RUHR-UNIVERSITÄT BOCHUM

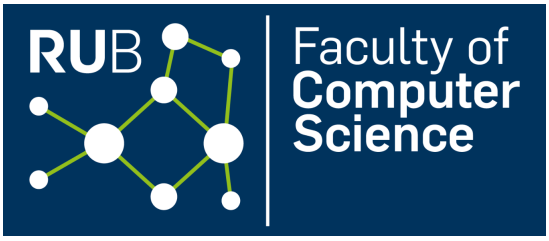
RUB

Extending IDE-based editing support for embedded feature annotations with a recommender system

Maurice Beintmann

Bachelors's Thesis – October 28, 2025
Chair of Software Engineering.

Supervisor: Prof. Dr. Thorsten Berger
Advisor: Kevin Hermann, M.Sc.



Contents

1 Introduction

1.1 Motivation

Features are a key concept in modern software development to represent the distinct functionalities of a software system [4]. They specify and manage the different parts of a software system to support developers in comprehending, reusing, or changing the system [15]. While developers may document a list of features that a software system contains, the location of those features is often not documented [21].

Feature location recovery is a common and time-consuming activity for developers, especially when features were implemented a long time ago, possibly by a developer who already left the organization, or when the feature has code spread across multiple packages, files, or code fragments [28, 24, 32]. The recovery therefore imposes a considerable overhead before developers can complete their tasks, such as fixing bugs or changing different aspects of a feature. This poses the challenging issue of how traceability between features and their corresponding software assets can be effectively maintained.

To address this feature traceability challenge, there are two main approaches: In the lazy strategy, the feature locations are retroactively recovered when they are required. This process relies on manual or automated feature location recovery techniques. However, manual feature location recovery is a laborious and error-prone process [32], and while several automated feature location recovery techniques have been proposed, they often require large training datasets, produce too many false-positives for effective usage in production, and recover only coarse-grained feature locations [25, 28]. In practice, features are more fine-grained entities [17].

The eager strategy asks developers to explicitly record traceability information during the development phase, when the knowledge of a feature is still fresh in the mind of the developer. For this purpose, feature information can be documented externally, for example in a feature traceability database like FEAT [27]. Alternatively it can be stored internally by embedding it directly into the software assets via embedded feature annotations [13, 29]. Although continuous recording will save a substantial amount of feature location recovery cost in the long-term [13], providing short-term benefits to developers has been identified as a crucial criterion in establishing feature traceability [19]. Developers might also forget to record and as a result

some feature locations may not be traced and aforementioned techniques are needed to recover their locations. The natural evolution of code can also cause preexisting feature location recordings to become inconsistent.

FeatRacer has been proposed to nudge developers towards a better documentation of their features by predicting when the developer forgets to annotate new code with a feature [21]. A study by Brown and Parnin [9] has shown that nudging techniques are effective in changing the behavior of software developers. To effectively incorporate FeatRacer into the developer's workflow, integration into mainstream tooling, such as IDE's or a version control system such as Git, is essential. This integration ensures that FeatRacer can automatically notify developers directly within their existing workflow, thus eliminating the need to switch between tools and minimizing disruptions to productivity [22].

1.2 Research Questions

To deal with the identified challenges and improve current feature traceability practices, we pose two research questions. The first question is *"How can a recommender system be integrated into an IDE plugin to provide support in maintaining feature traceability?"*. To answer this, we want to incorporate FeatRacer into the development workflow by providing tool support within an IDE. The aim is to integrate FeatRacer into the HAnS plugin for the IntelliJ IDE. This will increase the workflow efficiency for feature traceability by automatically triggering FeatRacer upon each version control commit.

To evaluate the effectiveness of the implementation, we formulate the second research question: *"What is the impact of the integration on usability during feature traceability tasks?"*. A usability study is conducted to determine the effectiveness of the plugin integration. In this study, the participants will perform a series of tasks designed to explore the plugin's functionalities. To evaluate the effectiveness, we record task completion times and measure the degree of correctness. Additionally, participants provide feedback through a questionnaire.

2 Background

2.1 Features

Features are high-level labels assigned to distinct and well-understood aspects of a software system. They are used to describe, manage, and communicate the different functional and non-functional elements within those systems. This abstraction supports different stakeholders with the communication and understanding of the underlying system's characteristics [4, 15, 2].

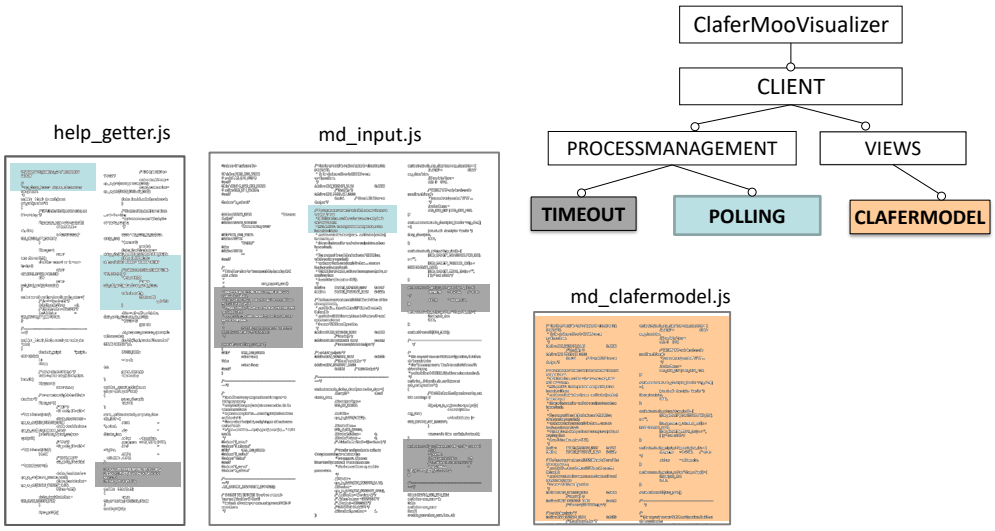


Figure 2.1: Example of feature scattering across multiple files in a codebase [21]

A fundamental characteristic of them is their cross-cutting nature, where individual features often span through multiple subsystems, files, or code fragments [23]. An example of this is illustrated in Figure 2.1, where the code fragments from the features *timeout* and *polling* are distributed across different parts of the codebase, with each feature's location marked in distinct colors. Notably, both *help_getter.js* and *md_input.js* contain code fragments belonging to both features and therefore demonstrate how multiple features scatter within the same file. This scattering presents significant challenges for maintenance and evolution, motivating the need for effective feature traceability approaches.

2.2 Feature Traceability

Feature Traceability refers to the practice of being able to establish a relationship between software assets and the features implemented by those assets. It is imperative to possess a comprehensive understanding of this relationship in order to effectively evolve, reuse, or change features [13, 18]. To address the traceability of features, there are two main strategies: The eager strategy and the lazy strategy.

In the **lazy strategy** the feature traceability information is not documented upfront and instead retroactively recovered when required. This recovery can be done manually, for example, through exploration of the source code, executing the application and triggering the feature, or by writing test cases that utilize the feature. [32, 16].

Automated feature location recovery follows a different approach. Instead of manual feature location recovery, they return a ranked list of files and methods, which is considered an implementation of the feature. The developer is then able to evaluate those candidates. Those techniques can further be classified as either static or dynamic [28]. *Static techniques* conventionally act as a search engine and prompt the developer for terms that specify what to search for in the code. This is usually done with the help of information retrieval techniques such as LSI and tf-idf [26], or machine learning [10]. *Dynamic techniques* monitor the execution stack trace, typically with the use of a given test case that has been written by the developer.

While the lazy strategy requires no upfront overhead in the development phase, it generates substantial effort in the recovery of feature locations in the later stages of a project. This recovery process itself is often costly, time-consuming, and error-prone. A survey by Wang et al. [32] observes an average of 15 minutes for feature location using a manual approach in systems with 73.000 lines of code or less.

Automated feature location recovery techniques come with their own set of additional challenges. They are often resource-intensive, difficult to set up, and have low precision and recall that is highly dependent on the prompt given by the developer [28, 25, 14]. Additionally, only coarse-grained feature locations can be recovered using automated techniques, while features are often fine-grained in practice [17].

The **eager strategy** involves the recording of feature traceability information during the development process [13]. It requires developers to explicitly document features and their location when the knowledge is still fresh in the mind of the developer. For this purpose feature information can be documented externally, for example with a feature traceability database like FEAT [27]. An alternative approach is to store

feature information internally by embedding annotations directly into the software assets [13].

Although continuous recording of feature traceability information will provide long-term benefits, it requires additional attention from the developer during the development process [7, 6, 12]. Developers might also forget to record features and their locations, alike developers not always commenting code during the development. As a result the locations of some features might need to be recovered.

2.3 Embedded Feature Annotations

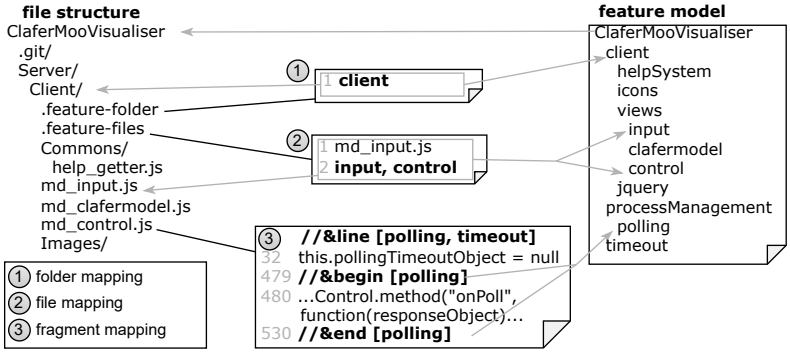


Figure 2.2: Example of embedded feature annotations for folder(1), file(2), and fragment(3) mapping

Embedded feature annotations can be utilized to implement the eager strategy in retaining feature traceability. This approach intends to embed the traceability information directly within the source code in the form of structured comments. Compared to other approaches, embedded annotations are lightweight and research suggests that it can substantially reduce feature location recovery costs [13]. A standardized notation for embedded feature annotations has been proposed by Schwarz et al. [29].

In their model, all features are organized in the *feature-model* file. Each feature is cataloged in its own line within the file, with the first line being the project name. The hierarchy of features is represented with tab-indentation within the file. To reference these features, the shortest possible unique path in the hierarchy is used, with "::" being used to conjoin the feature names.

Figure 2.2 shows an example of using embedded feature annotations. Code assets can be mapped to features on a folder, file, or code fragment level. Fragments and single lines are annotated with embedded code annotations in the form of comments. Developers annotate the asset with a comment, followed by an ampersand, and a keyword, which differs for fragment and line annotations. The features are then

written in square brackets after this keyword and separated with commas for multiple features.

To map a folder or file, the developer adds another file named *.feature-to-folder* or *.feature-to-file* to the relevant folder. The .feature-to-folder file lists features in individual lines, while the .feature-to-file file works in pairs of two lines. The first line defines the file name of the file that is mapped, and the succeeding line lists the features, separated with commas, that are associated with that file.

2.4 The HAnS Plugin

HAnS (**H**elping **An**notate **S**oftware) is a plugin for the IntelliJ IDE that supports developers in maintaining feature traceability with embedded feature annotations (see section 2.3) [20]. The plugin provides code completion and syntax highlighting for embedded feature annotations. Furthermore, HAnS also provides a feature browser for developers to browse through feature locations, as well as the option to refactor rename or delete features.

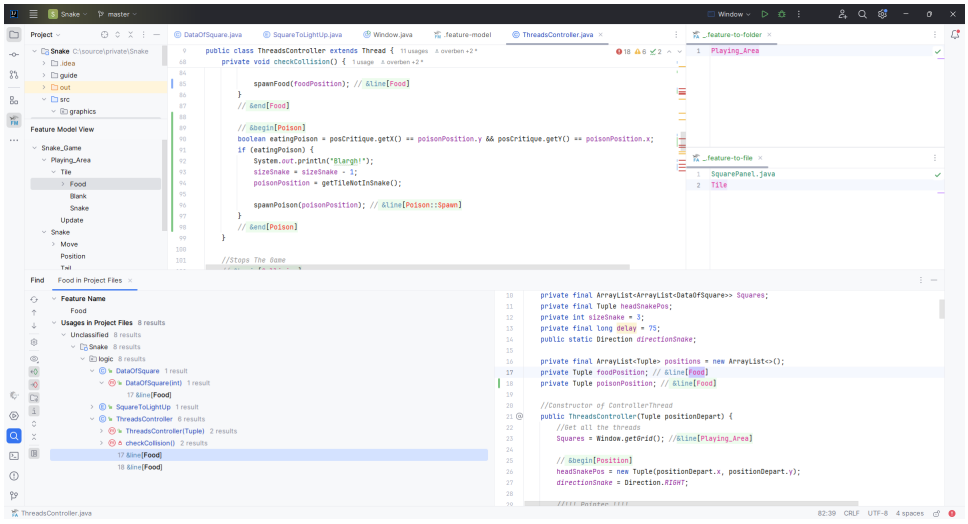


Figure 2.3: Overview of the HAnS plugin [20]

Figure 2.3 shows an overview of the HAnS plugin in action. In the main editor window, the line and block annotations are highlighted with colors. The side panel on the left-hand side has a browser displaying the feature model in a hierarchical manner. The selected feature *food* has all its occurrences listed in the search window below. Those occurrences can then be selected to display a new editor window at that location, as shown in the bottom right.

