RUHR-UNIVERSITÄT BOCHUM

RUB

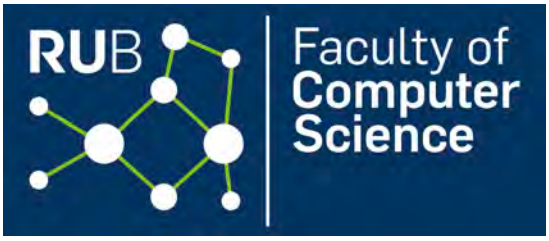
A Controlled Experiment on Using Embedded Annotations for Documenting Feature Locations in Code

Hasanain Qaddoori Hussein Al-Aassi

RUHR-UNIVERSITÄT BOCHUM

Master's Thesis – July 21, 2024
Chair of Software Engineering.

Supervisor: Prof. Dr. Thorsten Berger
Advisor: Wardah Mahmood



Abstract

Documenting the features of a software system is a fundamental yet challenging task in software development. Developers often invest considerable time during development, locating features in extensive software systems due to their spatial distribution in the code base [32, 39]. In response to this challenge, the HAnS (Helping Annotate Software) tool was developed to facilitate the documentation of these features in a managed way, providing a hierarchical overview for the annotated features [15, 22]. An extension of the HAnS tool, the Logger tool, was employed to record the feature annotation durations [33]. This thesis study evaluates the usability of documenting embedded feature annotations using the HAnS tool and the accuracy of the Logger tool in recording embedded feature annotations. To assess this, a user study was conducted in which participants utilized the HAnS tool to add, edit, and navigate feature annotations throughout the development process in various task scenarios. Quantitative and qualitative data were collected to evaluate the invested effort (in terms of time) in completing the tasks. The results of the controlled experiment show that the HAnS tool is user-friendly and practical for use in large systems. While the Logger tool showed overall accuracy in recording feature annotations, statistical tests revealed discrepancies, particularly with block annotations. These findings highlight the potential of embedded feature annotations to improve the traceability and management of these features, leading to a reduction in the development effort and an enhancement in code maintainability. This study also identifies potential areas for further development or refinement of the HAnS tool, providing invaluable insights into the benefits and difficulties associated with using feature annotations and establishing a foundation for future advancements in this field.

Acknowledgements

First, I would like to thank my supervisor, Professor Thorsten Berger, for his invaluable support and understanding throughout this thesis. His timely and constructive feedback was instrumental in improving and completing this thesis on time. I would also like to thank my second supervisor, Wardah Mahmood, for her exemplary work and consistent guidance throughout this thesis. The quality of this thesis has been invaluably enhanced by her dedication, encouragement, and positive feedback. Finally, I would like to thank Kevin Hermann and Johan Martinson for their interest in my work and helpful insights that have greatly improved the quality of this thesis.

Hasanain Al-Aassi, Bochum, July 2024

Contents

1 Introduction

Software development is a diverse and constantly evolving field. It is common for software development to be planned and organized around features. In the context of software, features are functional or characteristic aspects that are identified during the development cycle, maintenance, and evolution [6, 25]. These features are used in software planning and represent the common language and management of the development process for the developers [6, 19, 35]. However, the number of features grows substantially as software size and complexity increase. This expansion introduces new challenges that every developer needs to consider. In software engineering, feature location is the process of identifying and locating specific parts of source code that correspond to a particular software feature [19]. One of the challenges the developers face is the accurate documentation and location of software features within the codebase [19].

Despite its importance, the use of automated documentation features is not yet widely adopted by developers in large software projects, especially those involving code extension, maintenance, and cloning. Knowing the location of features throughout the software's evolution, including for code reuse and enhancements, is crucial for developers [20, 23].

This experiment aims to investigate the efficiency and effort developers require in terms of time to write embedded feature annotations in real-world scenarios during the development process. This will be achieved by measuring the time taken to complete the annotation process using the HAnS (Helping Annotate Software) tool. HAnS is a Java-based plugin that facilitates the annotation process by allowing the user to specify feature mappings to different types of assets within the code [15, 22]. It incorporates the Logger tool, which serves as an extension of HAnS to record the time taken to complete these feature annotations [33].

1.1 Motivation

The task of locating features presents a significant challenge. Developers often spend considerable time understanding the codebase and locating these features during the software development. This intense focus on feature location sometimes results in delays in other important tasks. In addition, even when they attempt to locate these features, they may overlook various potential locations where the features are

implemented because of their scattered nature. A limited understanding of the code or a lack of experience with a programming language can pose additional challenges. Studies by Wilde et al. [40], Revelle, Broadbent, and Coppit [28], Wang et al. [39], and Jordan et al. [17] acknowledge the importance of a developer's experience and knowledge but do not definitively establish the positive or negative effects of these factors on performance. Corresponding studies with a similar focus are essential for identifying and evaluating the factors significantly influencing the feature location process. It is essential to identify specific code portions related to features, but due to the intricacies of software systems and intricate interdependencies, the process of feature location is both time-consuming and prone to errors [39, 32]. Despite the notable achievements of various feature location techniques [32], feature location remains an activity that relies heavily on human expertise and domain-specific knowledge, making it risky to underestimate the role of human factors in this process [14].

This study aims to evaluate the effectiveness of using embedded annotations with the HAnS tool in facilitating documentation of embedded feature annotations in code. The evaluation examines the accuracy of the Logger tool for recording feature annotations and includes a user study experiment utilizing the HAnS tool. This experiment aims to measure the effort (in terms of time) developers require to write, edit, navigate, and document feature annotations during the development process in real-world scenarios. In addition to assessing the usability of the HAnS tool, this study also investigates the efficiency of feature annotations. The findings contribute to the field of embedded feature annotations by providing qualitative and quantitative insights into the annotations' effectiveness and the overall utility of the HAnS tool.

In brief, this thesis aims to address the significant challenges of feature annotations in software development using the HAnS tool to streamline the documentation process. This research also aims to provide valuable insights into the practical benefits and efficiencies of embedded feature annotation through a comprehensive evaluation, including an accuracy assessment of the Logger tool via a user study experiment, ultimately contributing to improved software development practices.

1.2 Related Work

Several studies exist on finding and evolving features in software product lines. These are third-party tools designed for extracting features from code, where their strength lies in the extraction process and application of heuristics. Therefore, these tools focus primarily on feature extraction and using heuristics without considering the realm of annotation creation. For instance, consider the product-line migration framework of Rubin et al. [30, 31]. They introduced operators that align with the

idea that mechanization leads to more efficient implementation and support, specifically from an operator-based perspective. Fischer et al. [11] present a framework and a tool (called ECCO) that heavily leans on heuristics to identify commonalities and enable the creation of new product variants using reusable resources. Similarly, the BUT4Reuse tool by Martinez et al. [21] specializes in extraction techniques for product-line migration, even incorporating support for feature-model synthesis. Their framework relies mainly on heuristics for feature extraction. These existing works are less than ideal, as they rely on heuristics, which can lead to inaccuracies in feature location. There is a notable shortfall in the systems available for benchmarking against actual revision histories that include accessible feature information. This challenge has become a central focus of their ongoing benchmarking initiative [36, 5]. Many feature visualization tools are created specifically for software product lines or systems with high configurability [23, 20]. However, only a few are specifically tailored for embedded feature annotations. For instance, in [35], FAXE is a tool for extracting feature annotations. This tool facilitates language independence and seamless integration into IDEs and other software tools. It is a Java library that can retrieve annotations, perform metric calculations, detect inconsistencies, and rename features. Nonetheless, it lacks the editing functionality. For example, in extensive and complex software systems, this feature location can swiftly become a challenging and error-prone endeavor [39, 28], particularly when features are spread out across software components [17]. Despite the existence of automated tools for locating features [39], they require specific configurations or setup efforts for each project and tend to produce numerous incorrect results, making them less practical for real-world applications [30].

FLORiDA [2] is a standalone Java application. It offers multiple feature views and metrics, such as tangling and scattering degrees. The VariantSync tool by Pfofe et al. allows users to add feature tags to code changes to make categorization easier [26].

Previous studies, including those by Wilde et al. [40], Revelle, Broadbent, and Copet [28], Wang et al. [39], and Jordan et al. [17], have investigated the impact of developer experience and knowledge on performance but did not establish any effects of these factors on performance.

However, no empirical studies currently measure the effort (in terms of time) involved in using the functionalities of documenting features to assets of different types during development. Although the previous tool appears useful, the lack of empirical data limits a comprehensive assessment of its efficiency in real-world development workflows. Further research may be needed to quantify the practical implications and benefits of embedded annotations using HAnS. In addition, these tools are challenged by the setup effort and the need for specific code entry points for feature locations, which also require focusing on tool training by developers.

In response to the challenges mentioned earlier and to facilitate software maintenance and reuse, the HAnS plugin was developed to help developers proactively create embedded feature annotations within code assets (rather than recovering them as in the earlier works mentioned). HAnS facilitates the development process by enabling the user to map and annotate features within different types of software assets, including folders, files, code lines, and code segments [22].

For the HAnS tool, as discussed in [22], researchers advocate that features should be recorded proactively instead of being extracted. This approach, in which developers write features themselves rather than relying on heuristics, is more accurate and requires no setup effort, as developers can write annotations during ordinary development. HAnS itself requires additional effort; however, this additional effort is expected to overcome the cost of manually locating features. Despite the effectiveness of using embedded feature annotations with HAnS, no study has been yet conducted to accurately measure the time investment required to use embedded feature annotations during development. Furthermore, there is a lack of subjective data regarding the tool's usability for this purpose. Therefore, an empirical study is essential to gain an accurate understanding of the effort (in terms of time) required to use embedded feature annotation with the HAnS tool. The goal of the proposed master's thesis is to contribute new knowledge to the software engineering field by conducting a controlled experiment using the HAnS tool to document feature locations. It also aims to provide empirical evidence of the HAnS tool's efficiency and usability in the context of embedded annotation in real software development scenarios.

1.3 Contribution

The goal of this thesis is to contribute to the field of software development by assessing the HAnS tool, which facilitates programmers' creation of embedded feature annotations by programmers [15, 22]. The research presents a combination of quantitative and qualitative data obtained from a user study, which provides insights into the effort required to document feature annotations in terms of time. This highlights the practical advantages of utilizing these annotations.

A key factor is evaluating the accuracy of the Logger tool in capturing feature annotations. The experiment demonstrates that the Logger tool reliably tracks annotation activity, ensuring reliable documentation of feature locations. This accuracy is critical for future reference, cloning, and maintenance tasks. The usability of the HAnS tool is also evaluated, providing insights into its intuitiveness and usability that can inform the development of similar tools and guide developers in using embedded feature annotations in code.

The research is necessary because keeping track of feature annotations within the code is difficult due to the lack of proper documentation during development. This

can lead to the loss of features, making it hard to maintain, clone, and understand the codebase in the future. One potential solution is to record feature locations, but it is often seen as time-consuming and challenging. This study measures the effort (in terms of time) required to create embedded feature annotations during development, allowing us to compare this effort to total development time.

The contribution includes a controlled experiment, quantitative and qualitative analyses, and an appendix containing all materials related to the user study. These comprise detailed data from the user study, reading handouts, a questionnaire with responses, and additional analyses [1]. This comprehensive approach provides a robust evaluation of documenting feature annotations using the HAnS tool and its practical implications for software development.

The results serve as a foundation for future research, suggesting enhancements such as feature visualization and metrics, integration with different development environments, and improvements to the logger and HAnS tools. Overall, this thesis advances the field by providing new insights and practical solutions for improving feature traceability, managing code maintenance, developing productivity, and documenting feature locations accurately.

1.4 Organization of the Thesis

This thesis is structured to present a comprehensive experiment on embedding feature annotations using the HAnS tool and its effectiveness in automatically documenting feature locations. The experiment focuses on measuring the effort required to write embedded feature annotations in code, thereby evaluating the accuracy of the Logger tool, conducting user studies, and analyzing both qualitative and quantitative data.

The thesis is organized as follows:

Chapter 2 presents the theoretical background, outlining the concepts related to the work, including features and embedded feature annotations. This chapter also provides an overview of the HAnS tool, explaining its functionality in the context of automatic documentation of feature locations, and describes the Logger tool.

Chapter 3 details the methodology used in this study, describing the design of the user studies, the experimental setup, the data collection methods, and the process of evaluating the accuracy of the Logger tool in documenting feature annotations. This chapter provides the necessary context for understanding the experiment conducted.

Chapter 4 presents the study's results. This includes the user studies' results, quantitative data (such as the effort required to complete feature annotations during development), and qualitative participant feedback. It also includes an analysis of the logger tool's accuracy compared to video recordings of the annotation process. The results related to the effort in navigating time and annotating are also presented.

Chapter 5 discusses the study results' implications in detail. This chapter examines the discrepancies between the Logger and video data and discusses the potential reasons for these differences. Additionally, it discusses the effort of navigating time and annotating, user perception, and qualitative insights.

Chapter 6 examines the study's validity threats. It discusses internal validity, external validity, construct validity, and reliability, providing a critical assessment of the factors that might have influenced the results and how these threats were mitigated.

In conclusion, Chapter 7 summarizes the essential findings and their contributions to the field of software engineering. The chapter highlights the practical benefits of using embedded feature annotations and the HAnS tool and suggests areas for future research. Additionally, the chapter considers the study's limitations and potential improvements for the HAnS tool.

By organizing this thesis in this manner, a logical progression will be evident from discussing the background and motivation behind the research through a detailed analysis of the results, concluding with insights and recommendations for future work.

2 Background

2.1 Feature

Features in software development serve as fundamental building blocks used for planning, managing, and communicating the functionality of the development process [6, 35]. These features provide a common language for defining, monitoring, and communicating how software systems work, making it easier for developers to understand, reuse, or modify them [19]. Consequently, features are used to describe the functionality and characteristics of the software throughout the phases of software development, maintenance, and evolution [6, 25]. Defining and understanding the concept of features can be challenging. Nevertheless, an explicit definition, presented earlier, can be stated as follows:

> *"A feature is a characteristic or end-user-visible behavior of a software system. Features are used in product-line engineering to specify and communicate commonalities and differences of the products between stakeholders, and to guide structure, reuse, and variation across all phases of the software life cycle."* [19]

In the previous definition, a feature in the software is a visible property or behavior that users can interact with. It's used in product line engineering to describe what's common or different between different software versions. Features guide how the software is structured, reused, and modified throughout its life cycle. Tool support is essential to use feature annotations efficiently and encourage developers to incorporate them seamlessly during development. That's why HAnS (Helping Annotate Software), an IDE plugin, was designed to assist developers in annotating [22].

A feature model represents a software system's characteristics or functionalities and provides a hierarchical view of these features and their interrelationships. It is a fundamental element of the software annotation system and organizes the various components of a software system [6].

The task of tracking specific source code segments responsible for implementing a given feature within a software system is called feature location. This process can be manual or automated and facilitates tasks such as feature updating, reuse, and maintenance [19].

