness. Usability is captured through the System Usability Scale (SUS), administered immediately after each task to measure user satisfaction and ease of use, as shown by Brooke([3]). After completing all tasks, participants indicate their preferred tool, providing insights into overall tool preference.

Questionnaire The questionnaire is designed to ensure comparability between the standard Git workflow and the Git extension with feature support. The links to the questionnaires as well as an overview of the questions is given in appendix B. For both tools, participants answer identical sets of questions, focusing on similar tasks such as identifying feature authors, locating files associated with features, and comparing branches. This parallel structure ensures that the performance metrics (time, accuracy, and usability) can be directly compared across the two tools. Additionally, questions are designed to capture both objective data (e.g., task accuracy) and subjective feedback (e.g., tool preference), providing an overview of each tool's impact on the developer's experience.

Experiment Execution The experiment is conducted asynchronously, allowing participants to complete the tasks at their own pace and in their own environment. At the beginning of the experiment, participants are informed about the study's purpose and asked for their consent to participate. Participation is entirely voluntary, and participants have the right to withdraw from the study at any time. Participants are selected from a pool of students, colleagues, and acquaintances to ensure a range of experience levels with Git. Before beginning the tasks, participants are asked to complete an initial survey that collects information about their experience with Git and feature-oriented development. This information helps to account for varying levels of expertise in the data analysis.
Each participant is provided with a set of accompanying materials, including task descriptions, a Git repository with the necessary setup, and documentation explaining how to use both the standard Git workflow and the extended Git with feature support. These materials, listed in appendix A, ensure that the experiment is self-contained and that participants have all the information needed to complete the tasks without additional guidance. The asynchronous nature of the experiment allows participants to work through the tasks independently, while still adhering to the standardized conditions required for reliable data collection.

Data Collection and Analysis For the analysis, the focus will be on visualizing the collected data through graphs to illustrate the impact of the Git extension compared to the standard Git workflow. The collected data, including task completion time, correctness, and usability scores, will be broken down by independent variables such as tool usage, participant experience, and group assignment. Basic statistical measures, such as means and standard deviations, will be calculated to compare performance across these variables. Usability scores from the SUS questionnaire will also be analyzed and compared between the two conditions. Qualitative feedback from participants will be categorized and summarized to provide additional context to the quantitative results.

4. Results for User Interface and Integration Concept

Building on the methodological work outlined previously, this section presents the results of the theoretical groundwork necessary for defining the *Git with Features* integration. The presented results address the research questions 1 and 2, focusing on the development of workflows, the design of the user interface, and the conceptual integration within Git. These elements form the basis for the subsequent implementation and evaluation, providing the framework for how feature management can be incorporated into Git's existing structure.
The workflows outlined below describe how developers can interact with features using new commands and operations. The integration concept defines how feature-related data is stored using a dedicated branch, with scripts that extend Git's current command set. The results presented in this chapter serve as the foundation for the practical implementation discussed in the next chapter, where the prototype and its evaluation will be further explored in the broader context of developer workflows and usability.

4.1. Workflows using Git with Features

Feature-oriented development requires systematic management of features throughout the software lifecycle, namely their implementation, modification and tracking. Integrating feature management into version control systems can enhance traceability and supports consistent development practices [10, 17]. This section presents a series of workflows demonstrating how *Git with Features* aims to aid with these tasks. Each workflow includes command outputs and explanations of how the output is utilized. The work is based on the papers "Concepts, Operations, and Feasibility of a Projection-Based Variation Control System", "Maintaining Feature Traceability with Embedded Annotations", "Feature-oriented variability management in product line engineering", "A Common Notation and Tool Support for Embedded Feature Annotations", "Concepts of Variation Control Systems" and *Feature-Oriented Software Product Lines*([17, 7, 10, 16, 13, 1]).

4.1.1. Understanding the Feature Codebase

Before developers can start to implement features or apply updates, they need to locate all relevant locations of the feature to understand its scope and decide where to place or revise code. They may also need to retrieve information about features for decision-making and collaboration in the planning process with their team or when deciding how a new product might be structured. The following commands and outputs aim assist in any process that profits from knowledge about feature locations and their evolution.

Listing 1: Find all features present in the codebase

```
$ git feature-list
    Available features:
        - authentication
        - payment-processing
        - user-profile
```

Once the feature of interest is found based on the output of listing 1, the steps shown in listing 2 help in understanding where, when and by whom it was developed.

Listing 2: Assessing file locations for feature `authentication`

```
$ git feature-info  authentication --files
    Files associated with 'authentication':
    - src/auth/login.py
    - src/auth/logout.py
    - src/auth/utils.py
```

Listing 3: Inspecting history of a feature, Providing the commit ids

```
$ git  feature-info authentication --log
    Recent commits for 'authentication':
    - mno7890: Improved password hashing algorithm
    - abc1234: Implemented login functionality
    ...
$ git show mno7890
    commit mno7890
    Author: Bob <bob@company.com>
    Date: Mon  Apr 01 11:12:13 2024
    Improved password hashing algorithm

    The one used before had a problem.
    diff --git a/src/auth/utils.py b/src/auth/utils.py
    < diff of the commit >
```

To ask questions about a feature and decision made in its evolution, it can be helpful to know how contributed to it. Listing 4 shows that for the feature of interest, two authors with a total of 8 commits are known.

Listing 4: Inspecting authors of a feature

```
$ git feature-info  authentication --authors
    Top contributors to 'authentication':
    - Alice Smith (5 commits)
    - Bob Johnson (3 commits)
```

With these information on hand, the developers can plan and outline their work without needing to manually search and grep the commit history of the project.

4.1.2. Staging Feature Changes

After the implementations are made, the next step is to add these changes to the codebase. Modifying the codebase to implement or update features is a central activity in software development.
Although there is a conceptual distinction between modifying an existing feature and implementing a new one, this difference does not affect the workflow for the feature association. In both cases, the same steps are followed to link changes to features, as no preparatory work is needed before this association can be made. In the workflows described below, the staging process is added with an additional step of associating code changes with their corresponding features. Two variants are shown, one adding the feature while preparing the commit, the other for using existing commits. The workflow differs only slightly from the current common practice of using **git** add . && **git** commit.

Adding feature information before commiting The first variation shows how the feature information are integrated in the staging process. When this workflow is used, additional checks can be put in place to ensure no commits without feature information.

1. **Analyze and stage changes** Before staging changes, the developer lists the overview of the current changes. Based on the output, they decide which files to add to staging area. In listing 5, a possible output for the workflow is shown, including a check from the developer.

   Listing 5: Terminal-based interaction to identify files for staging

   ```
   $ git feature-status
       Unstaged changes:
           src/main.py [add]
           src/operations.py [subtract, add]
           src/utils.py [core]
   ```

2. **Annotate with Feature Information:** The developer associates the staged changes with a feature using:

Listing 6: Terminal-based interaction to stage files while adding feature information

```
$ git feature-add --feature core src/utils.py
    Feature 'core' associated with staged changes.

$ git feature-status
Unstaged changes:
    src/main.py [add]
    src/operations.py [subtract, add]

Staged changes:
    src/utils.py [core]
```

   This output confirms that the staged changes are now linked to the specified feature. Internally, the command prepares the commit message with feature information and updates the current set of staged features.

3. **Commit the Changes:** Finally, the developer commits the changes to describe their changes and add them to the history. The commit message includes the feature name, and the output summarizes the changes. This association aids in tracking feature-specific modifications in the project history.

Listing 7: Committing changes

```
$ git commit
    [main abc1234] Implemented new functionality for
       core
    Feature: core
    1 file changed, 50 insertions(+), 10 deletions(-)
```

Attempted Commit Without Feature Information

If the developer attempts to commit without associating features, the system enforces a check.The commit is aborted to ensure that all changes are linked to a feature, maintaining consistency in the project's feature traceability.

Listing 8: Aborted commit due to missing feature information

```
$ git commit
    Error: No feature associated with staged changes.
    Please use 'git feature-add' to associate changes
       with a feature.
```

Adding feature information after commiting It is also possible to assign feature information after committing. This might be useful to not interfere with workflows developers already use. Instead, after the commit process is finished, the feature association is added retroactively. Just like in the previous example, the user selects and stages changes as shown in listing 5. Even though they did not associate feature information, they will not get an error. If users prefer this workflow, they would not setup the check for existence of feature information. In listing 9, the process of adding feature information to a commit is shown.

Listing 9: Assign feature for an existing commit

```
$ git commit
    [main abc1234] Implemented new functionality for core
    Feature: core
    1 file changed, 50 insertions(+), 10 deletions(-)

$ git log -n 1 --oneline
    abc1234 (HEAD -> main) Implemented new functionality for
       core

$ git feature-commit core abc1234
```

4.1.3. Updating a Feature Across Variants

Developers work on different parts of the code, maybe even in different product variants. If customers report a bug, they might hot-fix it within the product. However, the affected feature might also be part of other variants. Hence, developers need to find all potentially affected variants and apply the same update, maybe with minor modifcations, to them. The workflow for updating a feature from another variant is as follows:

1. **List Branches with the Feature:** This output lists all branches where the feature is present, helping the developer identify the source branch for the update.

   Listing 10: Listing information about a specific feature

   ```
   $ git feature-info --feature authentication --branches
       Branches containing 'authentication':
       - main
       - develop
       - release-v1.0
       - product-c
       - product-h
   ```

   From the naming conventions of the branches, developers probably know which branches contain shipped product variants. For the purpose of the workflow, we assume that this is possible. Even if the setup were slightly different, all code versioned in Git is part of a branch if it is part of the project history.

2. **Identify Missing Commits:** The output shows commits related to the feature that are present in the 'develop' branch but missing in the current branch.

   Listing 11: Finding potential update commits for a feature

   ```
       $ git status
           On branch product-h
           nothing to commit, working tree clean
       $ git feature-diff --feature authentication --branch
           product-c
           Commits in 'product-c' not in 'product-h' for '
              authentication':
           - commit ghi9012: Fixed issue with user login
       $ git show ghi9012
           commit ghi9012
           Author: Alice <alice@company.com>
           Date: Tue Apr 02 15:09:23 24

           Fixed issue with user login

           Error occured due to \dots
           <diff>
   ```

3. **Apply the Missing Commit:** Once the interesting commits to apply in the branch are identified, users can resume to traditional Git usage by cherry-picking the commits as shown in listing 12. This applies the commit to the current branch, updating the feature accordingly. It would also be possible to partially checkout this commit in the other project to explore and test. However, this is already part of the core of Git and will not be altered by the *Git with Features* concept just yet.

   Listing 12: Applying a feature udpate with cherry-pick

   ```
   $ git cherry-pick ghi9012
       [current-branch jkl3456] Fixed issue with user
          login
       1 file changed, 5 insertions(+), 2 deletions(-)
   ```

4. **Verify the Update:** The output confirms that the feature has been updated with the new commit.

   Listing 13: Investigating feature history

   ```
   $ git feature-info  authentication --log
       Commit history for 'authentication' on 'product-
          h':
          - jkl3456: Fixed issue with user login
          - def5678: Added additional test to find
             future bugs in CI
   ```

4.2. User Interface Design

The user interface is developed based on the workflows, listing all subcommands and flags used and providing an explanation for them. The commands are divided into three categories, covering both analysis and modification tasks:

- **Retrieving feature information**: Commands that allow users to gather and view data about existing features in the project.
- **Retrieving current workspace information**: Commands used to list current changes and prepare the next feature-oriented commit.
- **Updating features**: Commands to apply new commits, update features, or merge changes from feature-specific branches.

4.2.1. Commands to read the current workspace status

When deciding which code changes are going into the next commit, developers need to have an overview of changes. This command is similar to `git status`, however it lists assigned features as well. This command can potentially be extended when using concepts of feature-anntoation[7], as they can be parsed for additional information.

Listing 14: Command for displaying the current changes in the working directory and staging area, categorized by feature.

```
$ git feature-status --help
Usage: git feature-status [OPTIONS]

Displays the current changes in the working directory and staging
   area, categorized by feature.
```

4.2.2. Commands that read general information

These commands are used to read out all feature associations stored in the history. With the help of these commands, users can compare feature logs across different branches and find potential commits that need to be propagated to other variants as well.

Listing 15: Command for listing all features present in the project.

```
$ git feature-list --help
Usage: git feature-list

Lists all features present in the project.
```

Listing 16: Command for providing detailed information about a feature, with options to display associated files, authors, branches, and feature history.

```
$ git feature-info --help
Usage: git feature-info [OPTIONS] <feature-name>

Provides detailed information about a feature, including files and
   authors.

Options:
    --files          List all files associated with the feature.
    --authors      Show authors of commits related to the feature.
    --branches    Show all branches where the feature is present.
    --log            Display the history of the feature.
```

Listing 17: Command for showing the differences between the working directory and the last commit for a specific feature.

```
$ git feature-diff --help
Usage: git feature-diff [OPTIONS] <feature-name>

Displays the differences between the working directory and the
   last commit for a specific feature.
```

Listing 18: Command for displaying features associated with lines in a file, with an option to specify a particular line number.

```
$ git feature-blame --help
Usage: git feature-blame [OPTIONS] <file>

Displays features associated with lines in a file.

Options:
    --line <number>      Show features associated with a
        specific line number.
```

4.2.3. Commands related to associating feature-information

To be able to retrieve feature information, there need to be mechanismns to update them as well. The following commands show options to add feature information either before or after creating a commit. Also, helper commands are shown that help with retroactively migrating a history. They display commits that already have or do not have a feature assigned to them yet.

Listing 19: Command for adding files related to a feature to the staging area and associating them with the feature.

```
$ git feature-add --help
Usage: git feature-add [OPTIONS] --feature <feature-name> [<files
   >...]

Adds files related to a feature to the staging area and associates
   them with the feature.

Options:
    --staged          Associate only the changes that are currently
       staged.
```

Listing 20: Command for associating feature information with an existing commit.

```
$ git feature-commit --help
Usage: git feature-commit [OPTIONS] [FEATURE_NAME] [COMMIT_ID]

 Associates feature information with an existing commit.
```

Listing 21: Command for listing commits depending on whether they have feature information associated.

```
$ git feature-commits --help
Usage: git feature-commits [OPTIONS]

 List commits depending on whether they have feature information
    associated.

 Options:
    --missing         Display all commit IDs that are not related to
        a feature yet.
```

4.3. Analyzing Git

As explained in 3, a technical analysis of Git is conducted to gain insights into the storage model and extension possibilities. Below, it is explained how Git stores information and which extension points were found, completed by the example of *git-annex* which demonstrates the feasibility of Git extensions. The next part will then outline the integration concept derived from these findings.