2.5 FeatRacer

The FeatRacer technique has been proposed by Mukelabai et al. [21] to assist developers with the proactive and continuous recording of feature locations using embedded feature annotations. Although the FeatRacer technique could possibly work using external databases and other traceability strategies, the designers of FeatRacer chose embedded annotations due to their advantages over alternative solutions (see section 2.3) [21]. FeatRacer can be adopted at any point during the project's lifecycle, although ideally embedded annotations are adopted from the project start.

FeatRacer works with a recommender system, which should be triggered after a commit or a similar event that marks a significant step in software evolution. When it is triggered, it analyzes the assets from the changeset that do not have any feature annotation. Developers are informed when it predicts that they failed to annotate parts of the new source code with a specific feature. The predictions are made based on the project history and the assets that were previously annotated, on which FeatRacer learns. When this prediction is made, a ranked list of feature recommendations is returned. From this list the developer can choose whether to adopt a given recommendation and if they do, the specific asset is annotated with the chosen feature and the changes are committed to the version control system.

Metric	Description
CSDEV	Average cosine similarity between names of distinct developers contributing to a
DDEV	Number of distinct developers contributing to a
DCONT	Average percentage of lines contributed to a per developer
HDCONT	Percentage of lines contributed by the developer with the highest contribution to a
COMM	Amount of commits in which a has been modified
CCC	Total number of assets modified in the c alongside a
ACCC	Average number of assets modified alongside a in C
DNFMA	Number of distinct features mapped to all assets modified together with a in C
NFMA	Number of features modified with a in c
NLOC	Lines of code associated with a in c
NFF	Number of features within file that contains a in c

Table 2.1: Metrics used by FeatRacer where a be an asset modified in commit c, with C being the set of all commits up to c where a has been modified [21]

Internally, when FeatRacer is invoked, it analyzes the commit history of the project to collect all assets and the mapping of a specific asset to declared features, if feature annotations exist for that asset. For this data, metrics are calculated for each asset,

including aspects such as the number of developers that contributed to that asset or the number of commits that have modified that specific asset. A full list of asset metrics can be found in Table 2.1.

Afterwards, FeatRacer utilizes a machine-learning algorithm to build a machine-learning model using the data collected in prior steps. If the latest commit changeset includes assets without annotations, it uses the model to find recommendations for feature annotations that the developer forgot to annotate.

3 Related Work

Feature location encompasses a wide variety of techniques. A survey by Rubin and Chechik [28] identified 24 distinct techniques, highlighting the breadth of research in this area. Given this high amount of approaches, this chapter aims to provide an overview of techniques and tools that are most closely related to FeatRacer and HAnS.

TAFT

The Feature Tagging Approach (**TAFT**) was developed by Seiler and Paech [30], aiming to document feature knowledge continuously with the use of *Feature Tags*. At its core, it utilizes Java's annotation tag system with a **@Feature("x")** annotation in front of the relevant fragment. The tag has a *description* associated is used to provide a general specification of the feature. It can optionally have a *requirement*, which is defined as a specification that refines the feature, or a *work item*, which describes the development effort needed to implement the feature. Figure 3.1 displays an example for the full specification of the feature "Transportation" and an example of a feature annotation in Java code.

Feature: Integration of an API to retrieve data from public transportation such as stations and their departure schedules.

Requirement: As a user, I want to click on a station in order to view the departure schedule.

Work item: Implement service function to retrieve the departure schedule of a station.

```
// Package, imports and further code omitted
@Feature("Transportation")
public class OpnvManager implements IOpnvManager {
  public void queryStation(String stationID) {
  Request request = new Request.Builder().url(new HttpUrl.Builder()
    .scheme("http").host("rnv.the-agent-factory.de")
    .addQueryParameter("stationID", stationID).build();
  new OkHttpClient().newCall(request);
  }
}
```

Figure 3.1: Example of TAFT's documentation and code specifications [30]

The authors provide tool support for TAFT as plugins within the project management software Jira, and the Eclipse IDE. The tools provide a feature navigator, a feature dashboard, and a feature recommender that provides annotation recommendations. The feature navigator allows developer's to list the code files that implement a feature and navigate towards a location in the code when clicked on in the navigator, similar to the feature browser in HAnS (see section 2.4). The feature dashboard is a part of the Jira tool, and it tracks various metrics for the project to be displayed within this dashboard.

FLOrIDA

The Feature Location Dashboard (**FLOrIDA**) is a standalone Java application designed to encourage developers to use embedded feature annotations [1]. To achieve this, its user interface offers several features, including a feature browser to explore the documented feature locations, graphic visualizations that show relationships between features and the code fragments implementing them, and a variety of feature metrics, such as the scattering degree or the number of files associated with a feature.

Another noteworthy capability of FLOrIDA is the support for retroactive feature location recovery. For instances where a developer forgot to properly annotate a relevant code asset with a feature, the tool can suggest potential feature locations with the use of the information retrieval algorithm based on Lucene [1].

Feature Dashboard

Feature Dashboard is an open-source tool, usable as a plugin for the Eclipse IDE or as a standalone program, that supports developers with the visualization of the features in their software system [11]. Those visualizations are provided by extracting the traceability information stored within embedded feature annotations and displayed using an assortment of different views.

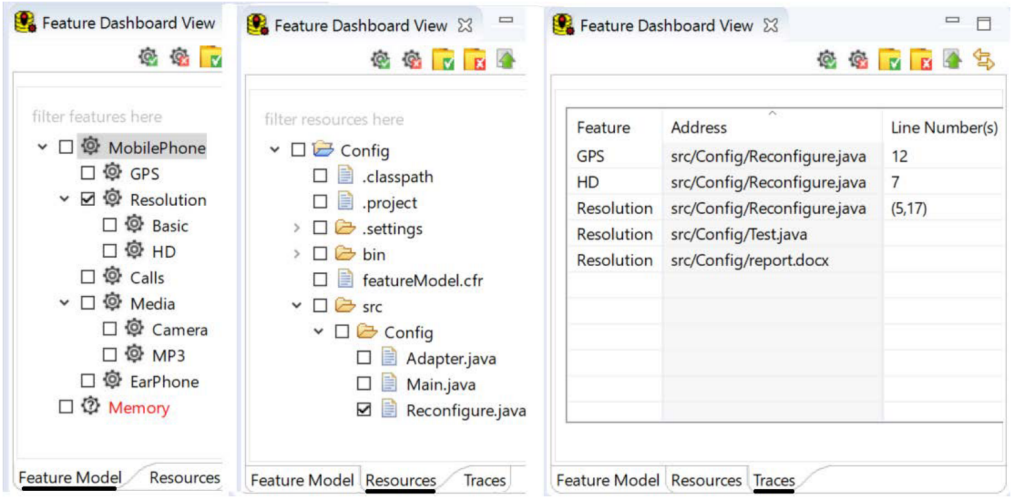


Figure 3.2: Overview of the Feature Dashboard View [11]

The tools primary interface is the *Feature Dashboard View* (see fig. 3.2), which represents a hierarchical overview of the feature model and all documented feature

[1] https://lucene.apache.org/

locations in the scope of the currently selected project. From this main view, developers can select specific features to explore them in more detail. For instance, the *Feature-to-File View* and *Feature-to-Folder View* visualize the relationship between these features graphically by drawing connections between nodes that represent the selected features and their corresponding files or folders.

In addition, FeatureDashboard offers several analytical views. The *Common Features View* uses a matrix to show which features are shared across multiple projects, while the *Feature Tangling View* highlights all features that are tangled with a selected feature. Finally, the *Metrics View* displays various feature metrics, similar to the metrics provided by FLOrIDA.

4 Methodology

This chapter outlines the methodology used to address the extension of HAnS with FeatRacer, as well as the evaluation of the resulting implementation.

4.1 Development

This section outlines the design decisions made for the integration of the FeatRacer recommender system into the IntelliJ plugin HAnS, directly addressing the first research question on providing IDE support for feature traceability. HAnS was selected as the integration target for FeatRacer because it already offers robust support for the kind of embedded feature annotation that FeatRacer utilizes, thus making it an ideal extension point. The technical approach consists of two distinct phases to ensure a robust integration. First, we refactor FeatRacer into a library to provide recommendations. This library is then integrated into the HAnS, which serves to present and annotate the recommendations given by FeatRacer.

Separating the recommendation engine from the presentation layer yields several advantages. It increases the maintainability of the library and the plugin as both can be tested and updated independently. Being a standalone library also allows for FeatRacer to be integrated into other tools, for example into an extension for a different IDE. Furthermore, it allows the IDE plugin to work with different libraries that provide a recommender system.

4.1.1 FeatRacer

The initial project plan assumed that FeatRacer was in a mostly functional state, requiring only minor adjustments and the development of a library interface to enable its integration. However, an analysis of FeatRacer's codebase revealed that the existing implementation was a research prototype for experiment evaluation. Consequently, the initial course of action shifted towards the refactoring of this codebase.

The first action taken was addressing the project's persistent data storage. FeatRacer relied on an external MySQL database to store information on identified code assets, feature mappings, metrics, and training datasets. To simplify the setup and reduce overhead for plugin users, the database was migrated to SQLite. SQLite is

a lightweight and serverless SQL database engine, thereby eliminating the need to create an external database server. Instead of relying on a separate server process, it utilizes a local file on the disk [5].

This migration required a substantial rework of the original database scheme, as SQLite does not support the procedures utilized in the original MySQL design. Since only an old snapshot of the database scheme was available, many procedures had to be recovered by analyzing the functions calling the procedures and their return values in FeatRacer's code.

Further investigation of the project revealed issues with the dependency management, as it contained duplicate and version-conflicting libraries. To address these issues and streamline the dependency management process, the build tool Gradle was introduced. This required a refactoring of the project's directory structure to align with Gradle's conventions and solving initial issues while adopting it.

The next step was to understand FeatRacer's internal workflow. While its *config.properties* file listed 23 execution modes, an analysis identified a sequence of four that, when executed in order, formed the core pipeline for generating feature recommendations(see Table 4.1).

D	Generates data for assets and feature annotations
GM	Calculates metrics over features and assets that implement them
GDT	Generate ARFF databsets from the metric data
EDB	Trains and runs classifiers based on the ARFF datasets to produce recommendations

Table 4.1: Execution methods used generate recommendations

With an improved understanding of the workflow, the class *FeatRacerAPI* was created to serve as the public interface for the library. This API defines two new methods, for project initialization and invocation after the initialization respectively. Both methods simplified the pipeline of four execution modes into a single method. The process of making the API functional entailed two major engineering efforts: Reworking the classifier method and implementing the methods for data and metrics generation on a per-commit basis.

The original classifier module not implemented for production use. Its purpose was to evaluate a wide range of classifiers from Mulan, an open-source Java library for multi-label learning that integrates with the WEKA machine learning toolkit [31]. For the implementation, this module was re-engineered in its entirety. The RAkELd

classifier was selected based on the original FeatRacer experiment, which identified it as the overall best-performing model for this task [21]. In addition, the output format was also refactored into a structured data format to facilitate its use within the plugin.

Additionally, the original **D** and **GM** execution methods could only process entire projects, which is considered to be too slow for displaying responsive feature recommendations within the IDE. New variations of those methods were created for the invocation method of our API that only processes one commit. These variations have been changed to accept a commit hash as a parameter and their scope is limited to that commit. This approach avoids the inefficiency of the previous methods, which re-process the entire project history for every execution.

Beyond these major changes, numerous minor improvements and bug fixes were applied throughout the codebase. While not individually detailed, these refinements were essential contributors towards the performance and stability of this library.

