RUHR
UNIVERSITÄT
BOCHUM

RUB

RUHR-UNIVERSITÄT BOCHUM

HAnS: Feature Visualisation

Mariana Hohashvili

Bachelor's Thesis – May 7, 2024
Chair of Software Engineering

1st Supervisor: Prof. Dr. Thorsten Berger
2nd Supervisor: Kevin Hermann, M.Sc.

Contents

Acronyms

FAXE Feature Annotation eXtraction Engine

HAnS Helping Annotate Software

JCEF Java Chromium Embedded Framework

LPQ Least-Partially-Qualified name

PSI Program Structure Interface

SUS System Usability Scale

1 Introduction

A feature is a distinct and well-understood aspect or functionality of a software system, which plays a fundamental role in specifying the system's capabilities, behavior, or data [1].

Developers often need to locate features in the code when evolving or maintaining the software. Locating features ranks among the most common tasks for developers. The essential process of feature location in software engineering is hindered by insufficient documentation, leading to a rapid decay of knowledge after development. This results in developers spending considerable time searching for feature locations. Existing automated techniques for retroactive recovery are often inaccurate and require significant effort, with identified locations being coarse-grained [2]. To address these challenges, embedded feature annotations serve as a vital solution, allowing developers to document feature locations directly within the codebase [3].

As the number of features and annotations increases, developers must quickly grasp specific feature realizations. Feature visualizations play a vital role by offering developers a clear overview of features and their locations. These visualizations enhance the understanding of fundamental feature characteristics, improving developer comprehension and productivity [2].

HAnS, or Helping Annotate Software, is an IntelliJ IDE plugin designed to streamline the recording and utilization of embedded feature annotations during software development. It addresses challenges related to locating features within code by allowing developers to map features to folders, files, code fragments, or lines. The plugin offers functionalities such as code completion, syntax highlighting, and a graphical *Feature Model View* window for easy browsing of features. HAnS supports refactoring annotations, making it easier for developers to manage and update feature information. This tool aims to enhance the efficiency and accuracy of feature location activities in software development [4].

The HAnS-Viz plugin, an extension of HAnS, was developed to provide various visualizations of the HAnS feature model and other metrics. It offers diverse visualizations,

including a tree view and treemap view of the feature model, as well as scattering and tangling views, among others [5].

The objective of this thesis is to augment HAnS-Viz by integrating functionalities for adding, editing, and removing features and annotations directly within the tree view, a crucial enhancement that would greatly elevate the tool's usability.

Additionally, another task involves integrating additional metrics into HAnS to provide deeper insights into the organization and complexity of annotated features within the project structure.

To tackle this challenge, two research questions were defined to serve as the foundation for this work.

RQ1 *How can the tree view visualization be leveraged within the HAnS-Viz plugin to offer users an intuitive method for modifying the feature model and feature annotations?*

This question entails identifying the core methods necessary for effectively modifying the feature model and associated annotations. Following this identification process, the next step involves integrating these essential actions into the HAnS plugin, thereby enhancing its functionality. The focus then transitions to the HAnS-Viz plugin, which involves developing user-friendly visualizations within the tree view interface to facilitate the execution of these actions.

RQ2 *How usable is the extended version of HAnS-Viz?*

Following the development phase, a usability study is planned to measure the effectiveness of the extended version of the plugin. Participants will perform various tasks during controlled experiments, providing feedback on usability. The aim is to assess crucial usability metrics, such as learnability, efficiency, error rate and recovery, and satisfaction. Addressing this question is vital to guide targeted enhancements that align with user needs and expectations, benefiting both current and future contributors to the plugin.

The thesis begins with an exploration of background concepts, starting with the definition of a feature, followed by an overview of feature metrics and visualizations, and then proceeds to an examination of the HAnS and HAnS-Viz plugins. The methodology employed in the study is detailed in the "Methodology" chapter, covering aspects of development, experimental objectives, survey setup, tasks, and questions. Subsequently, the "Implementation" chapter delves into the implementation of the extended HAnS and HAnS-Viz plugins, outlining metrics, additional functionalities, and advancements. The "Results" chapter presents the results of the experiment, including the evaluation of two research questions concerning tree view visualization and the usability of the extended HAnS-Viz plugin. Subsequently, the "Discussion" chapter engages in a discussion of findings, while the "Conclusion" chapter summarizes key insights and implications of the thesis.

2 Background

2.1 Feature

Features in software development encompass distinct pieces of functionality or behavior that fulfill specific requirements or user needs within a software system. They serve as identifiable components that provide value, ranging from simple functionalities to complex capabilities, aiming to deliver specific benefits or solve particular problems [6, 7, 8].

Feature traceability is crucial for establishing and maintaining transparent connections between features and assets throughout the development process. Through embedded feature annotations developers can locate features within project assets, including folders, files, and code. This traceability not only enhances development practices and decision-making but also optimizes resource allocation [3].

All features within the project are stored in the feature model, which comprises a hierarchical representation of feature names and their relationships in textual format [9]. Feature annotations, or feature mappings, are subsequently established to connect project artifacts to specific features within the feature model.

2.2 Feature Metrics and Visualization

PSI

The Program Structure Interface (PSI) is a fundamental component of IntelliJ IDEA. PSI forms the foundation of IntelliJ's abilities in code analysis and manipulation. It works by parsing the source code and representing it in a structured tree format, which consists of PSI elements (*PsiElement*). These elements include classes (*PsiFile*), methods (*PsiMethod*), comments (*PsiComment*), and custom language injections. This structured representation enables features like syntax highlighting, code completion, and error checking. Additionally, PSI enables dynamic code manipulation, facilitating

functions like code generation and refactoring. Overall, PSI plays an important role in enhancing developers' productivity by enabling efficient code navigation and manipulation within IntelliJ IDEA [10]. HAnS leverages PSI comments and custom language injections to establish feature mappings, encompassing folder, file, and code representations [11].

FAXE

FAXE is a lightweight Java library intended for the automated extraction and processing of embedded annotations from software assets based on a specific syntax. These annotations, specified within the code, help in proactively recording feature locations within the assets, aiding in tasks like feature traceability, visualization, and maintenance. FAXE's functionality includes extracting annotations recursively from sub-assets, handling syntax checks for consistency, calculating feature metrics, and facilitating feature-based partial commits, allowing developers to organize commits based on features. It offers integration options with IDEs and provides a command-line interface for interaction, making it versatile for various development environments and workflows [12]. The feature metrics and their calculation in my thesis work were inspired by FAXE's approach to extracting and processing embedded annotations in software assets.

The feature metrics integrated into HAnS include the calculation of nesting depths, which determine the maximum, minimum, and average levels of annotation nesting directly related to a feature, indicating how deep features are nested within each other or within files and folders inside the project structure. Additionally, the metrics involved counting the total number of different features within specific project elements, such as files or folders, and the total number of file annotations directly referencing each feature.

The feature metrics integrated into HAnS aimed to provide valuable insights into the organization and complexity of annotated features within the project structure. Nesting depths highlight potential hierarchical relationships and structural dependencies in the codebase. Furthermore, the total number of different features and file annotations helps in understanding the overall scope and distribution of features across project elements, aiding in better management and navigation within the feature model.

2.3 HAnS and HAnS-Viz

HAnS

HAnS is an IntelliJ IDE plugin designed to help developers record and manage feature locations within a codebase. It aims to streamline tasks such as mapping features to code, browsing features hierarchically, refactoring annotations, and reducing annotation mistakes through code completion and syntax highlighting, ultimately

enhancing the efficiency and effectiveness of feature management during software development [4]. To use HAnS effectively, developers start by creating a text file named ".feature-model" in the project's root folder, where they list all the features relevant to their project. This serves as a central repository for feature information. Then, developers can map these features to various assets such as folders, files, code fragments, or individual lines of code using specific text files named ".feature-to-folder" or ".feature-to-file" (see Fig 2.1). For instance, to map a folder to features, they add the feature names in the ".feature-to-folder" file within that folder. Similarly, to map files to features, they use the ".feature-to-file" file, listing the file names on the top line and their corresponding features on the bottom line.

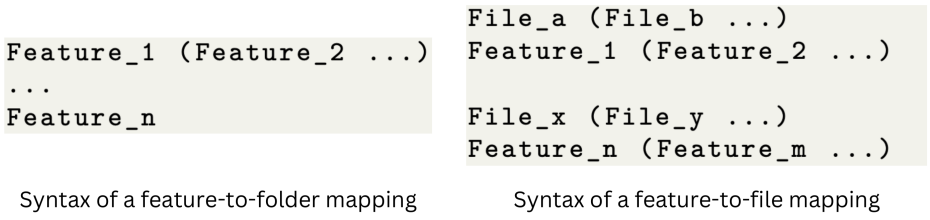

Figure 2.1: Syntax of folder and file annotations [11]

Additionally, developers can use annotations such as "&begin[...]" and "&end[...]" or "&line[...]" within Java comments to map features to code fragments or individual lines (see Fig. 2.2).

```
// &begin[Feature_1 (Feature_2 ...)]
... codeblock:
    ... (code) // &line[Feature_x (Feature_y ...)]
    ... (code)
...
// &end[Feature_1 (Feature_2 ...)]
```

Figure 2.2: Syntax of code annotations [11]

HAnS provides menus within IntelliJ IDEA to facilitate the creation and editing of these mappings, making the process more intuitive and efficient [11].

HAnS-Viz

HAnS-Viz is a plugin connected to HAnS, providing diverse visualizations to represent annotated features hierarchically, including a tree view and a treemap view of the feature model, as well as tangling and scattering visualization of features. It offers a comprehensive perspective on the structure and relationships of features, aiding in better understanding and analysis of complex software systems [5].

The *Tree View* (see Fig. 2.3), a fundamental component of HAnS-Viz, offers a hierarchical depiction of features within the feature model. This visualization assists in understanding feature relationships and their breakdown into sub-features. Each node in the Tree View corresponds to a feature in the model. Users can expand or collapse nodes to obtain a more comprehensive project overview [5].