Note

This chapter is heavily influenced from discussions with Ansel Sermersheim who provided insight into the fundamentals of Git as well as provisioning references to the early ideas of Git that helped to thoroughly understand the mental model of the software.

4.3.1. Git Storage Model

This subsection presents the results of the technical analysis of Git's storage architecture, as outlined in section 3.3.1 .Understanding Git's internal storage structure is essential for designing the storage model of the integration concept. Git employs a content-addressable storage system, where each object, such as files, trees, and commits, is identified by a unique SHA-1 hash. This hash also serves as the filename, ensuring that objects with identical content generate the

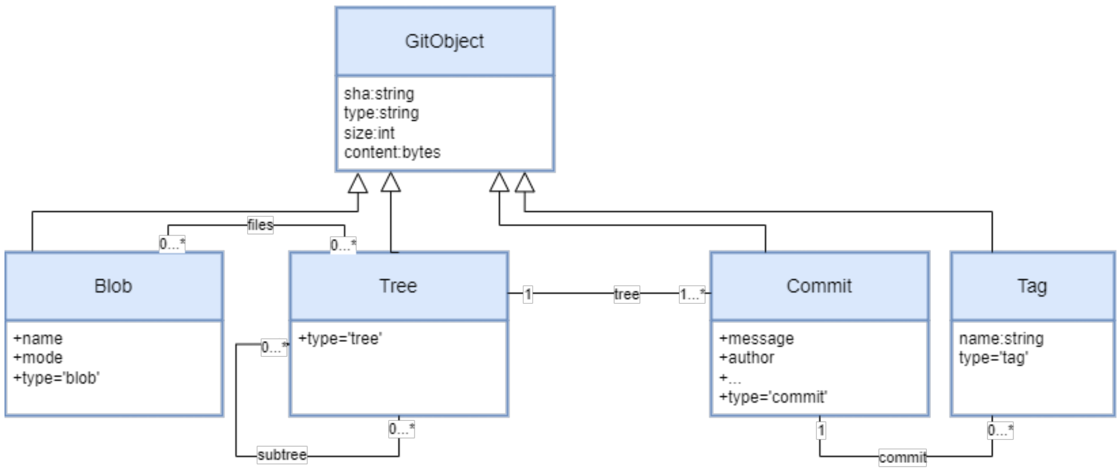


Figure 1: ERM Model of the intention to associate features with commits

same hash, thereby preventing duplication. Git organizes these objects into four types[12] : blobs (file contents), trees (directory listings), commits (pointers to trees and metadata), and tags (annotations for specific commits). As shown in figure 1, the core objects in Gitblobs, trees, and commitsare interlinked to maintain version histories.
The advantage of the SHA-filename approach is that identical content is stored only once, contributing to Git's efficiency in managing large codebases as identical content is easily identified. Additionally, objects can be easily retrieved by their hash values, streamlining access and versioning operations.

4.3.2. Options for Extension

No Fork of Git

Despite the initial idea of forking Git and creating a *Git with Feature* variant, this approach was swiftly dismissed. This decision was influenced by the observation that Git can be extended without source code modifications, and maintaining two Git installations would undermine the objective of integrating with an existing tool to facilitate the adoption of feature-centric workflows at the developer level.

As developers want to adjust their tools to their own needs, Git supports several custom configurations done in the *git config* level. This can be done both globally or specific for a certain repository. In this config, users can set aliases for new

[12]https://github.blog/open-source/git/gits-database-internals-i-packed-object-store/

commands or configure their own hook path. However, the config does not allow to add new data to retrieve information from. It is a config point useful in the setup process and when it comes down to refining the commands however. Git provides several customization options that allow developers to tailor the tool to their specific needs. To analyze which specific problem these options address to later decide what to use in the integration process, these three options are explored:

- Git Config
- Git Hooks
- Git Subcommands

Git config The Git config system allows users to set configuration variables that control Git's behavior. These settings can be scoped globally or per repository, providing flexibility in how Git operates in different contexts. Developers can create aliases for complex commands, adjust default behaviors, and configure external tools integration. While powerful for setup and refinement, Git config does not support adding new data sources for information retrieval. The settings for Git config can be done via the command line.

Git Hooks Git Hooks[13] are callback scripts triggered at specific points in the Git workflow, such as before a commit or after a merge. These hooks allow users to automate tasks like enforcing policies, validating code, or integrating with other systems. Each hook is tied to a predefined event with a specific name, such as pre-commit or post-merge, dictating when the script is executed. Only one script can be associated with each hook, so users need to customize the script if multiple tasks are required. While the scripts are flexible in their content and can execute any commands, they must be valid shell scripts or executable files compatible with the system's terminal.

Git Subcommands Git allows the addition of custom subcommands by recognizing scripts prefixed with *git-* in the system's PATH. This mechanism enables developers to create new commands that integrate seamlessly with Git's existing command set without modifying the core Git installation. These scripts can perform a wide range of tasks, from automating workflows to integrating with other tools, providing a powerful way to extend Git's functionality. Similarly to Git Hooks, the content of the script can contain any executable code for the operating system. Also, the location and naming of the script play an important role.

[13]`https://git-scm.com/book/ms/v2/Customizing-Git-Git-Hooks`

4.3.3. Case Study: git-annex

While searching for examples on how Git is extended and what ideas other developers have used, I found the tool *git-annex*[14] . git-annex is built on top of Git and also primarily command line based, hence being interesting to analyze. It enables users to handle large files by storing metadata about the file versions and their locations. By using this metadata, git-annex can track files stored across different locations, such as external drives or cloud storage services, without overloading the Git repository with large file content[15].
One of the core concepts of git-annex is its use of a separate branch, *git-annex*, which is dedicated to tracking metadata rather than file content. This branch operates independently from the main project history. Additionally, git-annex introduces custom Git commands (e.g., **git** annex add, **git** annex sync), showing how Git's subcommand framework can be extended for specific use cases[16].
Although the use case of git-annex is different from traditional version control tasks, it demonstrates how Git can be used to solve meta-data related-problems as well as that extending Git is doable.

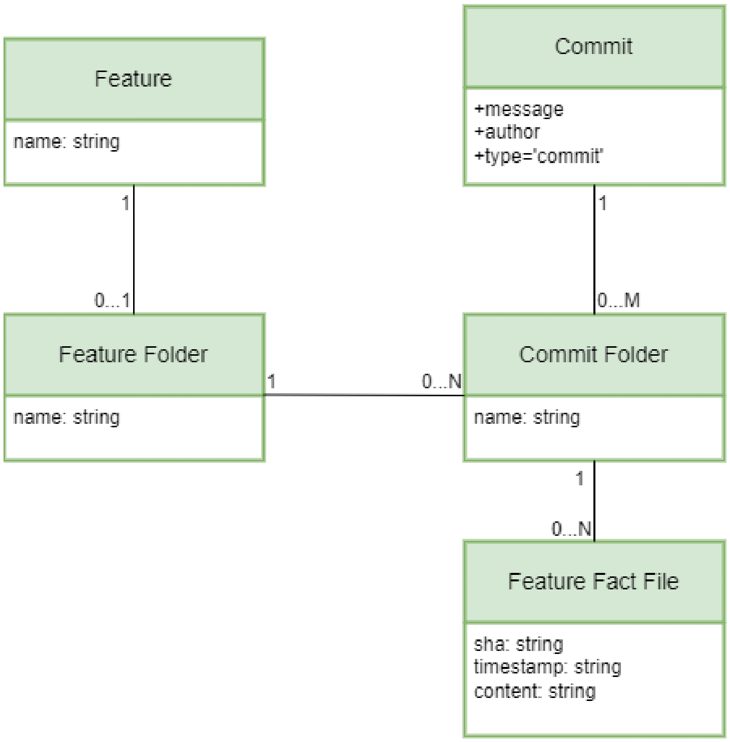
4.4. Git with Features Integration Concept

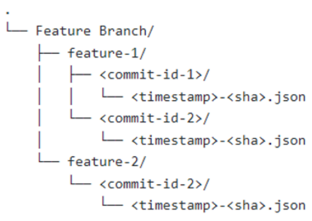
The objective of the integration concept is to propose a possible answer to 1, explaining how the *Git with Features* commands can be added and how the storage and retrieval of feature-related information can be accomplished. Building upon the analysis and methods discussed in the previous chapters, this section presents the integration concept for *Git with Features*. It aims to synthesize the findings from our earlier exploration of Git's extensibility and the workflows associated with feature management, thereby establishing a theoretical framework that will underpin the implementation detailed in the subsequent chapter.
To effectively implement the suggested commands, the following four questions grant some a prior insight. The answers already indicate the ideas of the integration concept that are explained in further detail below.
First, *what information do we store about features*? The system stores associations between features and commits, along with additional metadata such as feature descriptions or related notes that may be useful for developers.
Second, *where are they stored in Git*? The feature-related information is stored in a dedicated branch within the Git repository, separate from the main code branches. This ensures that the feature metadata does not interfere with the

[14]`https://writequit.org/articles/getting-started-with-git-annex.html`
[15]`https://en.wikipedia.org/wiki/Git-annex`
[16]`https://git-annex.branchable.com/`

standard development workflow. In this branch, a folder structure that represents a bidirectional mapping between features and commits is created.
Third, *how is the information retrieved and accessed*? A CLI tool acts as a wrapper around the actual information, providing an interface to add and retrieve feature data. For the actual implementation, Git's internal commands for parsing tree structures in other branches can be used.
Fourth, *how is information updated*? The feature metadata is updated whenever a new association between a feature and a commit is created or when additional information about a feature is added. This can be done explicitly by invoking specific commands designed for this purpose.

4.4.1. Feature Metadata Model





```
.
└── Feature Branch/
    ├── feature-1/
    │   ├── <commit-id-1>/
    │   │   └── <timestamp>-<sha>.json
    │   └── <commit-id-2>/
    │       └── <timestamp>-<sha>.json
    └── feature-2/
        └── <commit-id-2>/
            └── <timestamp>-<sha>.json
```

(a) Entity-Relationship Model illustrating the association between features and commits

(b) Folder structure of the integration concept

Figure 2: Overview of the file structure that allows to associate commits with features and features with commits

The presented metadata model, shown in figure 2, serves as the foundation for associating features with commits and provides the context required for the defined

commands to operate effectively.
Each feature is represented by their own folder as shown in figure 2b. Within this folder, all information about this feature is stored in the form of folders and files. If a text file with all commits was stored here, the feature-to-commit association would already exist.
However, the commit-to-feature association would be hard to parse. By maintaining a twofold binding between features and commits, it is ensured that for each feature, all relevant commits can be identified and vice versa. In order to achieve this, a two-level folder structure is chosen, where each individual top-level folder represents a feature and within it, subfolders contain the associated commits. This mapping enables developers to find all commits associated with a given feature and, conversely, to identify all features linked to a specific commit. By using this mapping, developers can extract further information on changes or use the metadata files to obtain feature details.
In line with Git's design, the same SHA-1 hash system is used for consistency. Each commit and feature is referenced by its unique hash or identifier, ensuring that the integrity of the system is maintained. By using Git's native hashing mechanism, collisions are avoided, and this feature management system can be integrated into Git's existing infrastructure.
The mapping is achieved by adding subfolder into the feature folders. Because the mapping is a property of the folder structure, no parsing of files is needed to check if a commit belongs to a feature. It is sufficient to search for the existence of the folder. The ERM of the resulting database can be seen in figure 2a.
The intended file-based data storage is visualized in figure 2a, where the Entity-Relationship Diagram illustrates how features and commits are connected. The structure demonstrates that starting from a specific commit, we can obtain all related information, and similarly, from a feature, we can trace all associated commits.
Merge conflicts are a challenge in version control software. They occur when Git cannot determine which changes to keep because two independent commits modify the same file. To prevent merge conflicts in the repository - particularly in code that is versioned but not directly modified by users - it is crucial to ensure that users are not burdened with resolving conflicts in files they have not worked on. This is avoided by ensuring that each change affects a unique file by assigning distinctive names and file paths.
By structuring file paths with unique identifiers such as `feature/commit/hash`, it is guaranteed that no two changes touch the same file. Git already uses hashes for file contents, so identical content yields the same hash, while differing content produces unique hashes. Since features and commits are also uniquely identified, this approach minimizes the possibility of merge conflicts. By preventing multiple

changes to the same file, Git can automatically merge changes without requiring user intervention.

Integration with the Git history Much like the approach used by *git-annex*, this model employs a dedicated branch where all feature information is stored in a structured file format, as shown in figure 2b. This separate branch ensures that feature metadata is isolated from the main codebase, preventing any interference with the standard development workflow while still being part of the repository.

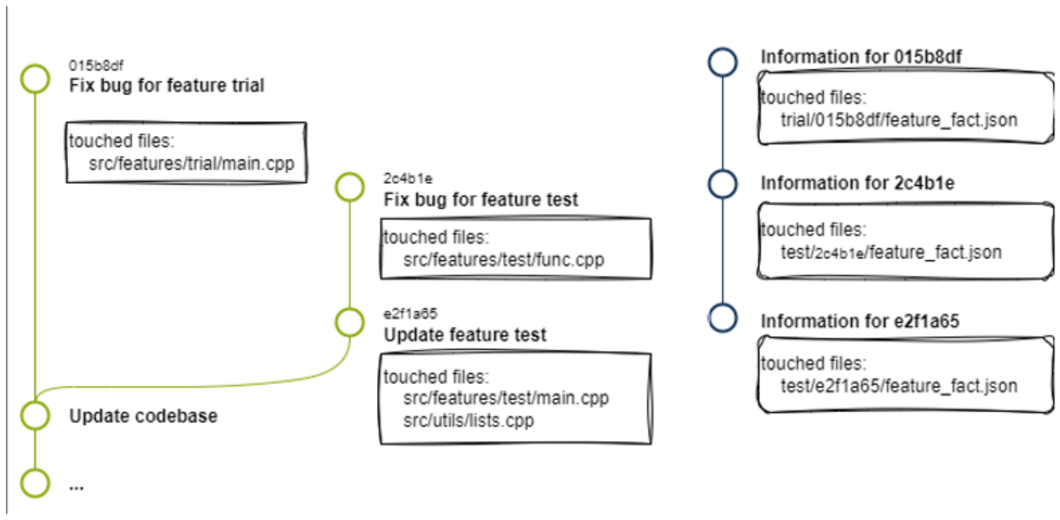


Figure 3: Exemplary history created from the integration concept

To visualize how the expected history of a repository using this concept might look, figure 3 depicts a section of such a history. The extra branch, dedicated to feature metadata, has no direct relation to the other branches containing code and contains all feature information from all branches.