4.1.2 HAnS Extension

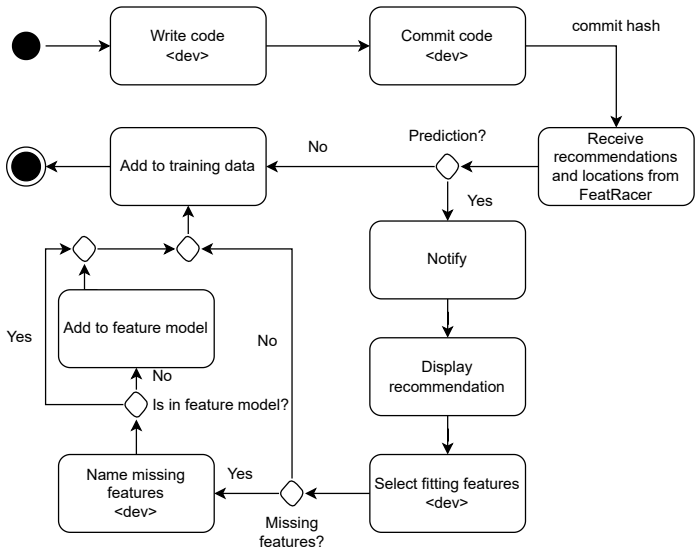


Figure 4.1: An overview of the conceptual workflow of the HAnS extension

The primary goal of this plugin extension is to integrate FeatRacer into the developer's daily workflow. Figure 4.1 visualizes the conceptual blueprint that was designed to guide the implementation of this workflow.

It is imperative to note that FeatRacer requires initialization for a project before this workflow becomes active. The developer can start the initialization process by interacting with a notification sent by the plugin when working on a project that has not been initialized yet.

The process is initiated when the developer writes their code and commits it to version control. The extension needs to listen to this event and trigger FeatRacer for that commit. When FeatRacer predicts that feature annotations are missing, the plugin notifies the developer and presents the recommendations in an interactive dialog. Within that dialog the developer can select the feature recommendations that fit, and the plugin will automatically annotate the features based on the selections. Finally, regardless of whether predictions were made, the commit is used to enrich the training data, allowing the model to learn continuously from the developer's annotations.

4.2 Usability Study

This section details the experiment conducted to evaluate the HAnS FeatRacer extension's usability and to address the second research question. Participants performed tasks using the plugin and their feedback was used to assess its usability. In the following, the experimental setup, task design and the questionnaire are outlined.

4.2.1 Setup

We gave participants access to a Google Forms document that we prepared for this experiment. The document contained an introduction, a pre-questionnaire, an explanation on the task setup, the tasks and the questionnaire for evaluation.

The introduction started with a brief explanation for the background topics features and feature traceability, to establish the core problem and the underlying challenges. Next, embedded feature annotations were presented as a lightweight solution to this problem. It explained that this approach can substantially reduce feature location recovery costs (see section 2.3) and an explanation on how to use them was given. HAnS and its core functionalities were introduced as a plugin, designed to support developers in using these annotations.

This introduction highlighted the challenge of a developer forgetting to add annotations. Subsequently, FeatRacer was introduced as a recommender system that attempts to predict missing feature annotations, to nudge developers toward better documentation. The workflow, as described in section 4.1.2, was also disclosed. We concluded the introduction section by outlining the purpose and structure of the

experiment. Following the introduction, we asked participants to complete a pre-questionnaire that was designed to gather information such as demographic details and prior experience to the experimental topic.

We gave participants detailed setup instructions for the tasks. As a prerequisite, they were required to have IntelliJ IDEA Ultimate Edition version 2025.1 or later, Java JDK 21, and Git installed. A ZIP archive containing HAnS, FeatracerAPI, and the task projects was provided, along with instructions for the setup of the IDE environment, utilizing Gradle's *runIde* command to launch an instance of the IDE running with the plugin.

4.2.2 Tasks

We gave all participants the same set of three tasks to familiarize themselves with the main functionality and interface of the plugin. The first task served as a warmup task while the second and third task evaluated for correctness, with participants additionally reporting the time required to complete each task. The tasks were based on a repository for the game Snake[1], which was written in Java and contained feature annotations in its code. We chose the project for its simplicity, as it contains 16 features and 8 files with fewer 500 lines of code in total, to minimize potential bias from participants having difficulties understanding the project.

Task 0 (warmup): We asked participants to annotate some aspects of the code using the recommendation view from our plugin. For this we setup a set of mock feature recommendations in a simulated recommendation view. Participants had to either apply or reject some or all of the recommended features, and additionally, add annotations for relevant features that had not been recommended.

Task 1 (Workflow): This task instructed participants to extend the Snake game with the feature *Poison*, similar to the preexisting feature *Food*, which adds consumable items to the play area. This required the participants to add code blocks in various parts of the project. After completing the implementation, we asked them to commit their changes, thereby triggering the plugin to display recommendations for the changeset of that commit, and to review this dialog and apply feature annotations where appropriate. After the task we asked participants to provide feedback, which we use to evaluate the usability in the plugin's main use case.

Task 2 (Recommendations): The last task revisited the same method employed for the warmup task, using different feature recommendations and a larger number of mock recommendations. This task ensured that participants are exposed to feature recommendations, as task 1 did not guarantee that concrete locations would appear for the modified code locations. Furthermore, in the event of issues with the

[1] `https://github.com/johmara/Snake`

FeatRacerAPI, this task ensured that the core functionality of the plugin could still be demonstrated and evaluated.

4.2.3 Analysis

The System Usability Scale (SUS) was utilized to assess the perceived usability of the plugin [8]. It consists of ten statements rated on a five-point Likert scale ranging from *Strongly disagree* to *Strongly agree*. The statements alternate between positive and negative phrasing, which helps mitigate acquiescence bias. It provides a SUS score, a value between 0 and 100, based on the responses given for those statements, representing the overall usability of the system.

In addition to the SUS, participants were asked supplementary questions in between and after the tasks. These questions aimed to further assess the usability of the plugin and to provide participants with the opportunity to share qualitative feedback. The full questionnaire can be found in the appendix.

5 Implementation

This chapter provides the implementation details for both the *FeatRacerAPI* and the *HAnS FeatRacer* extension. First we detail the development of the *FeatRacerAPI*, covering the undertaken refactorings. Subsequently, we will describe the implementation of the HAnS FeatRacer extension itself with a focus on the internal components and the user experience.

5.1 FeatRacerAPI

This section details the refactoring effort undertaken to transform the FeatRacer research prototype into a reusable library. The original codebase was designed with the FeatRacer experiment in mind and was not suitable and ready for integration, thereby creating a demand for foundational refactorings before it could be used with HAnS. While the main focus is the integration into HAnS, this library is generic and can be utilized in other projects, such as plugins to support different IDE's.

5.1.1 Data Storage

FeatRacer used an external MySQL database to store the information needed to create the training data for the classifier, which required manual setup and configuration of the database. This presented a significant usability barrier for a seamless IDE plugin experience. To eliminate the overhead of setting up an external database, we reworked the current *DataController* class by migrating it to use a lightweight and embedded SQLite database.

The connection is managed using the JDBC SQLite Driver, which connects to a local *.db* file and creates it if none is found. Our implementation checks for the existence of the required tables on every connection. If the tables are missing, the database schema is initialized from a *scheme.sql* file within the *resources* directory. However, the original scheme file provided with the MySQL implementation was an incomplete snapshot from a previous version, which made it necessary to reverse-engineer and reconstruct the missing parts of the snapshot during the conversion process.

The scheme defined tables and procedures, from which the latter act as functions as part of the SQL scheme. These stored procedures are not supported by SQLite, which necessitated a crucial refactoring of the DataController's access logic. All CallableStatements, which are used for the calling of procedures, were replaced with PreparedStatements, that are used to execute database queries. This involved extracting the SQL logic from the database scheme and embedding the queries directly into the corresponding methods in the DataController's code. A major challenge was that the scheme was incomplete, because the definitions for several procedures were missing in it. Their queries had to be reconstructed by analyzing the name, input parameters and expected output of the methods in the DataController that originally called them.

5.1.2 Dependency Management

During the testing of FeatRacer, the manual dependency management within the *lib* directory was found to cause issues, as runtime errors emerged due to conflicting versions of some libraries. To resolve those dependency issues and as part of the project refactoring process, we integrated the Gradle build tool to establish a reliable and well arranged dependency management and build process. This involved the creation of a *build.gradle* file and migration of all required dependencies, resolving the version conflicts by specifying a single version in the process. Using vulnerability data from the public source MVNRepository [1], dependencies were also updated to patched versions whenever possible to mitigate potential security threats.

At first Gradle failed to correctly build the project, because the *resources* folder, containing the scheme.sql that is essential for the database setup, was not included in the build artifact. To resolve this and align with best practices, we refactored the project's directory structure to follow the *Maven Standard Directory Layout*, which is natively supported by Gradle. This involved moving the project's java source code to *src/main/java* and all resource files to *src/main/resources*. The config.properties file containing FeatRacer's options was also relocated to the resources folder and the code was updated to load this configuration file as a classpath resource, to ensure the correct bundling into the build artifact and its availability at runtime.

5.1.3 Interface

The *FeatRacerAPI* class serves as the public entry point and facade for the new library. It exposes the methods *initializeProject()* and *invokeFeatRacer()* that share a common set of parameters: The project path on which FeatRacer is invoked, an

[1] https://mvnrepository.com/

output path for generated files, and a string of file extensions, separated with commas, that FeatRacer considers for the analysis. Both methods return a *Map<String, List<String»* where each key is a file path with a line specification representing a feature location, and the value is a list of recommended features for that location.

The *initializeProject()* method performs the initial analysis of an entire project and is intended to be run once as a setup, while the *invokeFeatRacer()* method performs the analysis on a single commit, for which it takes an additional parameter containing the commit hash. Both methods execute our feature recommendation pipeline in sequence, as described in chapter 4.1.1.

The classifier module **EDB** was completely overhauled for production use, while the data and metrics generation modules **D**, **GM**, and **GDT** received numerous fixes and adjustments to enhance their stability and performance. For the invocation method, **D** and **GM** have received an alternative version which only creates the data and metrics for a single commit. This change is critical for performance, as it allows the library to provide recommendations more quickly and efficiently after the initialization.

Data Generation

The original **D** module leveraged the RepoDriller [2] library, a Java framework for mining software repositories, to extract the information from the projects Git repository that populate the internal database of FeatRacer. This was managed with the *ProjectReaderWithDriller* class, which implements RepoDriller's *Study* interface to perform a comprehensive analysis of the entire project history. The visitor *ProjectDBVisitor* was used to insert this extracted data into the database.

To support the new per-commit analysis required for the *invokeFeatRacer()* method, the new class *CommitReaderWithDriller* was implemented. This class also implements the *Study* interface, but it is designed to mine data from a single commit only. While the *ProjectReaderWithDriller* class extracts all commit hashes for the project, the new class receives only the commit hash that should be analyzed as a parameter in its constructor. This hash is directly passed to the *ProjectDBVisitorSingle*, a modified version of the *ProjectDBVisitor* that allows it to process data for that commit only.

Metric Calculation

The *MetricCalculatorDB* class handles the calculation of metrics (see table 2.1) from the data gathered in the previous step. Originally, this class was designed to process

[2] https://github.com/mauricioaniche/repodriller

the entire dataset at once. The class was extended with the *calculateMetricsSingle()* method, which takes a commit hash as input, retrieves the collected data associated with that commit, and then calculates and inserts the corresponding metric data into the database. This targeted approach, combined with the data mining improvements, significantly reduces execution time for analyzes that run after the initial project setup.

Recommendation Service

As previously mentioned, the module providing recommendations was only a vessel for the FeatRacer experiment evaluation and therefore in need of an overhaul. To combat this, the new class *RecommendationService* was created for use within our API class. Its main method *runClassifier()* extracts the latest commit dataset in the internal database, checks if it has a test *.arff* dataset associated with the commit, and if it does, it will train a machine learning model with the training *.arff* dataset and use the RAkELd classifier from the Mulan library to make predictions [31].

For each instance of that test dataset, a prediction is made and a *MultiLabelOutput* is returned, that if not empty, is added to the result. The resulting HashMap is returned after predictions for each instance have been made.

5.2 HAnS-FeatRacer Extension

This section describes the process of extending the IntelliJ plugin HAnS with FeatRacer. While this extension was specifically developed for FeatRacer, it was designed with extendability and modularity in mind to facilitate future integrations with other classifiers or traceability tools.

The following subsections outline the components of the plugin. First we describe the modular integration to manage classifiers. The recommendation view is then introduced as the main interface for reviewing and applying recommendations. Finally, we present the implementation logic behind project initializations and the commit watcher, responsible for detecting commits and triggering our classifier accordingly.

5.2.1 Classifier Integration

The plugin extension was designed to increase the usability of FeatRacer by providing IDE support to increase FeatRacer's ability to nudge developers for an increased feature traceability. We do not want this extension to be limited to FeatRacer, as a modular approach increases the reusability, maintainability and scalability of

our project. For this reason we implemented a strategy design pattern to integrate our classifier. A visual representation of this implementation can be seen in Figure 5.1.