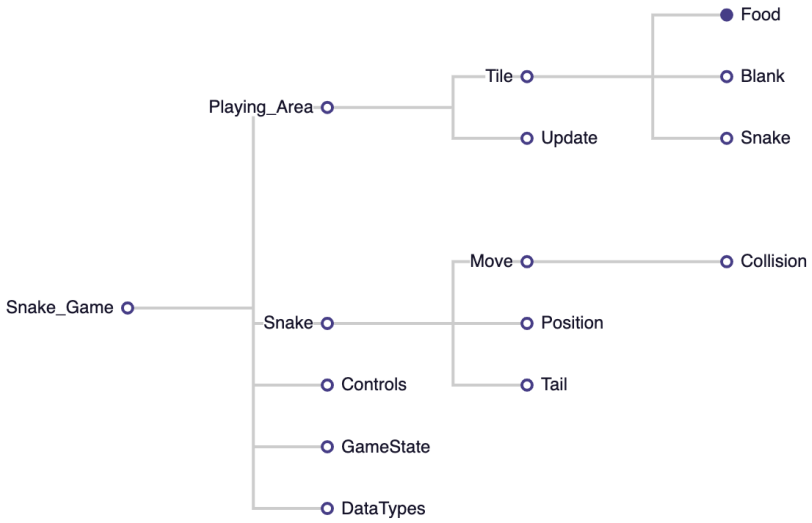

Figure 2.3: Tree View in HAnS-Viz

2.4 Related Work

This section provides an overview of existing tools relevant to the thesis topic. By introducing FeatureDashboard [13] and FeatureVista [14], it aims to place the thesis within the broader landscape of feature extraction and visualization tools. These tools contribute to the field of visualizations in feature-oriented development, providing insights into approaches used by other researchers and practitioners.

FeatureDashboard

FeatureDashboard [13] is an open-source tool that assists developers in extracting and visualizing features and their locations within software systems. It supports the use of embedded annotations added by developers during development to track feature locations. The tool provides various views, including the *Feature Dashboard* view, which displays the hierarchy of features and their locations (see Fig. 2.4).

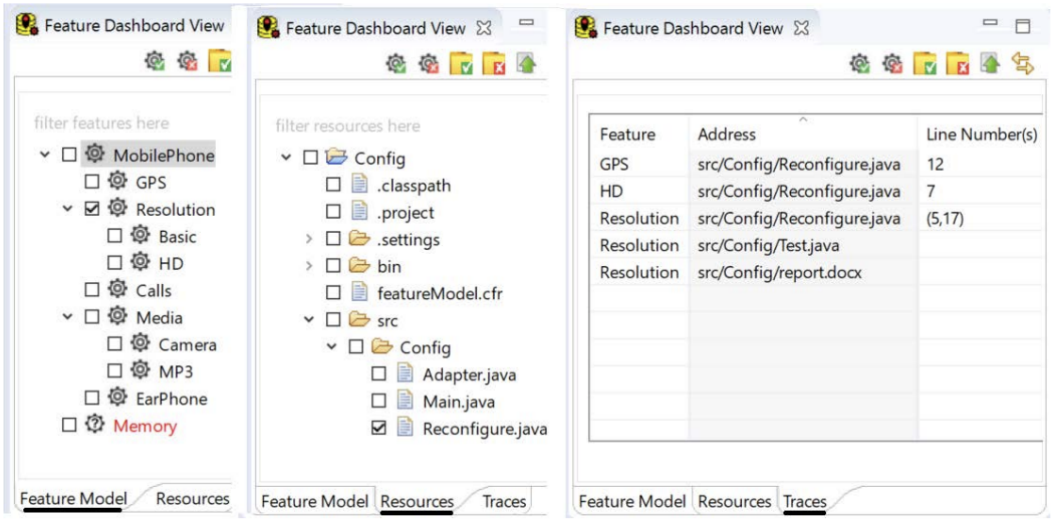

Figure 2.4: Feature Dashboard view [13]

It also offers the *Feature-to-File* and *Feature-to-Folder* views (see Fig. 2.5), illustrating the relations between features and files or folders through graphs, with green nodes representing features, blue nodes representing files or folders and edges indicating feature mappings within the project.

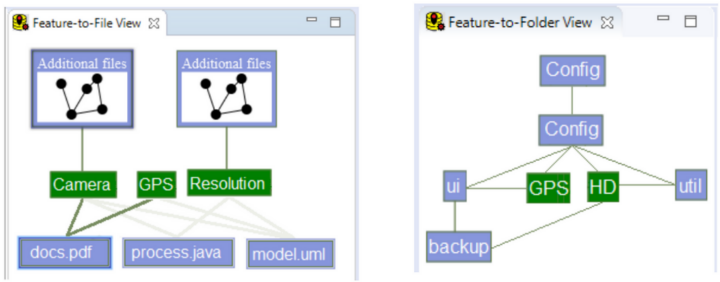

Figure 2.5: Feature-to-File and Feature-to-Folder views [13]

Another noteworthy visualization within the FeatureDashboard tool is the *Common Features* view, presenting a table that visualizes features shared across different projects or variants (see Fig. 2.6). Green cells indicate that a feature is contained in a project.

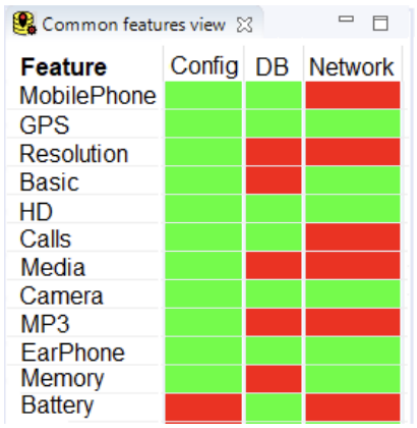

Figure 2.6: Common Features view [13]

FeatureDashboard aims to encourage continuous documentation of features early in development to facilitate understanding of the system and mitigate extensive feature location efforts [13].

FeatureVista

FeatureVista is an advanced software visualization tool designed for users to comprehensively visualize and analyze the features of a software system. It offers an array of visualization capabilities including feature model representations, class glyphs (depicted as vertical gray bars with annotated feature boxes), interactive highlighting for focusing on specific feature sets or source code units, dependency visualization, context scoping, etc. Through these visualizations, FeatureVista empowers users to explore the distribution of features within the system, distinguish their associations with project assets, and seamlessly navigate through the codebase to gain insights into its structure and functionality [14].

3 Methodology

This chapter outlines the methodology employed to address both research questions RQ1 and RQ2.

3.1 Development

Addressing the first research question, "How can the tree view visualization be leveraged within the HAnS-Viz plugin to offer users an intuitive method for modifying the feature model and feature annotations?" involved understanding the essential actions needed for modifying the feature model and annotations.

Initially, the process entailed identifying common tasks users would undertake when modifying the feature model. Some of these tasks were partially implemented in HAnS as part of the *Feature Model View*, including adding a single feature, renaming a feature, and deleting a feature. However, these implementations encountered several issues.

Firstly, renaming and adding features in the feature model failed to update the LPQs and feature annotations of other features with the same name, potentially leading to clashes. An LPQ, or Least-Partially-Qualified name, is a string formed by extending each feature with its ancestor within a feature model hierarchy to ensure unique reference, with components separated by "::" for annotation clarity [11].

Additionally, deleting a feature only removed the feature name from the feature model, leaving all associated annotations in the code without any feature association, making them inaccessible. Furthermore, the *Feature Model View* visualization is restricted, as users can only perceive features located in the top layer of the feature model hierarchy. This limitation hindered users' ability to fully grasp the hierarchical structure of the feature model.

The decision was made to incorporate all fundamental actions for modifying the feature model and feature annotations into the tree view of the HAnS-Viz plugin, leveraging its hierarchical feature model representation.

Firstly, the functionalities of adding and renaming features were incorporated, alongside modifications to ensure that features with the same name and their annotations were appropriately updated.

Another common task of developers involves removing features. Two scenarios were identified for deleting features. In the first scenario, the user intends to delete a feature along with its annotations but wishes to retain the code inside those annotations to remap it to a new feature. This scenario arises when the user desires to update the feature structure while preserving the functionality implemented within the code. Conversely, the second scenario involves the user wanting to delete a feature along with its annotations and the code contained within those annotations. This action effectively removes the entire functionality associated with the feature, which may be necessary in situations where the feature is no longer required in the project.

Finally, the action of moving features was identified, which becomes necessary when a user intends to restructure the hierarchy of the feature model, such as adding a new common parent for two existing features.

3.2 Experiment

This section addresses RQ2 and outlines the experiment created to evaluate the usability of HAnS-Viz. Different participants engaged in testing the plugin, and their input was utilized to evaluate its usability.

3.2.1 Objectives and Usability Metrics

The objective of this experiment was to assess the usability of the extended HAnS-Viz plugin. Usability served as the dependent variable, representing the effectiveness of the tool in facilitating feature-oriented development processes. The independent variable was the HAnS-Viz plugin itself.

In this experiment, various aspects of usability were assessed. *Learnability* was determined by evaluating how easy it was for users without prior knowledge of the plugin to perform a task with the tool. *Efficiency* was measured by assessing how quickly users could solve the tasks. *Error Rate and Recovery* focused on whether users could recover from errors they made while using the tool. This was observed through error counts in screen recordings. Furthermore, *Satisfaction* was evaluated based on the subjective level of contentment with the plugin's usability, determined using the SUS

score. Lastly, *Memorability*, or users' ability to recall interactions with the plugin over time, was omitted as it exceeded the thesis scope.

3.2.2 Setup

Participants were briefed on the motivations behind the creation of HAnS and HAnS-Viz, as well as the challenges associated with codebases lacking feature tracing. They received an overview of features, feature tracing, and feature annotations. Then, an introduction to the HAnS plugin was provided, covering its core functionalities such as creating features, mapping them, and locating them within a project.

Subsequently, an introduction to HAnS-Viz and its Tree View was given, along with an explanation of the toolbar functionalities and how they operate. Users were then provided with a link to the Google Forms document containing task explanations and a questionnaire. They were also given access to the GitHub repository [1] for installing both plugins, as well as the project called Snake [2], a Java-based small-scale game that contains feature annotations. This project was utilized to carry out the tasks and evaluate the functionalities of the plugin.

Additionally, users were requested to install and conduct a screen recording of their task completion process. This measure was implemented to capture the time required to complete the tasks and to assess the number of errors as a usability metric.

There were five tasks structured to encourage users to interact with the toolbar. Following the exploration of each functionality, users were required to answer several questions. Lastly, a SUS questionnaire was employed to evaluate the overall usability of the plugin.