4.4.2. Command Integration

To integrate the *Git with Features* commands as described in section 4.2, each command will have its own script that handles input flags and executes the desired behavior. By naming them `git-<subcommand>`, and installing them in the PATH, Git will find these new subcommands and integrate them into its CLI as described in section 4.3.2. The scripts themselves can be written in any language that can be interpreted and executed by the terminal in use.

5. Results from Git with Features Evaluation

Based on theoretical integration concept from section 4.4, a *Git with Features* prototype can be build and used to evaluate the benefits the integration brings for managing features. This section presents the results of the evaluation, focusing on both the implementation outcomes and the user feedback gathered through the study. Key metrics and insights from the qualitative and quantitative data collected during the user survey are discussed, with the methodical approach detailed in section 3.4.

5.1. Prototype Implementation

Building on the integration concept outlined in Section 4.4, the next step was to implement a research prototype according to the specification. The result of the implementation is a repository[17] containing the source code and build files, which allows the prototype to be built as a distributable Python package. In the following I will lay out the structure of the code base and the implemented subcommands based on the list in section 4.2.

5.1.1. Implemented CLI Commands

The tool implements a variety of commands that interact with Git and manage feature-based development. These commands are encapsulated in multiple Python scripts located across different directories. In contrast to the layed out commands in the previous section, here are the actual commands implemented. Currently, the support for feature annotations to help suggest the correct features and potential helpful commands around that scenario are still missing.

- `feature-status` This command displays the current status of files in the working directory, showing staged, unstaged, and untracked changes along with their associated features. Files without associated features will be highlighted, and suggestions for adding features will be provided.
- `feature-info` Displays feature information for the entire git repository, offering insight into the structure and details of each feature. This command is broken down into subcommands to provide more specific information:
  - `feature` This subcommand shows feature-specific information:
    - `authors` Lists all authors who have contributed to the specific feature, making it easier to track collaboration on that feature.

[17]`https://github.com/isselab/feature-oriented-git`

    * `files` Lists all files associated with the feature, providing a clear view of which parts of the codebase the feature affects.
    * `branches` Shows all branches where the feature is present, helping developers understand feature distribution across different branches.
    * `updatable` Checks if the feature has updates available on other branches and lists options for updating. This is useful for keeping features in sync across multiple development branches.
    * `branch` Used with the `-updatable` flag to specify which branch to check for updates.
  - `all` Lists all available features in the project, providing an overview of the feature set currently managed by the repository.

- `feature-add` This command associates files in the repository with specific features, making it easier to track changes. It can be used in different ways:
  - `by-add` Allows adding files to a feature based on user input:
    * `all-files` Adds all files in the repository to the selected feature.
    * `selected-files` Adds only specific, user-selected files to the feature.
  - `from-staged` Associates staged files with features, enabling feature tracking based on the current state of the Git staging area.
    * `feature-names` Allows the user to specify feature names. If no names are provided, the command attempts to associate the files with features that are already linked to them.

- `feature-blame` Find out which features are associated with a file. *This command is not fully implemented yet. Line-based blames are not fully functional yet*
  - `filename` The file to inspect.

- `feature-commit` Associate an existing commit with one or more features after the commit was created.

- `feature-commits` Use to manage commits with feature associations. *This command was added to help with migrating an existing history to feature-based*
  - `list` Shows all commits that are already associated with at least one feature

  - `missing` List all commits without a feature

- `feature-pre-commit` Checks if all staged changes are properly associated with features. Returns an error if any issues are found. *This command is intended to be used within Git Hooks to perform pre-commit chekcs*

- `feature-commit-msg` Generates feature information for the commit message. *Experimental command, intended to be used with a Git Commit Message Hook and Template to add feature information in Message Trailers*

Each of these commands supports the tool's goal of enhancing feature-based development in Git.

5.1.2. Installation Setup

The prototype is packaged as a Python `.wheel` package, and the installation is handled by the `pyproject.toml` file. The `pyproject.toml` file, which controls the installation process, is shown below. The command for building is simple as the build-in Python build tools are able to parse the file python –m build pyproject.toml. This file defines the necessary metadata for building the package and managing dependencies. Installation is done using a package manager, e.g.`pip`, and users must ensure that the package's scripts are included in their system's `$PATH` to make the commands executable within Git environments. This should be the default behavior during installation, however, user feedback has shown that this sometimes does not work as intended.

5.1.3. Prototype Architecture and Project Structure

The *Git with Features* prototype is structured with a clear separation between the CLI and the internal logic. The CLI commands, built using Typer, allow developers to manage features through Git-integrated commands without directly handling low-level operations. The internal logic handles tasks such as reading and writing feature data, managing the repository state, and interacting with the metadata branch. This separation ensures that the UI is responsible for structuring the output, while the core operations are managed by internal functions.
The internal logic consists of two main types of functions: low-level state management and specific information retrieval. The state management functions deal with setting up the repository context and accessing the metadata branch, ensuring that the necessary Git environment is correctly configured. On the other hand, the specific information retrieval functions target particular datasets, such as feature-specific commit history or branch information. These functions are

combined and reused to implement higher-level user commands, making it easy to extend the system by adding new commands that operate in different contexts. The prototype also includes example Git hooks that can be used to enforce specific commit policies. For example, one hook ensures that commits are only possible if feature-related information has been added, helping to maintain consistency in feature tracking across the repository. The hooks automate checks during the commit process, preventing actions that could lead to incomplete or untracked feature data.

5.1.4. Resolved Challenges during Implementation

While the integration concept provided a general framework for design decisions, certain challenges emerged during implementation that required resolution. The challenges and implemented solutions are shown below. Apart from the help flag issue, most of them could be addressed within the scope of this thesis. Solutions were designed to ensure cross-platform compatibility and requiring as little user interaction as possible. However, not all of these challenges could be addressed finally.

- **Path Management**: Managing file paths consistently across Windows and Unix systems was essential to ensure the Git hooks and scripts functioned properly. Hardcoded paths would potentially lead to errors in different environments. The solution was to use Git's internal path resolution functions in combination with Python's Path library to avoid OS-specific code, making paths valid on any platform.
- **Git Worktrees**: Worktrees present a non-standard `.git` directory structure, which complicated the construction of the Repository object needed for the GitPython library to execute Git commands. Using the Git `rev-parse` command, I could dynamically resolve the base path of the repository given the current folder belonged to a repository, allowing the tool to find the correct Git repository directory even in worktree setups. This ensures compatibility with a variety of user configurations.
- **Automating Metadata Branch Setup**: Normally, only the current branch is updated when working with Git. To support feature sharing across all branches, I automated the creation and management of a metadata branch. While this reduced user errors, synchronization issues still exist, particularly when users lack write permissions or work across multiple machines.
- **Custom Help System**: Typers text-based help system did not align with Git's use of man-pages, which made it impossible to display help as expected.

Without the knowledge of how to automatically generate and integrate man-pages from within my installation process, I implemented a workaround where help text is displayed by default when no arguments are provided. While this works, a more complete solution involving a potential man-pages integration remains to be found.

- **Reading and Updating Feature Information without Checkout**: Git typically requires a clean workspace to switch branches, and switching branches disrupts user workflows when they have uncommitted changes. To avoid this, I used `ls-tree` to read files from the metadata branch without needing to switch to the branch. For writing, I leveraged Git's `fast-import` command, allowing me to create commits without switching branches, ensuring smoother user operations. This command is intended to migrate from SVN to Git, but the mechanism allows to create arbitrary commits from a given file, following a certain format.

5.2. Evaluation Results

This section presents the principal findings of the evaluation of Git with Features, based on the raw data provided in appendix A in section C. As outlined in section 3.4.2, in this section I analyze whether the claims made are valid. Furthermore, I present the data necessary to answer research question 3 by conducting a comprehensive analysis of the obtained user data and determining whether Git with Features fulfils the claims made.

Recruited Participants Participants were recruited from a diverse pool of developers, comprising half RUB students and staff, and half contacts from outside the institution. Among the 15 participants (7 in Group A and 8 in Group B), a small number had prior experience with feature-oriented tools. Specifically, only 4 participants across both groups had such experience. Figure 4 shows the number of participants who rated themselves in each category to get an overview of the distribution of prior knowledge in the groups.
When comparing general programming experience and Git knowledge across the groups, Group B displayed a slightly higher level of experience on average, with more participants rating themselves as having strong Git knowledge. Group A, which started with traditional Git, had a more balanced distribution of skill levels, including participants with moderate to high Git expertise. The initial group assignment ensured no prior advantage for any group due to asynchronous participation.

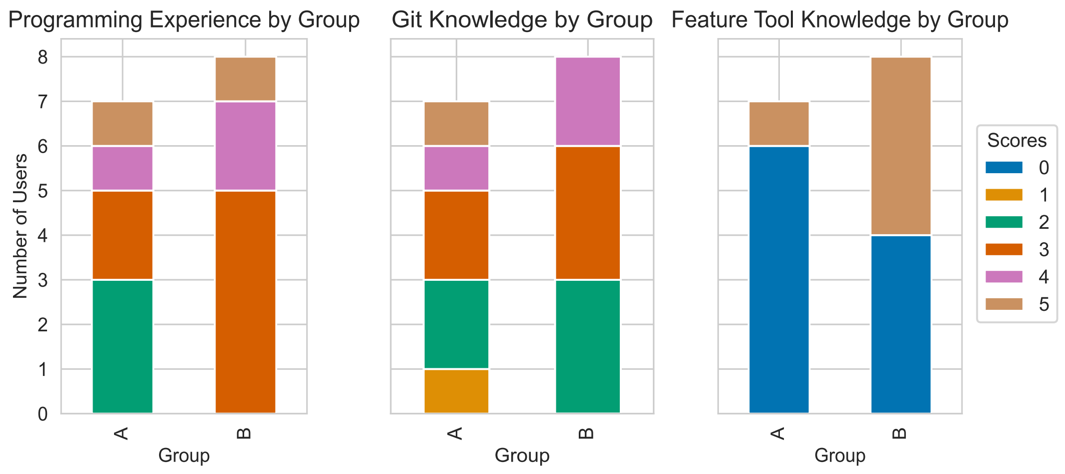


Figure 4: Previous experience of participants, sorted by group. Group A had 7, Group B 8 participants. For the tool knowledge, users answered either `Yes` or `No`, where Yes is displayed as a value of 5 and No as a value of 0

5.2.1. Correctness of Task Completion

When analyzing the correctness of task completions across different setups, *Git with Features* consistently led to higher correctness rates than traditional Git. Across all tasks, participants using *Git with Features* produced more correct results, particularly for complex, feature-related tasks where tracking features across branches was necessary. The difference in performance is most noticeable in tasks requiring feature management, where the integrated tool significantly improved accuracy.

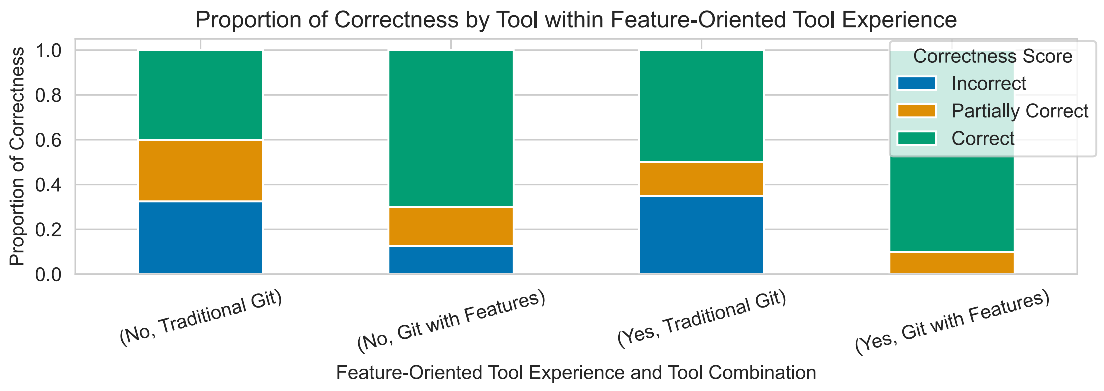


Figure 5: Overview of correctly solved tasks showing the influence of tool experience and current setup

The figures 14, 13, 12 and 15 show the influence of the various independent variables (programming experience, group membership, Git knowledge, experience with feature-oriented tools) on the accuracy of the task solutions.
The influence of tool use on the correctness of the tasks is shown independently of the respective values of the independent variables. By normalizing the proportions to 1, one can clearly see that, regardless of the participants' previous experience, the accuracy of the solutions is higher if the tool *Git with Features* was used. This is shown by higher proportions of completely correct answers in the corresponding diagrams. The results indicate that the use of *Git with Features* has a positive effect on the correctness of the task solutions, regardless of the users' previous experience.

5.2.2. SUS Evaluation

Even though the SUS is not a metric I want to improve, I still wanted to compare the ergonomics of the tool. When looking at the general overview, the usability seems to pretty comparable as shown in figure 7. However, when looking at the variable-based showings in figure 6, there is no consistent trends. Just like with the tool preference, users of Group A perceived traditional Git more useable whereas Group B found the new tool more usable. For users with a medium experience level, the new tool was also perceived as more usable then Git.