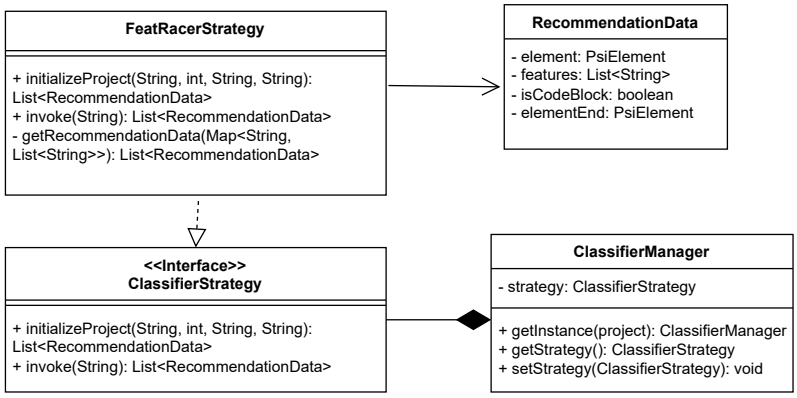


Figure 5.1: UML diagram of the strategy pattern for the classifier integration

We define the interface *ClassifierStrategy* that provides two methods: *initializeProject()* for a full project analysis at the initialization stage and *invoke()* for subsequent commits after the initialization. Both methods return a list of *RecommendationData* objects, which encapsulates all information for a single recommendation, such as the location data of the recommendation and the features that were recommended. To manage the current ClassifierStrategy, the *ClassifierManager* project level service is utilized. Whenever the classifier should be triggered, the current strategy is retrieved from the ClassifierManager and the corresponding method is called.

For our FeatRacer implementation we created the class *FeatRacerStrategy*, implementing the *ClassifierStrategy* interface, which acts as an adapter by calling the FeatRacerAPI methods and then using the private helper method *getRecommendationData()* to convert the output into a *List<RecommmendationData>*, which is returned. The methods are called as part of a *Backgroundable* task to ensure that the FeatRacerAPI call does not impair the performance of the main IDE thread, as that would cause the IDE to freeze.

5.2.2 Recommendation View

Whenever the call of the FeatRacerAPI returns one or more recommendations, a dialog window is presented to the developer to support the developer in reviewing and applying the suggested feature annotations. The dialog utilizes a CardLayout with each panel corresponding to one feature recommendation. As shown in figures 5.2 and 5.3, each panel consists of a read-only editor to provide context, and an

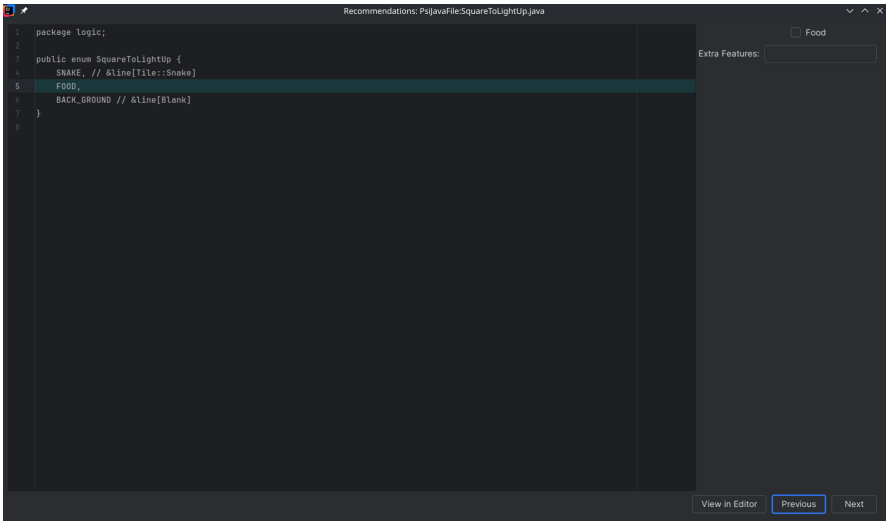


Figure 5.2: Example of a line annotation in the recommendation dialog

adjacent panel on the east with a checkbox for every recommended feature and an additional text field for custom input.

The panel is implemented with the class *RecommendationDialogPanel*, which extends JPanel. Each instance stores the information for a given recommendation, for example, the location of the recommendation or the suggested features. It also handles the logic for our automatic annotation process.

If the developer ticks a checkbox or types into the text field, an action listener is triggered to update the annotations based on the selected features. To remember the exact boundaries of our annotations, we leverage the *RangeMarker* from IntelliJ's Platform API. A RangeMarker represents a range of text in the document that tracks its position as the document is modified. When our listener is triggered and no annotation exists, we will apply the annotation and create corresponding RangeMarker for each comment. Subsequent calls then update the text within this marker. The annotation process is designed to handle both single-line and multi-line code selections, depending on if the recommendation contained a single-line or a range of lines for the location.

5.2.3 Initialization

The entry point for the initialization of a project is the *InitializationActivity* class that implements IntelliJ's ProjectActivity interface and is registered in the plugin.xml as a *postStartupActivity*. When the project has finished loading, this activity checks if the current project has already been initialized. If it has not, a

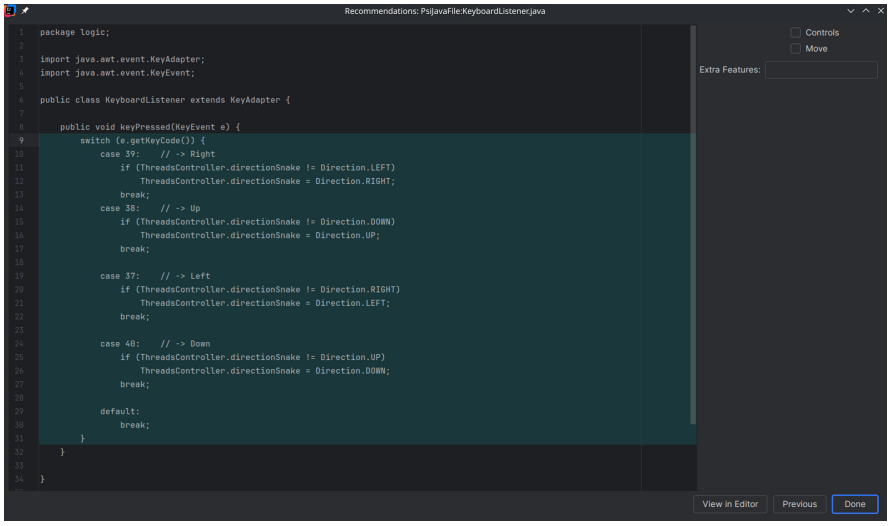


Figure 5.3: Example of a block annotation in the recommendation dialog

sticky balloon notification is displayed at the bottom right of the IDE, prompting the developer to initialize the current project.

The notification triggers an action event when clicked, opening an initialization dialog that gathers information from the developer, such as the id of the commit at which the initialization analysis should begin. The dialog was also designed to ask the developer for a custom analysis folder path, in which FeatRacer stores all its information, but this feature is currently deprecated due restrictions with IntelliJ's sandboxing.

Upon clicking the *initialize* button, the given information is saved using a project level service implementing the *PersistentStateComponent* interface. Finally, the *initializeProject()* function from the current classifier strategy is called to perform the initial analysis of the full project. In the case that this analysis returns any recommendations, they are immediately presented to the developer with the recommendation dialog.

5.2.4 Commit Watcher

In order to nudge the developer and provide recommendations, our extension must be able to listen when a successful commit is made. This functionality was implemented using the *CheckinHandlerFactory* and *CheckinHandler* extension points provided by the IntelliJ Platform API.

To ensure modularity and extendability, the implementation of our handler does not directly invoke FeatRacer. Instead, we use a publish-subscribe pattern to separate the logic of detecting a commit from the logic of processing it. This means

that other components can subscribe their own implementation of the *CommitListener* interface to the *CommitWatcher* service, without requiring any changes to our implementation of the CheckinHandler.

Utilizing this structure we implemented the *FeatRacerCommitListener* that implements our *CommitListener* and is registered to our *CommitWatcher* service. Our implementation invokes the FeatRacerAPI on every successful commit. If the FeatRacerAPI returns any number of recommendations, those recommendations are displayed to the developer with our recommendation view (see chapter 5.2.2). If no recommendations are returned, the call still serves purpose by extending FeatRacer's internal dataset with the new information from the commit.

6 Usability Study Results

This chapter details the findings from our usability study, which was conducted to evaluate the usability and design of the HAnS plugin extended with FeatRacer, by presenting the data collected during the experiment. From this data we will then calculate and interpret the SUS score to assess the overall perceived usability of the plugin.

Demographics

The participants comprised 14 university students from three different universities at the undergraduate and bachelor level, with more than two-thirds having completed their bachelor's degree (see fig. 6.1). All participants were enrolled in study programs related to computer science.

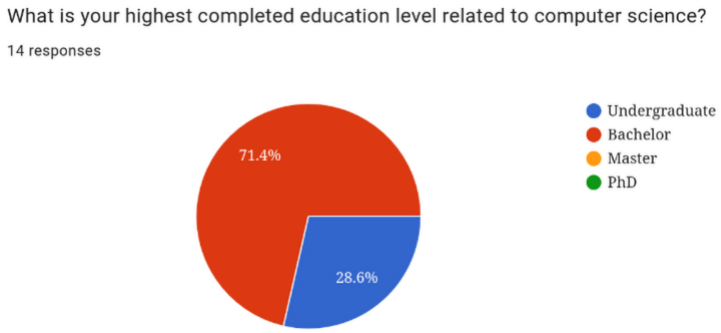


Figure 6.1: Education levels between participants

Regarding programming experience, 12 out of the 14 participants reported having more than three years of experience, with six of them reporting over five years of experience (see fig. 6.2). In addition, the participants were asked to rate their familiarity with HAnS and embedded feature annotations on a Likert scale ranging from one to five, labeled as *very unfamiliar* to *very familiar*. All participants indicated being "very unfamiliar" with the HAnS plugin, and 86% reported the same for embedded feature annotations.

For the evaluation, participants were asked to respond to both Likert-scale items and open-ended questions. Subsequently, the results will be shown. In the following figures, the responses range from 1 (Strongly disagree) to 5 (Strongly agree),

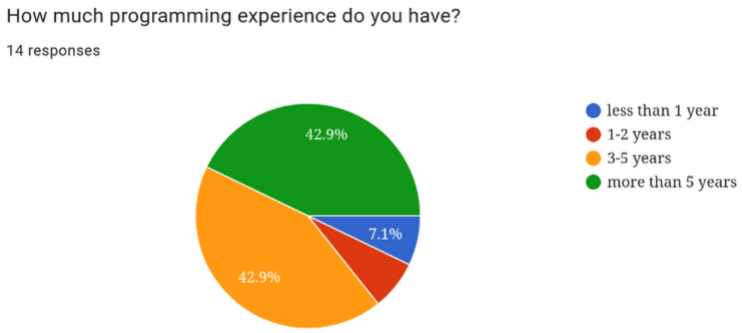


Figure 6.2: Programming experience between participants

to represent the Likert statements *Strongly disagree*, *Disagree*, *Neutral*, *Agree*, and *Strongly agree*, respectively.

Recommendations after committing

During task 1, FeatRacer posed a considerable challenge for the participants in completing the task. 78% of participants disagreed with the statement that the locations displayed in the recommendation view were relevant to the changes they made during the task (see fig. 6.3). The comments provided in response to the question about encountered problems offered further insight, as several participants reported that the plugin highlighted incorrect code locations following the commit. One participant noted: *"Probably only in one instance were the highlights correct. While sometimes additional code to the actually changed code was included, even cases happened where the highlighted code was not in the relevant region at all"*. This suggests that FeatRacer might experience internal issues in identifying the exact location of code changes within a commit.

The interface of the recommendation view itself received positive feedback, with 11 out of 14 participants providing favorable comments about the view. The user interface was recognized for its simplicity, clarity, and intuitive design. One participant remarked: *"A really simple design one can understand intuitively"*. The processing time for a commit was also evaluated positively, with 11 participants agreeing that the time required for the recommendation view to appear after a commit was appropriate.