3.2.3 Tasks

All participants were given tasks to actively utilize the toolbar within the Tree View. These tasks aimed to familiarize them with the functionalities, and implications of each action within the toolbar. There were two types of tasks assigned to users during the evaluation.

The first type involved simple observation tasks, where users were instructed to perform actions such as adding a feature to the tree view, renaming a feature within the tree view, or moving a feature in the tree view, and then observe the resulting changes in the feature model.

[1] https://github.com/hohashvili/snake-game-experiment
[2] https://github.com/johmara/Snake

The second type of task required users to navigate to a specific file, examine particular feature annotations (and the corresponding code), perform a specified action (such as deleting a feature along with its code or annotations), and then return to the file to observe the changes in the code.

For tasks involving the "deleting feature with code" functionality, users were also prompted to interact with a modal window and make modifications to the code within it.

3.2.4 Questions

Following each section, users were prompted with several questions aimed at assessing key usability metrics including learnability, efficiency, number of errors, and satisfaction. Once all tasks were completed, a survey for the entire toolbar was conducted. The questionnaire is included in the Appendix for reference.

SUS

The toolbar's overall usability was assessed using the System Usability Scale (SUS), a commonly employed tool for evaluating the usability of various interactive systems including software and websites. The SUS comprises ten statements, each rated on a five-point scale ranging from "Strongly Disagree" to "Strongly Agree". These statements consist of five positive and five negative ones, arranged alternately [15]. They can then be used to form an overall rating.

4 Implementation

This chapter presents an overview of the implementation details for both the HAnS and HAnS-Viz plugins. It emphasizes the significant improvements incorporated into these plugins, such as the addition of supplementary metrics in HAnS to enhance project structure comprehension. Additionally, it explores the integration of functionalities allowing for the modification of the feature model and annotations within HAnS, followed by their integration into the new toolbar in HAnS-Viz, aimed at enhancing user interaction. The chapter also discusses the methodologies employed in implementing various features and functionalities, offering insights into the technical aspects of plugin development.

4.1 HAnS

In the present version of HAnS, certain metrics essential for comprehending the structure and complexity of the codebase and its features remain uncomputed. These metrics include the maximum, minimum, and average nesting depths of features, which delineate the hierarchical organization of features within the project structure. Specifically, the depth of a feature is determined by factors such as its position within the project's directory hierarchy—each folder and file incrementally increases this depth by one. Additionally, the presence of code annotations further contributes to the depth, with each annotation augmenting it by 1.

HAnS also lacks some further metrics that could provide a deeper understanding and analysis of the codebase's organization and relationships between features and files. For instance, it does not compute the number of features directly referenced in annotations within a folder or any of its subfolders. Additionally, it does not calculate the number of annotated files (or the number of mappings between features and files).

4.1.1 Metrics

To compute the mentioned metrics, the *FullFeatureTree* class was developed, which serves as a representation of the project in a tree structure. In this structure, each node corresponds to either a file or a folder within the project, with leaves representing code annotations within the parent file node. Every node maintains a list of features mapped to it. During the construction of the tree, the project's folder structure is recursively traversed, and information such as depths and lists of mapped features are added to each node.

To access feature LPQs within folder or file annotations, the *PsiRecursiveElement-WalkingVisitor* provided by the IntelliJ PSI package was utilized. This visitor pattern facilitates traversing the PSI tree and extracting relevant information. Additionally, to access feature LPQs within code annotations, the Custom Language Injection manager (*InjectedLanguageManager*) was employed. This manager allows for handling custom language injections, enabling the retrieval of LPQs from code annotations, which are not represented as *PsiElements*, but as custom language injections within the code.

To compute the metric "number of annotated files", another recursive traversal of the existing feature tree is performed. During this traversal, nodes of type "FILE" that have at least one mapped feature associated with them are counted. Similarly, for the metric "number of features", each node is traversed and the number of features stored in the list associated with that node is counted.

4.1.2 Further Functionalities

In order to enhance the HAnS-Viz plugin with additional capabilities for modifying the feature model and feature annotations, the corresponding methods within the HAnS plugin have been implemented. This involved extending the utility class for the *.feature-model* file (*FeatureModelPsiImplUtil*) as well as the utility class for accessing feature references (*FeatureReferenceUtil*).

Adding and renaming features

When adding or renaming features, the PSI structure of the feature model file undergoes alterations. Upon addition, a new PSI element is inserted into the tree. In the case of renaming, the PSI element with the old name is substituted with the updated one, and all associated annotations are accordingly updated to reflect the new feature name. In both cases, if the feature name of the new element matches any of the children of the parent feature, the operation will be cancelled to prevent conflicts.

Subsequently, a reference search is conducted to identify any potential clashes between the newly added feature name and all the existing ones. If such clashes occur, the

annotations of the conflicting features are updated to align with their new, unique LPQ.

Moving features

Likewise, when moving features, the PSI element of the feature to be moved along with all its children are extracted from the feature model and inserted at the new parent location. The parent feature must not contain any direct children with the same feature name as the feature being moved to avoid conflicts.

Deleting features with annotations

Deleting a feature and all of its annotations entails several steps. Initially, a reference search is conducted to locate and store all the file, folder, and code annotations associated with the target feature and its children. Subsequently, these PSI elements and language injections are removed from the project. Following this, the feature and its children are removed from the feature model. Finally, another reference search is performed to identify features with the same name as any of the removed features. If such features are found, their annotations are updated to match a new, unique LPQ, typically a shorter one.

Deleting features with code

When deleting features with code, two different execution paths emerge. The first scenario occurs when no tangled features are associated with the target feature or any of its children. In this case, the process mirrors that of deleting features with annotations. Feature and file mappings are deleted, along with in-file annotations containing code enclosed within the *&begin* and *&end* markers, or on the same line as the *&line* marker. Subsequently, the *.feature-model* file is updated, along with any remaining features sharing the same name.

In the second scenario, intertwinements with other features are present. For instance, *FeatureA* may have a code annotation in *File1.java*, while *FeatureB* may have a file mapping with *File1.java*. As a result, *FeatureA* and *FeatureB* become tangled. Due to this entanglement, deleting any of the annotations becomes problematic, as it would disrupt the logical feature mapping structure. To address this issue, a modal window was developed to display two tangled features and illustrate their relationship to each other (see Fig. 4.1).

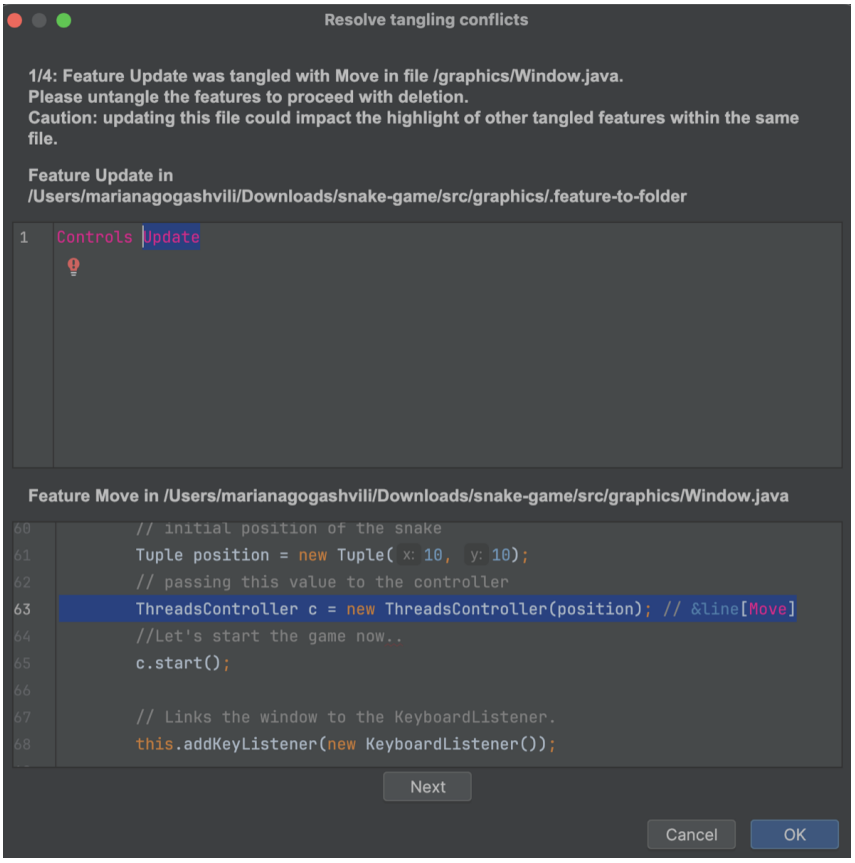

Figure 4.1: Example of the modal window for deleting tangled features with code

When a user attempts to delete a feature with associated code, this modal window appears, providing the user with the opportunity to untangle the features. For instance, the user can achieve this by deleting specific annotations or by rearranging them. This interactive approach allows users to manage feature dependencies effectively and maintain the integrity of the feature mapping structure. Once the user has finished untangling the features, they can click the *OK* button. This action triggers the process of searching for tangled features and their respective locations once again. If no tangled features are found, the feature and its children are deleted, following the same process as in the first scenario.

4.2 HAnS-Viz

Following the extension of HAnS, HAnS-Viz underwent modifications to incorporate a new toolbar featuring all the associated actions (refer to Fig 4.2). This toolbar is accessible directly from the tree view.

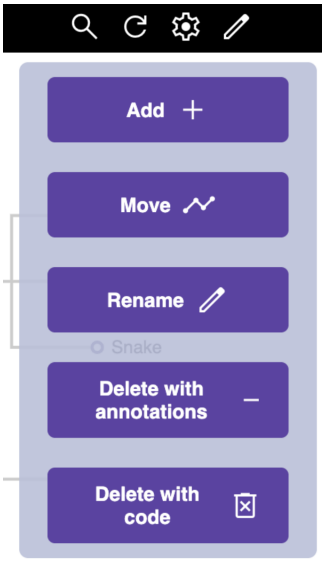

Figure 4.2: HAnS-Viz Toolbar

Since HAnS-Viz was developed utilizing the JCEF browser, all elements within the window, including the toolbar, were implemented using HTML for structure, CSS for styling, and JavaScript for interaction with the controller and task processing [5]. To execute a specific action, the user selects it from the toolbar and then clicks on the corresponding feature. Subsequently, the user will be prompted with either an input field or a confirmation pop-up window, depending on the action selected (see Fig. 4.3).