The concept of software feature location traceability can be defined as the process of establishing a traceability link, a term with broader implications [3, 24]. This term has a broad implication, capturing multiple relationships between different artifact types, including both code and non-code elements such as requirements.

2.2 Embedded Feature Annotations

Embedded feature annotations (EFA) are comments that map code locations to specific features, indicating that the annotated code fragment is part of the feature's implementation [35].

Embedded feature annotations provide a solution by embedding feature details directly into the source code. This approach ensures that feature locations remain easily accessible and can be updated during development [35].

Developers can use this annotation system to document the location of features in folders, files, code fragments, and even individual lines of code. A feature model provides a comprehensive enumeration of all the features present within a software system, organized in a hierarchical tree structure [15]. Figure 2.1 below illustrates how the features hierarchy in DARTPlus is designed with embedded feature annotations.

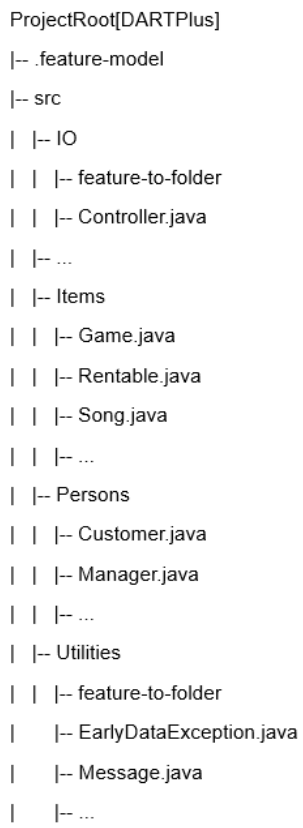


Figure 2.1: Illustrates the Embedded Feature Annotations of the DARTPlus rent system.

2.3 HAnS (Helping Annotate Software)

HAnS is an open-source plugin for the IntelliJ integrated development environment (IDE) that was presented by a team of researchers and developers from various institutions, including Mukelabai Mukelabai, Thorsten Berger, Alexandre Bergel, and Truong Ho Quang [22]. Additionally, Johan Martinson and Herman Jansson, who also collaborated on the research of the HAnS (Helping Annotate Software) tool, contributed to its development to ensure the effective use of feature annotations and to facilitate developers to document them with the least impact on their development tasks [15, 22].

HAnS provides detailed yet easy-to-use annotations to help developers understand software features. With this tool, developers can effectively generate and document various features within a feature model. It assists developers in annotating various software elements like files, folders, and code fragments with the specific features they are implementing. It provides helpful functionalities for creating and evolving feature models over time. Additionally, the tool facilitates mapping features to assets, including code completion for feature names and code syntax highlighting. Developers can also explore feature definitions and easily locate where they are implemented within the system. Furthermore, the tool facilitates the redesign of features, for instance, enabling developers to delete or rename features as needed. This enhances flexibility and clarity in managing software features effectively.

HAnS offers proper functionality for tracking features and their associated code assets throughout the software system. In addition, the Logger tool was developed as an extension of the HAnS plugin, which assists in recording embedded feature annotations created by HAnS [33].

2.3.1 Feature model

A feature model represents the characteristics or functionalities of a software system. It provides a hierarchical view of the different features of a system and how they relate to each other. The feature model is the basis of the annotation system [6]. Creating a feature model using HAnS is done by creating a file in the root directory of a project and naming it with ".feature-model" at the end. In this file, all the features of the projects must be added and organized in a sort of hierarchy. Each feature model should begin with the project root name and then the features added below in hierarchical order, with indentation to show which features are part of other features [15]. Whenever a developer makes changes to the ".feature-model", it is akin to adding feature annotations to the project. An example of a project for the feature model of a snake game is shown in Figure 2.2 below. It is significantly crucial for developers to use the feature model when engaged in software development projects

in order to gain an understanding of and to be able to manage the functionality of the system.

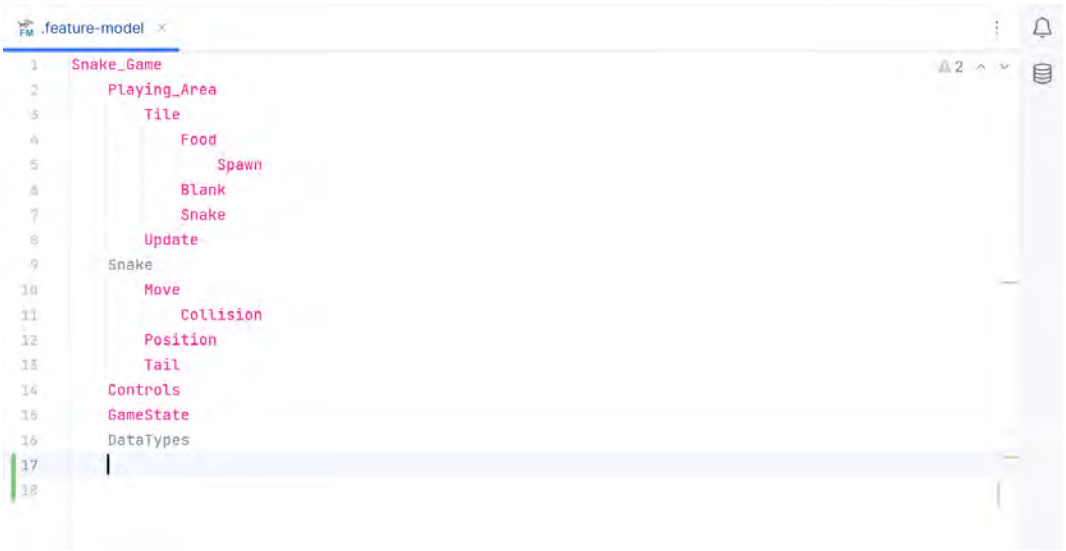


Figure 2.2: Feature model of the Snake game.

2.3.2 Feature-to-file mapping

The concept of feature-to-file mapping implies that the entirety of a file represents either a complete or partial implementation of a feature. This allows for the association of individual features with the entire file. The mapping of features to files is stored in a dedicated file identified by the ".feature-to-file" extension [15].

Figure 2.3 illustrates the mapping between features within the files. The file names and feature mappings are written in alternating lines. For instance, the SquarePanel.java file includes implementing the "Tile" feature.

Figure 2.3: Feature-to-file mapping.

2.3.3 Feature-to-folder mapping

Feature-to-folder is the mapping of entire folders and their contents to one or more features. This allows specific features, including files and sub-components, to be associated with the entire folder.

In the following example, the mapping of features to the folder is stored in a dedicated file identified by the ". feature-to-folder" extension, as shown in Figure 2.4.

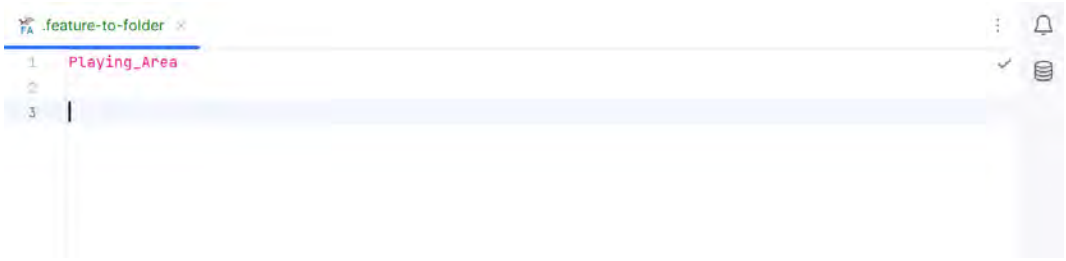


Figure 2.4: Feature-to-folder mapping.

2.3.4 Feature-to-code mapping

The block annotation of the feature-to-code mapping function helps to associate specific blocks of code with one or more features. The source code segments mapped to a particular feature are called annotation scopes. An annotation scope is enclosed by annotation markers and includes at least one feature reference. The implementation of block annotation is defined as "//&begin" and " //&end" expressions for the HAnS tool [15].

Line annotation serves as the method of "//&begin" and " //&end" annotations but to map a single specific line of code and can be replaced by them. The implementation of line annotation is defined as an "//&line" expression for the HAnS tool [15]. In Figure 3.4, an example shows both block and line annotations; the [Food] feature is mapped to the block enclosed by the "//&begin" and "//&end" markers, and the [Spawn] feature is mapped to the code line by the &line where it is located.

Both line and block annotation may have a similar expression explicitly defined for practical use. In practice, the code annotation of any source code fragments or lines may consist of several code lines or just a single line of code. Even in the case of a class or method, only the code fragment or code line annotated belongs to this identifier [34].

2.4 Logger Tool

The Logger tool is an extension of the HAnS plugin for IntelliJ IDEA, which is named "annotationLogger". It was developed to measure the time taken to write embedded feature annotations using the HAnS tool within the IntelliJ Integrated Development Environment (IDE) [33].

The logging process is implemented by recording timestamps in a JavaScript Object Notation (JSON) file. This file contains all the recorded items. Whenever an action is performed within the software project, it is recorded in the JSON file. The JSON file consists of an array containing all logged annotations of each type and the computation time spent on all annotation types, as shown in the example in Figure 2.5 below.

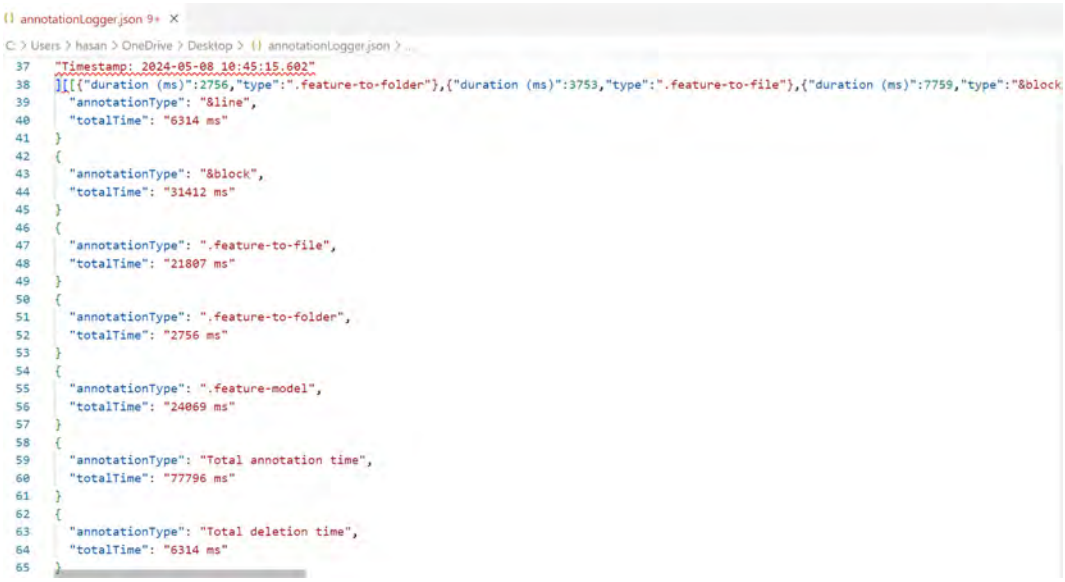


Figure 2.5: Example of a Logger tool JSON log file.

This includes annotations such as ".feature-to-file", ".feature-to-folder", ".feature-model", "&block", "&line", and annotation deletions. The total time spent on annotations and the total duration of development sessions are also included. The Logger tool tracks the time spent on each annotation by recording the timestamps of the first and last character changes within the annotation. If no further changes are made to the same annotation within a ten-second window, the logger marks the annotation as complete. The Logger calculates the time between the first and last changes to determine the duration [33].

All timestamps and durations are meticulously recorded in milliseconds to ensure precise measurement accuracy. The development sessions were implemented to compare the time spent annotating features with the total development time, including

coding time. These sessions begin when a software project is opened in the IntelliJ IDE and end when the IntelliJ IDE is closed. The duration of an active session is recorded and logged when it ends. If the user is inactive for 30 seconds, the session is inactive for the duration of the inactivity period until session activity is resumed. Inactivity is defined as the absence of mouse movement or keystrokes within the IntelliJ environment. This functionality is implemented in the code through listeners that detect events such as keystrokes or mouse movements within the IntelliJ and update the user activity accordingly [33].

2.5 The JetBrains IntelliJ IDE Support Platform

The IntelliJ platform offers several features that benefit researchers and developers involved in tool development and implementation. This platform is not a standalone software, which serves as a means of developing integrated development environments (IDEs) [37]. Other companies like Google can use it for their IDEs because it is open source and runs JetBrains products. This platform equips these IDEs with the necessary robust language development support infrastructure. It's a component-based, cross-platform application environment based on the JVM (Java Virtual Machine). It has an easy-to-use toolkit for building various interface elements like tree views, tool windows, lists (with a quick search), pop-up menus, and dialogues. It also includes a full-text editor with features like syntax highlighting, code folding, code completion, and text editing features, as well as an integrated image editor [37]. Open APIs are used to build on standard IDE features such as project models and build systems that the IntelliJ Platform also provides. It also offers an infrastructure for efficient debugging, supporting advanced features like breakpoints, call stacks, and expression evaluation independent of programming languages. The essential strength of the IntelliJ Platform is based on the Program Structure Interface, a set of functions used to analyze files, create rich syntactic and semantic code models, and generate indexes. PSI enables various functionalities, from fast navigation to code completions, searches, inspections, and refactoring. The platform includes parsers and a PSI model for many languages, and its extensible nature allows support for additional languages to be added. JetBrains is a market leader that offers software development tools [37].

Dedicated Integrated Development Environments (IDEs) are available not only for Java programming and various Plugins but also offer broad support for many languages, such as C, HTML, PHP, SQL, Kotlin, and JavaScript. JetBrains IDEs like IntelliJ IDEA, WebStorm, and PyCharm are based on it, as well as external IDEs like Android Studio and Comma IDE for Raku [16]. This IDE is based on the IntelliJ platform, which extensively reuses features across languages. Developers can save time implementing their concepts through the platform's powerful code-processing capabilities. The IntelliJ platform simplifies the process of developing IDE plugins. Platform-specific APIs allow plugin developers to handle code and

artifacts seamlessly. Plugins can be easily published to the marketplace after development, providing access to millions of IDE users and encouraging the evaluation of ideas. The IntelliJ Platform is open source under the Apache License and hosted on GitHub. While it's often called a standalone entity, there's no separate "IntelliJ Platform" repository.

Products built on the IntelliJ platform are plug-in applications, where the platform manages the creation of extensions [16]. It effectively supports plug-ins that are distributed through the JetBrains marketplace or custom repositories, which can serve the platform in numerous ways, from simple menu items to full language support. Creating actions for a plugin is also supported and can be used to perform specific tasks in IntelliJ.