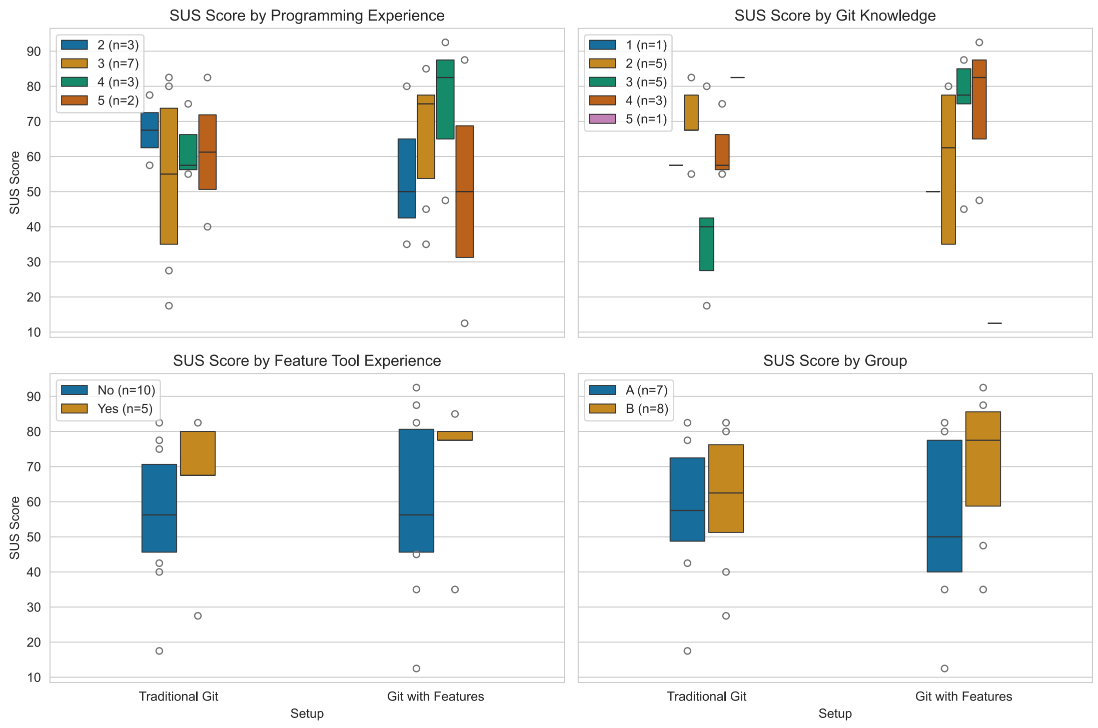


Figure 6: SUS Score evaluation shown with respect to the users previous expertise

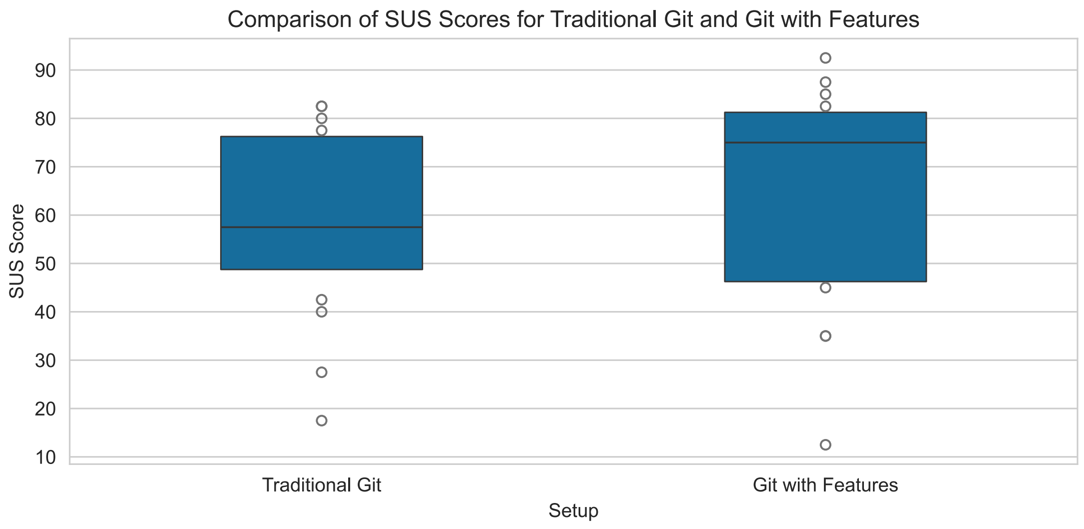


Figure 7: SUS Score evaluation shown with respect to the users previous expertise

5.2.3. Time Evaluation

In figure 8, the average time needed to complete the tasks with the different tools is shown. As the questions are structured such that the effort needed and information searched for is comparable, the times can be compared.
The timing shows that the average time needed is not dependent on the tool per se. For easy information retrieval tasks like finding authors and files associated with a certain format in the commit message, standard Git seems to be on average faster. However, when feature evolution between branches needs to be compared, the *Git with Features* Tool is faster again. Adding new information to the history currently takes longer for the new tool.

5.2.4. Prefered Setup

At the end the users were asked to state their tool preference on a scale of 1-5, where 1 indicated strong favor of traditional Git and 5 strong favor of *Git with Features*. In figure 16 and figure 10, the results are visualized. The largest group of users chose *Git with Features*, as 7 out of 15 participants opted for a strong or very strong preference of the tool. However, for 5 people, the traditional Git setup is more preferable. When looking at figure 11a, it is interesting to see that Group B, who started with the tool extension, has a larger preference rate than Group A who started with traditional Git. In figures 18, 17 and and 11b, the influence of the other independent variables on the selection can be seen. As there is only one participant in the most extreme knowledge groups, these are omitted. For the other groups, higher Git knowledge seems to correspond with a higher extension preference as well. A similar trend can be seen in the influence of programming expertise. Users already familiar with feature tools also prefer the tool more than those without tool expertise,
To summarize, a general tendency to use the *Git with Features* approach becomes apparent. When dividing the data based on the previous Git knowledge of users, apart from the outliers for very strong and very weak Git knowledge, a higher Git knowledge seems to indicate a stronger preference for the new setup.
To gain a better understanding of the reasoning, looking into the comments explaining the users choice can be interesting. From the comments, the following reasoning in favor and against the tool were given. The full list of comments, sorted by preference, is listed in listing 22.

- **Plain Git preferred** Users that already know how to use Git and feel unfamiliar with the feature-oriented workflows don't see a need to switch. Interface Inconsistencies make the tool less usable then the intended integration concept.

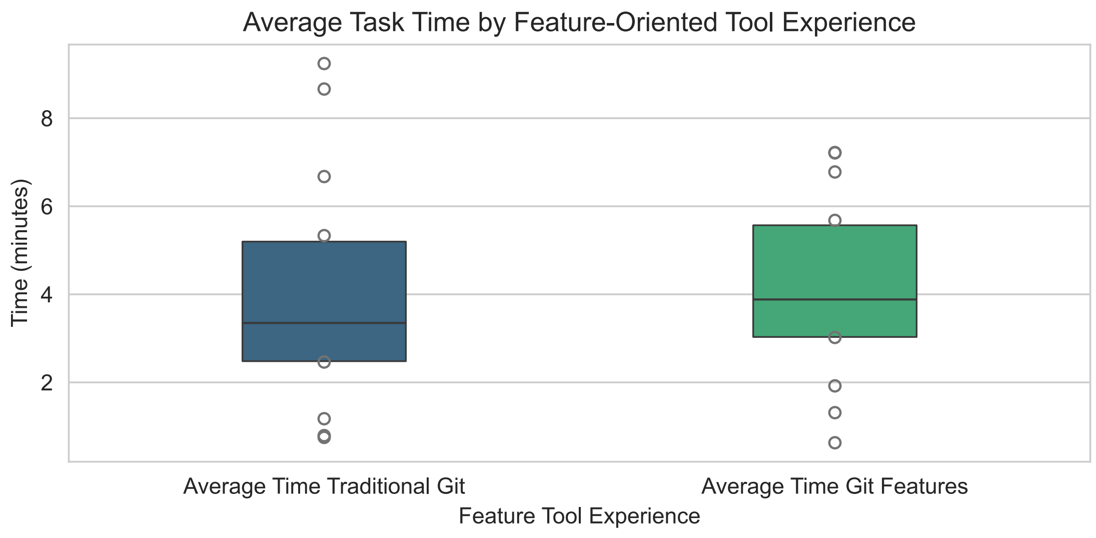


Figure 8: Overview of average times needed, broken down by the independent variables of this experiments

- **Indecided** Users without Git skills or without a strong preference but a general openness for third-party tools chose the neutral option.
- **Git with Features preferred** The improved organization and clarity of management are reasons in favor of using the tool. Especially retrieving information was perceived easier than with plain Git as a lot of output parsing becomes unnecessary. Here, users also liked the concise commands. The tool was deemed interesting for complex setups.

Users against Git with Features prefer traditional Git due to familiarity, perceived interface inconsistencies, or reliance on third-party tools for simplicity. On the other hand, users in favor appreciate its clarity in feature management, ease of information retrieval, and concise commands, particularly for complex projects.

Listing 22: Overview of user feedback, structured by selected tool preference

```
    Preference 1 (Strongly prefer Traditional Git):
["I already know a little about git and learning how to use
   git with features would take some time. I would spare my
   time and just do it the old way since I don't got why to
   use git with features anyways.", 'Inconsistencies in user
   interface, poor integration with builtin git commands.']
```

```
Preference 2 (Moderately prefer Traditional Git):
["I don't think I can give a correct statement on this yet
   because my skills with plain git are not yet sufficient",
   nan, nan]

Preference 3 (Neutral):
['I normally use third party tools to simplify the process
   and not use the command line.', 'For simple projects, I
   would use plain Git. For complex projects, I would use Git
    with features, but I still need to learn how to use them
   better.', 'usability']

Preference 4 (Moderately prefer Git with Features):
[nan, nan, 'Git with features makes the repository clearer,
   especially when working with features. Information related
    to a specific feature are much easier to find']

Preference 5 (Strongly prefer Git with Features):
[nan, 'Gathering information about features in plain git
   requires a lot of manual parsing', 'It makes it easier to
   make changes to features and retrieve information about
   features history in a project ', 'I prefer not using grep
   simulator, and git log tends to either give way too much
   or not enough information. While the git feature extension
    thingy is concise, and the commands are very self
   explanatory (aka give me a cheat sheet and i will be fine)
   . And i cant mess up the commit message.']
```

5.2.5. Key Results and Interpretation

The evaluation demonstrates that Git with Features improves workflow efficiency, particularly in the management of feature-related tasks, such as the tracking of features across different branches. In comparison to traditional Git, the tool provides superior organization and simplifies complex feature management tasks. The evaluation largely corroborates the assertions posited in the hypothesis. The implementation of Git with Features resulted in a reduction in task completion time and an enhancement in the capacity to oversee features across branches. However, the tool currently offers limited advantages in the context of non-feature-based tasks. Approximately 25% of users indicated a strong preference for the new configuration, which aligns with the hypothesis that the tool

would be favored by those engaged in feature-centric workflows.
An unplanned but beneficial outcome of the tool's development is its ability to retroactively associate commits with features, allowing existing repositories to be adapted without altering the original Git history. Since the tool only adds references and does not modify the objects, regular Git operations remain unaffected, with the only addition being the metadata branch. This functionality allows to adopt the tool for existing projects, making it useful for a broader range of projects.
These results support the hypothesis that Git with Features enhances task efficiency in feature-centric development without sacrificing usability. While the tool is particularly beneficial for complex feature management, traditional Git remains competitive for simpler tasks. In summary, Git with Features meets its primary objective of improving feature-centric task efficiency, providing valuable insights for further developments in version control systems.
The order of using the tool seems to make a huge difference, at least greater then expected, in the setup. It might be due to the fact that the second part of the experiment was less nice for users so their mood was worse. Also, it might be due to the fact that some users didn't know either approach and switching from one learned setup to another was perceived difficult as well. When looking at the reasoning for the choice, lack of experience was a thing.

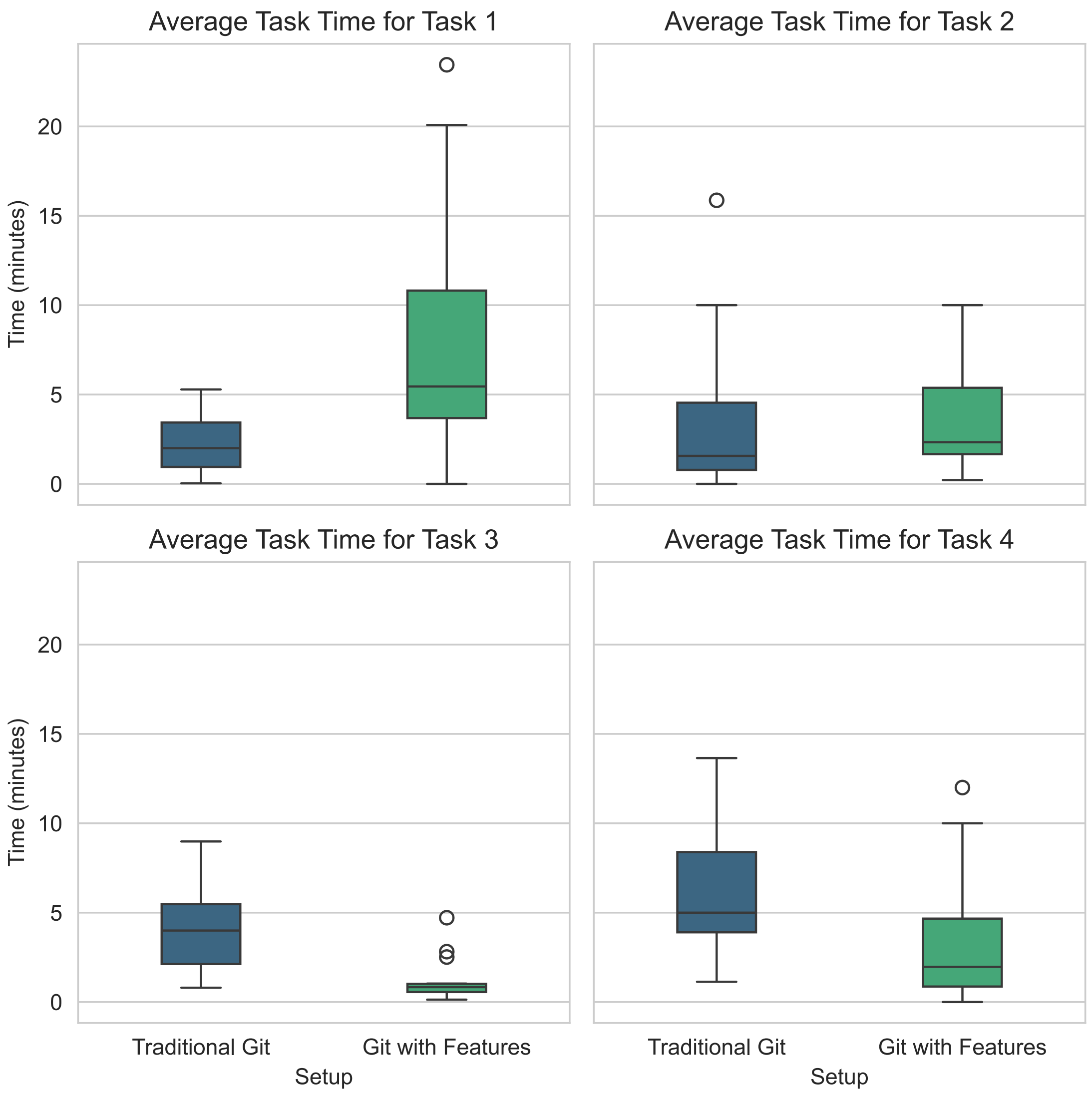


Figure 9: Overview of average times needed, broken down by the independent variables of this experiments