Post-Task Questionnaire

After the participants completed the tasks, we asked them if they thought the plugin to be too intrusive. Except for one participant, all other participants indicated that they did not find the plugin intrusive, with eight participants strongly disagreeing

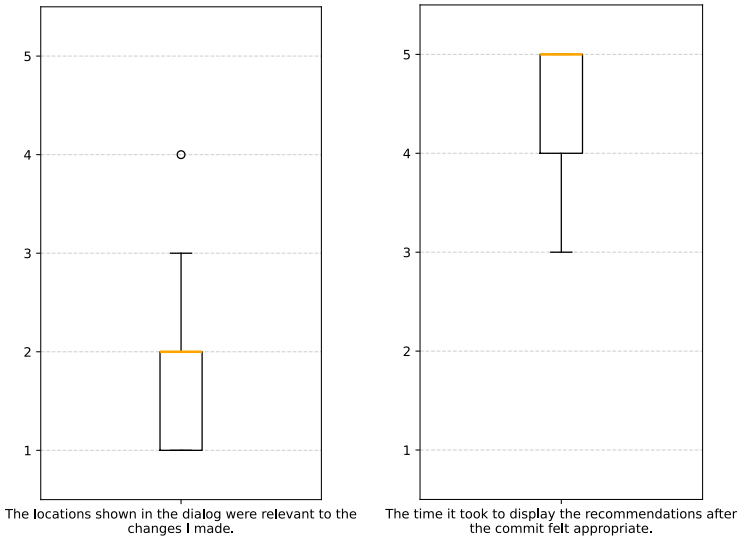


Figure 6.3: Results of the questionnaire after task 2

with the statement (see fig. 6.4). Furthermore, 12 out of the 14 participants agree with the statement *"This plugin would help me remember to annotate feature I might forget"*, suggesting that the plugin supports users in maintaining feature traceability without disrupting their workflow too much.

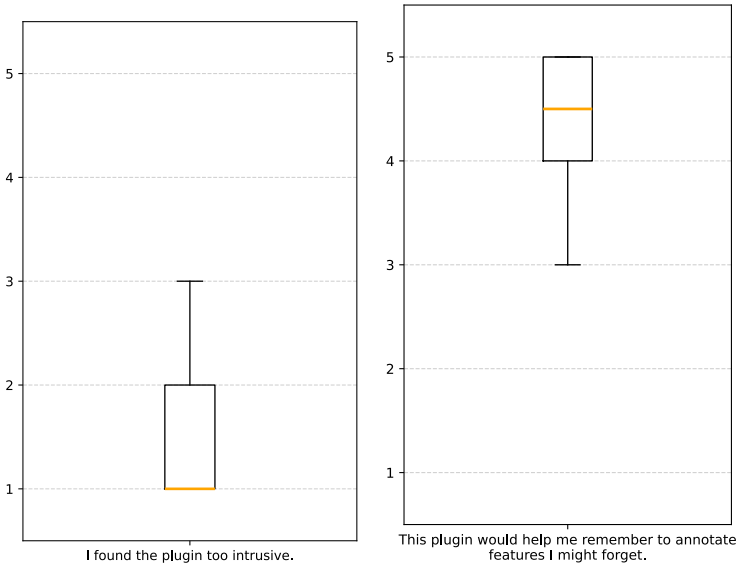


Figure 6.4: Results of the post-task questionnaire

As part of the post-task questionnaire, we asked the participants about their opinion on being reminded after a commit when parts of the code have not been annotated,

to which 13 of the 14 participants expressed a positive opinion. One participant responded with *"It can help to remember to annotate. I think it is not too bothersome to be reminded every time, since it is shown in only a notification"*, which also reinforces the observation from the previous section that participants do not find the plugin too intrusive.

When it comes to the aspects participants found the most challenging when using our plugin, we received a variety of different answers. Two participants mentioned difficulties with the error recovery process, noting that the recommendation view cannot be reopened once it has been closed. Fixes for this specific problem were also suggested, with one participant proposing to *"Allow the user to reopen it/open it at any time instead of only after committing"*. Some participants also expressed frustration regarding the lack of flexibility in the recommendation view, because the highlighted locations were not adjustable. This made it difficult to annotate the code locations returned by FeatRacer during task 1, as the highlighted locations often encompassed more code than what was actually modified in the commit.

Task completion time

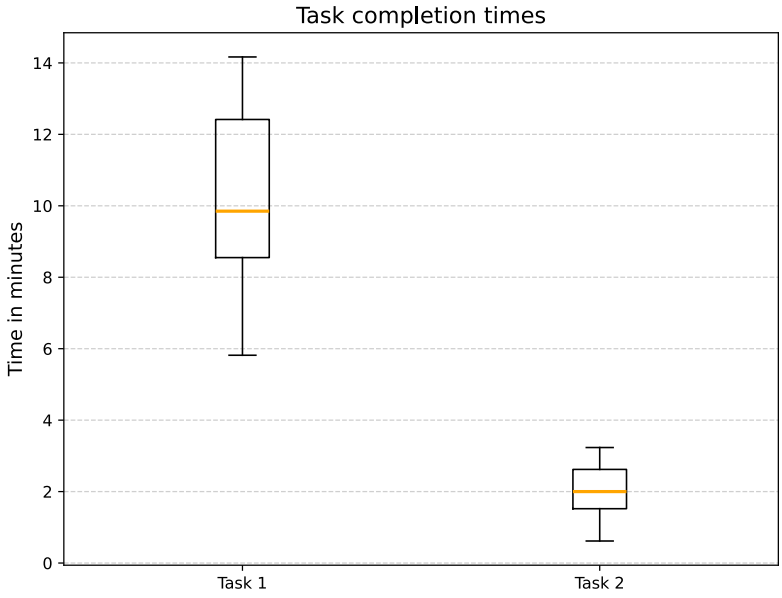


Figure 6.5: Task completion times

Figure 6.5 displays the completion times for the tasks. The mean completion time for task 1 was 10 minutes and 20 seconds, while task 2 had a mean completion time of 2 minutes and 2 seconds. For task 1 the recorded time included both the coding and the subsequent review of the recommendation view, whereas task 2 measured only

the time required to evaluate all recommendations within the test recommendation view.

Task correctness

No participant was able to correctly solve task 1 due to the aforementioned issues. Of the 14 participants, nine made no annotation at all, while four annotated the *Poison* feature at incorrect locations, and one participant incorrectly labeled a section with an unrelated feature. In task 2, participants correctly annotated 51 assets out of 70 recommendations (72%), making a total of 19 mistakes. 10 out of 14 participants made at least one mistake, with a median of one mistake per participant.

System Usability Scale

As part of the post-task questionnaire, the participants were asked ten questions based on the System Usability Scale (SUS) [8]. The results of the SUS questionnaire are presented in this section to evaluate the plugin's usability and, subsequently, to calculate and interpret the overall SUS score.

Figure 6.6 displays the results for the first three questions of the SUS questionnaire. The results indicate that the majority of participants expressed an interest to use the plugin more frequently. It is notable that all participants disagreed with the statement *"I found the plugin unnecessarily complex"*, and 86% of those disagreed strongly. The statement *"I thought the plugin was easy to use"* also received highly affirmatory feedback, with 86% of participants agreeing or strongly agreeing with it.

While most participants disagreed with the sentiment *"I think that I would need the support of a technical person to be able to use this plugin"*, a subset of three participants agreed that they might need help. The responses to the statement *"I found the various functions in this plugin were well integrated"* and *"I thought there was too much inconsistency in this plugin"* were mixed, as seen in figure 6.7.

The results yielded for the last four questions of the SUS questionnaire can be seen in figure 6.8. The statements *"I would imagine that most people would learn to use this plugin very quickly"* received positive resonance, as all participants either agreed (4 out of 14) or strongly agreed (10 out of 14). The findings for the statement *"I needed to learn a lot of things before I could get going with this plugin"* support this, with 12 out of 14 participants disagreeing that they needed to learn a lot. Additionally, a vast majority of participants did not think that the plugin was too cumbersome to use, with three participants being neutral towards that statement.

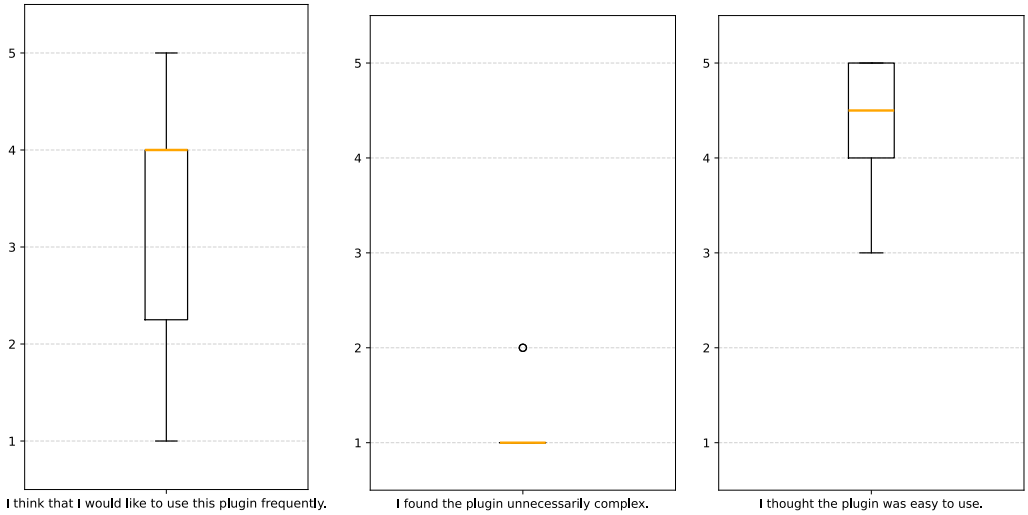


Figure 6.6: Results for SUS questions 1-3

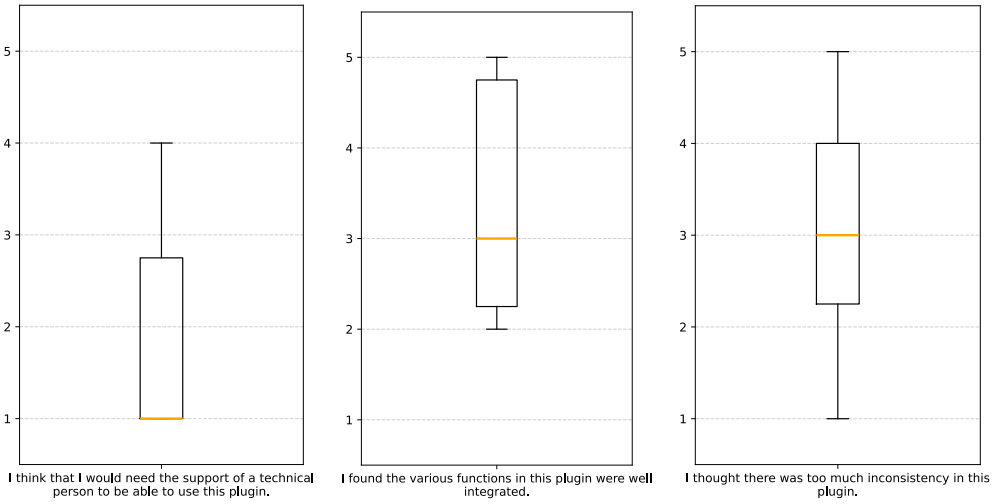


Figure 6.7: Results for SUS questions 4-6

Using the results from the SUS questionnaire, the overall SUS score was calculated. Figure 6.9 illustrates the distribution of SUS scores among participants. The mean score is 75, with the median being 76.25. The lowest recorded score was 52.5, while the highest reached 87.5.

Additionally, figure 6.10 displays different interpretations for the mean SUS score, as defined by Bangor et al. [3]. Based on this interpretation, a mean score of 75 places our plugin slightly above the *good* category on the adjective rating scale, and corresponds to a C on a school grade scale.