HAnS uses 15 different extension points where it can add additional functionality and includes five new actions [15, 22]. Many of these extensions, such as "filetype" and "parser definition", have been used multiple times to set up how annotation languages work. Other extensions are used to add a new tab view via the "toolWindow" extension and feature name validation via the "renameInputValidator" extension. These new actions are also used for different purposes, like renaming, adding, or deleting features. Some actions create new annotation files, and others map annotations around a block of code.

2.6 Feature-Oriented Software Development and Evolution

Feature-Oriented Software Development (FOSD) is a paradigm that employs the concept of features to manage commonality and variability throughout the software development life cycle. This approach emphasizes the systematic use of features at each of the various stages, from initial analysis to final implementation, thereby facilitating the reuse and structuring of software systems based on these features [4]. Feature-Oriented Software Development (FOSD) is an approach to software development that emphasizes the identification, analysis, and implementation of significant or noticeable aspects, qualities, or attributes of a software system [18].

The concept of Feature-Oriented Software Development (FOSD) is complex due to its broad scope. However, an earlier definition provides clarity. In the article "An Overview of Feature-Oriented Software Development", Sven Apel and colleagues define FOSD as follows:

> *"The concept of a feature is useful for the description of commonalities and variabilities in the analysis, design, and implementation of software systems. FOSD is a paradigm that favors the systematic application of the feature concept in all phases of the*

> *software life cycle. Features are used as first-class entities to analyze, design, implement, customize, debug, or evolve a software system. That is, features not only emerge from the structure and behavior of a software system, e.g., in the form of the software's observable behavior, but are also used explicitly and systematically to define variabilities and commonalities, to facilitate reuse, and to structure software along these variabilities and commonalities. A distinguishing property of FOSD is that it aims at a clean (ideally one-to-one) mapping between the representations of features across all phases of the software life cycle. That is, features specified during the analysis phase can be traced through design and implementation."* [4]

This definition emphasizes the pivotal role of features in FOSD, influencing various stages of the software development process. By prioritizing features, developers can explicitly and systematically address the variabilities and commonalities within the software, thereby enhancing reuse and maintaining a consistent structural framework. A notable feature of FOSD is achieving a seamless mapping of features from the analysis phase to design and implementation, ensuring traceability and coherence [4].

Also, in the same article, Sven Apel [4] describes that the Feature-Oriented Software Development (FOSD) process consists of four phases: Domain Analysis, Design and Specification, Implementation, and Configuration/Generation. During domain analysis, developers identify relevant features and their support status. Design and Specification involve the creation of detailed plans for consistent feature implementation. Implementation translates the specified features into software components. Finally, Configuration/Generation assembles the software based on the specified features, allowing for customization. FOSD maintains a clear mapping of features from analysis to implementation, ensuring software consistency, customization, and evolution. This systematic approach improves software management and makes FOSD valuable in modern software engineering.

In their publication [38], Thomas Thüm et al. discusses how FOSD is utilized to originate and implement Software Product Lines (SPLs) in the engineering domain and to create specific application programs. FOSD involves producing functional software through several sequential phases: analysis, implementation, requirements analysis, and software generation. Various methodologies exist for implementing SPLs within the FOSD framework, primarily aimed at establishing a link between features and source code to automate the generation of software systems based on a given configuration. These methodologies are diverse but generally rely on a specific programming language known as the host language, such as Java or C++. Feature-Oriented Programming (FOP), as proposed by Prehofer [27], represents a significant advancement in object-oriented programming. This approach involves dividing classes into feature modules, each representing a distinct feature. These

feature modules may include methods and fields from multiple classes and can be combined into a program based on a defined configuration and sequence of features.

Software systems constantly change to keep up with new needs, technologies, and regulations. This evolution is influenced by different factors such as laws, market trends, and technological advancements [12]. According to Leonardo Passos et al. [25], this evolution is especially tricky for extensive systems due to the variety of involved artifacts and the breadth of expertise required. For example, adding a new feature to existing systems involves changes to multiple items and requires the knowledge of different experts. This complexity can lead to problems such as communication difficulties, software bugs, architectural degradation, and increased maintenance costs.

A way to deal with these challenges is to manage change at the feature level. Features are groups of requirements that define important parts of the system. They are also useful for building software product lines (SPLs), which are collections of similar software systems [7]. Sven Apel et al. [4] describe the idea that the features can be used to find out which parts of the system are the same across different versions and which parts can change, making it easier to control how the system is put together and how it changes over time. Incorporating feature-oriented development into existing ways of working shows that dealing with the tricky parts of software evolution is practical.

3 Methodology

In this chapter, we outline the methodology used to investigate the research questions outlined in Section (3.2). The methodology encompasses the experimental setup, data collection procedures, and analysis techniques used to address each research question systematically.

3.1 Experimental Design

This study uses a single-subject design [10], in which all participants experience the same conditions and tasks, allowing for direct comparisons within the same group. This design reduces variability between individuals, thereby increasing the study's statistical power. It was chosen because it is suitable for assessing the impact of the HAnS tool on feature annotation tasks. In this experimental study, the independent variable is the HAnS tool used to document feature annotations in a single experimental group. The dependent variables include the accuracy of the Logger tool in capturing embedded feature annotations, the effort (in terms of time) taken to complete annotation tasks, and the usability ratings of the HAnS tool. This experiment aims to evaluate the accuracy of the Logger tool in recording feature annotations. For this purpose, participants were instructed to perform these tasks using the HAnS tool, which integrates the Logger tool, for writing embedded feature annotations. Finally, the time invested in navigating and annotating these features was measured during the development process. The single-subject design also aids in evaluating the Logger tool by comparing it to video recordings of the participants' tasks. This helps to ensure that any differences observed are due to the functionalities of the tools rather than individual differences among participants. An overview of the experiment is shown in Figure 3.1.

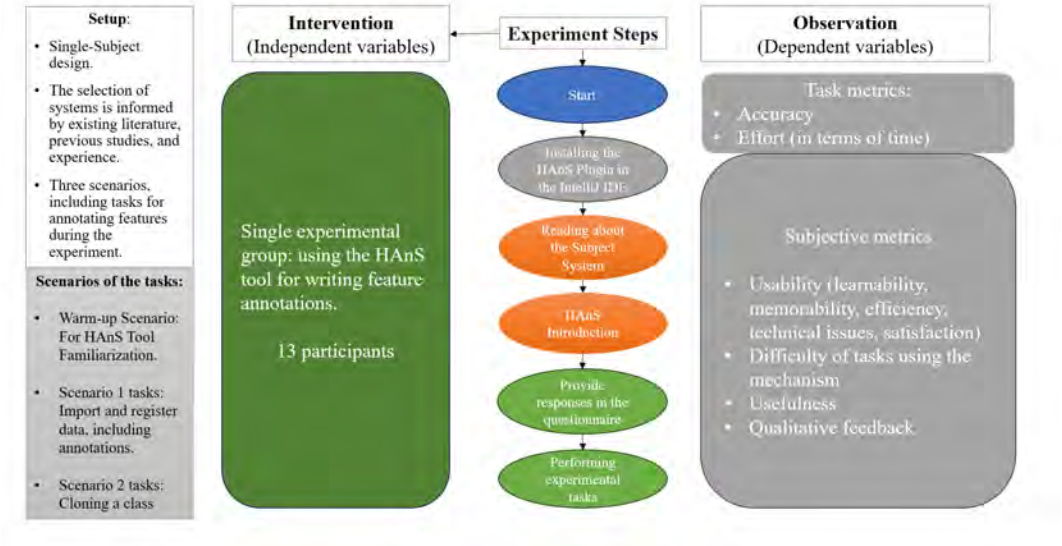


Figure 3.1: Overview of the experiment.

3.2 Research Questions

To guide our research, we will investigate the following research questions:

RQ1: How accurate is the Logger tool for recording embedded feature annotations?

This question aims to evaluate the accuracy and dependability of the Logger tool in accurately capturing and logging feature annotations within the codebase through the utilization of the HAnS tool.

RQ2: What is the effort (in terms of time) of using embedded feature traceability annotations during development?

This question measures the time it took to write embedded feature annotations during the development process.

- **RQ2.1:** What is the effort of recording annotations?
  This sub-question determines developer time spent initially on writing feature annotations during development.

- **RQ2.2:** What is the effort of editing and removing annotations?
  This sub-question examines the effort required to change or remove existing annotations based on manual observations of captured videos.

RQ3: How is the annotation process perceived by developers?

This question aims to gather qualitative data on how developers feel about and perceive the annotation process. Specifically, the aim is to understand the circumstances under which developers decide to add annotations to their work.

RQ4: How is the IDE-based tool support perceived by developers? (HAnS-specific)

This question examines the overall perception of the HAnS tool within the IDE across several aspects.

- **RQ4.1:** Difficulty ratings
  In this sub-question, developers' difficulty in using the HAnS tool is to be assessed.
- **RQ4.2:** Usability (efficiency, satisfaction)
  This sub-question explores the usability of the HAnS tool, focusing on efficiency and overall user satisfaction.
- **RQ4.3:** Usefulness
  This sub-question determines how developers find the tool useful in their workflow and code navigation.
- **RQ4.4:** Technical issues
  In this sub-question, document any technical issues or bugs developers may encounter while using the tool.
- **RQ4.5:** Suggestions for improvement.
  This sub-question collects developer recommendations for tool improvements based on their experience.

RQ5: How does the developers' experience and understanding of the annotation paradigm align with their mental model after being introduced to the concept?

This question examines the degree to which developers' expectations match their experience using the embedded feature annotations and their ability to adapt to the feature annotation paradigm introduced by the HAnS tool.

The formulated research questions will serve as a structure for the study, allowing us to investigate the accuracy, efficiency, perception, and usability of the HAnS tool and embedded feature annotations during software development.

3.3 Experimental Setup

The experimental setup encompassed several materials uploaded to the public GitHub repository named "hasanainq/HAnS-experiment-" [1]. In this section, we will explain

the essential materials used for the experiment, including "Subject Systems Introduction", "Tasks Design", "HAnS Tutorial Introduction", and "Installation Guidelines for the HAnS Plugin and Logger Tool".

3.3.1 Subject System Introduction

DART (viDeo gAme and song RenTal System): DART is software for a video game and music rental system. The company has previously been using old paper and pen to record the data about games, personnel, transactions, etc. With items being added to the system every day and the number of customers increasing, the company decides to digitalize its system by creating a simple Java-based console application. As a result, they created a software system called "DART " and distributed it to various franchise stores. DART can be used by three different types of users: Managers, Employees, and Customers. Managers are responsible for high-level tasks, such as adding, editing, and removing employees. Employees are responsible for registering customers, upgrading or downgrading customer memberships, adding items to the inventory, etc. Both managers and employees can view different statistics to monitor the sales in their respective stores, for example, the most profitable item, rental frequency of each item, best-reviewed item, etc. The system permits users to engage with it in various ways. They can rent video games and songs. They can also interact with one another and provide feedback on the items they have rented. The system specifications, which include a combination of feature descriptions and a feature model, give participants a list of feature descriptions and pointers to the source code where the features are implemented. For additional clarification, we direct your attention to the appendix in the DARTPlus Subject System Features document, which provides a comprehensive overview of the system's features.

The DARTPlus project was developed using the Java programming language. Figure 3.2 illustrates the feature model and enumerates the specific features included in DARTPlus. The total code base of the DARTPlus project consists of approximately **1500** lines of code, encompassing **18** different features.

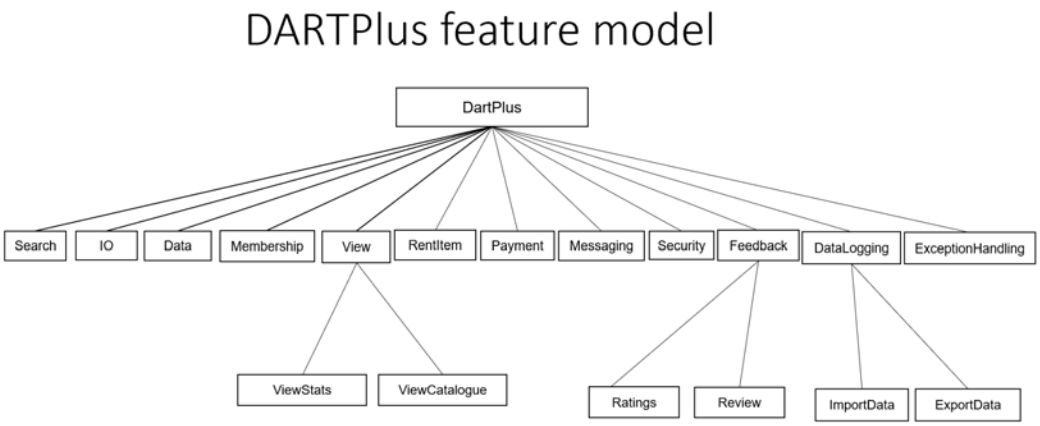


Figure 3.2: Feature model of the DARTPlus subject system.

3.3.2 Tasks Design

The tasks were designed to simulate real-world scenarios and were assigned to participants in a single experimental group. The intention was to permit participants to implement specific feature annotations, including writing a simple code and cloning from other parts of the source code. Additionally, they were required to perform maintenance tasks, such as refactoring, to observe the effects. Participants were required to complete some development tasks that involved writing embedded feature annotations, thereby recording their screens during the experiment. The Appendix Section (A) of the "Scenarios-Tasks" document provides more details regarding each task within the scenarios, as well as including the "dartData.txt" file. The tasks of the scenarios constituted of the following three parts:

- **Warm-up task:**
  The objective of this task is to familiarize the participants with the process of adding feature annotations using the HAnS tool. This will involve the following:

  1. Add a new feature mapping named "feature-to-folder".
  2. Add a new feature mapping named "feature-to-file".
  3. Navigate all the usages of a feature using the feature model.
  4. Adding a "feature-to-code" mapping via begin and end annotation for the feature annotation.

- **Implementing ImportData feature task:**
  The objective of this task is to add a new feature to the DARTPlus system, specifically implementing the ***ImportData*** functionality. This feature will enable the system to read data from the "dartData.txt" text file containing information about employees, songs, games, and customers. The data will then be registered accordingly. Employees and customers have dedicated array lists (employees and customers), whereas rentable items (songs and games) can be added to the same array list (i.e., items).

  For file reading, the following code was provided as a help reference:

```
1  public void readFile(ArrayList<Employee> employees,
   ↪  ArrayList<Customer> customers, ArrayList<Rentable> items, Scanner
   ↪  input){
2      try {
3          File dartData = new File("dartData.txt");
4          FileReader fr = new FileReader(dartData);
5          BufferedReader br = new BufferedReader(fr);
6          String line;
7          while((line = br.readLine()) != null) {
8              String[] dartInfo = line.split(";"); // parse data
9              //register employees, songs, and games based on the first
                   ↪  word of the line (employee, game, song)
10             //For registering, use methods this.registerEmployee(),
                   ↪  items.add(new Game or Song), and customers.add()
11         }
12     } catch (Exception var12) {
13         System.out.println("File does not exist");
14     }
15 }
```

- **Allow customers to rent movies by cloning the rental class task:**
  The objective of this task is to clone the class **Song ()** to create a new class, ***Movie***, within the DARTPlus system. This new class, ***Movie***, should be created with a "feature-to-file" mapping for the ***RentItem*** class, which should be implemented in the ***Movie.java*** file.