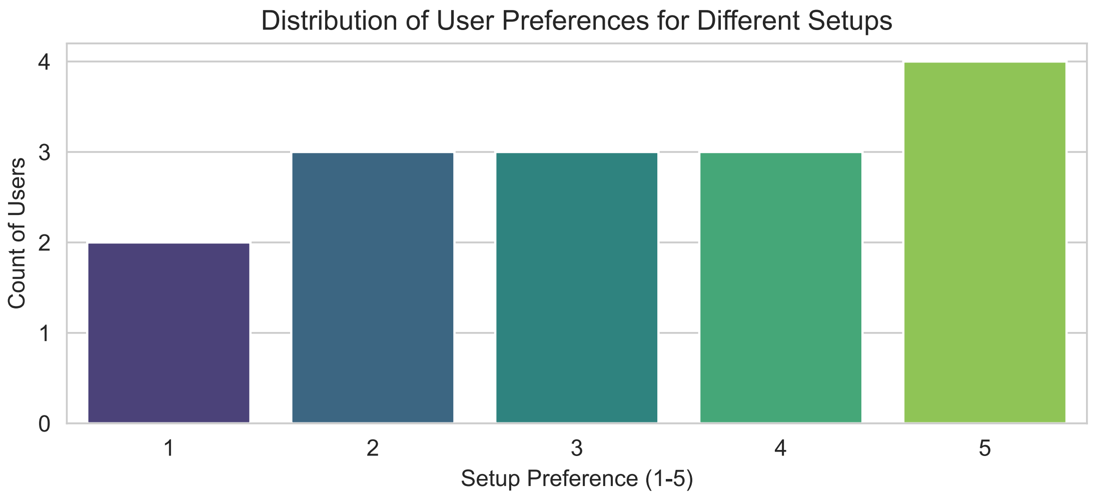
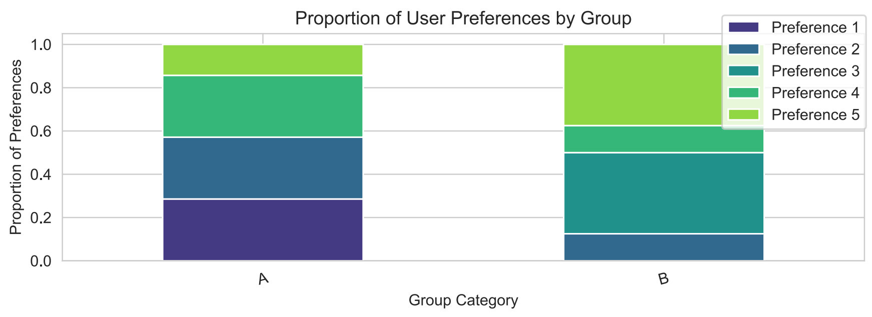
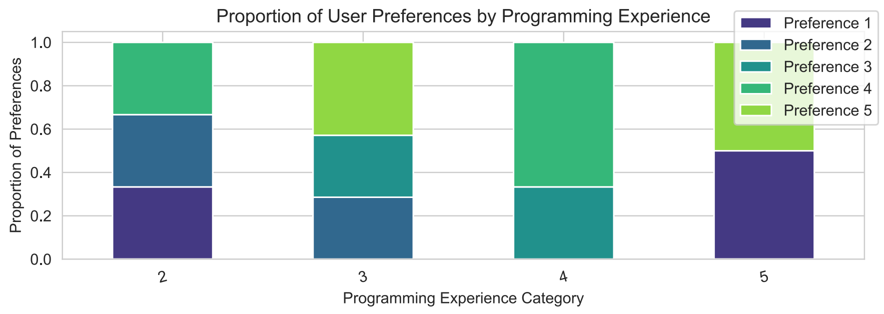


Figure 10: Distribution of user preferences across the different experience levels and groups. Group A had 7 participants, and Group B had 8 participants. The displayed values represent the distribution for each preference category.



(a) Proportion of user preferences by group. This figure illustrates the distribution of preferences for Git with Features versus Traditional Git, separated by Group A and Group B.



(b) Proportion of user preferences by programming experience. This figure shows how users' programming experience influences their preference for either Git with Features or Traditional Git.

6. Discussion

With the results of the previous analysis in mind, it is now possible to address the research questions. In this chapter, the results are discussed, including threats to validity, and answer the questions first posed in the introduction of the work. The objective of this thesis was to provide and evaluate a feature-oriented Git extension that allows to obtain feature-information from the entire project to in turn support maintenance tasks for features.
Below, the three research questions regarding the requirements and conceptual approach as well as the measured improvement outcomes are answered. Finally, potential threats to validity that could compromise the found outcomes are discussed.

6.1. Answering RQ 1

Research Question 1: What commands should a command-line interface provide to facilitate feature-oriented workflows that assist developers in feature management, while reflecting the syntax typically used in Git commands?

The list of obtained commands can be found in section 4.2. They are derived from the workflow descriptions of section 3.2, which features in total 5 workflows. The needed commands cover all the needs to find and update information on features both globally in the project as well as in regards to the changes in the current worksapce.

- `git feature-info`
- `git feature-status`
- `git feature-add`
- `git feature-commit`

To refine the output information, the commands are designed to receive flags that adjust the information being displayed.
Additionally **git** feature−commits helps to review the history as a tool to ensure that all commits have feature assignments.
These commands facilitate workflows focused on finding features in files, searching for updates in other branches, and providing an overall project overview. The idea was to have commands that are as close to the current git commands as possible.

6.2. Answering RQ 2

Research Question 2: How can Git be extended to support feature-oriented development, and what integration concept enables this while ensuring that the commands feel native to the user?

From the extension possibilities explored in section 4.3.2, the addition of custom CLI commands by adding scripts following the naming convention was selected for the integration of the new commands. Git detects these scripts and integrates them into its command list. These scripts are arbitrary and must be executable on the system.
The integration concept stores feature information in a separate branch where all feature-related data from all branches is stored. Each feature has its own folder, and within these folders are subfolders for each commit associated with the feature. This file structure can be read by Git and used to find associations between code changes and features. Together with the user interface from research question 1, this forms the complete integration concept.
The biggest challenge when the users add to branches via side effects is to ensure no merge conflicts occur. this is solved by never modifying results but rather adding new files when there are new information. this way, the files have different names and won't create merge conflicts.

6.3. Answering RQ 3

Research Question 3: How does the *Git with Features* prototype enhance the management and traceability of features in software development?

The evaluation was conducted with the research prototype obtained from the last research question's work. The Python-based command-line tool was installed as a package and automatically added the defined scripts mentioned in research question 2. This tool, along with an exemplary calculator project repository, was given to participants who were asked to solve four tasks using both traditional Git and Git with Features. The results were evaluated using metrics The analysis shows that the usability of the Git extension is comparable to traditional Git, while improving task completion time. Additionally, users preferred the extension. Thus, the extension helps developers work more efficiently with features, even when their history is ideally structured.

6.4. Threats to Validity

In this section, we outline potential threats to the validity of the study's results, categorized into internal, external, and construct validity.

Internal Validity A key internal threat stems from the controlled environment used during the evaluation. The test repository featured a well-documented and consistent Git history, allowing for an idealized scenario where feature-related changes could be easily identified. This does not reflect the complexity of real-world projects, where inconsistent commit histories and tangled feature associations are common. In these cases, the tool might face challenges not observed during the controlled testing, particularly when identifying features linked to ambiguous commits or resolving historical technical debt.
Another internal threat is that some features of the tool, particularly those enhancing user experience (e.g., comprehensive help functions or onboarding processes), were not fully implemented. As such, the tool's usability might have been underestimated, and the study does not account for the additional benefits these features could offer in real-world usage.

External Validity The artificial repository used in this study limits the generalizability of the findings to other settings. Real-world projects often have more complex branching structures, inconsistent commit messages, and varying levels of documentation. While the tool performed well in the simplified environment, it remains unclear how it would handle repositories with substantial technical debt, incomplete documentation, or inconsistent feature tracking. Thus, further evaluation is needed to assess its effectiveness across different project types, particularly in large-scale, multi-team environments.
Additionally, the study's focus on a single type of project (a calculator application) restricts its application to other domains. Larger projects with more intricate dependencies or those managed by multiple teams could present challenges that were not covered in the scope of this evaluation.

Construct Validity The construct validity of the study is impacted by the evaluation metrics and tasks chosen for the user study. The tasks focused on basic feature management operations, which do not cover the full complexity of real-world scenarios, such as handling commits with intertwined features or navigating highly complex branching strategies. As a result, the study does not fully capture the potential limitations or performance issues the tool might face in more complicated workflows.

Furthermore, while the use of a separate metadata branch was intended to preserve the integrity of the existing commit history, this approach may not suit all development workflows, particularly in projects where maintaining additional branches could lead to increased complexity. Alternative approaches, such as embedding feature information directly within commits or using an external database, could have yielded different results and should be explored in future work.

7. Conclusion

In conclusion, the presented *Git with Features* extension allows developers to adhere to feature-oriented implementation workflows. To achieve this, the tool implements the integration concept derived in research question 2 which explains what kind of feature information and how it is stored. With this concept, the workflows described for research question 1 are enabled by providing the derived interface. The study shows promising results regarding the preference of users for the tool over traditional Git, indicating a speed-up in locating features and examining update states without compromising on usability. However, the documentation of usage needs to be improved for the feature-commit process to ensure the intended workflow becomes obvious to users.

Limitations

While the development of the *Git with Features* tool has shown promising results, the following limitations affect the tool's current usability and scope.

1. **Incomplete Implementation of Key Features** The planned support for feature annotations and the feature-blame functionality were not fully implemented, primarily due to time constraints. As a result of this incomplete integration, the prototype currently exhibits a subset of the intended functionality originally envisioned for the tool. However, with additional time, it would be feasible to incorporate these features.

2. **Python Dependent** Currently, developers are required to have a new version of Python installed in order to utilize the tool, as the distributed artifacts are not yet compiled for the respective operating systems. Given that Python is a dependency for numerous tools and is frequently employed for prototyping, the majority of developers have already installed it on their machines. Should the necessity arise, Python projects can be compiled to standalone distributables; however, the effort was deemed unnecessary in the absence of an immediate requirement.

Future Work

Based on the limitations and results presented here, this section outlines a number of promising directions I see for future research and development.

In-Depth Research Across All Steps In keeping with the principles of the design science research approach, future work should focus on iteratively refining each development step by conducting deeper, more detailed investigations of both the concept and the implementation approach. This iterative cycle of design, implementation, and evaluation can help to incrementally improve the effectiveness and usability of the tool by refining the workflows, user interface, and integration options with existing structures.

- **User Interface Design and Workflows**: Some feedback suggests that the naming of the subcommands is not as in line with the traditional Git commands as it could be. Additionally, options to suggest feature mappings based on comments found in code or based on previous pattern is not explored in depth yet. Work on both can help to improve the user experience by offering maximum guidance for the additional step of associating changes with features as possible. Also, deriving new variants by selecting features might be supportable through the command line interface. However, the expected behavior of these commands might be highly context sensitive and needs more investigations on the user level.
- **Experimental Methodology**: Revising the experimental design with clearer instructions, demonstrations, and iterative testing would improve the reliability of the improvements found from the studies. Future experiments should incorporate structured feedback at each stage to evaluate the tool under optimal conditions, allowing for adjustments based on real-world usage. Long-term studies could also provide valuable insights by measuring benefits over extended periods, particularly considering the overhead of managing and transitioning to new workflows. Additionally, focusing on a subgroup already familiar with feature-oriented development functions may yield more relevant and targeted feedback.
- **Iterating the prototype**: Developing comprehensive guides iteratively, including case studies and practical examples, would support new users in understanding how to effectively integrate the tool into their workflows. Each iteration should refine the documentation, incorporating user feedback to enhance clarity and usefulness.

Expanding the Meta Concept The concept of linking metadata with Git commits is not limited to feature management. Future research could explore how this approach might be adapted to other domains, such as project management, bug tracking, or release planning. By associating commits with specific tasks, issues, or broader concepts, developers could gain more detailed insights into their projects, improving workflow management and traceability. Expanding this meta concept could lead to a more generalized framework within Git for handling diverse types of metadata and solve similar problems related to information not being tracked alongside the code.

Personal Thoughts

I am satisfied with the results of the project, even though not all implementation goals were achieved. In particular, I was pleased with the successful evaluation, which confirmed the concept and showed that the tool has practical applications. One unexpectedly positive finding was how easy it was to add feature information to existing Git histories. It was possible to retrofit an existing history within a day, as long as the commit messages were well structured and the scope of the changes was clear. This shows the potential of the approach for wider application, even in existing projects. Organizing my work and keeping a structured schedule helped me make progress despite unexpected challenges. The exchange and critical discussions with other developers also gave me new perspectives on the topic, which proved to be important for success as they outlined potential problems and lead to further investigations on my part.
Overall, the tool shows potential for extracting information from Git histories. It may encourage developers to write more thoughtful commits by showing how this data can be used to gain insight into a project's feature structure. The project has not only increased my own knowledge of Git, but also provides a foundation for future development. I hope the tool will actually be used and can prove its usefulness in practice.