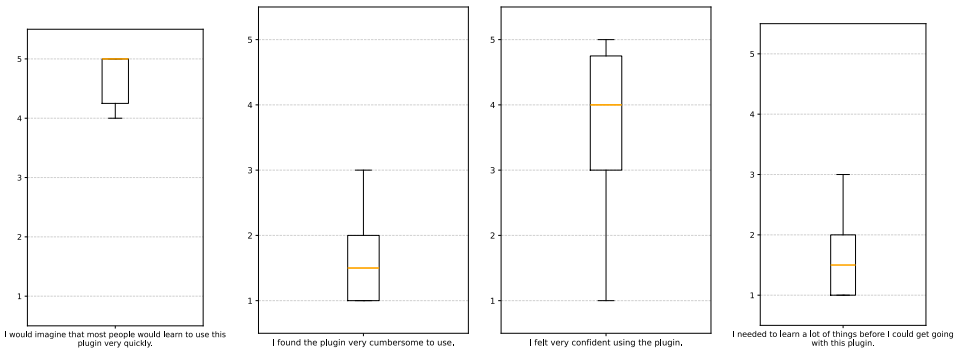


Figure 6.8: Results for SUS questions 7-10

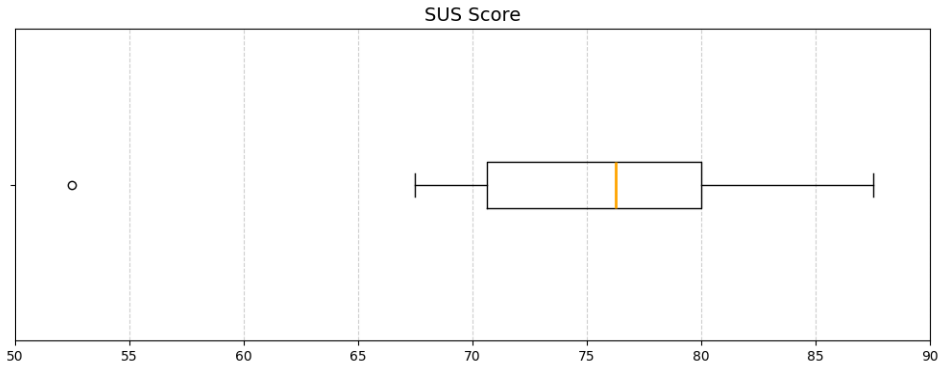


Figure 6.9: Distribution of SUS scores

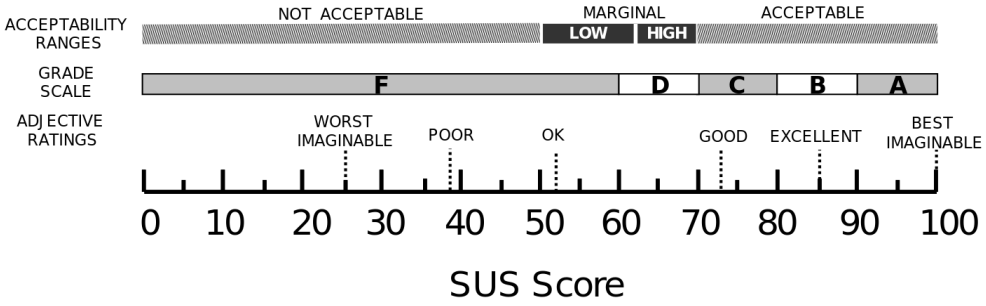


Figure 6.10: Acceptability ranges, school grading scales, and adjective ratings for the average SUS score according to Bangor et al. [3]

7 Discussion

This chapter discusses the findings from the usability study and discusses their implications in the context of the plugin's overall usability. Subsequently, we will discuss the limitations of our usability study to provide context for the implications of the results.

7.1 Implications

We invited 14 computer science students from three different universities with varying levels of experience and expertise for a usability study. The results for task 1 indicate that FeatRacer encounters internal issues when identifying the correct location of changesets, as most participants experienced issues with the task due to incorrectly highlighted code locations. The logs confirm that the plugin logic correctly highlights the locations it receives from FeatRacer, but FeatRacer outputs incorrect locations. This implies that FeatRacer's data generation module (see section 5.1.3) might store incorrect code locations into its internal database, which are subsequently included in the generated .arff test file and returned by the API. Further investigation also revealed issues in the logic responsible for recognizing annotated assets, potentially causing FeatRacer to return locations that were previously annotated.

The RepoDriller library, which FeatRacer currently leverages in its *Data Generation* module (see section 5.1.3) to extract the changeset data and store it in FeatRacer's internal database, relies on outdated versions of dependencies, with a total of 50 known vulnerabilities reported by MVNRepository, for example, a remote code execution exploit in the dependency *Log4j2*[1]. This severely limits the ability for the plugin to be adopted in a professional software development environment, as its core functionality does not work as intended and it also poses a substantial security risk. Further iterations should prioritize resolving these issues and consider a rewrite of the Data Generation module to ensure security and location accuracy for this tool.

Task 2 allowed us to demonstrate the functionality our plugin independently of FeatRacer's recommendation engine. The overall interface design of the recommendation view was received positively by most participants, who reported no difficulties

[1] https://www.cve.org/CVERecord?id=CVE-2021-44832

in understanding its usage. This suggests that even users with limited software development experience or little familiarity with embedded feature annotations were able to use the recommendation view intuitively.

Participants highlighted issues with the plugin's limited error-recovery capabilities. Currently, when a user closes the recommendation view, the recommendation data is lost. As a result, if a user accidentally closes the dialog or wants to make retrospective changes, all recommendations are lost and the code segments need to be manually recovered again. This impacts usability because a user mistake can negate the benefits of this plugin.

One potential improvement could be the creation of a dedicated sidebar panel, which stores and displays the data of past commits, allowing users to reopen the recommendation view for a given commit by clicking on its entry. This would need adjustments with the current logic of the recommendation view, as modifications in the code would shift the offset and therefore invalidate location highlighted by the view. A simpler and more immediate enhancement could be enabling users to reopen the latest recommendation view. Furthermore, usability could be improved by allowing the user to modify the highlighted locations within the view. This would make the recovery process easier in the case of an unexpected issue due to the plugin or recommender system.

For a practical adaptation into a developer's workflow, future revisions should focus on resolving the current security vulnerabilities and the incorrectly highlighted code locations. Additionally, an implementation of the mentioned suggestions would help in increasing the robustness and usability of the plugin, and conducting a second study after resolving those issues could provide further insights into how effectively this plugin nudges developers towards a well-maintained documentation of their feature locations. Despite the problems with the plugin, with our mean SUS score of 75 the usability can be considered acceptable (see fig. 6.10), and a majority of participants found the idea to remind developers on a commit helpful and not intrusive to their workflow.

7.2 Threats to Validity

Internal Validity. Differences in participants programming experience could have influenced task completion times, as they may have completed the tasks faster. Furthermore, those times were self-reported and could thereby have measurement inaccuracies. To mitigate bias, the participants were given explicit instructions on when to start and stop the timer, and we additionally supervised participants to ensure the correct measurement and reporting of the completion times. Additionally, the programming task may have been too simple to reflect real-world usage, which could have affected completion times and may have introduced a bias to the study's findings. Still, we observed that the completion times for participants had a

high standard deviation, ranging from low completion times (5:49 minutes) to high completion times (14:10 minutes), suggesting that the level of difficulty was appropriate when taking into account the varying levels in participants programming experience.

External Validity. It is important to highlight that the university context and the small sample size constrain the generalizability of the findings, as this group is not representative of the broader population of software developers and their diverse perspectives. Future studies should consider a larger and more diverse sample size to obtain more representative insights into the usability of the extension. Nevertheless, the study found initial insights into the usability of the plugin and identified several areas for potential improvement in further iterations.

Another limitation of the study's design is that it does not provide an accurate evaluation of how effective the plugin can nudge developers to maintain proper annotations over the course of project, as the study's duration was too short to simulate a realistic development workflow. In task 2, participants were instructed not to annotate the code manually, as such this is not an accurate representation of a developer naturally forgetting annotations and relying on the plugin to get reminded. A long-term usability study observing developers using the plugin over several weeks or months would provide more meaningful insight into the plugin's usability in a real-world scenario.

8 Conclusion

Embedded feature annotations are a lightweight method in retaining feature traceability within a project, which have been found to substantially reduce the cost to recover feature locations when adopted [13]. However, this benefit is lost when developers forget to consistently apply annotations, and as a consequence the feature locations must be recovered through more laborious and time-consuming recovery techniques. FeatRacer addresses this challenge by reminding developers when it predicts missing feature annotations for code locations in a commit, thus helping developers to proactively record feature locations.

This thesis aimed to extend the IntelliJ plugin HAnS with FeatRacer to enhance the usability and effectiveness of FeatRacer by integrating it into the workflow of developers, and subsequently, to evaluate the perceived usability of such an integration through a usability study. For the purpose of this integration, FeatRacer received optimizations and was refactored into a library to enable its incorporation into the plugin. The functionality of HAnS was extended with a recommendation view, allowing developers to review and automatically apply feature annotations suggested by FeatRacer directly within the IDE. The integration was designed to be generic, modular and extendable, ensuring that the plugin is not limited to FeatRacer.

We evaluated the plugin in a usability study, which provided valuable insights and feedback. While certain limitations were identified, particularly regarding the recommender system and the plugin's error-recovery, participants feedback found the plugin to be helpful and intuitive in applying feature recommendations. This is reflected by the SUS score, which indicates that the plugin has an acceptable level of usability.

Future work should address FeatRacer's current limitations, particularly the issues regarding security and the accuracy of recommendation locations. Additionally, extending the plugin with more robust error-recovery capabilities would be valuable in improving its usability. Conducting a follow-up long-term study could also yield useful insights into the long-term usability of recommender systems to assist with proactive feature location recording. We hope that this thesis can provide a solid foundation to inspire future work.

Bibliography

[1] Berima Andam, Andreas Burger, Thorsten Berger, and Michel RV Chaudron. Florida: Feature location dashboard for extracting and visualizing feature traces. In *Proceedings of the 11th International Workshop on Variability Modelling of Software-Intensive Systems*, pages 100–107, 2017.

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. Feature-oriented software product lines. 2013.

[3] Aaron Bangor, Philip Kortum, and James Miller. Determining what individual sus scores mean: Adding an adjective rating scale. *Journal of usability studies*, 4(3):114–123, 2009.

[4] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. What is a feature? a qualitative study of features in industrial software product lines. In *Proceedings of the 19th International Conference on Software Product Line*, SPLC '15, page 16–25, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336130. doi: 10.1145/2791060.2791108. URL `https://doi.org/10.1145/2791060.2791108`.

[5] Satish Tanaji Bhosale, Tejaswini Patil, and Pooja Patil. Sqlite: Light database system. *Int. J. Comput. Sci. Mob. Comput*, 44(4):882–885, 2015.

[6] A. Blackwell and M. Burnett. Applying attention investment to end-user programming. In *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, pages 28–30, 2002. doi: 10.1109/HCC.2002.1046337.

[7] Alan F Blackwell and Thomas RG Green. Investment of attention as an analytic approach to cognitive dimensions. In *PPIG*, page 5. Citeseer, 1999.

[8] John Brooke et al. Sus-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.

[9] Chris Brown and Chris Parnin. Nudging students toward better software engineering behaviors. In *2021 IEEE/ACM Third International Workshop on Bots in Software Engineering (BotSE)*, pages 11–15, 2021. doi: 10.1109/BotSE52550.2021.00010.

[10] Christopher S Corley, Kostadin Damevski, and Nicholas A Kraft. Exploring the use of deep learning for feature location. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 556–560. IEEE, 2015.

[11] Sina Entekhabi, Anton Solback, Jan-Philipp Steghöfer, and Thorsten Berger. Visualization of feature locations with the tool featuredashboard. In *Proceedings of the 23rd international systems and software product line conference-volume b*, pages 1–4, 2019.

[12] T.R.G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996. ISSN 1045-926X. doi: https://doi.org/10.1006/jvlc.1996.0009. URL `https://www.sciencedirect.com/science/article/pii/S1045926X96900099`.

[13] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. Maintaining feature traceability with embedded annotations. In *Proceedings of the 19th International Conference on Software Product Line*, SPLC '15, page 61–70, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336130. doi: 10.1145/2791060.2791107. URL `https://doi.org/10.1145/2791060.2791107`.

[14] Rainer Koschke and Jochen Quante. On dynamic feature location. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, page 86–95, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1581139934. doi: 10.1145/1101908.1101923. URL `https://doi.org/10.1145/1101908.1101923`.

[15] Jacob Krüger, Wanzi Gu, Hui Shen, Mukelabai Mukelabai, Regina Hebig, and Thorsten Berger. Towards a better understanding of software features and their characteristics: A case study of marlin. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*, VAMOS '18, page 105–112, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450353984. doi: 10.1145/3168365.3168371. URL `https://doi.org/10.1145/3168365.3168371`.

[16] Jacob Krüger, Thorsten Berger, and Thomas Leich. Features and how to find them: a survey of manual feature location. In *Software Engineering for Variability Intensive Systems*, pages 153–172. Auerbach Publications, 2019.

[17] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 105–114, 2010.

[18] Salome Maro, Anthony Anjorin, Rebekka Wohlrab, and Jan-Philipp Steghöfer. Traceability maintenance: factors and guidelines. In *Proceedings of the 31st*

IEEE/ACM International Conference on Automated Software Engineering, pages 414–425, 2016.

[19] Salome Maro, Jan-Philipp Steghöfer, and Miroslaw Staron. Software traceability in the automotive domain: Challenges and solutions. *Journal of Systems and Software*, 141:85–110, 2018. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2018.03.060. URL https://www.sciencedirect.com/science/article/pii/S0164121218300608.

[20] Johan Martinson, Herman Jansson, Mukelabai Mukelabai, Thorsten Berger, Alexandre Bergel, and Truong Ho-Quang. Hans: Ide-based editing support for embedded feature annotations. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume B*, SPLC '21, page 28–31, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384704. doi: 10.1145/3461002.3473072. URL https://doi.org/10.1145/3461002.3473072.

[21] Mukelabai Mukelabai, Kevin Hermann, Thorsten Berger, and Jan-Philipp Steghöfer. Featracer: Locating features through assisted traceability. *IEEE Transactions on Software Engineering*, 49(12):5060–5083, 2023. doi: 10.1109/TSE.2023.3324719.