Subsequently, participants were required to annotate the implemented code with embedded feature annotations after each scenario. Moreover, after completing each scenario of the tasks, they were provided with an explanation of the expected outcomes, which helped them ensure that the required tasks had been completed and that the results had been understood.

3.3.3 HAnS Tutorial Introduction

As part of the experimental procedure, each participant was presented with a PowerPoint slideshow that provided an overview of the HAnS tool. The "HAnS Tutorial Introduction" file is provided in the Appendix Section (A). The slides offer guidance on how developers can utilize embedded feature annotations directly within their code assets, such as files, folders, code fragments, individual lines of code, and navigating feature models. Furthermore, the slides offer a detailed explanation of the types of embedded annotations supported by the HAnS tool, illustrated with screenshots and tutorials. Some illustrations are shown in Figures 3.3, 3.4, and 3.5 below.

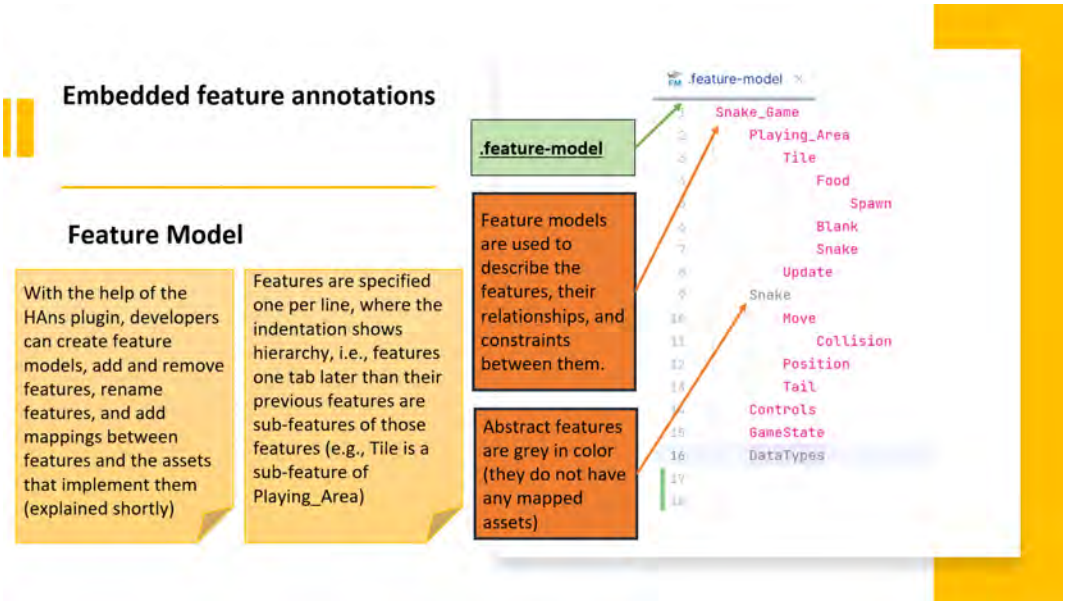


Figure 3.3: Embedded feature annotations.

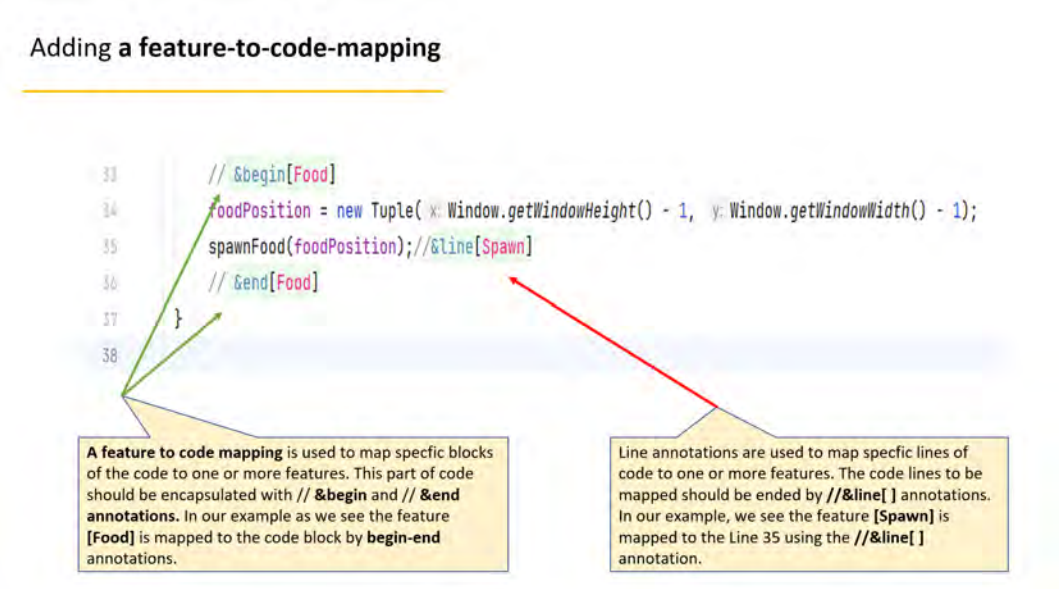


Figure 3.4: Adding a feature-to-code mapping.

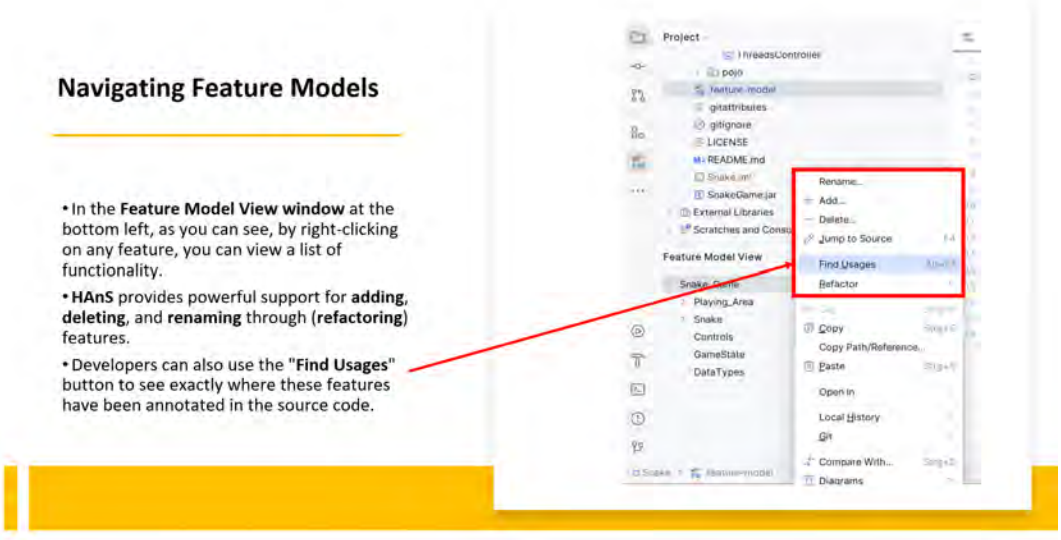


Figure 3.5: Navigating feature models.

3.3.4 HAnS and Logger Tools Setup Guidelines

The document "Installation Guidelines for the HAnS Tool" contains detailed instructions for installing the HAnS (Helping Annotate Software) tool. It explains how developers can set up the HAnS and Logger plugins within the Integrated Development Environment (IDE). In the case of a manual installation, developers needed to access Settings/Preferences > Plugins > Install plugin from disk... [1].

The "HAnS annotationLogger Guidelines" document delineates the essential steps for configuring the Logger tool, which were instrumental in data collection [1]. This process automatically generates a JSON file for data logging, where each feature annotation using the HAnS tool is recorded in a JSON file. Once the IDE has been closed, the logging stops automatically, and the logged JSON file, identified by the name "annotationLogger", is saved to a specified local directory [33]. For further details on the HAnS and the Logger Tools setup guidelines, refer to Appendix (A). Experiment Materials, the "Installation Guidelines for the HAnS Plugin and Logger Tool".

3.4 Participants

3.4.1 Participant Selection Criteria

For the recruitment of participants in this experiment, an approach was chosen based on personal networks, academic staff in the software engineering chair, and WhatsApp groups, with a focus on students as the main target group. This method aimed to attract a diverse and heterogeneous group of participants in order to capture a broad spectrum of user experiences with feature documentation techniques and programming. The programming knowledge criterion for participation in the experiment was a lack of any previous experience with feature annotations using the HAnS tool. This criterion guarantees the impartiality of all participants' evaluations and assessments conducted during the user study of using the HAnS tool for writing embedded feature annotations. This targeted selection is intended to collect authentic and representative data that provides a clear picture of user acceptance and experience with Embedded Feature Annotations and the usability HAnS tool.

3.4.2 Recruitment Process

The study aimes to recruit approximately 10–15 participants from various disciplines, including scientific researchers, practitioners, software developers, and students at the PhD, MSc, and BSc levels. The participants were required to complete

a questionnaire and engage in tasks related to developing and annotating embedded features using the HAnS tool, as well as responding to the posed questions. Additionally, the participants were asked to answer some experience questions in the questionnaire to provide qualitative data. The collected data helps to gain insights into the participants' experiences.

3.5 Procedure of the Experimental Study

3.5.1 Preparation Phase

At the study's outset, the experimental setup included several components designed to introduce the experiment materials to the participants in the experimental group. These components were uploaded to a public GitHub repository named "hasanainq/HAnS-experiment-" [1], which also contains supplementary materials, in addition to those presented in the Appendix Section (A).

This introductory phase was crucial for creating a solid understanding of implementing the feature annotation methods using the HAnS tool, which was to be investigated, and for preparing the participants for the experimental setup processes. The "Subject Systems Introduction" and "HAnS Tutorial Introduction" documents were provided to the participants to review, allowing them to familiarize themselves with the DartPuls subject system and the HAnS tool. In addition, participants were instructed to clone and open the "DartPuls" project folder in the IntelliJ IDE. Additionally, they were provided with the "HAnS-0.0.4.zip" file, which contained the integrated logger tool to be set up in the IntelliJ IDE. The main repository also includes guidelines for installing HAnS and Logger tool configurations [1].

Finally, participants were given the "Scenario - Tasks" for performing the assigned tasks in different scenarios and the "dartData.txt" file document, which was utilized in the tasks. Further details can be found in the "(A). Experiment Materials" Section of the Appendix.

3.5.2 Pilot Study

Before beginning the implementation phase of the experiment, a pilot study was conducted to test and refine the experimental setup and procedures. We endeavored in the pilot study to ensure the clarity of the questionnaire and the preparation steps. These steps consisted of installing the IntelliJ IDE, setting up plug-ins and using the HAnS (including the Logger), identifying and resolving complications, and recording the time spent conducting the pilot study. Two participants, selected from undergraduate and graduate computer science students, were assigned to complete

the same tasks designed for the main experiment. After completing the tasks, the participants provided feedback on the clarity of the instructions, the usability of the tools, and any challenges they encountered. They were also encouraged to suggest potential improvements and refinements to the experimental process. The data obtained from the pilot study, including screencast recordings and timestamps from the Logger tool, were carefully extracted and reviewed to ensure the data collection process's accuracy, correctness, and comprehensiveness.

The pilot study outcomes assisted us in refining and solving issues that could arise in the main experiment. For instance, based on the feedback and observations, we could enhance the clarity of task instructions by redesigning each scenario of the tasks and the HAnS tutorial introduction to use the embedded annotations refined procedures in the main experimental study. Additionally, measuring the experiment's duration helped us to determine the required time for the participants to complete the experiment.

3.5.3 Execution Phase

The execution of the experiment focused first on setting up the HAnS tool, including the Logger tool, within the suitable version of the IntelliJ IDE. Each participant was required to download the GitHub repository containing all the experiment documents [1]. The participants were given an introduction tutorial to the HAnS tool and familiarization handouts regarding the subject system. This process was crucial to avoid possible learning effects or biases that the results could have been influenced to minimize. Immediately after finishing the setup process on each laptop system, participants were asked to fill out a standardized questionnaire. The questionnaire was designed to record the participants' immediate impressions and evaluations of the user-friendliness of writing embedded feature annotations using the HAnS tool. Next, in the crucial part of the experimental study, participants conducted an experiment where they recorded all their screens while performing annotation tasks during development. Each participant was then asked to upload both files for screen recording and log files for data collecting and analysis purposes. This approach provided qualitative data and deeper insights into their personal experiences. Integrating quantitative and qualitative data collection techniques offered a comprehensive basis for the user study of using the HAnS tool for writing embedded feature annotations.

3.6 Data Collection

This section outlines the data collection methods and tools employed in the experimental study, including questionnaires, observations, and participant feedback regarding feature documentation during the experiment.

3.6.1 Quantitative Data

Logger Tool: A review of the Logger tool's source code reveals that the start and stop logging functions are integrated within the tool. The annotation functions are designed to be executed via keystrokes, and they are also tracked and added to the total annotations when the user highlights text with the mouse. The listeners are responsible for logging various events, such as text highlighting and right-clicking for both line and block annotations. Subsequently, these events are then implemented to calculate the time taken by the user to highlight text and right-click on "Surround with Feature Annotation" for (line []) or (begin [] end []) annotations. This functionality enables the annotations to be created instantly. Subsequent changes to the annotation, such as deletions, are also recorded. However, these changes were not included in the total annotation time. The Logger tool records and calculates the annotations when the text begins with the specified syntax.

Below are some lines from the Logger tool's source code:

```
1 private static final String LINE_ANNOTATION = "// &l";
2 private static final String END_ANNOTATION = "// &e";
3 private static final String BEGIN_ANNOTATION = "// &b";
```

Moreover, when the deleted text matches a specific pattern of annotations, such as "line" and "being or end," then the time of deletion is calculated and stored in the "annotationLogger" file. These logging times are then also stored in the JSON file. The following lines of code present the process of determining whether the deleted text matches the annotation pattern, including the potential for spaces:

```
1 Pattern pattern = Pattern.compile("^\\s*//\\s*&(.+?)\\[");
2 Matcher matcher = pattern.matcher(deletedText);
```

To summarize the Logger tool workflow, it records the times of keypresses and mouse clicks. Should these events lead to an annotation, the required time to complete it will be calculated by subtracting the times for the first keypress from the time of the last for that annotation. Besides, there is an implementation for recording the time of "quick annotations." This will be logged when the user starts code highlighting and right-clicks on "Surround with Feature Annotations".

The Logger tool was employed to automatically record timestamps while writing annotation tasks within the IntelliJ IDE. These timestamps were logged into log files in the JSON format while completing the tasks during the experiment.