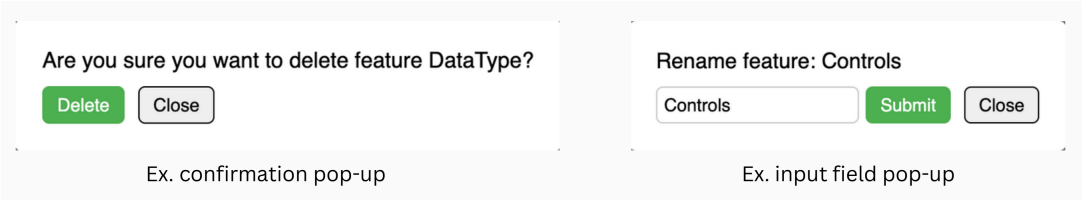

Figure 4.3: Examples of popups

After the user interacts with the interface, a request containing the specified action and data is sent to the back-end. This request is then executed by HAnS to perform the desired operation, utilizing the methods described in Section 4.1.2. Subsequently, the updated data is retrieved from the controller and the tree view is refreshed to display the latest version.

5 Results

In this chapter, I will first address RQ2, focusing on analyzing the usability of individual actions within the toolbar and the overall usability of the plugin. Additionally, I will present subjective feedback gathered from the participants. Lastly, I will evaluate research questions RQ1 and RQ2 as a whole.

5.1 Experiment

The participants in the experiment constituted a diverse group of 10 individuals, with more than two-thirds (70%) being either PhD students or PhD holders. The remainder was divided between Bachelor's and Master's students, all specializing in computer science (see Fig. 5.1).

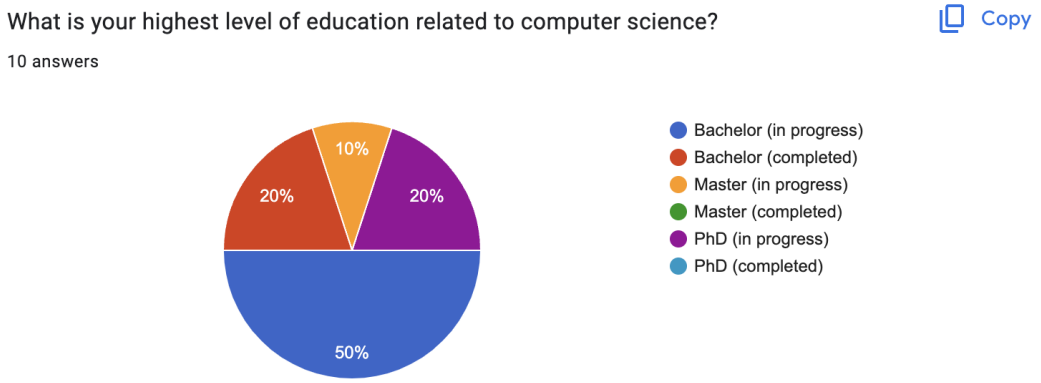

Figure 5.1: Educational levels among participants

Regarding software development experience level, half of the participants had less than one year of experience, while the other half was split between those with 1-2

years and 3-5 years, with one participant having over 5 years of experience (see Fig. 5.2).

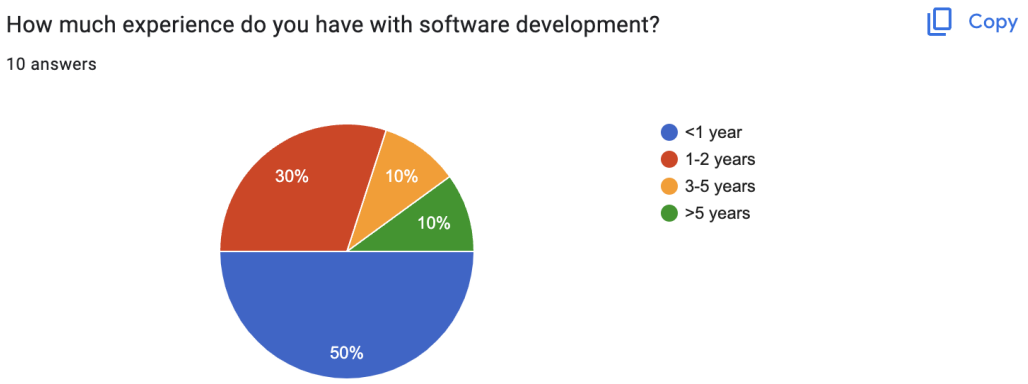

Figure 5.2: Software development experience among participants

Notably, over half of the participants had previously used the HAnS plugin (70%), which is likely attributable to the university environment in which the survey was conducted.

During the experiment, participants were assigned tasks that involved trying out each action in the toolbar and subsequently providing feedback on the usability of each action, as well as the overall usability of the plugin.

Following the completion of the tasks, participants were asked to respond to statements from the questionnaire, with their responses ranging from 1 to 5, corresponding to Strongly Disagree (1), Disagree (2), Neither Agree nor Disagree (3), Agree (4), and Strongly Agree (5). In the figures presented below, the Y-axis illustrates the number of participants who responded accordingly, while the X-axis indicates the corresponding statements from the questionnaire.

Additionally, participants were requested to record their task completions and upload the recordings to the questionnaire. This facilitated the evaluation of completion time and the identification of any mistakes or unwanted behaviors during the task execution.

Adding a feature

The evaluation results regarding the addition of features revealed promising insights. A significant majority (80%) of participants strongly agreed that adding a feature was easy, reflecting a high level of user-friendliness. One participant commented, *"... adding took around 1 minute without any problems coming up... it was easy to do*

and understand... ". This feedback highlights the simplicity of the feature addition process as perceived by the participants.

Similarly, an equal proportion found the tasks easy to solve quickly, suggesting efficient usability (80%). However, regarding error recovery, responses varied: 20% strongly agreed, 40% agreed, and 30% neither agreed nor disagreed, indicating room for improvement in error management. Nonetheless, a substantial portion of participants (80%) agreed that adding a feature was satisfying, underscoring a positive overall experience with the feature addition process (see Fig. 5.3).

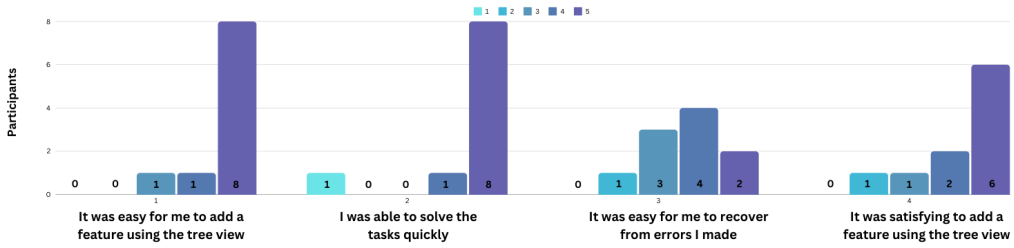

Figure 5.3: Evaluation results for adding a feature

Renaming a feature

The evaluation results for renaming features exhibited consistent trends. Most participants (80%) strongly agreed that renaming a feature was straightforward, indicating a high level of ease in the process. Furthermore, an overwhelming majority (90%) found the tasks quick to solve. 80% of the participants found error resolution to be easy. This was likely due to the fact that either they made minimal errors or they could rectify mistakes simply by repeating the renaming operation. Also, a significant portion of participants (90%) agreed that renaming a feature was satisfying, highlighting a positive overall sentiment towards the renaming process (see Fig. 5.4).

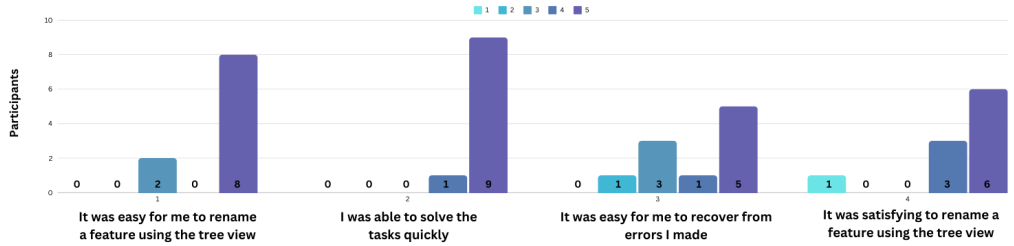

Figure 5.4: Evaluation results for renaming a feature

Moving a feature

When it came to moving a feature, 80% of participants found it easy, while 90% were able to complete the task quickly. However, only 60% reported being able to recover from their mistakes. Nonetheless, a significant majority (90%) still found the process satisfying. One participant did not find the process satisfying nor could recover from errors. This may be attributed to the fact, that they lack prior experience with HAnS and HAnS-Viz, leading to difficulty in quickly grasping the concept of feature model hierarchy (see Fig. 5.5).

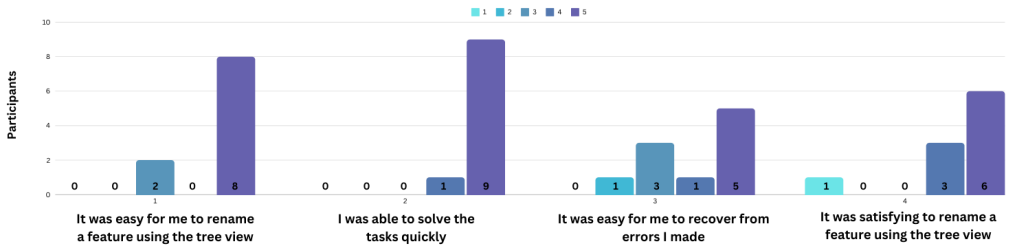

Figure 5.5: Evaluation results for moving a feature

Deleting a feature with annotations

In the case of deleting a feature with annotations, 90% of participants found it easy, and all were able to solve the tasks quickly. Additionally, 90% reported finding the overall experience satisfying. One participant mentioned, *"...deleting with annotations was the easiest task so far..."*, which underscores the smoothness and efficiency of the deletion process with annotations. However, only 40% could recover from errors quickly. This limitation is likely attributed to the absence of an undo feature, making it challenging to rectify accidental deletions. If the user mistakenly deletes the wrong feature, there is essentially no recourse available for recovery (see Fig. 5.6).