[22] Chris Parnin and Spencer Rugaber. Resumption strategies for interrupted programming tasks. *Software Quality Journal*, 19(1):5–34, Mar 2011. ISSN 1573-1367. doi: 10.1007/s11219-010-9104-9. URL https://doi.org/10.1007/s11219-010-9104-9.

[23] Leonardo Passos, Rodrigo Queiroz, Mukelabai Mukelabai, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Jesus Alejandro Padilla. A study of feature scattering in the linux kernel. *IEEE Transactions on Software Engineering*, 47 (1):146–164, 2021. doi: 10.1109/TSE.2018.2884911.

[24] Denys Poshyvanyk, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, 2007. doi: 10.1109/TSE.2007.1016.

[25] Francisca Pérez, Jorge Echeverría, Raúl Lapeña, and Carlos Cetina. Comparing manual and automated feature location in conceptual models: A controlled experiment. *Information and Software Technology*, 125:106337, 2020. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2020.106337. URL https://www.sciencedirect.com/science/article/pii/S095058492030104X.

[26] Abdul Razzaq, Andrew Le Gear, Chris Exton, and Jim Buckley. An empirical assessment of baseline feature location techniques. *Empirical Software Engineering*, 25(1):266–321, 2020.

[27] Martin P. Robillard and Gail C. Murphy. Representing concerns in source code. *ACM Trans. Softw. Eng. Methodol.*, 16(1):3–es, February 2007. ISSN 1049-331X. doi: 10.1145/1189748.1189751. URL https://doi.org/10.1145/1189748.1189751.

[28] Julia Rubin and Marsha Chechik. *A Survey of Feature Location Techniques*, pages 29–58. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-36654-3. doi: 10.1007/978-3-642-36654-3_2. URL https://doi.org/10.1007/978-3-642-36654-3_2.

[29] 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, page 5–8, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375702. doi: 10.1145/3382026.3431253. URL https://doi.org/10.1145/3382026.3431253.

[30] Marcus Seiler and Barbara Paech. Documenting and exploiting software feature knowledge through tags. In *SEKE*, pages 754–777, 2019.

[31] Grigorios Tsoumakas, Eleftherios Spyromitros-Xioufis, Jozef Vilcek, and Ioannis Vlahavas. Mulan: A java library for multi-label learning. *The Journal of Machine Learning Research*, 12:2411–2414, 2011.

[32] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. How developers perform feature location tasks: a human-centric and process-oriented exploratory study. *Journal of software : evolution and process.*, 25(11), 2013. ISSN 2047-7473.