Screen Recordings: During the experiment, participants' screens were recorded while they completed development tasks and implemented embedded feature annotation tasks. These recorded videos captured the entire process within the IntelliJ

IDE, including the time taken to complete each task. The time durations were manually extracted from the videos using Clipchamp Microsoft software. The extracted timestamps for the required tasks related to embedded feature annotations were then utilized.

Clipchamp is a Microsoft software video editor suitable for creating YouTube videos, slideshow videos, and other content. Its user-friendly interface is ideal for students lacking experience in video editing [8]. For the experiment, the free version of Clipchamp was used to manually measure the precise durations of the screencast series recorded during the completion of experiment tasks. Clipchamp allows tracing the video duration in a format of "MM: ss. ms/ HH:mm.ss", which is the equivalent of "minutes: seconds. milliseconds/hours: minutes. seconds". This is consistent with the development design time unit in milliseconds of the Logger tool.

Although these screen recordings were primarily used to gather the actual times required to perform a task, they also generated insights that proved helpful for our qualitative analysis.

Questionnaire: The online Google form was used to administer the questionnaire to the participants. This form facilitated the collection of quantitative and qualitative data regarding participant demographics, technical background, perceived difficulty of tasks, and opinions on embedded feature traceability annotations. Closed-ended questions were used to collect responses regarding participant demographics and technical background, programming expertise, and the rating of their knowledge in feature annotations. A series of Likert scale questions were included to gain insight into the perceived difficulty of the tasks, opinions on embedded feature traceability annotations, and experience and benefits. The types of questions are listed as follows:

- **Demographic and Technical Background Questions:** The demographic questions were used to gather fundamental participant data. This questionnaire records technical background information, such as the participant's level of education, experience with programming, and knowledge of feature annotations. The demographic and technical background data is crucial to characterizing the study population, identifying possible demographic influences, and identifying factors affecting user experience. It also allows for assessing familiarity with feature location tasks and proficiency in programming languages.
- **Likert Scale Questions:** The Likert scale was utilized to assess perceptions of the HAnS tool, the difficulty of annotation tasks, questions regarding the embedded feature traceability annotations, and experience and benefits.
- **Open-Ended Questions:** These Provided crucial detailed feedback regarding embedded feature annotations, the difficulty of adding annotations, their own experiences in this regard, and suggestions for improvement.

Structure: The structure was designed first with a Pre-Task Questionnaire to collect basic information about the participant's demographic and technical backgrounds. Second, a Post-Task Questionnaire was used to gain insight into the participants' experiences regarding the annotation tasks and their feedback on the HAnS tool. Finally, a follow-up questionnaire was included as an optional open-ended question for further elaboration regarding the HAnS tool, embedded feature annotations, and overall experience.

Data Collection Process: The participants received a Google form link via Skype, email, or WhatsApp. To ensure anonymity and confidentiality, anonymized responses and reporting of only aggregated data were performed.

3.6.2 Qualitative Data

Open-Ended Questionnaire Responses:

In the post-task questionnaires, participants were also asked open-ended questions to provide additional insights about the difficulty of adding feature annotations, detailed feedback on their experience with the HAnS tool, and suggestions for improvement.

3.7 Data Analysis

The methodology of this study involved a detailed analysis of data collected from a user study experiment using the HAnS and Logger tools, along with tasks and a questionnaire administered to participants, including students and academic researchers. The aim was to gather both quantitative and qualitative data for comprehensive analysis. Quantitative data were collected through screen recordings, capturing participants' screens while they performed feature annotation tasks using the HAnS tool and through the annotation logger, which tracked the time and effort participants spent on writing embedded feature annotations. Additionally, empirical evidence was collected as participants performed the tasks, with their interactions being logged for subsequent analysis.

Qualitative data were gathered via a questionnaire containing open-ended questions, which participants answered after completing the tasks. This provided insights into their experiences and opinions regarding the usability of the tools and the annotation process. The data analysis process began with examining the screen recordings and Logger tool data to measure the effort spent on writing annotations and assess the Logger tool's accuracy. Timestamps were manually extracted from the screen recordings and compared with those in the log files generated by the Logger tool, aiming to verify its accuracy. The time required for writing feature annotations and

navigating within the IntelliJ IDE was measured, and these times were organized into tables and Excel spreadsheets for further analysis.

Statistical analysis was conducted to interpret the data. The Shapiro-Wilk test [29, 13] was employed to assess the normality of the data collected from the logger files and manual screen recordings for each feature annotation task.

This test is crucial for determining between a non-normal and a normal distribution:

Normal Distribution: If the output data from both methods are normally distributed (p-values > 0.05), parametric tests like the paired t-test will be used to compare the means.

Non-Normal Distribution: Non-parametric tests such as the Wilcoxon signed-rank test will be used if the data is not normally distributed (p-values < 0.05).

In all cases where there was a disparity in normality between the two sets of data, as determined by comparing the p-values, indicating one as normal distributed and the other as non-normal distributed, the null hypothesis was rejected. This suggests that the data were non-normal. Therefore, the non-parametric tests, such as the Wilcoxon signed-rank test [9, 41], were the most suitable for this situation.

In the Wilcoxon signed-rank test, the obtained p-value is typically compared to a set significance level, which is often referred to as alpha (α). This is commonly set to 0.05.

1. **Significant Difference:** If the p-value is ≤ 0.05, it indicates a statistically significant difference between the paired observations.
2. **No Significant Difference:** If the p-value is > 0.05, it indicates no statistically significant difference between the paired observations.

The Wilcoxon signed-rank test was applied to assess the significance of differences between the various annotation types, including feature model, file annotation, folder annotation, block annotation, line annotation, and total annotation. The p-values were compared to 0.05 to evaluate significance.

The accuracy of the Logger tool was evaluated by comparing the recorded annotation times in the log files with the corresponding times in the screencasts. The percentage of time spent on annotations compared to the total development time was calculated using the equation:

$$\text{Percentage of total spent time in Logger tool} = \frac{\text{Total annotation time (ms)}}{\text{Total developing time (ms)}} \times 100$$

Each participant's data was presented using the percentage equation to demonstrate the time Logger spent on annotations relative to the total development session.

To ensure data uniformity in this study, the Logger tool's unit of time, expressed in milliseconds, is used. The time taken for annotations recorded by the Logger tool was manually compared with the time recorded in videos, and the Wilcoxon test was applied to determine if there was a significant difference between the various annotation types and the time measurements. P-values were compared to 0.05 to evaluate the statistical significance of the results.

3.8 Limitations and Challenges

Only 13 participants were identified in this study, which represents a relatively small sample size and may potentially influence the statistical power of the empirical study. However, the participants were primarily students with programming experience, which adds value to the work and increases the relevance of their contributions. Because of these issues, our findings may not generalize well to a larger population. To address these limitations, we have reported effect sizes and confidence intervals to provide a clearer picture of the results. We applied robust statistical techniques to improve the reliability of our estimates. We acknowledge that further research with larger sample sizes is needed to confirm and extend our findings. We also faced challenges in finding and preparing an appropriate experimental subject system. After investing time in developing the subject system for library management and explaining its features in Software Product Line Engineering (SPLE), including the code, we encountered new bugs in connecting the database using the HAnS tool in the IntelliJ IDE. This problem forced us to change the system before conducting the experiment. Despite these limitations and challenges, this controlled experiment provides valuable insights into using embedded feature annotations with the HAnS tool for documenting feature locations in code.

4 Results

In this chapter, we present the results of our experimental study, organized according to our research questions outlined in Section (3.2) and the questionnaire. Each research question has a subsection presenting our experiment's relevant results.

4.1 RQ1: Accuracy

To address RQ1 (How accurate is the Logger tool for recording embedded feature annotations?), the data extracted from the Logger tool and videos were analyzed using a statistical test in the R language and Microsoft Excel sheets. Differences between the Logger tool and the videos of normally distributed populations were assessed by analysis of the Shapiro-Wilk test. Based on the normality of the data, the differences between paired observations were then assessed by implementing the Wilcoxon signed-rank test. The results of the analysis were conducted using statistical test techniques. In this section, we present the most interesting results and the most important implemented methods.

Statistical Testing Methods:

The following table presents the results of the Shapiro-Wilk normality test for each annotation type. It indicates whether the data from logger files or manual screens is classified as normal or non-normal distribution.

Table 4.1 below provides a clear overview of the normalization status of the data collected from log files and manual recording videos for each annotation type. It shows that the null hypothesis of normality was rejected since the outputs were one as normal distributed and the other as non-normal distributed or both as non-normal distributed.

Annotation Type	Logger Data (p-value)	Manual Screen Data (p-value)	Normality
Feature Model	0.1461	0.01233	One non-normal
File Annotation	0.003246	0.07582	One non-normal
Folder Annotation	0.0003657	0.0001185	Both non-normal
Block Annotation	0.0004415	0.058	One non-normal
Line Annotation	0.001456	0.05913	One non-normal
Total Annotation	0.0007183	0.5512	One non-normal

Table 4.1: Output results of the Shapiro-Wilk test.

Based on the Shapiro-Wilk test in output table 4.1, we conclude that the data are non-normally distributed.

Annotation Type	Logger Data Normality	Video Manual Data Normality	Recommended Test
Feature Model	Normal	Non-Normal	Wilcoxon Signed-Rank Test
File Annotation	Non-Normal	Non-Normal	Wilcoxon Signed-Rank Test
Folder Annotation	Non-Normal	Non-Normal	Wilcoxon Signed-Rank Test
Block Annotation	Non-Normal	Normal	Wilcoxon Signed-Rank Test
Line Annotation	Non-Normal	Normal	Wilcoxon Signed-Rank Test
Total Annotation	Non-Normal	Normal	Wilcoxon Signed-Rank Test

Table 4.2: Statistical test selection.

Table 4.2 indicates that the data are non-normal for either the logger or video manual data. Therefore, the recommended statistical test is the Wilcoxon Signed Rank Test.

In the Wilcoxon signed rank test, a p-value is typically compared to a significance level (often referred to as α). This level is usually set at 0.05. If a p-value is equal to or less than this level, it indicates a significant difference between the paired observations.

	Feature Model		**File Annotations**		**Folder Annotations**		**Block Annotations**		**Line Annotations**		**Total Annotations**	
Ps	**Logger tool**	**Videos**	**Logger tool**	**Videos**	**Logger tool**	**Videos**	**Logger tool**	**Videos**	**Logger tool**	**Videos**	**Logger tool**	**Videos**
P1	5101	7000	21148	20000	-	1500	18762	14000	3771	4000	46161	46500
P2	19656	20000	28005	27000	894	800	17995	18000	8837	10000	75387	75800
P3	49036	54000	53093	48000	579	800	20271	25000	-	14000	102708	102800
P4	11234	11000	29517	25500	503	900	75375	54000	20782	21000	132678	112400
P5	15350	13000	36894	38000	1526	2000	30786	22000	-	-	81510	75000
P6	25803	27000	73186	66500	358	800	12370	10000	-	4000	110945	108300
P7	16017	16000	15272	16300	21869	22500	10662	8500	-	-	63820	63300
P8	8845	18000	32468	33000	546	700	18869	16300	-	-	58522	61000
P9	14184	11000	108149	50000	6358	5500	125518	59000	-	-	254209	132500
P10	-	-	31347	29500	6326	4500	31702	32500	-	13800	69375	80300
P11	24069	23000	21807	21200	2756	2200	31412	26500	6314	5000	77796	77900
P12	25263	13000	15321	16250	11128	11000	16192	14500	20829	23900	85826	78650
P13	13672	13400	17965	18700	451	465	39007	34300	14995	15500	80401	82365

Table 4.3: Time comparison between Logger tool and videos for all 13 participants.

Table 4.3 summarizes the time taken by 13 participants to complete various types of annotations using a logger tool and video recordings. The annotations include feature model, file, folder, block, line annotations, and the total time required for annotating. The comparison provides an overview of the logger tool's efficiency compared to video recordings. For a more detailed analysis of the accuracy of the annotations, see Appendix Section (B). Times for validating the accuracy of the Logger tool.

After presenting the findings of the initial evaluation, it is evident from Table 4.4 that, in many cases, there are no significant variances in the comparison between the Logger tool and the manually recorded videos. Through the analysis of the line annotations, it can be interpreted that the results are likely similar according to the p-value = 0.058, which is close to the conventional significant level of a = 0.05. This indicates that the tool is relatively accurate in the line annotations. However, noticeable differences were observed in the block annotations.

Annotation Type	Logger vs. Manual Videos (p-value)	Interpretation
Feature Model	0.968	No significant difference: Logger tool accurate
File Annotation	0.094	No significant difference: Logger tool accurate
Folder Annotation	0.635	No significant difference: Logger tool accurate
Block Annotation	0.013	Significant difference: Logger tool may be inaccurate
Line Annotation	0.058	Potential difference: Logger tool is approximately accurate, and further investigation is needed
Total Annotation	0.454	No significant difference, except for block annotations

Table 4.4: Interpretation results for each annotation type.

According to the results of our experiment, the Logger tool is generally considered accurate. However, in real-world scenarios, the cases are very different from how the Logger tool works appropriately using the HAnS tool. Even though the Logger tool shows a high degree of accuracy and effectiveness, Table 4.4 provides a summary for interpreting the results for each annotation type.

4.2 RQ2: Effort

The user study experiment presents both qualitative observations and quantitative results to address the research questions RQ2 (What is the effort (in terms of time) of using embedded feature traceability annotations during development?), RQ2.1 (4.2.1 What is the effort of recording annotations?), and RQ2.2 (4.2.2 What is the effort of editing and removing annotations?).

4.2.1 Effort of using embedded feature traceability annotations during development and the effort of recording annotations

This research study evaluates the effort (in terms of time) required to use embedded feature traceability annotations during development using the HAnS tool. Thirteen participants were assigned the same tasks as detailed in Tasks Design (3.3.2) of the Methodology chapter and were asked to complete the annotations for these tasks throughout the development process. Some participants made errors in their

annotations, which were then corrected by editing the annotations, resulting in longer annotation times.

Table 4.5 illustrates that the mean time across all tasks was approximately 84370 ms, with notable variation among participants. These observations underscore the importance of user training and familiarity with feature annotations, particularly the HAnS tool, in order to minimize the effort (in terms of time) required for using embedded feature traceability annotations during development.