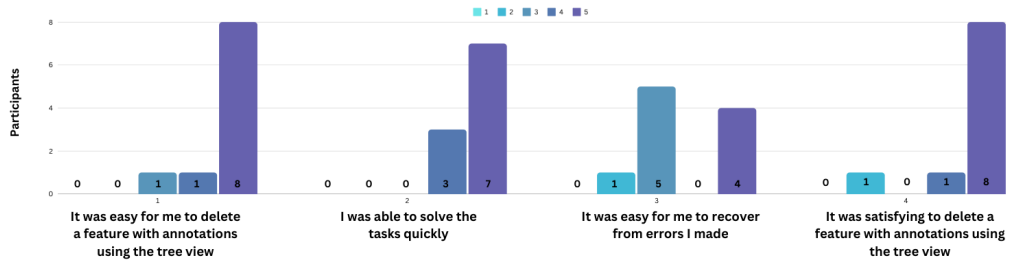

Figure 5.6: Evaluation results for deleting a feature with annotations

Deleting a feature with code

In the case of deleting a feature with code, 100% of participants found it easy to accomplish, while 70% reported finding it satisfying. However, only 60% were able to solve the task quickly, likely due to the additional steps required, such as investigating the modal window and performing extra actions, such as deleting several lines of code. Furthermore, only 40% could recover from errors, likely for similar reasons as with deleting features with annotations, namely the absence of an undo functionality (see Fig. 5.7).

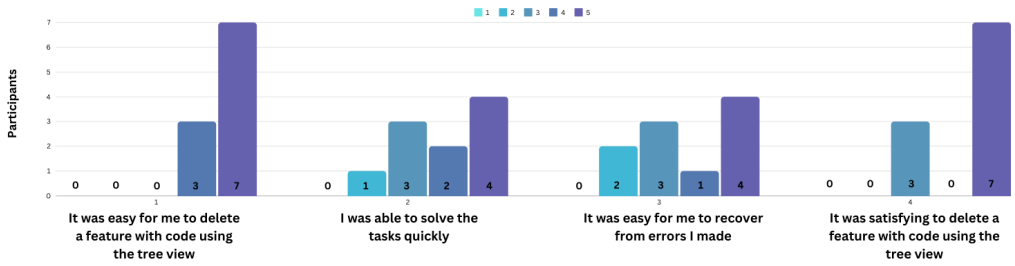

Figure 5.7: Evaluation results for deleting a feature with code

Evaluation of User Interaction and Feedback

Upon reviewing the screen recordings, the number of errors made by users and any undesirable user behaviors were assessed.

Three participants encountered confusion with the modal window during the deletion process involving code. They've attempted to click the OK button multiple times and were puzzled when the same window reappeared. Additionally, one participant misunderstood the task outlined in the questionnaire, specifically regarding the deletion of two lines with code annotations. Instead of utilizing the modal window, they attempted to delete code directly from the file. Moreover, one participant provided feedback expressing difficulty in resolving tangling conflicts, stating, *"... without the help of the questionnaire I think, I wouldn't be able to solve the tangling conflict ... I had some problems understanding the message within the dialog without the questionnaire."*

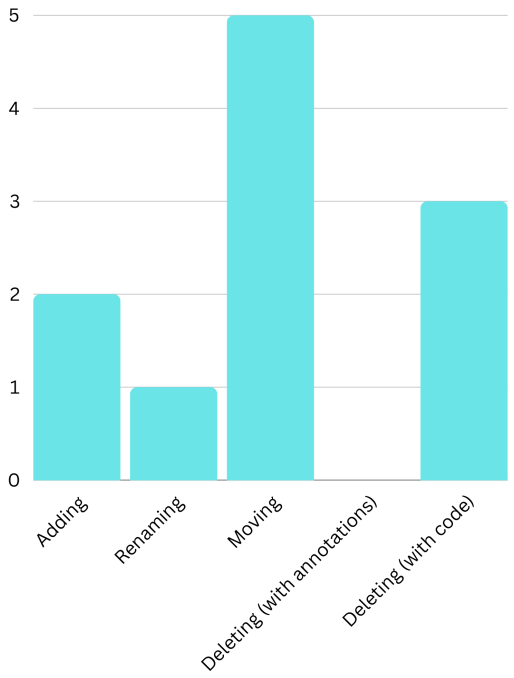

Figure 5.8: Total number of mistakes made for each task

The majority of errors (see Fig. 5.8) occurred during the process of moving a feature. Two participants encountered confusion with collapsed features and struggled to locate the required feature by solely relying on the tree view. Additionally, another issue arose when the same users attempted to utilize the search functionality, which failed to display features that were collapsed in the tree view. As a result, participants skipped the task altogether.

Another common issue observed among users is the lack of highlighting when clicking on features, leading to confusion about whether the click was registered. This lack of visual feedback can make it unclear whether the desired feature has been selected. One user provided feedback, stating: *"...it took me some time to see that I already selected the node I wanted to move, so I clicked again on it because I thought it wasn't selected."*

Furthermore, there were two mistakes observed when adding a feature. This could be attributed to the fact that it was the first task in the survey, and users may not have been fully acquainted with the toolbar at that point.

Additionally, I've noticed some conflicts arising from double-clicking a feature, which triggers the scattering information, and single-clicking a feature to select it for performing certain action from the toolbar.

Lastly, one participant suggested that the current process of opening the toolbar and selecting the feature each time to modify the tree view is verbose. They proposed that there should potentially be a way to perform an action directly from the node itself.

Completion time

On average, it took 12 minutes and 52 seconds to complete a survey. Users who had prior experience with HAnS and HAnS-Viz demonstrated faster completion times.

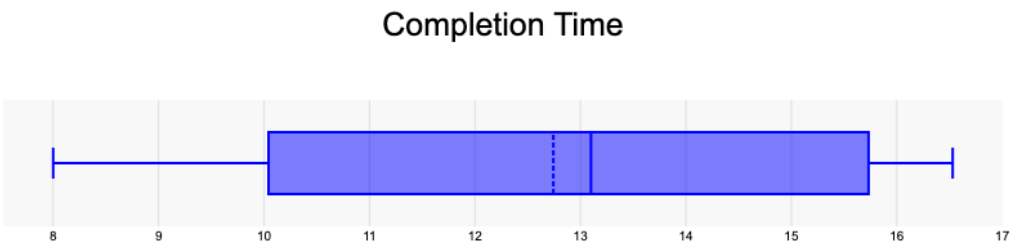

Figure 5.9: Task completion time

SUS Score Evaluation

The SUS score is derived from the SUS questionnaire designed by Brooke [16]. As depicted in Figure 5.10, individual participants' SUS scores range from 47.5 to 97.5, with most falling within the range of 82.5 to 97.5. The average SUS score is 83.25.

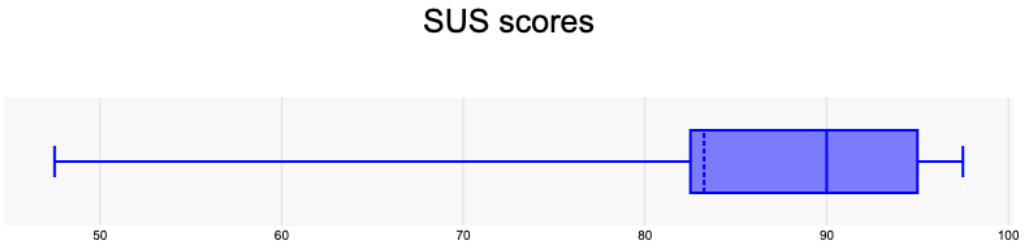

Figure 5.10: SUS scores

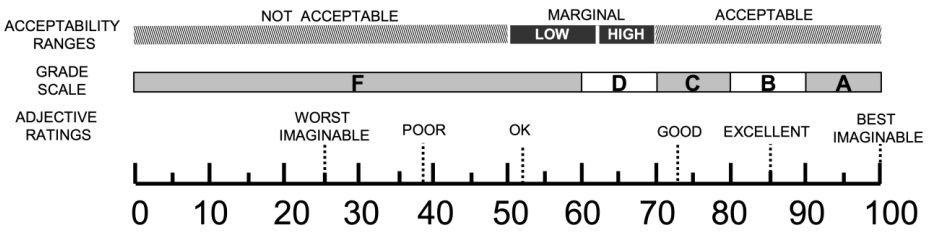

Figure 5.11: Grade rankings [15] of SUS Scores

In Figure 5.11, the grade rankings provided by Bangor, Kortum, and Miller [15] are presented to facilitate SUS score interpretation. The SUS score falls between good and excellent, akin to a grade of B in school ratings, strongly indicating good usability.

5.2 Evaluating research questions

In this section, I will evaluate the research questions posed in the thesis. These investigate the effectiveness and usability of the extended functionalities within the HAnS and HAnS-Viz plugins. Specifically, the investigation centers on leveraging tree view visualization within HAnS-Viz to enhance user interaction and intuitively modify feature model and annotations. Through evaluations and assessments, I can explore the findings and insights derived from addressing these research questions.

5.2.1 RQ1: Leveraging Tree View Visualization

How can the tree view visualization be leveraged within the HAnS-Viz plugin to offer users an intuitive method for modifying the feature model and feature annotations?

Based on the implementations presented in Sections 4.1.2 and 4.2, I can now address research question 1. By augmenting the HAnS plugin with five additional functionalities—namely, adding, renaming, moving features, and deleting features with annotations or code — I facilitated the modification of the feature model and annotations. These enhancements were subsequently integrated into the HAnS-Viz plugin as part of the new toolbar within the tree view visualization, with the goal of offering users a user-friendly and intuitive method for modifying the feature hierarchy and corresponding feature mappings.

5.2.2 RQ2: Usability of Extended HAnS-Viz

How usable is the extended version of HAnS-Viz?

The extended version of HAnS-Viz exhibited promising usability, as indicated by the evaluation results outlined in Section 5.1. Participants, a diverse group of computer science students and researchers, found the toolbar actions intuitive and user-friendly. Based on the survey results, they demonstrated high learnability and efficiency in using the plugin, with tasks completed quickly and mostly error-free. While satisfaction levels were generally positive, challenges in error recovery, particularly during more complex tasks, suggest areas for improvement to enhance overall user experience. Across various actions, the evaluation yielded positive feedback, and participants generally reported satisfaction with the overall experience. Additionally, the SUS score of 83.25 further supports the favorable usability of the extended HAnS-Viz plugin.