List of Figures

List of Tables

A Questionnaire

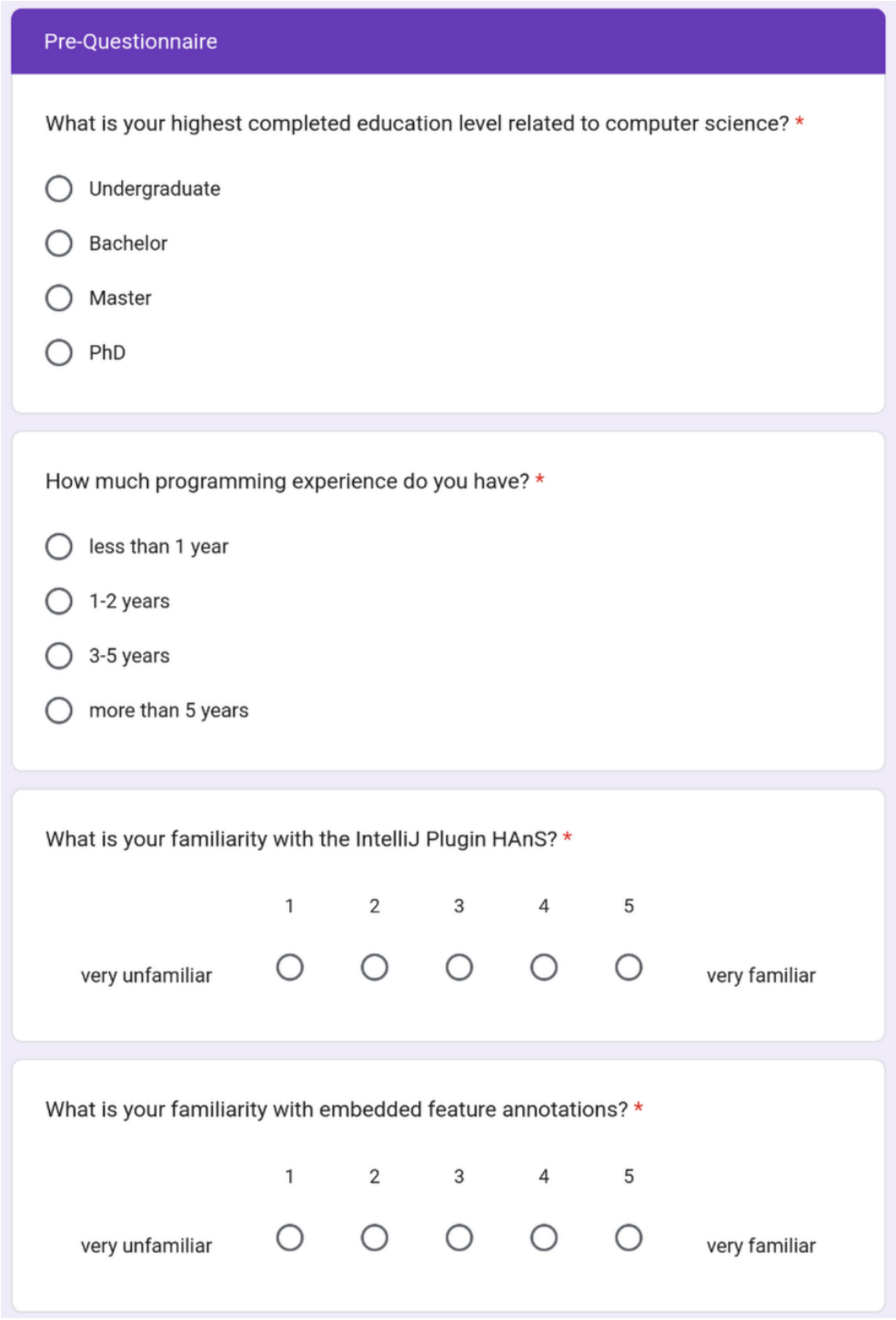


Figure A.1: Questionnaire: Demographics

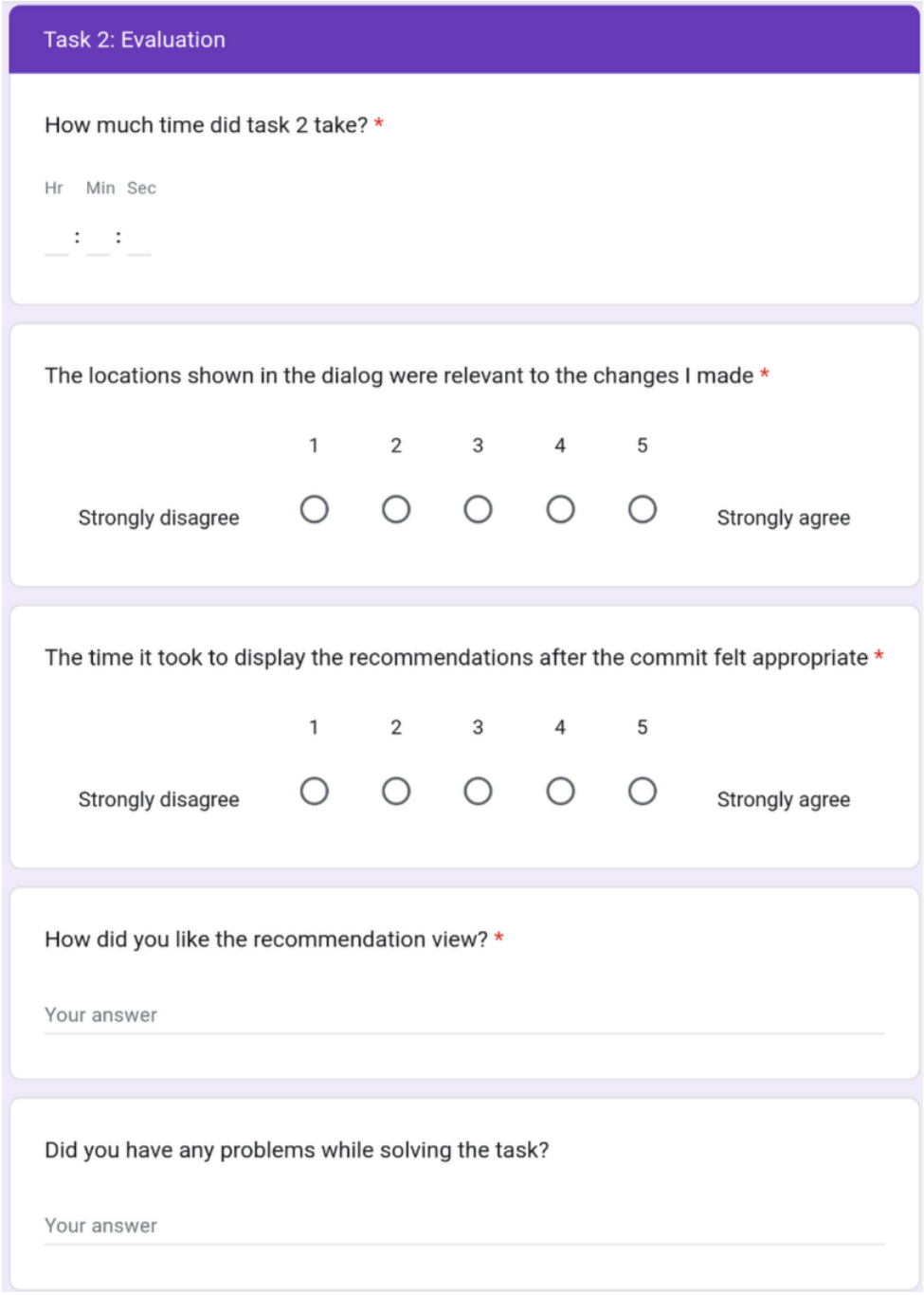


Figure A.2: Questionnaire: Evaluation after the second task

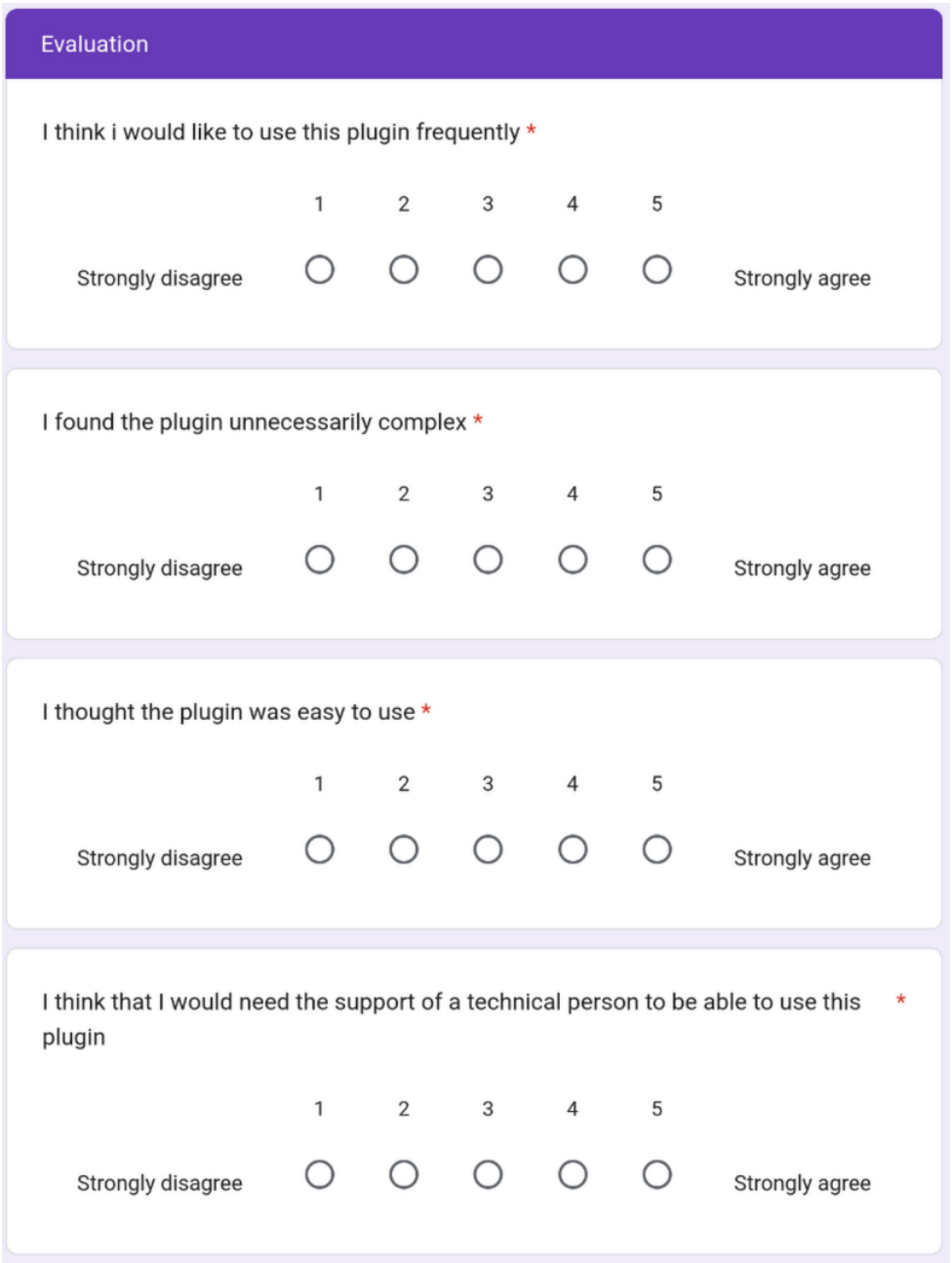


Figure A.3: Questionnaire: SUS

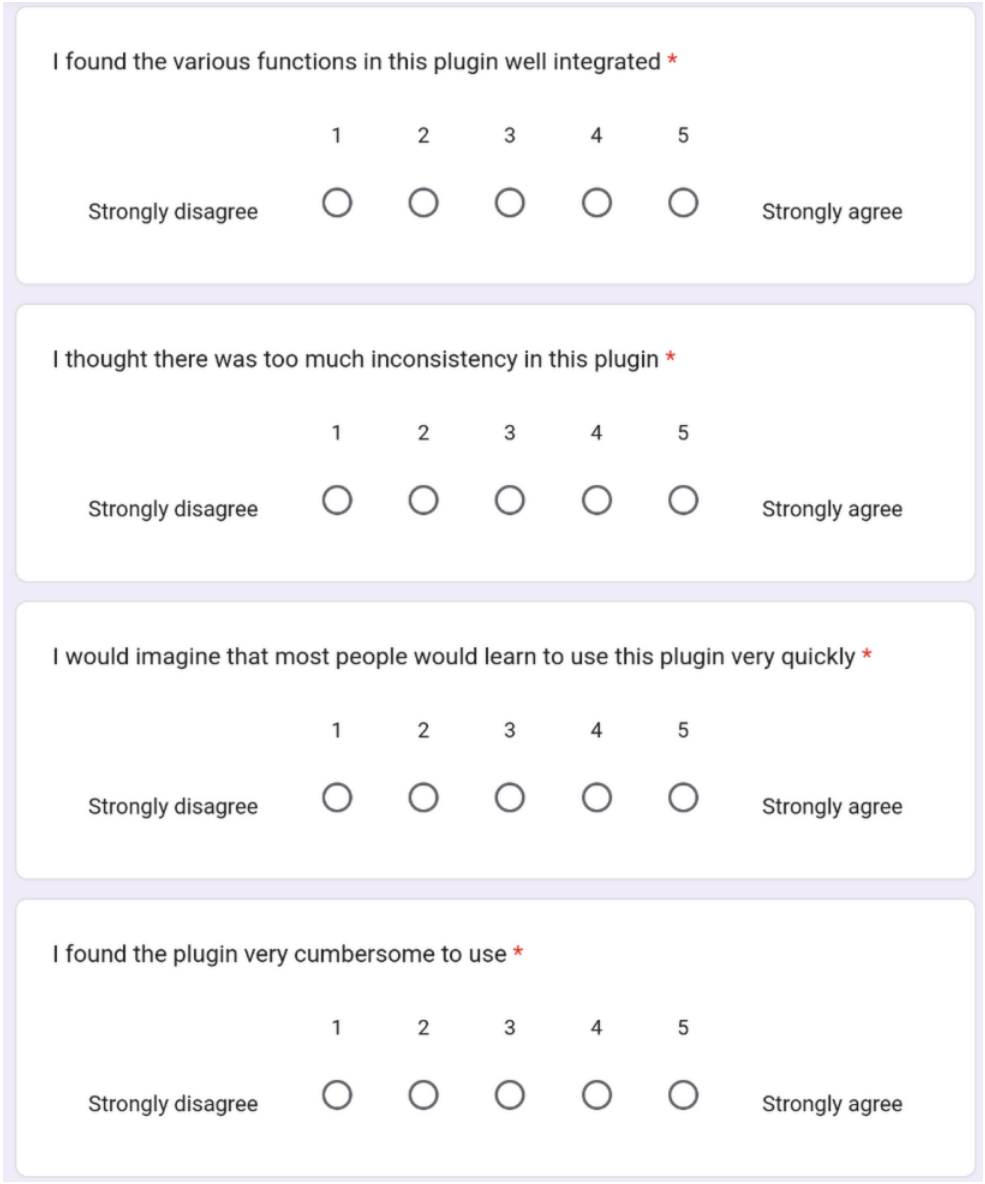


Figure A.4: Questionnaire: SUS 2

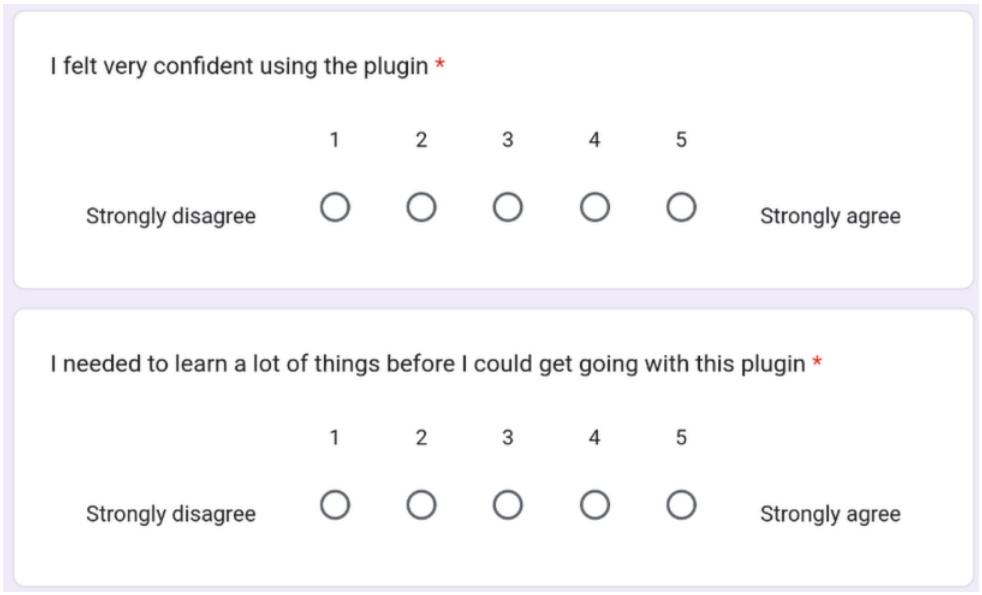


Figure A.5: Questionnaire: SUS 3

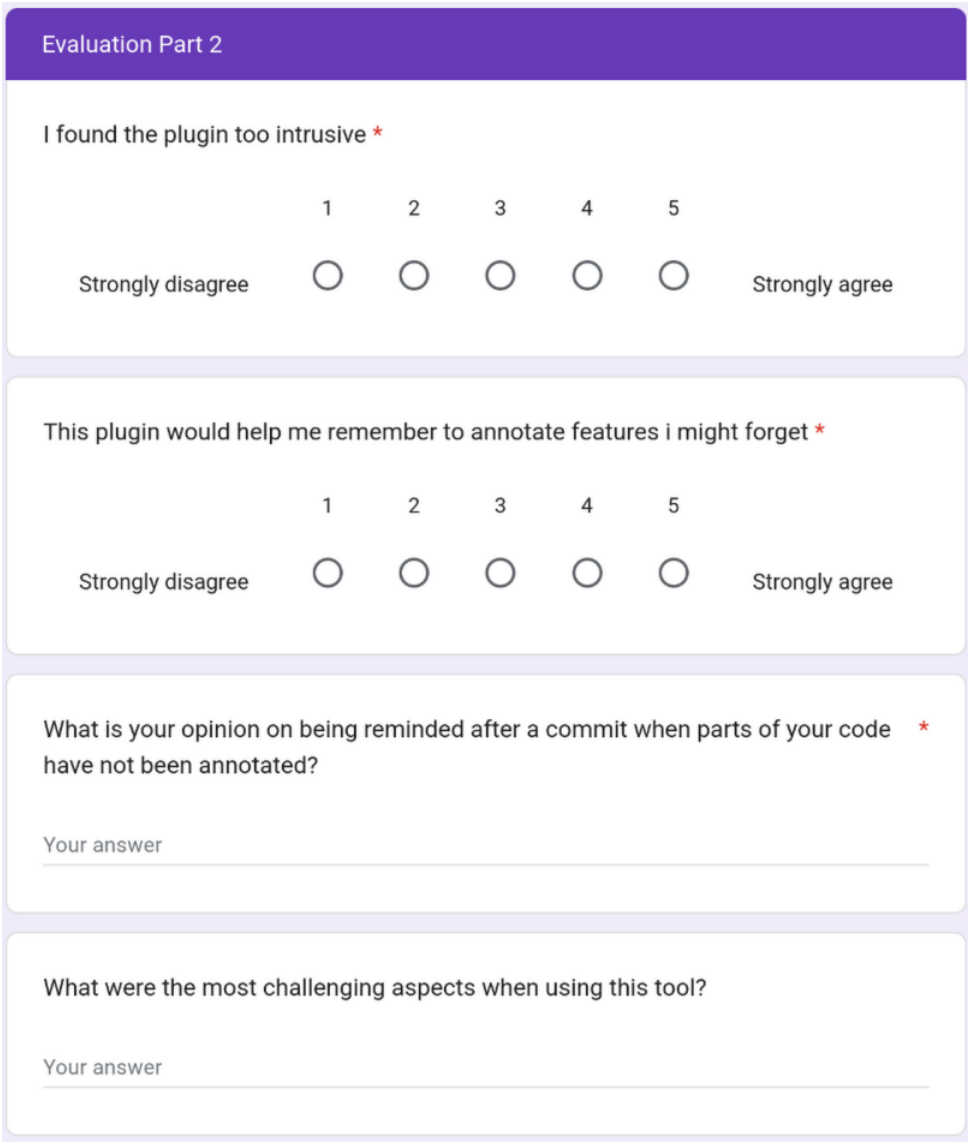


Figure A.6: Questionnaire: Post Task

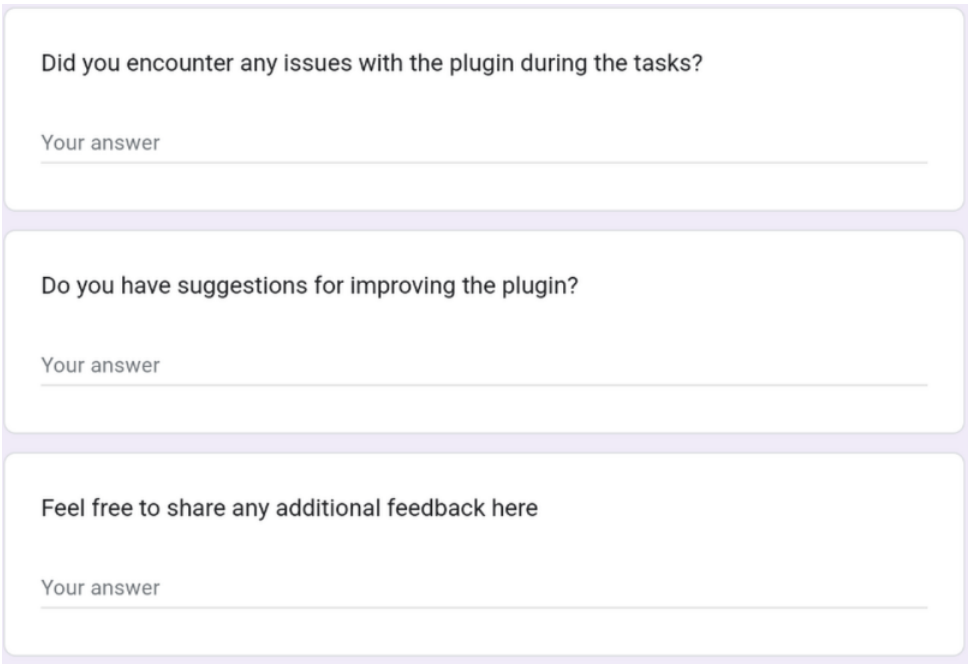


Figure A.7: Questionnaire: Post Task 2