	Feature Model		**File Annotations**		**Folder Annotations**		**Block Annotations**		**Line Annotations**		**Total Annotations**	
Ps	**Logger tool**	**Videos**	**Logger tool**	**Videos**	**Logger tool**	**Videos**	**Logger tool**	**Videos**	**Logger tool**	**Videos**	**Logger tool**	**Videos**
P1	5101	7000	21148	20000	-	1500	18762	14000	3771	4000	46161	46500
P2	19656	20000	28005	27000	894	800	17995	18000	8837	10000	75387	75800
P3	49036	54000	53093	48000	579	800	20271	25000	-	14000	102708	102800
P4	11234	11000	29517	25500	503	900	75375	54000	20782	21000	132678	112400
P5	15350	13000	36894	38000	1526	2000	30786	22000	-	-	81510	75000
P6	25803	27000	73186	66500	358	800	12370	10000	-	4000	110945	108300
P7	16017	16000	15272	16300	21869	22500	10662	8500	-	-	63820	63300
P8	8845	18000	32468	33000	546	700	18869	16300	-	-	58522	61000
P9	14184	11000	108149	50000	6358	5500	125518	59000	-	-	254209	132500
P10	-	-	31347	29500	6326	4500	31702	32500	-	13800	69375	80300
P11	24069	23000	21807	21200	2756	2200	31412	26500	6314	5000	77796	77900
P12	25263	13000	15321	16250	11128	11000	16192	14500	20829	23900	85826	78650
P13	13672	13400	17965	18700	451	465	39007	34300	14995	15500	80401	82365
Mean	**17556**	**17415**	**37244**	**31535**	**4100**	**4128**	**34532**	**25738**	**5810**	**8381**	**95334**	**84370**
Std Dev.	**12173**	**12973**	**26780**	**15303**	**6298**	**6263**	**32102**	**15804**	**8083**	**8064**	**52902**	**23620**

Table 4.5: Mean (average) and Std Dev. (standard deviation) for the Logger tool and videos of the effort in terms of time for each participant.

It was also observed that the minimum total development time required to complete the experiment was approximately 22 minutes for Participant 8, with the greatest time required for Participant 3 being approximately 1 hour and 12 minutes, with an average time of 44 minutes.

The time required for development is recorded in the log file of the Logger tool, allowing the mean time across all 13 participants to be calculated. Table 4.6 shows the percentage of time spent writing annotations using the HAnS Tool, which is estimated at approximately 3.5% of the total mean time for all development periods.

	Total annotation time		**Total Developing time**	
Participants	**Logger tool**	**Videos**	**Logger tool**	**Videos**
Participant 1	46161	46500	1487297	1445079
Participant 2	75387	75800	1767893	1868093
Participant 3	102708	102800	3773677	4326069
Participant 4	132678	112400	1285628	1740070
Participant 5	81510	75000	3812627	3878030
Participant 6	110945	108300	3110786	3627050
Participant 7	63820	63300	3115269	2896053
Participant 8	58522	61000	1378505	1377003
Participant 9	254209	132500	5787350	2997056
Participant 10	69375	80300	3594069	4089040
Participant 11	77796	77900	1856612	1709026
Participant 12	85826	78650	1706352	1656043
Participant 13	80401	82365	2427531	2962086
Mean	**95334**	**84370**	**2700277**	**2659284**
Average time spent on feature annotations during development			**3.5%**	**3.2%**

Table 4.6: Average time spent annotating features during development for all 13 participants, including the mean of total Logger and video times.

To determine the effort required for recording annotations, the total time taken for recording feature annotations was measured using the Logger tool and verified through video recordings. Table 4.5 summarizes the recorded times.

- Mean time for total annotations (Logger Tool): 95334 ms.
- Mean time for total annotations (Video Recordings): 84370 ms.
- Standard deviation (Logger Tool): 52902 ms.
- Standard deviation (Video Recordings): 23620 ms.

These results indicate that, on average, participants took slightly longer to write feature annotations as measured by the Logger tool than the video recordings. This discrepancy may be attributed to some participants not using the tool features correctly, which led to some annotations being inaccurately logged.

4.2.2 Effort of editing and removing annotations

In order to evaluate the effort involved in editing and removing annotations, it was necessary to manually analyze the screen recordings of the participants as they performed these tasks during the experiment. This was because the Logger tool did not measure the editing time and was inaccurate in recording the time taken to remove annotations. Qualitative observations from the video recordings highlighted several major points. It was observed that participants frequently edited and removed annotations, thereby increasing the total effort in terms of time. The most prevalent tasks included correcting block and line annotations, mapping features, and editing existing annotations. The use of common edits and removals, such as the cutting and pasting of annotation markers, directly impacted the time spent on annotation tasks. For example, some participants needed to edit or remove the automatic insertion of double slashes (//) in block annotations and remove features to edit them. Those participants who were more familiar with the HAnS tool or quickly understood its functions (e.g., Participants 4 and 12) could edit and remove annotations more efficiently, resulting in lower times for these tasks. Conversely, participants with less experience took longer to complete these tasks, indicating that familiarity with the HAnS tool significantly impacts the effort in terms of time.

Manual Observations:

- Participants such as Participant 2 effectively used the "Surround with Feature Annotation" function, with minimal deletions and efficient editing.
- Participant 5 encountered issues where some annotations were not logged accurately, requiring additional time for correction.
- Participant 9 and others who relied on writing the annotations without the use of the automatic HAnS tool suggestions faced an increased time requirement due to the necessity of multiple edits and deletions.

An example of writing annotations using the HAnS tool suggestion is shown in Figure 4.1 below.

Although the HAnS tool supports embedded feature traceability annotations, the effort (in terms of time) is influenced by the frequency and complexity of editing and removing annotations and the user's familiarity with the HAnS tool.

Figure 4.1: HAnS tool suggestion for writing annotations.

4.2.3 Effort required for navigation and annotation

Using embedded feature annotations requires both navigating and annotation writing efforts (in terms of time). In this study, we measured the time spent by 13 participants navigating the IDE and annotating during various development tasks. This process was captured in video recordings to ensure accuracy and detail.

During the development phases, participants faced the dual challenge of thinking about what to annotate and deciding how best to annotate it, tasks that required thoughtful consideration and added to the overall time invested in navigation. To measure these efforts, we manually extracted data from the video recordings and organized it into tables within the appendix Section (C). Effort for annotating, including the navigating time. These tables categorically detail the time spent on each annotation type in milliseconds.

To analyze the time spent navigating and annotating, the aggregated data were summarized in table 4.7. This table presents not only the recorded times for each annotation type, but also the total time spent navigating and annotating. Furthermore, it provides the mean and standard deviation of annotation times across all participants, offering a comprehensive overview of the distribution and consistency of effort.

	Feature model		**File annotation**		**Folder annotation**		**Block annotation**		**Line annotation**		**Total annotation**	
Ps	**Ann. Time**	**Nav. Time**	**Ann. Time**	**Nav. Time**	**Ann. Time**	**Nav. Time**	**Ann. Time**	**Nav. Time**	**Ann. Time**	**Nav. Time**	**Ann. Time**	**Nav. Time**
P1	7000	2500	20000	8300	1500	2000	14000	8200	4000	3500	46500	24500
P2	20000	2000	27000	5300	800	1500	18000	25800	10000	7200	75800	41800
P3	54000	9900	48000	7500	800	3400	25000	-	14000	-	102800	20800
P4	11000	5000	25500	14200	900	3000	54000	45400	21000	8500	112400	76100
P5	13000	8400	38000	12900	2000	2600	22000	19700	-	-	75000	43600
P6	27000	7400	66500	32250	800	4200	10000	21900	4000	7500	108300	73250
P7	16000	15500	16300	8800	22500	6300	8500	45000	-	-	63300	75600
P8	18000	64700	33000	25100	700	2000	16300	48000	-	-	61000	139800
P9	11000	2000	50000	17100	5500	2800	59000	86900	-	-	132500	108800
P10	-	-	29500	9000	4500	7800	32500	36200	13800	14300	80300	67300
P11	23000	6000	21200	12500	2200	6000	26500	30000	5000	1300	77900	55800
P12	13000	5500	16250	11700	11000	4700	14500	41500	23900	13800	78650	77200
P13	13400	6000	18700	10600	465	1000	34300	41300	15500	6500	82365	65400
Mean	**18867**	**11242**	**31535**	**13481**	**4128**	**3638**	**25738**	**37492**	**12356**	**7825**	**84370**	**66919**
Std Dev.	**12399**	**17254**	**15303**	**7556**	**6263**	**2053**	**15804**	**19802**	**7263**	**4498**	**23620**	**32401**

Table 4.7: Time measurements for Ann. (annotating) and Nav. (navigating) in video recordings for all 13 participants.

In Table 4.8, the total time spent by the participants on navigation and annotation tasks is highlighted. The table shows that, on average, approximately 5.69% of the total development time was spent on these activities. This analysis provides a detailed understanding of the time and effort involved in using the HAnS tool and demonstrates the importance of efficient navigation and annotation in the development process.

Participants	**Total time (Annotation + Navigation)**	**Total Developing time**
Participant 1	71000	1445079
Participant 2	117600	1868093
Participant 3	123600	4326069
Participant 4	188500	1740070
Participant 5	118600	3878030
Participant 6	181550	3627050
Participant 7	138900	2896053
Participant 8	200800	1377003
Participant 9	241300	2997056
Participant 10	147600	4089040
Participant 11	133700	1709026
Participant 12	155850	1656043
Participant 13	147765	2962086
Mean "AVG"	**151290**	**2659284**
Std dev.	**43570**	**1080320**
AVG time spent on a total of annotating and navigating during development	**5.69%**	

Table 4.8: Total time required for navigating and annotating, including the average time during development.

4.3 Participants Knowledge

All study participants were introduced to the HAnS tool, including embedded feature annotations. They were then asked to perform development tasks in Java. Before starting the development tasks, participants completed a series of warm-up annotation tasks, followed by the main tasks, which were divided into two scenarios. The tasks are described in the subsection entitled "Tasks Design" (3.3.2) within the Methodology chapter.

After completing the tasks, participants were required to fill out a questionnaire to provide feedback on their experiences with the HAnS tool for feature traceability annotations. The tool's evaluation is based on the questionnaire responses, which are presented with diagrams for clarity.

Occupation:

The participants were mainly students with programming backgrounds enrolled in bachelor's, master's, and doctoral programs. Their occupations were recorded to gain insight into the context of their experience with feature annotations while programming simple Java code.

A total of thirteen participants were asked to indicate their current occupation. The distribution of occupations among the participants is shown in Figure 4.2 below. The largest group, comprising 46.2% of the participants, represents six individuals pursuing a bachelor's degree (BSc or BEng). The second largest group includes five participants (38.5%) pursuing a master's degree (MSc). The smallest group, comprising 15.4% of participants and two individuals, engaged in PhD studies. Notably, none of the participants were categorized within the academic/scientist or practitioner categories. This data indicates that most participants are engaged in undergraduate and master's studies, with a smaller portion involved in PhD. studies.

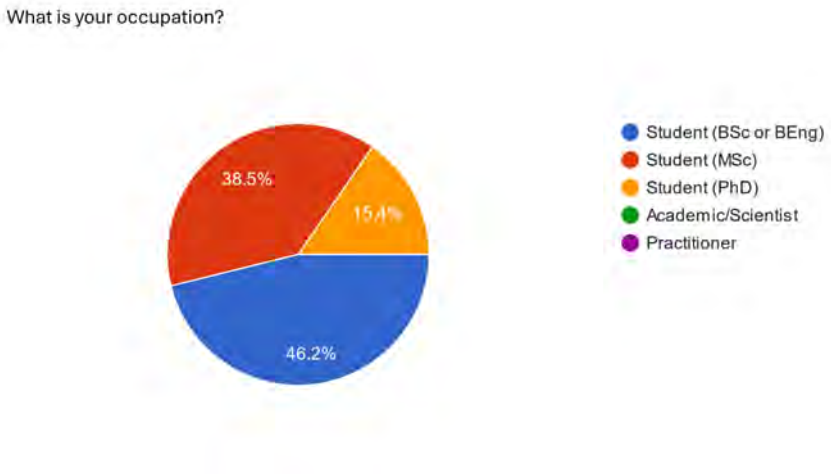


Figure 4.2: Occupation.

Programming Expertise:

All thirteen participants were requested to rate their programming expertise on a scale from 1 (I can't program at all) to 5 (I am an expert). The results, shown in Figure 4.3, indicated that seven participants rated themselves at level 3 on the scale, four at level 4, and two at level 2. The average rating for the participants was 3.15, which indicated a medium level of programming expertise. This data provides insight into the participants' abilities in the development field.

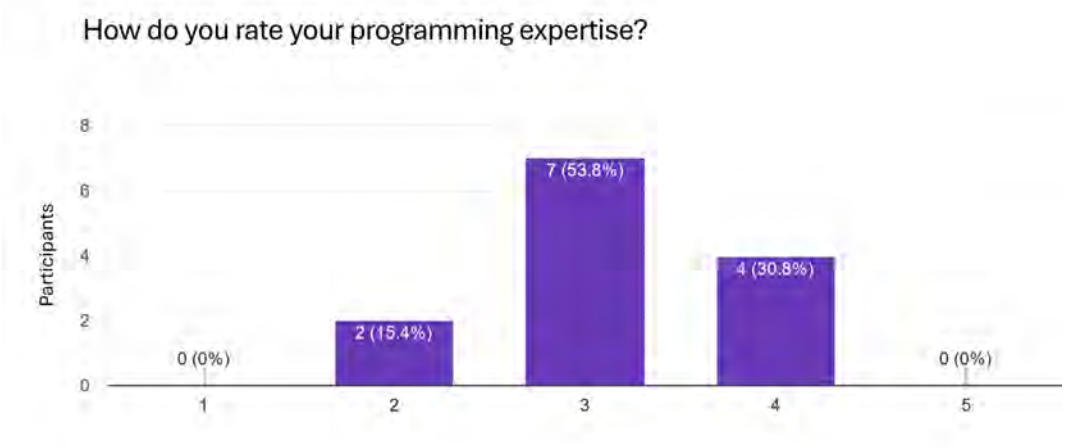


Figure 4.3: Rating of programming expertise.

Knowledge of Feature Annotations:

Thirteen participants were additionally requested to rate their knowledge of feature annotations on a scale from 1 (I am a beginner) to 5 (I am an expert). As depicted in Figure 4.4, five participants rated themselves at 1 on the scale, five at 2, and three at 3. The average score indicated a relatively limited familiarity with the concept of feature annotations, with an overall average of 1.85.

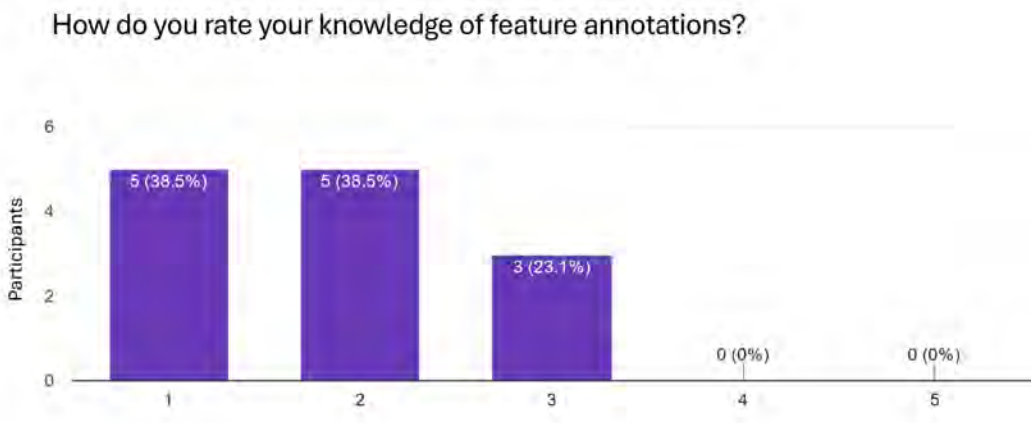


Figure 4.4: Knowledge of feature annotations.