6 Discussion

This chapter discusses the findings from the experiment, which shed light on the usability of the plugin and highlight areas for potential improvement based on feedback and observations.

The experiment involved the evaluation of 10 participants, all of whom were affiliated with the university and had backgrounds in computer science. While the feedback provided valuable insights, it's important to note that the participants were exclusively from the faculty of computer science and had professional experience in software development. This limitation restricts the diversity of perspectives, as insights from individuals outside the software development realm or those not directly involved in coding were not captured. Nonetheless, how the tool is being utilized by developers, the target group of the experiment, could be assessed.

Furthermore, the sample size of 10 participants is relatively small, which may impact the generalizability of the findings. Despite these limitations, the feedback obtained from the experiment serves as a valuable starting point for future iterations and enhancements to the plugin's usability.

Several users encountered difficulties in locating and manipulating collapsed features within the tree view. A potential solution to this issue could involve modifying the search functionality to automatically expand searched features, enhancing user accessibility and navigation.

Confusion arose among users when interacting with the modal window during the "deleting feature with code" action. To address this, implementing an informative tooltip within the modal window could provide users with explanations of its functions and displayed content, aiding in clarity and comprehension.

The presence of keyword event listeners led to potential clashes due to multiple types of clicking events associated with each node in the tree view. To mitigate this, restructuring the mappings between keywords and actions could prevent conflicts, ensuring smoother user interactions and reducing confusion.

Participants also suggested enhancing usability by adding a tooltip with multiple options for each node in the tree view. This improvement would offer users quick access to various actions, streamlining workflow and enhancing overall efficiency.

Despite the identified challenges, users generally found the extended version of HAnS-Viz to be usable, as indicated by the SUS evaluation score (see Fig. 5.10). However, to further validate and improve its usability, conducting a second experiment with HAnS-Viz, incorporating the suggested improvements to the toolbar, would offer greater insights into the plugin's effectiveness and user satisfaction.

Expanding the participant pool to include a more diverse range of individuals, beyond those solely affiliated with computer science, would also be beneficial. This broader participant pool would provide a more comprehensive understanding of user needs and preferences, enhancing the reliability and applicability of the experiment's findings.

7 Conclusion

Feature manipulation, including adding, renaming, and removing features, is a critical aspect of software development that influences the functionality, design, and maintainability of a software product. Each manipulation pattern incurs costs, including annotation recording and editing, as well as potential impacts on the software's architecture and user experience. These manipulations also offer opportunities for enhancing the software's functionality, improving its design, and reducing complexity. Therefore, successful feature manipulation necessitates a thoughtful evaluation of the associated advantages and disadvantages to ensure the software product's successful evolution [3].

Although HAnS and HAnS-Viz already provide developers with tools for managing and modifying software features and annotations, my goal was to enhance the usability and effectiveness of these plugins by broadening the range of actions available to users and conducting an experiment to evaluate user experience.

Through the addition of functionalities such as adding, renaming, moving, and deleting features within the plugin's toolbar, along with improvements to the tree view visualization, significant strides were made toward providing developers with smooth experience in managing feature annotations.

The experiment evaluating the extended HAnS-Viz provided valuable insights. Participants found the plugin user-friendly and efficient. While the results were promising, users highlighted several challenges, particularly with error recovery and modal window interactions. They also suggested improvements, including enhancing the search functionality and providing informative tooltips.

Overall, HAnS-Viz shows promise for feature-oriented development, with room for refinement based on user feedback. Expanding the user pool and conducting follow-up experiments would further enrich the understanding of its usability and effectiveness across different development contexts.

In conclusion, this work aimed to make contributions to feature visualization and management tools, with the ultimate goal of enhancing productivity and efficiency in software development workflows.

List of Figures

Bibliography

[1] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. What is a feature? pages 16–25, 07 2015.

[2] Julia Rubin and Marsha Chechik. A survey of feature location techniques. In *Domain Engineering, Product Lines, Languages, and Conceptual Models*, 2013.

[3] 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.

[4] 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, pages 28–31, New York, NY, USA, 2021. Association for Computing Machinery.

[5] David Stechow and Philipp Kusmierz. HAnS: Feature Visualization. Bachelor's thesis, Ruhr-Universität Bochum, 2024.

[6] Sven Apel and Christian Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology (JOT)*, 8:49–84, 07 2009.

[7] 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.

[8] Jacob Krüger, Thorsten Berger, and Thomas Leich. *Features and How to Find Them: A Survey on Manual Feature Location*, pages 153–172. 01 2019.

[9] Tobias Schwarz. Design and assessment of an engine for embedded feature annotations. Master's thesis, University of Gothenburg, 2021.

[10] JetBrains. Program structure interface (PSI): IntelliJ Platform Plugin SDK, 2022. Accessed: 01/05/2024.

[11] Johan Martinson and Herman Jansson. HAnS: IDE-based editing support for embedded feature annotations. Master's thesis, Gothenburg, Sweden, 2021.

[12] Tobias Schwarz, Wardah Mahmood, and Thorsten Berger. A Common Notation and Tool Support for Embedded Feature Annotations. pages 5–8, 10 2020.

[13] Sina Entekhabi, Anton Solback, Jan-Philipp Steghöfer, and Thorsten Berger. Visualization of Feature Locations with the Tool FeatureDashboard. pages 1–4, 09 2019.

[14] Alexandre Bergel, Razan Ghzouli, Thorsten Berger, and Michel Chaudron. FeatureVista: interactive feature visualization. pages 196–201, 09 2021.

[15] Aaron Bangor, Philip Kortum, and James Miller. Determining what individual SUS scores mean: adding an adjective rating scale. *J. Usability Studies*, 4(3):114–123, may 2009.

[16] John Brooke. *SUS – a quick and dirty usability scale*, pages 189–194. 01 1996.

A Questionnaire

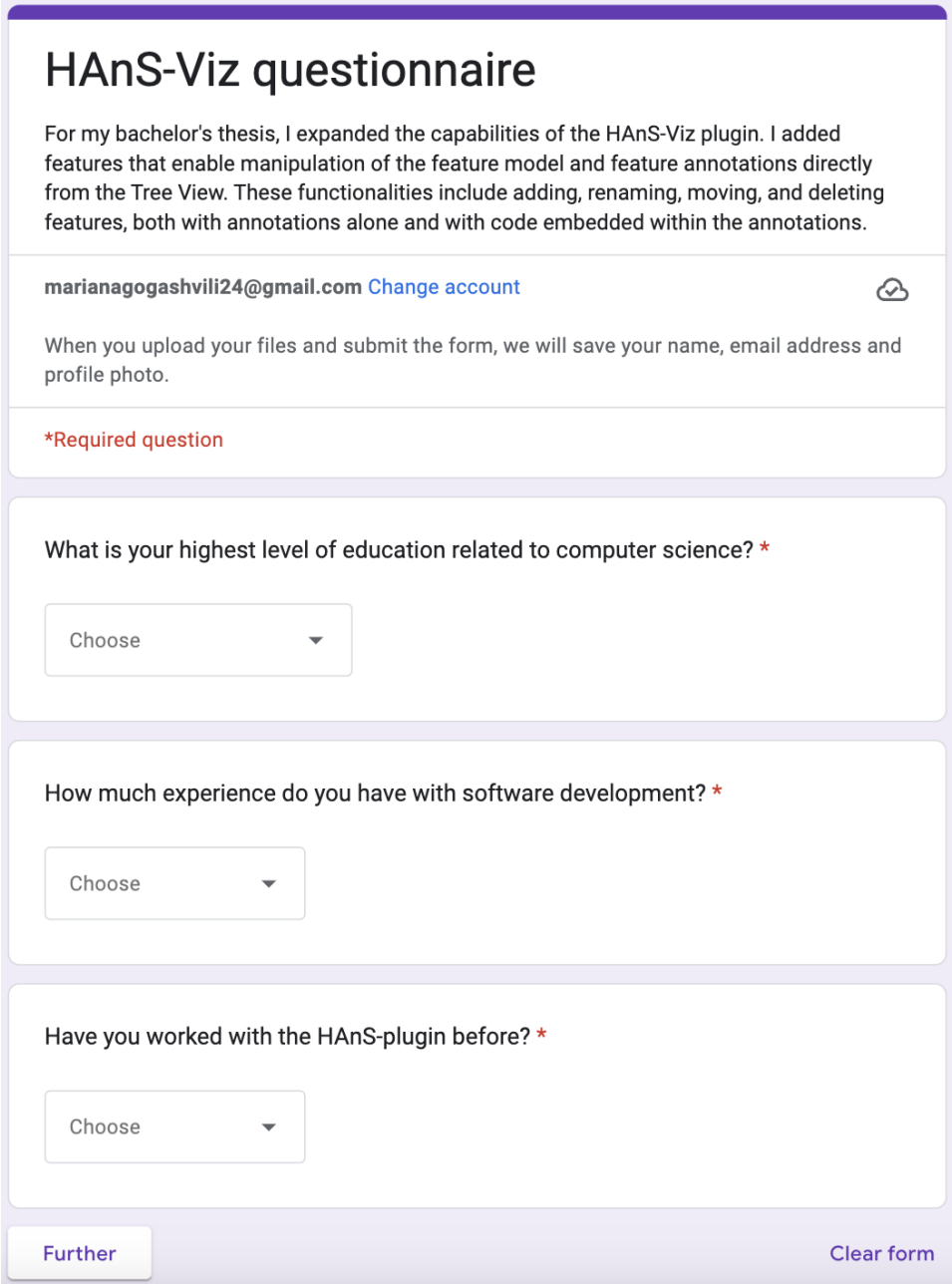

Figure A.1: Questionnaire: Introduction

Setup

Before starting the survey, please follow instructions to install HAnS and HAnS-Viz.

1. Go to https://github.com/hohashvili/snake-game-experiment
2. **Download/Clone** the repository
3. Open Intellij (min. build must be 223.x)
4. Go to IntelliJ -> Settings... -> Plugins -> settings icon (at the top, next to "Installed") -> "**Install plugin from disk...**"
5. Go to **plugins** folder and install **HAnS-0.0.5.zip** first
6. Then install **HAnS-Viz.zip**
7. After installing both, **restart** the IDE
8. Open project "**snake-game**" in the downloaded repository

Install screen recorder

Please install **Google Chrome screen recorder** for this experiment.
1. Go to Screen recorder
2. Follow instructions to start recording
https://drive.google.com/file/d/1AjU0nZFsSUqjyluE-Y5lxLV682oYdhlV/view?usp=sharing

Alternatively, feel free to use **your preferred screen recorder**, just adjust the quality to the lowest resolution to minimize storage usage.