References

[1] Sven Apel et al. *Feature-Oriented Software Product Lines*. Jan. 2013. ISBN: 978-3-642-37520-0. DOI: 10.1007/978-3-642-37521-7.

[2] Thorsten Berger et al. “Software Evolution in Time and Space: Unifying Version and Variability Management (Dagstuhl seminar 19191)”. In: *Dagstuhl Reports*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2019.

[3] John Brooke. “SUS: A quick and dirty usability scale”. In: *Usability evaluation in industry* 189 (1996), pp. 4–7.

[4] Gary Charness, Uri Gneezy, and Michael A. Kuhn. “Experimental methods: Between-subject and within-subject design”. In: *Journal of Economic Behavior & Organization* 81.1 (2012), pp. 1–8. ISSN: 0167-2681. DOI: https://doi.org/10.1016/j.jebo.2011.08.009. URL: https://www.sciencedirect.com/science/article/pii/S0167268111002289.

[5] Reidar Conradi and Bernhard Westfechtel. “Version models for software configuration management”. In: *ACM Computing Surveys (CSUR)* 30.2 (1998), pp. 232–282. DOI: 10.1145/280277.280280.

[6] Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl. “Reporting experiments in software engineering”. In: *Guide to advanced empirical software engineering* (2008), pp. 201–228.

[7] Wenbin Ji et al. “Maintaining Feature Traceability with Embedded Annotations”. In: *19th International Software Product Line Conference (SPLC)*. 2015.

[8] Barbara Kitchenham and Stuart Charters. “Guidelines for performing Systematic Literature Reviews in Software Engineering”. In: 2 (Jan. 2007).

[9] Eric Knauss. *Constructive Master’s Thesis Work in Industry: Guidelines for Applying Design Science Research*. 2021. arXiv: 2012.04966 [cs.SE]. URL: https://arxiv.org/abs/2012.04966.

[10] Jaejoon Lee and Dirk Muthig. “Feature-oriented variability management in product line engineering”. In: *Commun. ACM* 49.12 (Dec. 2006), pp. 55–59. ISSN: 0001-0782. DOI: 10.1145/1183236.1183266. URL: https://doi.org/10.1145/1183236.1183266.

[11] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. “A Classification of Variation Control Systems”. In: *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE)*. Vancouver, Canada: ACM, 2017, pp. 14–25. DOI: 10.1145/3136040.3136054.

[12] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. “A classification of variation control systems”. In: *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2017. Vancouver, BC, Canada: Association for Computing Machinery, 2017, pp. 49–62. ISBN: 9781450355247. DOI: 10.1145/3136040.3136054. URL: https://doi.org/10.1145/3136040.3136054.

[13] Lukas Linsbauer et al. “Concepts of Variation Control Systems”. In: *Journal of Systems and Software* 171 (2021), p. 110796.

[14] Wardah Mahmood et al. *Seamless Variability Management With the Virtual Platform*. 2021. arXiv: 2103.00437 [cs.SE].

[15] Johan Martinson et al. “HAnS: IDE-Based Editing Support for Embedded Feature Annotations”. In: *25th ACM International Systems and Software Product Line Conference (SPLC), Tools Track*. 2021.

[16] Tobias Schwarz, Wardah Mahmood, and Thorsten Berger. “A Common Notation and Tool Support for Embedded Feature Annotations”. In: *Proceedings of the 24th ACM International Systems and Software Product Line Conference - Volume B*. SPLC ’20. Montreal, QC, Canada: Association for Computing Machinery, 2020, pp. 5–8. ISBN: 9781450375702. DOI: 10.1145/3382026.3431253. URL: https://doi.org/10.1145/3382026.3431253.

[17] Stefan Stnciulescu et al. “Concepts, Operations, and Feasibility of a Projection-Based Variation Control System”. In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2016, pp. 323–333. DOI: 10.1109/ICSME.2016.88.

A. Experiment Material

Git Feature Command Documentation

This document provides an overview of the available commands and options for the `git feature` tool. It allows you to associate features with commits and files in a Git repository, manage feature metadata, and inspect feature-related information.

Overview

The `git feature` tool provides commands to:

- Stage files and associate them with features.
- Associate features with existing commits.
- Display feature associations for files.
- Show the status of features in the working directory.
- Inspect features by displaying information about their authors, files, history etc.

Adding Feature Information

git feature-status

Displays the current status of files in the working directory, showing staged, unstaged, and untracked changes along with their associated features.
This command is intended to be used with the following commands to add feature information while staging files. It can also be used to aid in the selection of files for the next commit.

Usage:
```
git feature-status
```

git feature-add

The `git feature-add` command is used to stage files and associate them with features. Alternatively, you can use the git feature-commit workflow to add features after creating a commit.

git feature-add by-add

Stages specific files or all files and associates them with the provided features.

Usage:

```
git feature-add by-add [FEATURE_NAMES...] [OPTIONS]
```

Arguments:

- `FEATURE_NAMES`: A list of feature names to associate with the staged files.

Options:

- `-a`, `--all`: Stage all tracked changes and associate them with features.
- `-f`, `--files [FILES...]`: Specify a list of files to stage and associate with features.

Examples:
```
git feature-add by-add FeatureA --all

git feature-add by-add FeatureX --files src/main.py --files
src/utils.py
```

*git feature-add from-staged

Associates features with the currently staged files. Currently, it derives the feature list from the previously selected features for those files and is not fully implemented yet.

git feature-commit

Description:

Associates feature information with an existing commit. This is useful when you want to add feature metadata to a commit after it has been created. After executing this command, the output of feature-info will be updated as well.

Usage:

```
git feature-commit [COMMIT_ID] [OPTIONS]
```

Arguments:

- `COMMIT_ID`: The ID of the commit to associate the feature information with.

Options:

- `--features [FEATURES...]`: Manually specify feature names. Each feature name should be prefixed with `--features`. If this option is provided, staged feature information will be ignored. Otherwise, the program will try to derive the information.

Example:

Associate the last commit with a new feature (can be repeated multiple times to add multiple features)

```
git log -1 // outputs the latest commit info
git feature-commit <commit-id> --features FeatureX
```

git feature-commits

Description:

Provides information about commits that are either already associated with a feature or have not been associated with any feature yet. This is especially useful when transitioning to feature-based workflows or when tracing commits for specific features. It helps developers identify which commits have feature metadata associated with them and which do not, ensuring that no commit is left untracked.

Usage:

```
git feature-commits missing
git feature-commits list
```

Reading Feature Information

git feature-blame

Displays features associated with file lines. Optionally specify a line or a range of lines.

Usage:

```
git feature-blame [FILENAME]
```

Arguments:

- `FILENAME`: The file to display feature blame for.

Examples:

```
git feature-blame src/main.py
```

git feature-info

The `git feature-info` command provides information about features in the repository.

git feature info all

Description:

Lists all available features in the project. Use these names with `git feature info feature [FEATURE_NAME]` to inspect details.

Usage:

```
git feature-info all
```

git feature-info <feature>

Description:

Shows detailed information about a specific feature, including associated authors, files, branches, and update status.

Usage:

```
git feature-info feature [FEATURE_NAME] [OPTIONS]
```

Arguments:

- `FEATURE_NAME`: The name of the feature to inspect.

Options:

- `--authors`: Show all authors who contributed to this feature.
- `--files`: List all files associated with the feature.
- `--branches`: Show all branches where the feature is present.
- `--updatable`: Check if the feature has updates available on other branches and list the update options.
- `--branch [BRANCH]`: Specify a branch for checking updates (used with `--updatable`).

Examples:

Display all information about a feature:

```
git feature-info feature FeatureX --authors --files --branches
```

Check if a feature is updatable on another branch:

```
git feature-info feature FeatureY --updatable --branch develop
```

Note: This documentation assumes that you have the `git feature` tool installed and configured in your Git environment. The commands and options provided are designed to help you manage and inspect feature associations within your Git repository effectively.

Git With Features - Reference for Traditional Git Tasks

Git Introduction

Git is a distributed version control system that allows developers to track changes in their codebase over time. It helps teams collaborate by providing tools to branch, merge, and revert code while maintaining a detailed history of all changes. With Git, you can experiment with new features, fix bugs, or roll back to previous versions without affecting the main project. Common commands like `git add`, `git commit`, and `git push` form the foundation of this workflow.

Feature-Oriented Development Introduction

Feature-Oriented Development (FOD) organizes software development around features—units of functionality that meet specific requirements. In FOD, code assets like classes or methods are associated with the features they implement, making it easier to track and manage changes. By linking code to features, developers can modify or extend functionality without impacting unrelated parts of the system. This approach enables better modularity, flexibility, and maintainability in large, evolving codebases.

Conventional Commit Format

The Conventional Commit format can help to have a clean and consistent Git history. In the experiment, all commits follow this format to make it easier to read the history. Therefore, all commit messages look as follows:.

```
<type>(feature): <description>

<explanation>
```

- **Type**: The purpose of the commit, such as `feat` (feature), `fix` (bug fix), or `chore` (routine tasks).
- **Feature**: The name of the feature being affected.
- **Description**: A short and clear description of the change.
- **Explanation**: Optional, used to describe in more detail what was changed and why

Helpful Git Commands for History Exploration

All git commands have their own **documentation**, which can be found when appending the --help flag to any command.

git log

Displays the commit history.
Useful flags, that can also be combined:

- --oneline: Shows a simplified commit history, with one line per commit.
- --all: Displays the commit history for all branches.
- --author=<name>: Filter commits by author.
- --help: Shows all options for the log command
- --format='%aN': Format takes multiple values, this would show the authors
- --grep="<string>": Searches for a string in commit messages.
- <flags> -- <filename>: Displays the commit history of a specific file
- <branchA>..<branchB>: Shows the commits between two branches.

git branch

Lists all branches.

- --all: Outputs all branches, including remote ones

git checkout <branch/commit>

Switches to the specified branch or commit.

B. User Study - Questions

Previous experience The links to the Google Forms for Group A and Group B can be found in the footnote[18]

1. How would you rate your programming experience?
2. How would you rate your Git knowledge?
3. Have you ever worked with any feature-oriented tools before?
4. If yes, which tools have you used, and for what purpose?

Traditional Git Questions

1. If you have questions about feature subtract, whom do you ask? Write down name and e-mail address.
2. Which files were touched to implement feature add? Put in the filenames.
3. Compare the branches fix-input and main. Which features exist on main but not in fix-input?
4. Commit your changes! What commit message did you choose?
5. System Usability Scale Questions 1

Git with Features Questions

1. How many authors does the feature add have?
2. Which files are touched to implement feature subtract?
3. Which branch has the newest commit for feature core? And is the main branch up-to-date regarding that feature?
4. What commit message did you choose?
5. System Usability Scale Questions 2

Final Decision Quesionts

1. Which setup do you prefer?
2. What are your reasons for this choice?

[18]Group A: `https://forms.gle/zPRmm33hwwR2KMpJ9` Group B: `https://forms.gle/ATzcPRqR47p9cXZx5`

C. Experiment Data

	By continuing, you agree to participate in this study.	How would you rate your programming experience?	How would you rate your Git knowledge?	Have you ever worked with any feature-oriented tools before?
0	I agree to the use of my anonymized data for research purposes.	2	2	No
1	I agree to the use of my anonymized data for research purposes.	3	3	No
2	I agree to the use of my anonymized data for research purposes.	2	2	Yes
3	I agree to the use of my anonymized data for research purposes.	2	1	No
4	I agree to the use of my anonymized data for research purposes.	4	4	No
5	I agree to the use of my anonymized data for research purposes.	5	5	No
6	I agree to the use of my anonymized data for research purposes.	3	3	No
7	I agree to the use of my anonymized data for research purposes.	5	3	No
8	I agree to the use of my anonymized data for research purposes.	4	4	No
9	I agree to the use of my anonymized data for research purposes.	4	4	No
10	I agree to the use of my anonymized data for research purposes.	3	2	Yes
11	I agree to the use of my anonymized data for research purposes.	3	3	Yes
12	I agree to the use of my anonymized data for research purposes.	3	3	Yes
13	I agree to the use of my anonymized data for research purposes.	3	2	Yes
14	I agree to the use of my anonymized data for research purposes.	3	2	No

	If you have questions about feature subtract, whom do you ask? Write down name and e-mail address.	Q1/4: How long did it take?	Which files were touched to implement feature add? Put in the filenames.	Q2/4: How long did it take?	Compare the branches fix-input and main. Which features exist on main but not in fix-input?	Q3/4: How long did it take?	Commit your changes! What commit message did you choose?
0	Tabea, tabea.roethemeyer@rub.de	0 days 00:00:02	input.py, main.py, operastions.py	0 days 00:01:34	subtract	0 days 00:00:56	introduce feature divide to calculator
1	Tabea Rthemeyer <tabea.roethemeyer@gruppe.ai>	0 days 00:05:17	git-with-features-experiment				
src operations.py	0 days 00:15:52	REDACTED; TOO	0 days 00:07:59	feat(add): add division operation and division function - Add the division operation to the operations module - Add the division function			
2	Jane Doe (jane.doe@example.com)	0 days 00:03:40	0000	0 days 00:10:00	docs, subtract	0 days 00:04:45	"feat(divide): Implemented divide function"
3	0000	0 days 00:02:35	src/main.py	0 days 00:02:47	The installation guide in README.md exists in main but not in fix-input.	0 days 00:03:31	feat(division): add divide function
4	jane.doe@example.com	0 days 00:03:45	operations.py main.py	0 days 00:02:18	docs substract	0 days 00:03:06	feat(divide): Add feature for division
5	Jane Doe <jane.doe@example.com>	0 days 00:00:20	input.py, main.py, operations.py	0 days 00:00:34	subtract	0 days 00:00:48	feat(divide): add divide function