4.4 RQ3: Participants Perceptions

To assess the difficulty of the tasks completed during development, participants in the questionnaire were asked to respond to five questions regarding the difficulty of tasks related to writing embedded feature annotations using the HAnS tool in the experiment. Table 4.9 presents the questions related to the tasks' difficulty.

	Questions related to the difficulty of tasks
1	How difficult was it to perform "Adding a feature to a feature model"?
2	How difficult was it to perform "Adding a feature-to-code mapping" using block annotations?
3	How difficult was it to perform "Adding a feature-to-code mapping" using a line annotation?
4	How difficult was it to perform "Adding a feature-to-file mapping"?
5	How difficult was it to perform "Find usages of a feature"?

Table 4.9: Questions regarding the difficulty of tasks related to feature annotations.

The participants were asked to rate each task's perceived difficulty on a scale from 1 (very easy) to 5 (very difficult). Table 4.10 presents the results, including the difficulty ratings provided by each participant. As well as providing the standard deviation and mean values for the task difficulty ratings.

	Task 1 difficulty	**Task 2 difficulty**	**Task 3 difficulty**	**Task 4 difficulty**	**Task 5 difficulty**
P1	2	2	1	1	1
P2	1	2	2	2	1
P3	2	2	2	2	3
P4	2	2	2	3	2
P5	1	1	1	1	1
P6	1	1	1	1	2
P7	1	1	1	2	1
P8	2	3	3	2	1
P9	1	2	2	2	1
P10	3	3	2	2	1
P11	1	3	2	3	3
P12	2	2	2	2	3
P13	1	1	2	2	1
Mean	**1.54**	**1.92**	**1.77**	**1.92**	**1.62**
Std Dev.	**0.66**	**0.76**	**0.60**	**0.64**	**0.87**

Table 4.10: Assessment of ratings, including difficulty ratings for various tasks performed using the HAnS tool (1: very easy, 5: very difficult).

The HAnS tool was generally well received by the participants, with the majority rating the tasks as either (easy) or (very easy). Among the tasks, Task 1 and Task 5 received the highest percentage of (very easy) ratings, making them the easiest. Although Tasks 2 and 4 had slightly higher mean difficulty ratings, they still fell within the easy range. Additionally, the low overall standard deviations indicate that the difficulty ratings were consistent across all participants. These consistent low difficulty ratings affirm that the tool effectively simplifies the task of annotating features in code, making it a valuable resource for developers.

4.4.1 RQ3.1: How difficult was it to decide when to add annotations?

Participants were asked to assess the challenge of deciding when to add annotations. The following most intriguing elaborations are presented in detail below:

> *"When working on a feature, that was easy. However, there was quite some guidance on what steps to perform."*

This participant found the process uncomplicated when focused on a particular function, primarily because of the clear guidance provided on the steps.

> *"Since I was not sure about the current annotation style, it was overall neutral but without respect to best practices it was rather very easy."*

This response demonstrates a neutral appreciation for the difficulty of the task, likely due to uncertainty about the existing annotation style. Despite this, the participant found the process relatively straightforward, although they acknowledged the potential discrepancy with established best practices.

> *"Deciding on what to annotate is always a bit difficult in my opinion, because if it's overdone the code can become cluttered. In this case it was only a few so I think it can be very useful when searching for features."*

This participant highlighted the difficulty of determining which elements to annotate, noting that while this could result in an overly complex code, in this case, with only a few annotations, it could be highly beneficial for identifying specific features.

The responses of the 13 participants to the question "How difficult was it to decide when to add annotations?" revealed a range of experiences and insights. While some found the process easy or straightforward, others encountered challenges, primarily due to a lack of familiarity with the embedded feature annotations and uncertainty about the need for annotations. Additionally, there was a need to balance the level of detail in annotations to avoid cluttering the code.

4.4.2 RQ 3.2: How difficult was it to decide where to add annotations?

Similarly, participants rated the difficulty of deciding on where to add annotations. Responses varied. Important elaborations will be presented in more detail in the following section:

> *"Neutral. the specifics of how the annotations will be used isn't 100% clear so it's tricky to know where sometimes."*

This response reports difficulty in determining the most appropriate locations for the annotations due to a lack of clarity regarding the intended purpose of the annotations. This resulted in a neutral experience.

> *"That was easy, too. I only annotate code I touched. Also there was no interference with other code features."*

This participant found the process of annotating the code easy. This was because the code was annotated only with the code with which the participant interacted, and there was no interference with other code features.

> *"In this case it was easy if done correctly. But I can imagine more complicated projects where features are intertwined or not that clearly separable."*

It was observed that annotations could be easily added in their current context, but it was also acknowledged that more complex projects with interconnected features might pose certain difficulties.

> *"After I decided when to annotate the decision of where to set the feature annotation was not difficult at all."*

The participant found that once a decision had been made regarding the timing of annotation additions, the whereabouts of their placement were identified as an easily resolvable issue.

The responses to the question "How difficult was it to decide where to add annotations?" show that people had different experiences. Some found it simple, while others faced challenges, mainly depending on their knowledge of the embedded feature annotations. Few found it hard to decide where to annotate. This diversity in responses underscores the importance of clear guidelines and familiarity with the annotation process.

Participants generally found the annotation process manageable. However, few participants expressed difficulty determining when and where to add annotations. This indicates that while the tool provides helpful guidance, further improvements could enhance usability.

4.4.3 What do you think are the benefits of embedded feature traceability annotations?

The participants shared various opinions about the benefits of embedded feature traceability annotations. This is exemplified by the following interesting quotes:

> *"Makes managing constantly changing databases easier."*

This response points out that embedded feature traceability annotations help manage databases that change frequently, likely by providing a clear and consistent way to track changes and features.

> *"Improved Understanding of the codebase, easier to find features and their implementations, easier to understand how features are interconnected."*

The annotations facilitate a deeper comprehension of the codebase, allowing for the identification of features and understanding of their interconnections.

> *"They save time and effort in understanding how the code is related."*

It is recognized for the time and effort saved by annotating, especially by clarifying the relationships within the code.

> *"Easier documentation, especially for later documentation. The annotation is directly in the source code and not somewhere else (word documentation, etc.), not related to a 'person's knowledge."*

This demonstrates that annotations enhance documentation by integrating it into the source code, thereby facilitating accessibility and eliminating the need for external documents or individual expertise.

> *"Easy navigation, Easier refactoring, Improved security, implements the principle of 'divide and conquer'."*

This response revealed multiple benefits, including improved navigation, easier refactoring, improved security, and implementation of the "divide and conquer" principle.

> *"Better documentation and for the project, new people are able to read the code and understand it better."*

It shows that better documentation, facilitated by annotations, makes it easier for new developers to understand the code.

> *"Due to this feature traceability, it is now much easier to find the implementation of a feature in several software assets. I find it very useful."*

This response notes the usefulness of feature traceability in helping to locate the implementation of features in a wide variety of software assets.

> *"It is fast to find the code needed to do modification to parts of the codebase."*

It explains how embedded feature traceability annotations can be used to quickly find and fix fragments of code in the code base.

The participants identified several benefits of embedded feature traceability annotations. These benefits were summarized as improved understanding and management of the codebase, more straightforward navigation and refactoring, time-saving in finding and understanding features, and better documentation. These annotations are particularly useful for new developers and maintaining a clear project overview.

4.4.4 In your opinion, what are the advantages of browsing embedded feature traceability annotations?

Participants also provided their perspectives on the advantages of browsing embedded feature traceability annotations, explaining how it enhances project navigation and comprehension. Below are a few quotes:

> *"Work like comments in the code and have similar strengths. They also help to find related code."*

This means that the embedded feature traceability annotations act like comments and provide similar benefits, such as clarifying the code and helping to locate related code segments.

> *"That makes coding easier when there are many contributors to the same project, helps avoiding errors as well."*

This aspect of these annotations facilitates coding in collaborative environments, thereby reducing errors and enhancing coordination among multiple contributors.

> *"Browsing makes it easier to find features in code, instead of manually searching 1000+ lines of code."*

The text highlights the efficiency gains that can be achieved using annotations, as they facilitate the identification of specific features without the necessity for manual spotting of extensive lines of code.

> *"More intuitive to me as programmer than navigating like inside of a UML class- diagram. Also faster in the development process. Projects are overall easier to read."*

The evidence indicates that browsing annotations are a more intuitive and quicker method than using UML class diagrams, which facilitates project understanding and reading.

> *"Browsing the annotations helps getting an overview over the code and to find parts of a feature. I think it will be especially useful on larger projects, even more so with code written by someone else."*

It appears that annotations can be a helpful tool for providing an overview of the code and assisting in finding feature locations, which could be particularly beneficial for larger projects and when dealing with code written by others.

> *"It fits well for other programmers to know how methods in Implementation work."*

It seems that annotations can help other programmers understand how methods work within the implementation.

> *"It makes the process of writing code faster, by reducing the search time with only adding minimal time consumption to add the annotations."*

This response shows that developers appreciate the speed gained in the coding process by minimizing the time spent searching for specific sections of code while requiring minimal effort to add annotations.

To summarize the quotes previously presented, the participants pointed out several advantages of browsing embedded feature traceability annotations. These advantages encompass improved project organization and comprehension, optimized decision-making and troubleshooting, intuitive navigation, and streamlined coding practices. These annotations facilitate the identification of features within the code, promoting teamwork and faster development.

4.4.5 How intuitive is it to browse embedded feature traceability annotations?

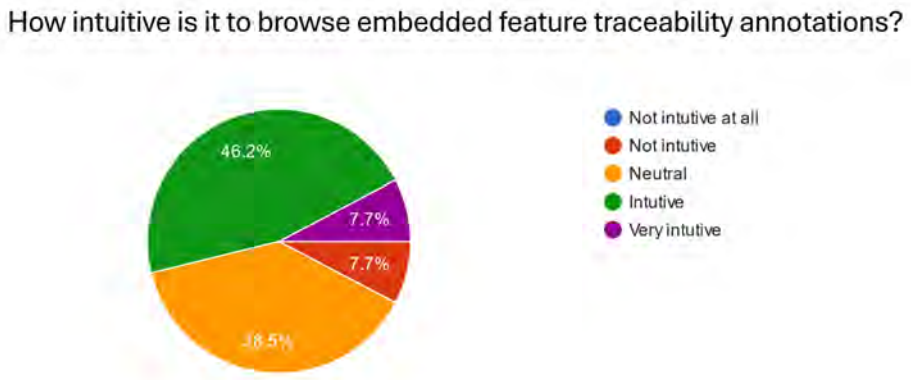


Figure 4.5: Intuitiveness of browsing embedded feature traceability annotations?

In general, participants found the process of browsing embedded feature traceability annotations to be intuitive and user-friendly. Most ratings fell within the (Intuitive) to (Neutral) range. As shown in Figure 4.5, 46.2% of the participants rated the process as (Intuitive), 38.5% as (Neutral), and 7.7% as (Very intuitive). A smaller proportion of participants found it less intuitive, with 7.7% rating it as (Not Intuitive). This indicates that while some participants had a neutral experience, the majority found the process intuitive and user-friendly, providing reassurance about the feature traceability.

4.4.6 How would you rate the usability of using embedded feature traceability annotations?

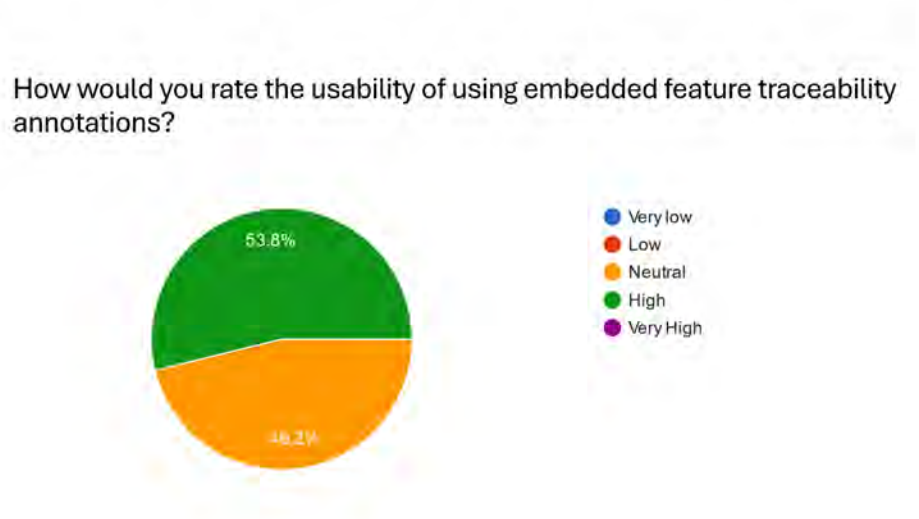


Figure 4.6: Rating the use of embedded feature traceability annotations?

Thirteen participants rated the usability of embedded feature traceability annotations highly. Figure 4.6 illustrates that the scores were predominantly between (High) and (Neutral). Specifically, 53.8% of participants rated it as (High), and 46.2% as (Neutral). This suggests that many participants found the embedded feature traceability annotations effective and reliable for their tasks, instilling confidence in its usability.

4.4.7 The process of adding annotations enhances my understanding of the codebase.

A total of 13 participants' responses to the questionnaire were evaluated. Participants generally agreed that the process of adding annotations enhanced their understanding of the codebase, as shown in Figure 4.7, with most ratings fell between (Agree) and (Strongly Agree). Specifically, 61.5% of participants rated it as (Agree) and 23.1% rated it as (Strongly Agree). Additionally, 15.4% of participants remained (Neutral). This finding demonstrates that embedding feature annotations not only enhances the documentation but also significantly helps in comprehending the code structure and logic.

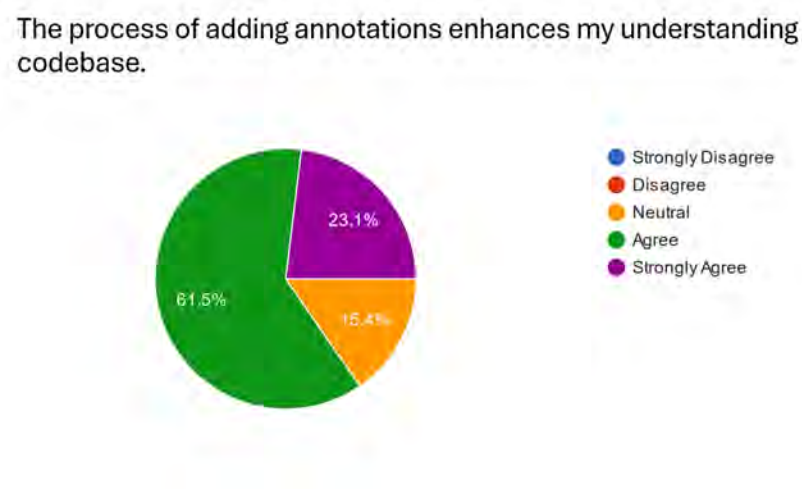


Figure 4.7: The process of adding annotations has improved their understanding of the process.