I am kindly asking you to submit your recording at the end of the survey.

Please start the recording now.

If you open the **tree view** and click on the **pen icon** in the top right corner, a **toolbar** will appear showing a list of buttons that correspond to different actions.

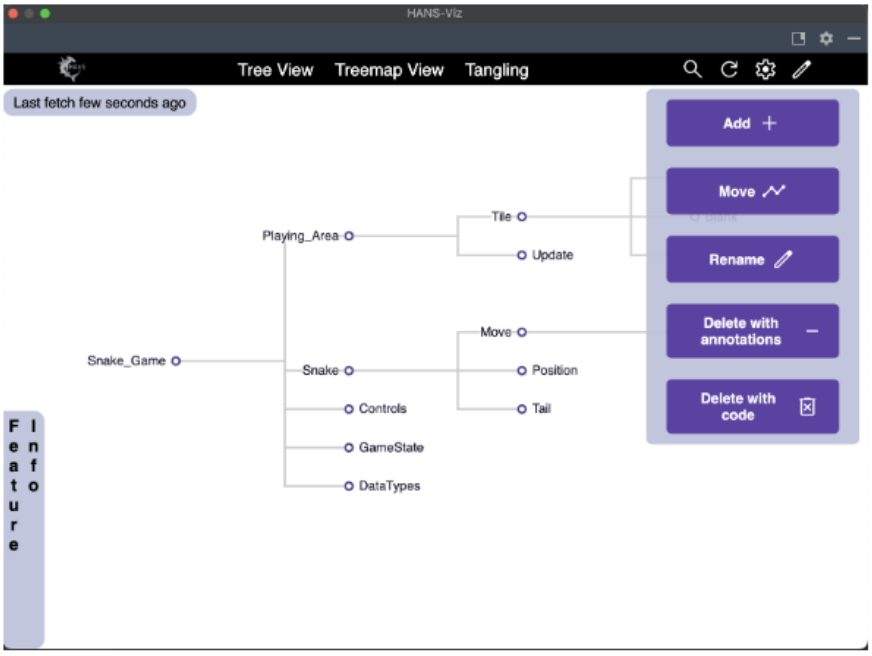

Figure A.2: Questionnaire: Installation instructions

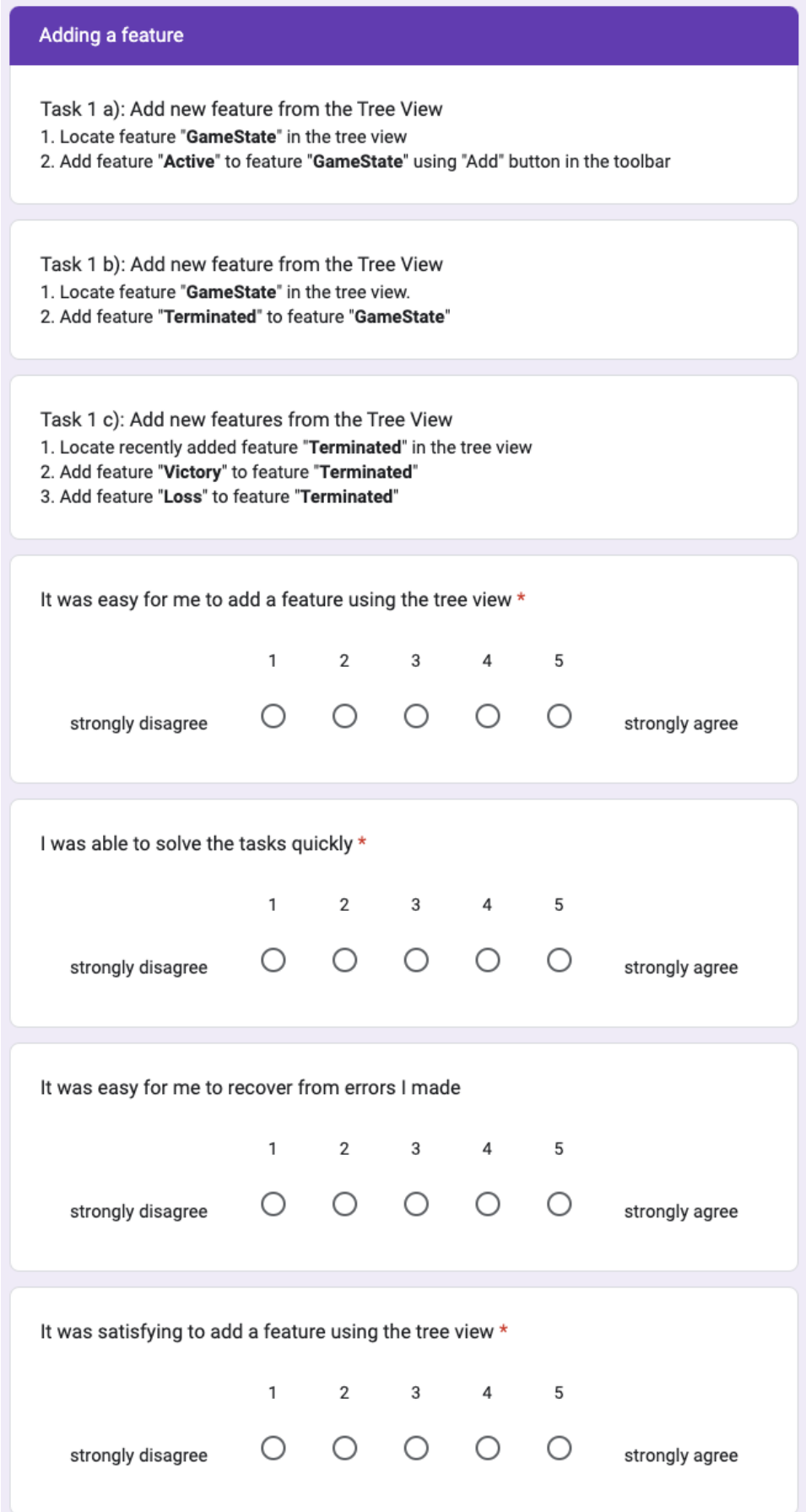