6	Jane Doe <jane.doe@example.com>	0 days 00:02:15	0000	0 days 00:04:05	none	0 days 00:06:30	git commit -m "feat(divide): add divide function - Implemented a divide function that divides two numbers and checks for division by zero. - Added the divide function to the operations list to make it accessible for use." [new f8fed78] feat(divide): add divide function 1 file changed, 18 insertions(+), 3 deletions(-)
7	Jane Doe	0 days 00:00:15	4	0 days 00:05:00	keine Ahnung wie ich das rausfinde	0 days 00:05:00	feat(divide): integrate divide functionality in program - user can now select divide operation - output is the division of the parameters *Tests none *Limitations all
8	Jane Doe <jane.doe@example.com>	0 days 00:00:32	src/operations.py, src/main.py	0 days 00:00:24	subtract, docs	0 days 00:01:06	feat(divide): add divide operation
9	Jane Doe	0 days 00:05:05	src/main.py src/-operations.py	0 days 00:00:31	all exist	0 days 00:05:07	git commit -am "added devide"
10	Jane Doe	0 days 00:02:00	main.py operations.py	0 days 00:01:13	0000	0 days 00:04:00	add divide function to operations list
11	author Jane doe?	0 days 00:02:00	src/operations.py src/main.py	0 days 00:01:00	0000	0 days 00:02:00	feat(divide): <added a division op>

12	Jane Doe <jane.doe@example.com>	0 days 00:03:12	main, operations	0 days 00:08:49	docs	0 days 00:08:59	feat(divide): added division feature with non-null divisor check
13	Jane Doe <jane.doe@example.com>	0 days 00:02:00	no idea	0 days 00:00:00	0000	0 days 00:05:50	git commit -a -m "added divide feature, i know good commit messages look different"
14	Jane Doe, jane.doe@example.com	0 days 00:01:22	src/operations.py	0 days 00:01:15	subtract	0 days 00:02:15	added division

	How did you like traditional Git? [I think that I would like to use this system frequently]	How did you like traditional Git? [I found the system unnecessarily complex.]	How did you like traditional Git? [I thought the system was easy to use.]	How did you like traditional Git? [I think that I would need the support of a technical person to be able to use this system.]	How did you like traditional Git? [I found the various functions in this system were well integrated.]	How did you like traditional Git? [I thought there was too much inconsistency in this system.]	How did you like traditional Git? [I would imagine that most people would learn to use this system very quickly.]	How did you like traditional Git? [I found the system very cumbersome to use.]	How did you like traditional Git? [I felt very confident using the system.]	How did you like traditional Git? [I needed to learn a lot of things before I could get going with this system.]
0	4	1	4	3	5	1	3	3	5	2
1	4	2	2	4	3	2	2	4	2	4
2	5	1	4	2	3	2	4	2	3	5
3	4	3	2	3	4	1	3	3	2	2
4	4	2	3	2	3	2	1	3	4	4
5	5	1	5	1	4	4	2	1	5	1
6	2	5	4	4	2	5	1	5	2	5
7	1	1	1	3	4	3	2	4	2	3
8	3	4	4	1	4	1	4	2	4	1
9	3	3	3	1	3	2	3	3	3	3
10	5	1	3	2	4	2	4	2	5	1
11	3	2	4	1	5	1	4	2	4	2
12	4	4	2	2	1	4	1	5	2	4
13	5	2	3	3	3	2	4	2	4	3
14	4	3	2	2	3	3	3	2	3	3

	How long did it take you?	How many authors does the feature add have?	Q1/4: How long did it take you?	Which files are touched to implement feature subtract?	Q2/4: How long did it take you?	Which branch has the newest commit for feature core? And is the main branch up-to-date regarding that feature?	Q3/4: How long did it take you?	What commit message do you choose?
0	0 days 00:04:22	0000	0 days 00:10:00	operations.py	0 days 00:02:31	0000	0 days 00:12:00	0000 / dont know how to associate the feature with a new commit?
1	0 days 00:20:05	2	0 days 00:07:44	src/operations.py src/main.py	0 days 00:01:02	0000	0 days 00:00:00	git feature-commit 9a53dd4cb27dc2e83813b –features MultiplyFeature Selecting features from cli parameters [’MultiplyFeature’] Features [’MultiplyFeature’] assigned to 9a53dd4cb27dc2e83813b Fetching feature-metadata from origin Pushing feature-metadata to origin Warning: Feature Updates could not be pushed to remote
2	0 days 00:00:00	2	0 days 00:05:45	2	0 days 00:01:00	Yes, the main branch is up to date	0 days 00:05:20	Added feature multiply
3	0 days 00:05:27	1	0 days 00:01:56	0000	0 days 00:02:49	main, Yes	0 days 00:01:58	Added multiply operation
4	0 days 00:05:13	2	0 days 00:01:50	main.py operations.py	0 days 00:00:40	main yes	0 days 00:04:50	Add operation for multiplication

5	0 days 00:00:35	2	0 days 00:06:31	main.py, operations.py	0 days 00:00:37	0000	0 days 00:10:00	Added multiply feature
6	0 days 00:15:20	Tabea Röthemeyer Tabea	0 days 00:01:30	src/operations.py src/main.py	0 days 00:00:30	0000, yes	0 days 00:04:30	git commit -m "feat(multiply): add multiply function Implemented the multiply function and added it to the operations list."

7	0 days 00:03:00	2	0 days 00:01:00	2	0 days 00:00:30	main	0 days 00:00:45	maxclerkwell@SURFnTE /git-with-features-experiment-ssh (sb/multiply)> vim src/operations.py maxclerkwell@SURFnTERF /git-with-features-experiment-ssh (sb/multiply)> git add src/operations.py maxclerkwell@SURFnTERF /git-with-features-experiment-ssh (sb/multiply)> git commit -m "Adds multiplication" Running feature-pre-commit Fetching new feature-metadata Error: No features associated with the staged changes. Error: Pre-commit checks failed. maxclerkwell@SURFnTERF /git-with-features-experiment-ssh (sb/multiply) [1]> git feature-add by-add multiplication -f src/operations.py Staging selected files: ['src/operations.py'] Staging files Staged file: src/operations.py

8	0 days 00:01:49	2	0 days 00:00:13	src/main.py, src/-operations.py	0 days 00:00:08	main, it’s up to date	0 days 00:00:21	Added new operation multiply
9	0 days 00:04:36	2	0 days 00:02:20	2	0 days 00:00:18	main. JA	0 days 00:00:27	git feature-commit b6d8e6c283582e1bae1af –features multiply
10	0 days 00:10:00	2	0 days 00:05:00	src/operations.py, src/main.py	0 days 00:04:43	main and the main branch is up to date	0 days 00:03:00	Multiply function
11	0 days 00:23:27	2	0 days 00:01:00	src/main.py src/-operations.py	0 days 00:00:50	Main. Yes	0 days 00:01:50	add feature multiplication
12	0 days 00:09:45	2 authors	0 days 00:02:32	2 files, src/-main.py and src/operations.py	0 days 00:00:46	main has the newest update	0 days 00:02:29	"added multiplication feature"
13	0 days 00:10:46	2	0 days 00:02:22	src/main.py, src/-operations.py	0 days 00:00:50	main, no	0 days 00:01:40	$ git commit -a -m "added multiply feature" $ git feature-commit HEAD –features multiply
14	0 days 00:10:52	2	0 days 00:01:52	src/main.py; src/-operations.py	0 days 00:00:56	main; yes	0 days 00:00:59	added multiply

	How did you like Git with Features? [I think that I would like to use this system frequently]	How did you like Git with Features? [I found the system unnecessarily complex.]	How did you like Git with Features? [I thought the system was easy to use.]	How did you like Git with Features? [I think that I would need the support of a technical person to be able to use this system.]	How did you like Git with Features? [I found the various functions in this system were well integrated.]	How did you like Git with Features? [I thought there was too much inconsistency in this system.]	How did you like Git with Features? [I would imagine that most people would learn to use this system very quickly.]	How did you like Git with Features? [I found the system very cumbersome to use.]	How did you like Git with Features? [I felt very confident using the system.]	How did you like Git with Features? [I needed to learn a lot of things before I could get going with this system.]
0	2	3	2	4	3	1	2	4	2	5
1	4	2	2	5	3	2	3	3	2	4
2	4	1	5	2	3	1	5	1	4	4
3	3	3	2	4	3	2	4	3	3	3
4	4	2	4	1	5	1	3	2	4	1
5	2	5	2	5	1	4	2	4	1	5
6	4	2	3	2	4	1	4	2	4	2
7	3	1	4	1	5	1	4	1	4	1
8	3	1	5	1	5	1	4	1	5	1
9	3	4	4	3	4	2	2	4	2	3
10	4	1	3	2	4	1	4	2	4	2
11	5	3	4	1	5	2	4	3	4	2
12	4	1	3	1	5	1	4	2	5	2
13	2	3	2	2	2	3	3	5	1	3
14	4	2	3	3	5	3	4	3	3	3

	Task 1 - Traditional Git - Correctness	Task 2 - Traditional Git - Correctness	Task 3 - Traditional Git - Correctness	Task 4 - Traditional Git - Correctness	Task 1 - Git with Features - Correctness	Task 2 - Git with Features - Correctness	Task 3 - Git with Features - Correctness	Task 4 - Git with Features - Correctness
0	0	1	1	0	0	1	0	0
1	0	1	0	1	2	1	0	2
2	2	0	2	2	2	1	2	2
3	0	1	0	2	2	0	2	2
4	2	2	2	2	2	2	2	2
5	2	1	1	2	2	2	1	2
6	2	0	0	2	2	2	1	2
7	1	0	0	2	2	1	2	1
8	2	2	2	2	2	2	2	2
9	1	2	0	0	2	1	2	2
10	1	2	0	0	2	2	2	2
11	1	2	0	2	2	2	2	2
12	2	2	1	2	2	2	2	2
13	2	0	0	0	2	2	1	2
14	2	1	1	0	2	2	2	2

	Which setup do you prefer?
0	1
1	2
2	4
3	2
4	4
5	1
6	5
7	5
8	4
9	3
10	3
11	5
12	5
13	3
14	2

D. Additional Figures from the User Study Evaluation

Figures regarding correctness of tasks

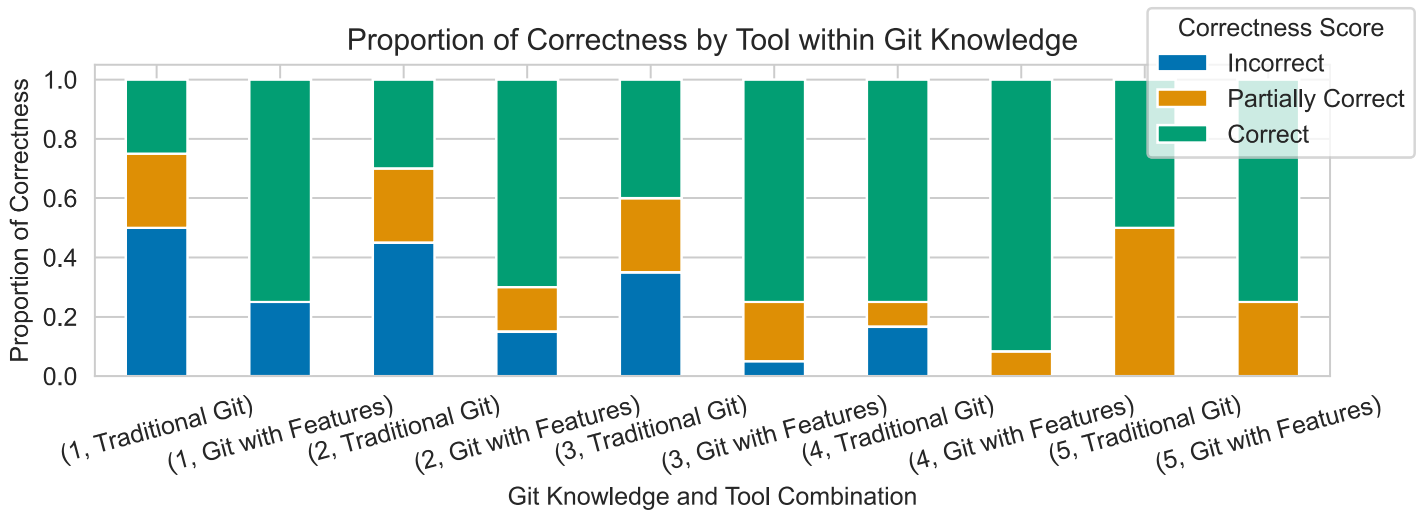


Figure 12: Overview of task correctness based on Git knowledge and tool setup, showing the proportion of correct and partially correct answers for each group.

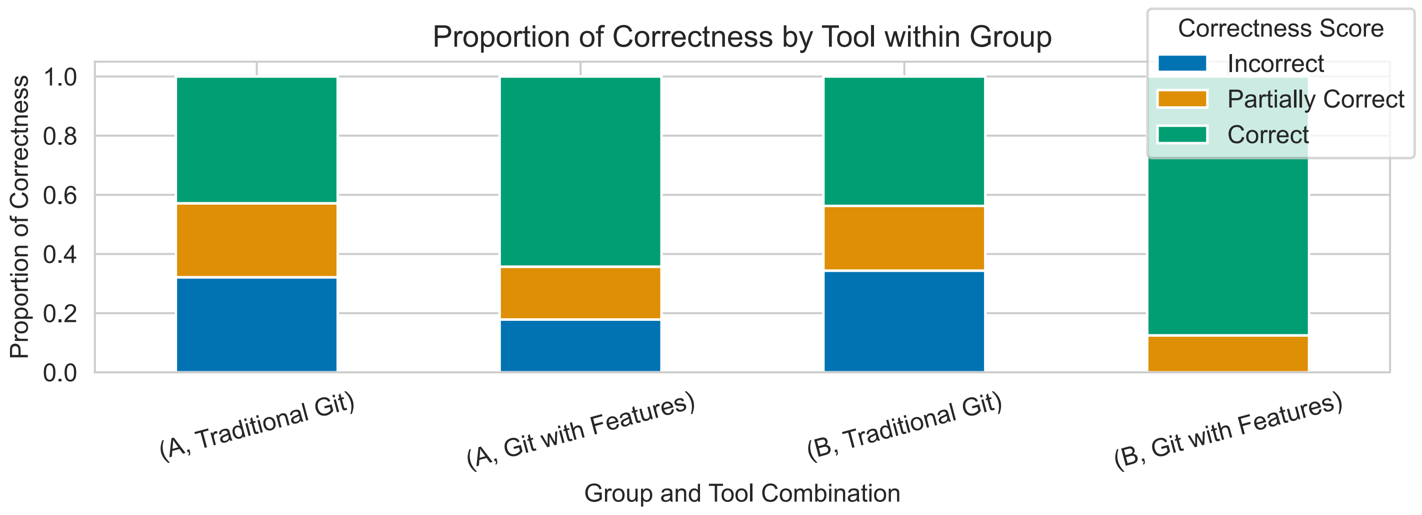


Figure 13: Comparison of correctness between groups A and B, highlighting how the use of traditional Git and Git with Features affects task performance.

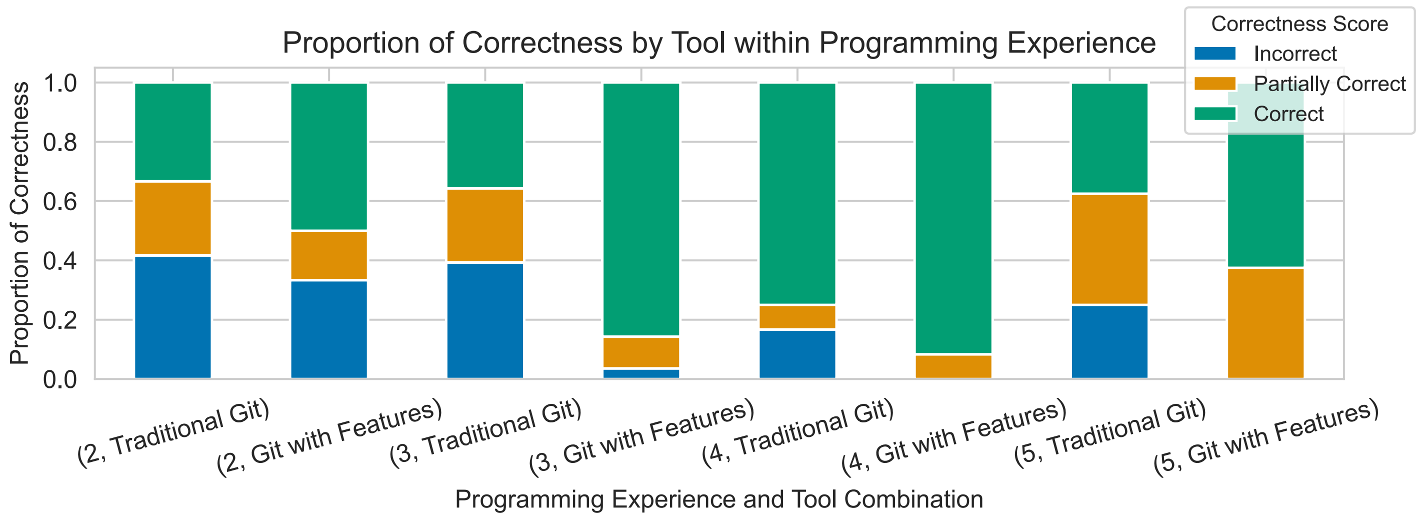


Figure 14: Correctness of task solutions in relation to programming experience and tool setup, illustrating how experience levels influence accuracy with different tools.

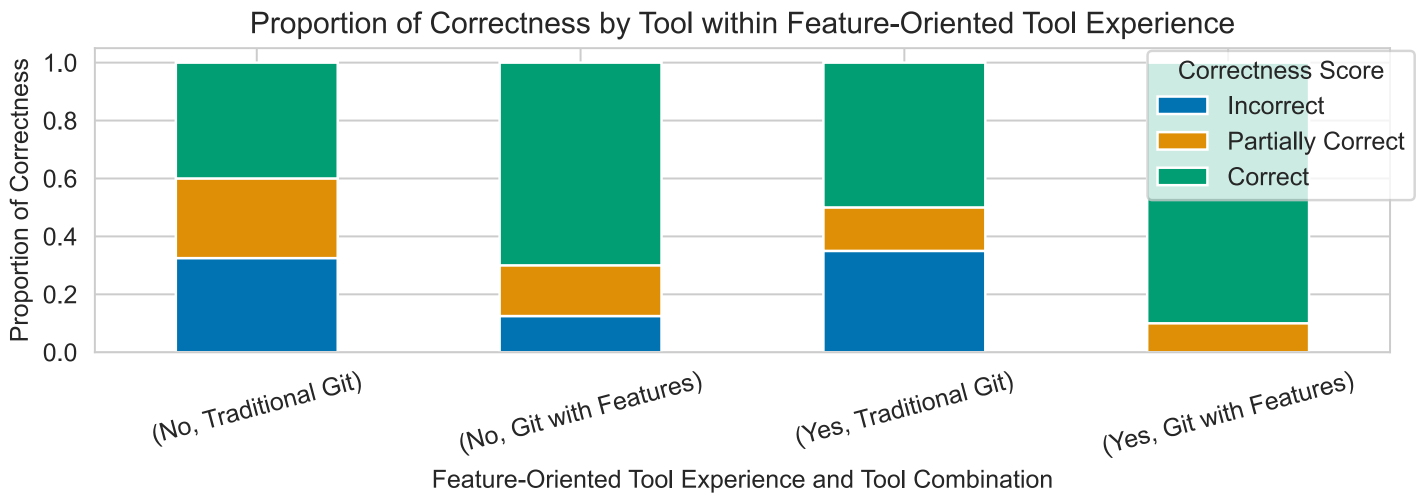


Figure 15: Impact of feature-oriented tool experience on task correctness, comparing outcomes between traditional Git and Git with Features.

Figures regarding tool preference

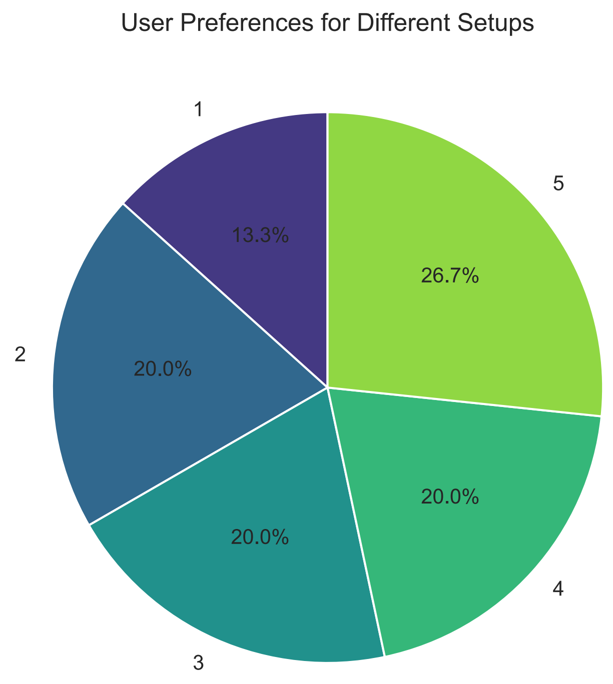


Figure 16: Pie chart showing the proportion of user preferences between Group A and Group B. This chart illustrates the breakdown of participants' tool preferences between the two groups.

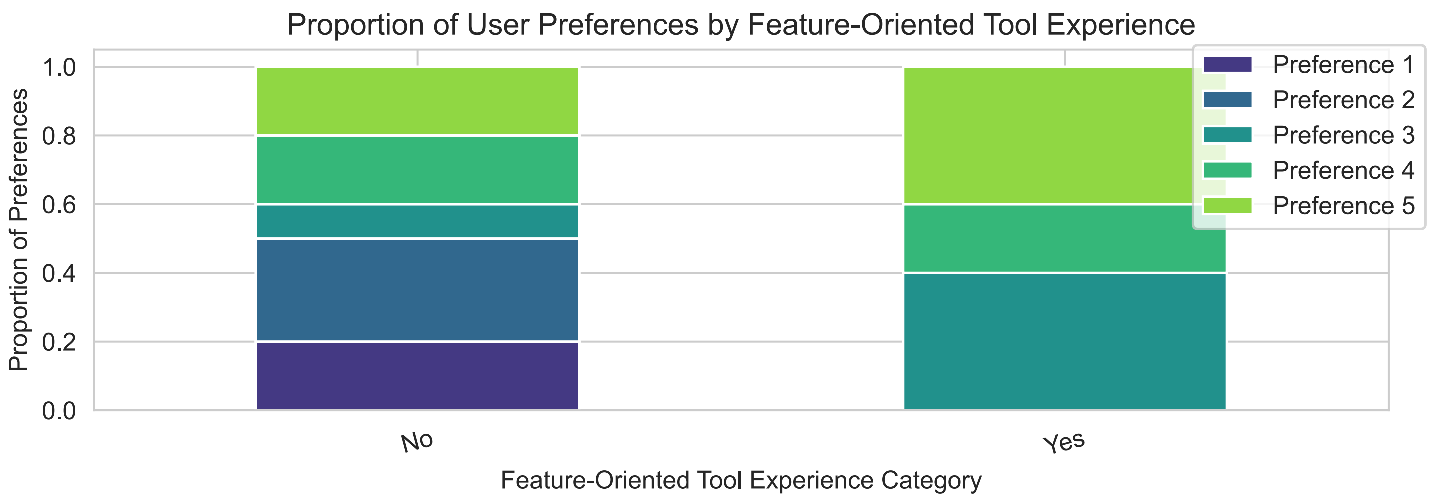


Figure 17: Proportion of user preferences by feature-oriented tool experience. Participants were divided based on their prior experience using feature-oriented tools. The chart illustrates how these preferences vary between Git with Features and Traditional Git.

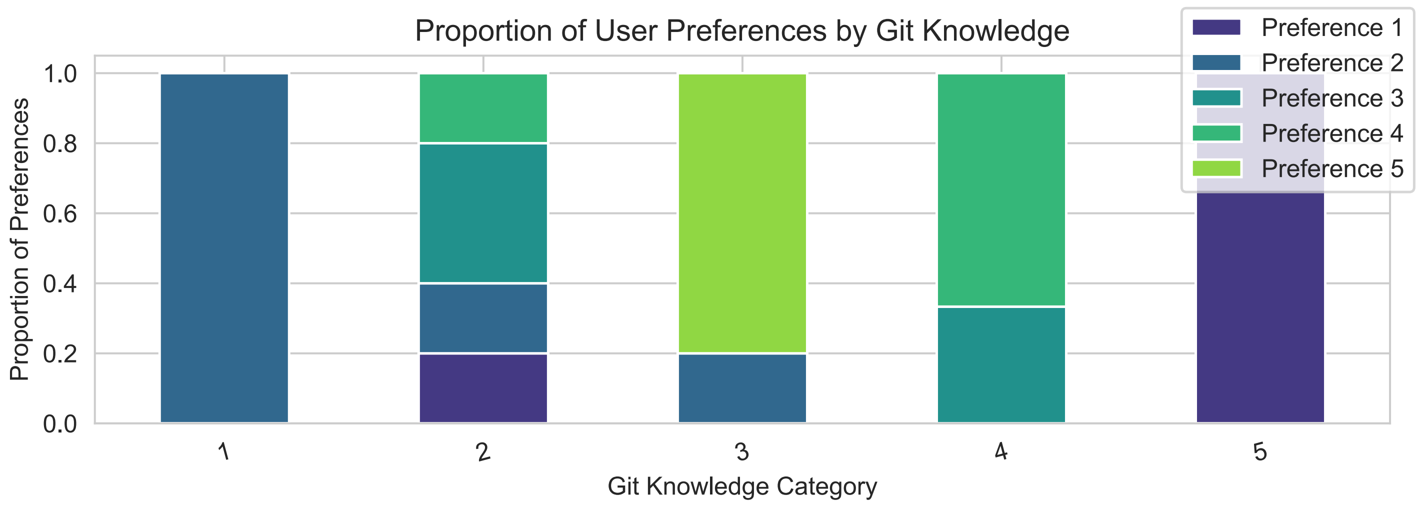


Figure 18: Proportion of user preferences by Git knowledge. Participants were grouped according to their level of Git knowledge, showing their preferences for either Git with Features or Traditional Git.

E. Code Artifacts

The link to the research prototype can be found on GitHub: `https://github.com/isselab/feature-oriented-git`. The code for the experiment in the user study can also be found on GitHub: `https://github.com/tabeatheunicorn/git-with-features-experiment`

Listing 23: Build file - pyproject.toml. In this file, all package information including the dependencies and scripts to be installed are listed.

```
[project]
name = "git_tool"
authors = [{ name = "Tabea Röthemeyer", email = "tabea.roethemeyer@rub.de" }]
license = { file = "LICENSE" }
description = "Support feature-oriented development workflows with git"
readme = "README.md"
version = "1.0.0"
requires-python = ">=3.10"
classifiers = [
  "Programming Language :: Python :: 3",
  "License :: OSI Approved :: MIT License",
  "Operating System :: OS Independent",
  "Development Status :: 3 - Alpha",
]
dependencies = [
  "typer>=0.12,<0.13",
  "pydantic[email]>=2.8,<3.0",
  "GitPython>=3.1,<4.0",
  "python-dotenv>=1.0,<2.0",
  "prompt_toolkit>=3.0,<4.0",
]

[project.scripts]
git-feature-status = "git_tool.ci.subcommands.feature_status:app"
git-feature-info = "git_tool.ci.subcommands.feature_info:app"
git-feature-add = "git_tool.ci.subcommands.feature_add:app"
git-feature-blame = "git_tool.ci.subcommands.feature_blame:app"
git-feature-commit = "git_tool.ci.subcommands.feature_commit:app"
git-feature-commit-msg = "git_tool.ci.subcommands.feature_commit_msg:app"
git-feature-pre-commit = "git_tool.ci.subcommands.feature_pre_commit:app"
git-feature-commits = "git_tool.ci.subcommands.feature_commits:app"
```

```
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"


[tool.hatch.build.targets.wheel]
packages = ["git_tool"]

[tool.pytest.ini_options]
testpaths = ["test"]
addopts = "-s"
pythonpath = ["."]

[tool.black]
line-length = 80
target-version = ['py38']
include = '\.pyi?$'
extend-exclude = '''
/(
  # The following are specific to Black, you probably don't want those.
  tests/data/
  | profiling/
  | scripts/generate_schema.py  # Uses match syntax
)
'''
```