Figure A.3: Questionnaire: Adding a feature

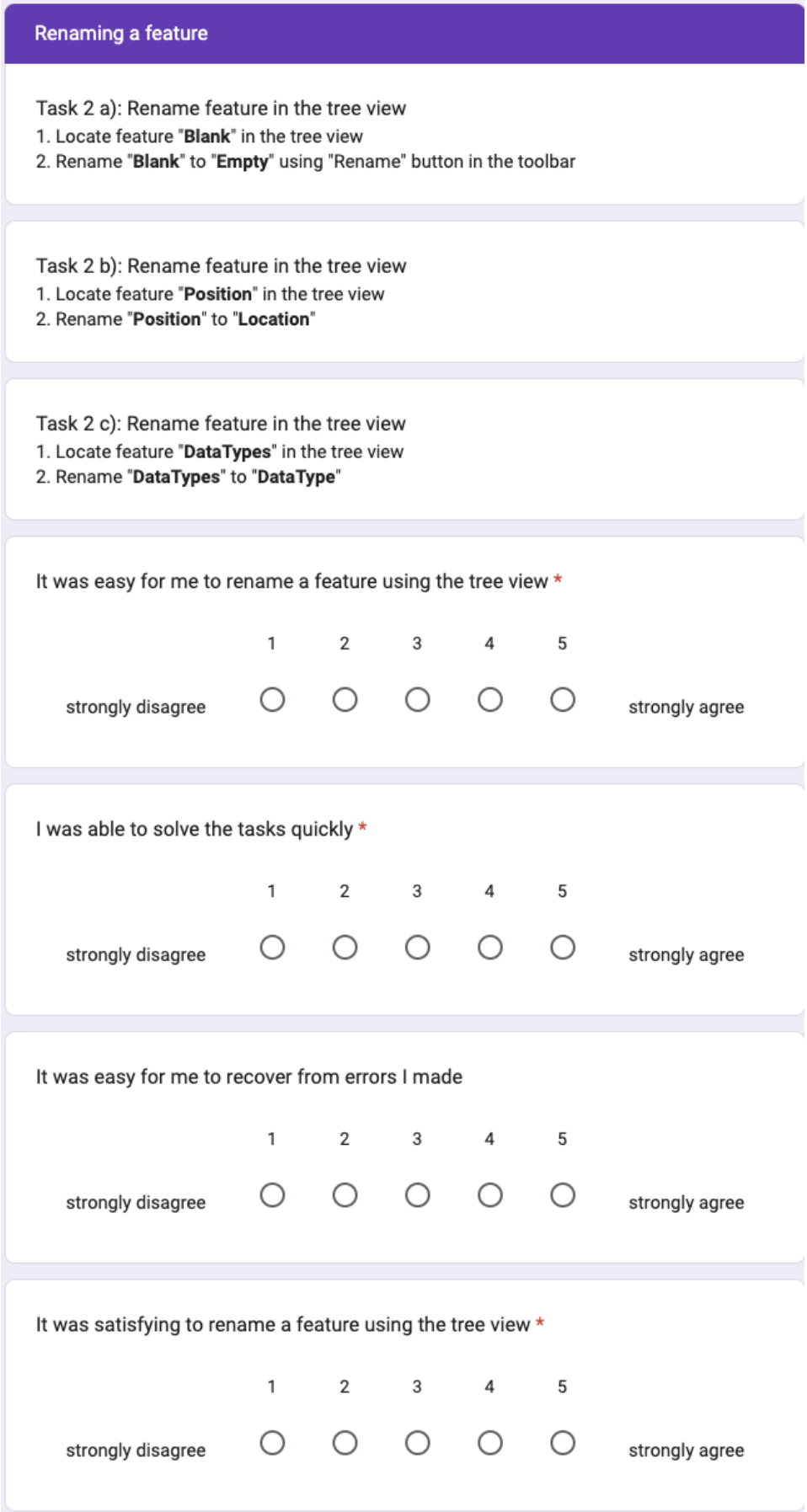

Figure A.4: Questionnaire: Renaming a feature

Figure A.5: Questionnaire: Moving a feature

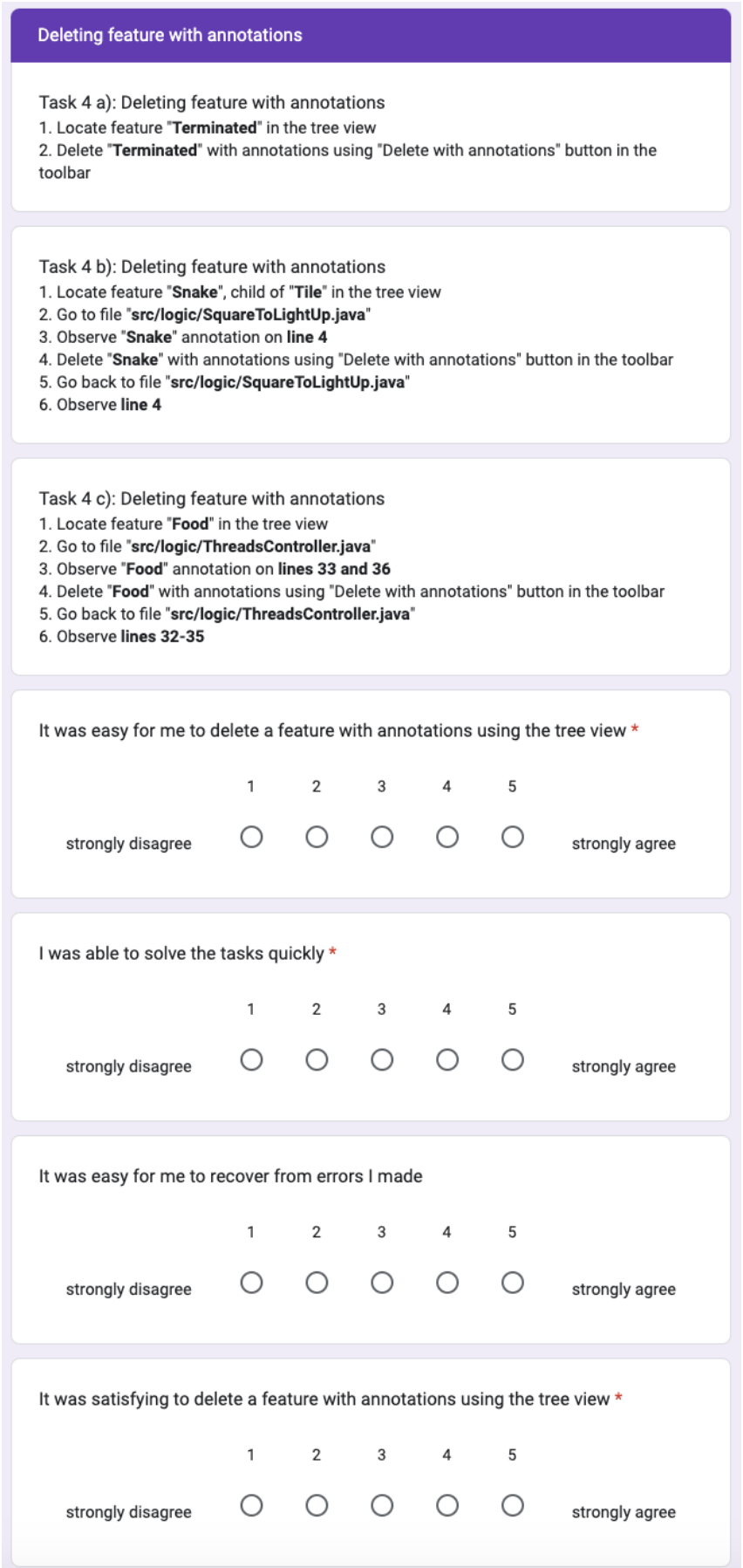

Figure A.6: Questionnaire: Deleting a feature with annotations

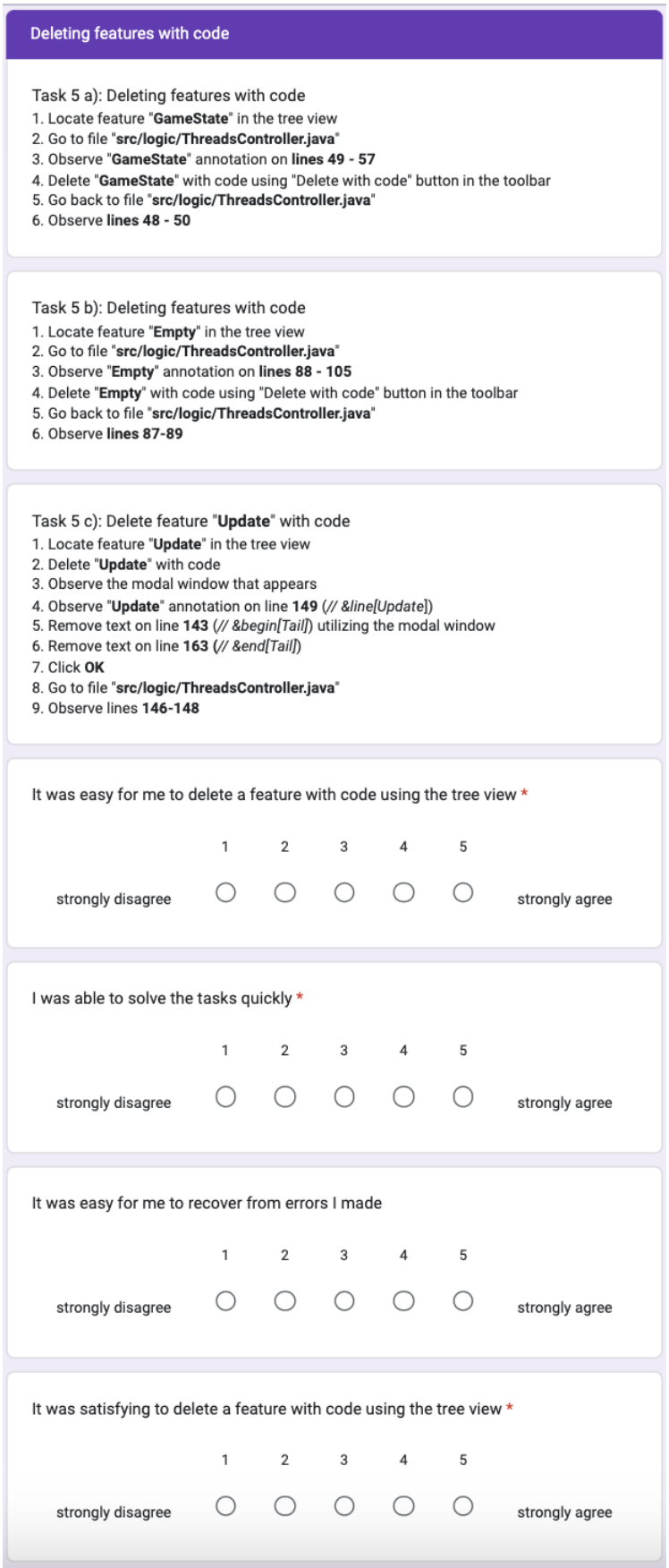

Figure A.7: Questionnaire: Deleting a feature with code

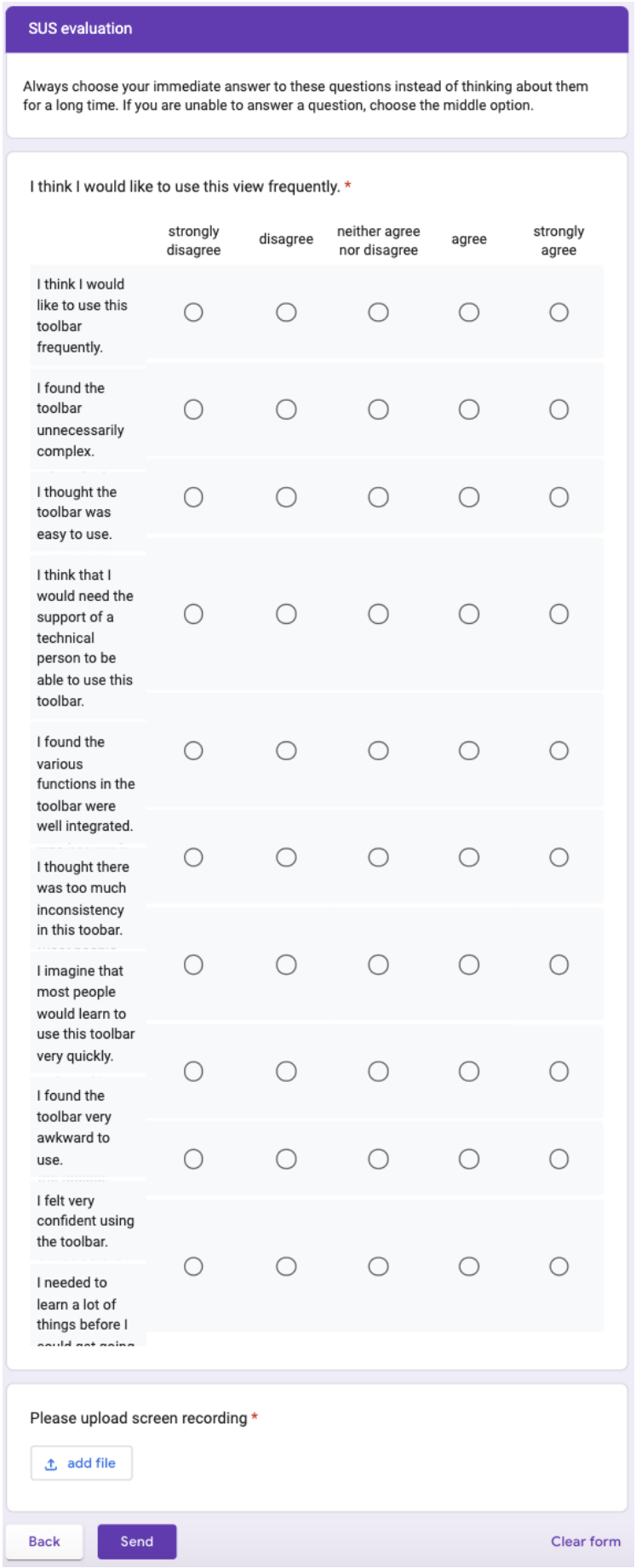

Figure A.8: Questionnaire: SUS evaluation