4.5 RQ4: Experience and Benefits

The participants provided various insights into their experience and the benefits of using the HAnS tool for embedded feature traceability annotations. Their responses highlight numerous advantages of the tool, ranging from improved traceability to better code navigation.

4.5.1 What do you think are the benefits of the HAnS tool?

Participants provided their responses on the benefits of the HAnS tool, elaborating their perspectives. A few of these perspectives are presented below:

> *"Makes the coding process easier, especially for new coders."*

This underscores that the HAnS tool makes coding manageable, especially for beginners needing additional guidance and support.

> *"Allows developers to explicitly document features and their relationships and helps locate the code."*

The participant believes HAnS allows developers to document features and how they relate to each other, making it easier to locate specific code.

> *"Easy feature traceability and annotation over a whole project (also bigger scale)."*

HAnS provides easy-to-track and annotate features that work well in large projects.

> *"HAnS helps me as a developer to keep the locations of features implemented in code clear for later usages. Especially for other developers who will start working on the same project. It will be much easier for them to understand the codebase."*

Here, the participant explains that the HAnS tool makes it easier to maintain clarity about the locations of implemented features, which is particularly useful for onboarding new developers and ensuring they can quickly understand the codebase.

> *"It increases the velocity by enabling the developer to find the code faster."*

The report indicated that the HAnS tool increased the development process's speed, enabling developers to locate code more rapidly.

The participants commended the HAnS tool for facilitating the coding process, enhancing feature clarity, improving code traceability, facilitating project organization, and simplifying navigation. These benefits have the potential to improve collaboration, faster development, and overall project management.

Participants provided answers on various aspects of RQ4, which focused on developers' perception of IDE-based tool support specific to HAnS. These answers included ratings on difficulty, usability, usefulness, technical issues, and suggestions for improvement.

Critical aspects of the HAnS experience, such as difficulty, usability, and usefulness, were evaluated and presented in Table 4.11.

Question	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
Have you ever used the HAnS tool for feature annotation before practicing in the study?	No	Yes	Yes	No	Yes	No	No	No	No	No	Yes	Yes	Yes
How difficult is it to learn how to use the HAnS tool?	3	2	3	2	3	1	2	2	3	2	2	2	2
How did you find the user interface of the HAnS tool in terms of usability and user-friendliness?	2	2	2	2	2	2	2	3	2	3	3	1	2
How useful is HAnS based on the tasks you performed in the experiment?	2	1	2	4	2	5	2	1	2	2	3	1	2

Table 4.11: HAnS tool ratings

More than half of the participants had no prior experience with HAnS. They rated the usability and user-friendliness of the HAnS tool highly, with scores mainly between 1 (Very User-Friendly) and 3 (Neutral). The learning difficulty of the HAnS

tool was also rated between 1 (Very Easy) and 3 (Neutral), indicating that participants generally found the tool easy to use and effective for their tasks. Most participants found the HAnS tool useful for managing feature annotations, with scores mainly between 1 (Very Useful) and 5 (Not Very Useful), suggesting that it adds value to the development process.

4.5.2 Technical issues or bugs encountered with the HAnS tool

The participants were queried to answer the question, "While using the HAnS tool for feature annotation, did you encounter any technical issues or bugs?" Only three participants reported encountering technical issues and provided substantiating quotes:

> *"The model view was not on by default which made it confusing."*

This issue shows that the HAnS tool's default settings can lead to confusion. The Model View is important for understanding the code structure and should be enabled by default.

> *"I had to name files even though that does not seem to be necessary. When adding the annotations manually, the closing is not automatically updated."*

This response identifies two issues: the unnecessary requirement to name files and the lack of automatic updates for closing annotations when they are added manually. Both of these can interrupt the workflow and lead to potential errors.

> *"GitHub Co-Pilot was annoying but also helping with annotations."*

These issues were noted for further improvement. Other participants either answered "no" or passed on the question, indicating no significant technical issues.

The feedback highlights specific areas for potential improvement, including adjustments to default settings, enhancements to the manual annotation process, and other modifications to improve the HAnS tool's usability and efficiency.

4.5.3 Suggestions for improvement:

Several participants offered suggestions for enhancing the HAnS tool, including improvements to the logging mechanism, more intuitive prompts, and integration with the IDE to reduce bugs. Specific suggestions included:

> *"Make visualization better for adding and deleting features with their annotations."*

This proposal suggests the necessity for improved visualization tools within HAnS, making adding and deleting features along with their annotations easier, thereby optimizing the user experience and reducing potential errors.

> *"Syntax highlighting when selecting begin/end annotation."*

It is recommended here that syntax highlighting be added for annotations, which would make them more transparent with the "begin and end" annotations, assisting in avoiding errors and improving code readability.

> *"None specific but needs to be better clarified."*

The feedback pointed out that some users found the current features and functionalities of HAnS unclear, suggesting a need for better documentation or more intuitive user guidance.

> *"Automating functionality for tasks such as adding, removing,' or moving features annotations."*

This suggests the potential for automating common tasks related to feature annotations, including adding, removing, and modifying annotations.

These suggestions were noted for future improvements, making the HAnS tool more efficient, easier to use, and better integrated into developers' workflows, enhancing productivity and satisfaction.

4.5.4 Experience with the HAnS tool

The most interesting elaborations related to the question "Feel free to share your experiences with the HAnS tool. " are presented below in quotation marks:

> *"It was tricky at first to figure out what help is included in the tool. Especially marking code and then adding an annotation was pretty useful. There are no options to go to the next or previous occurrence of code with the same feature."*

This response reported that the users may find it challenging to understand the available functionality of the HAnS tool. However, once users become familiar with annotating using the tool, such as code-block annotations, they will likely find it valuable. Also, a potential issue was identified regarding the lack of navigation options to facilitate the movement between occurrences of the same feature in the code and suggested as an improvement to enhance the usability of the HAnS tool.

> *"I find HAnS really helpful and can imagine myself using it in daily software."*

One of several participants elaborated on the positive user experience with the HAnS tool, where the participant found the HAnS tool very helpful and could see its potential for regular use in their daily software development activities.

These experiences illustrate the initial learning curve associated with the HAnS tool and its eventual usefulness once users become familiar with using feature annotations. Improvements in navigation options and more explicit help documentation could enhance the user experience, making the tool more user-friendly and practical for daily use.

4.6 RQ5: Alignment of Developers' Experience and Understanding with their Mental Model

The responses of 13 participants in the questionnaire were evaluated concerning the RQ5.

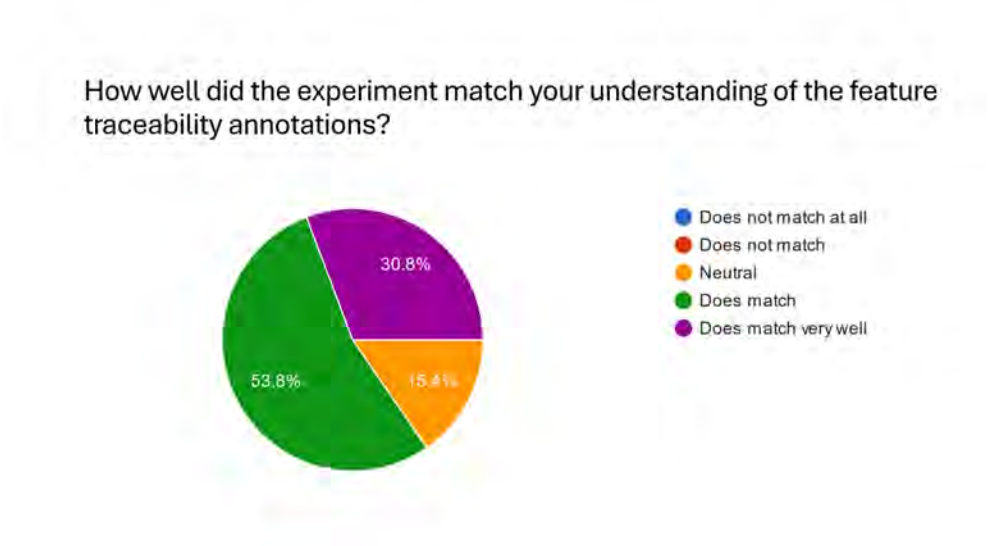


Figure 4.8: Alignment of experiment and understanding?

Participants' comprehension of the annotation paradigm and its alignment with their mental model was evaluated after the presentation of the overall concept. Figure 4.8 visualizes that most participants felt that the experiment was well-matched to their understanding of feature traceability annotations. Specifically, 53.6% responded with (Does match) while 30.8% responded with (Does match very well), indicating a comprehensive understanding of the feature traceability annotations. 15.4% of participants responded with (Neutral). These positive experiences enhanced their overall experience with the tool, and they found the annotations in

the experiment helpful in organizing their code. Overall, the results show that the experiment effectively met participants' expectations and understanding of the annotations.

5 Discussion

This chapter discusses the findings and implications of a user study experiment that evaluated the usability of the HAnS and Logger tools in documenting embedded annotations within the IntelliJ integrated development environment. It provided specific observations and insights gathered during the evaluation process. Key topics covered include discrepancies in the total times recorded by the Logger tool and videos during development, challenges associated with detecting block and line annotations, the impact of navigation and annotation efforts on the results, experiment execution times, and users' perceptions regarding the usability of embedded feature annotations.

5.1 Discrepancies in Total Development Time

Notably, the total development times recorded by the Logger tool and the one captured by the video differ. One of the reasons explaining these discrepancies lay in the fact that some video recordings (made by participants) started at different phases of their development process, either before or after launching the Integrated Development Environment (IDE). The Logger tool starts recording once the IDE is active, which may not fully capture the initial preparation phases of the development.

5.2 Inconsistencies in Total Annotation Time

The data from the Logger tool corresponding to the total annotation time were generally consistent with the video data, but there were some notable exceptions. For instance, Participant 9 recorded 254 seconds of the total annotation time using the Logger tool, whereas the video only captured 132 seconds. Similarly, Participant 4 recorded approximately 133 seconds of the total annotation time using the Logger tool, while the video recorded 112 seconds. Such discrepancies were also observed in other participants, particularly with block annotations.

Several factors likely contributed to these discrepancies. First, human errors occurred. Participants spent more time than necessary on certain tasks, such as thinking or pausing, which the Logger tool recorded as active annotation time. Second, Logger sensitivity has been an issue. The Logger tool was more sensitive to quick

annotations within the IDE, resulting in the recording of annotation time, even when the participant was not actively annotating.

5.3 Differences in Block and Line Annotations

A discrepancy in the recording times of annotations was observed between the Logger tool and the video data. Participants 4 and 9 exhibited particularly notable differences in annotation times. The following reasons likely contributed to this discrepancy:

- **Complexity of Block Annotations:** Block annotations often involve extensive changes and code reviews, resulting in longer recording times. The Logger tool captures these feature annotations by starting keypresses and recognizing the specified syntaxes. For example, it recognizes "// &l"; &line, "// &b"; &begin, and "// &e" ; &end.
- **Annotation Process:** Participants spent additional time reviewing or editing the block/line annotations, which inflated the Logger's recorded times.

This also explains why the block annotation times recorded by the Logger tool were inaccurate. The complexity and additional steps in creating block annotations contributed to this inaccuracy. Few participants invested more time and effort to ensure accuracy and comprehensiveness, which the Logger properly documented as extended annotation time.

To improve the accuracy of block annotation times, future releases of the Logger tool would benefit from integrating more advanced tracking mechanisms that can detect most annotations within real-world software development scenarios. These mechanisms would facilitate processing various annotations, including those involving copying and pasting feature annotations.

5.4 Unaccounted Block and Line Annotations

The Logger tool missed some block and line annotations, particularly when participants used copy-pasting. In real-world development scenarios, copy-pasting annotations can bypass the Logger's tracking mechanism, leading to discrepancies. This behavior highlights a limitation of the Logger tool, which cannot reliably detect copy-paste annotations. The inclusion of these cases in the evaluation may not provide a fair comparison. It is essential to distinguish between manually created annotations and those added through copy-pasting, as the latter cannot be accurately tracked by the Logger.

The act of copying and pasting a moment in milliseconds was inherently difficult to measure. Consequently, there are no Logger implementations for a copy and paste, resulting in a relatively short time when performing this kind of annotation. This is comparable to the speed of the refactor feature annotations. It is important to note that just annotating the "//&begin" does not result in a complete annotation, nor does just annotating the "//&end", result in the Logger tool recording. Furthermore, the Logger tool will not record the action if the annotation is not performed correctly or in the correct format for both line and block annotations. This is the case regardless of whether the user presses the suggestion on the keyboard tab or selects the HAnS tool suggestions with a mouse click, as shown in Figure 4.1 Deletions of "//&block" and "//&line" annotations were logged into the log file with the following annotation types: "//&block" and "//&line".

5.5 Effort in Navigating and Annotating

The study measured the time spent on navigation and annotation using the embedded feature annotations through the HAnS tool. On average, participants allocated 5.69% of their total development time to these activities. This relatively small percentage suggests that while the HAnS tool incurs some additional time, it is not excessively time-consuming, especially when considering its potential benefits in terms of code comprehension and maintenance.

Additionally, it was observed that the fastest participant was number 1, who took approximately 1.11 minutes to annotate and navigate, while the slowest participant was number 9, who took approximately 4 minutes.

Table 4.8 shows the variability in the time spent on annotation and navigation, possibly due to differences in individual practices with the HAnS tool. Some participants required more time to become proficient in the annotation process, while others were quicker, possibly reflecting their diverse thinking and decision-making processes.

The findings suggest that the use of embedded feature annotations via the HAnS tool is manageable in terms of the additional effort required and that it can be integrated into the development workflow without causing significant disruption. Further research in real-world scenarios could investigate methods to improve the efficiency of the annotation process and the efforts involved. In addition, the impact of feature annotation usability in large-group samples on long-term code maintenance and collaboration can be explored.

Despite some variability, the consistency of the findings across participants suggests the potential for wider adoption. However, the results also indicate areas

for improvement. For example, the user interface could be improved, and annotation decisions could be better supported, which would further reduce the time required.

5.6 Experimental Execution Times

The experimental study showed that Participant 8 completed the experiment in the shortest time, approximately 22 minutes. In contrast, Participant 3 took the longest time, about 1 hour and 12 minutes, with an average time of 44 minutes for all participants.

The variability in the experiment execution times can be attributed to participants approaching the tasks differently. For instance, Participant 3 may have employed a more meticulous development approach, prioritizing accuracy over speed, which resulted in a longer total development time. Conversely, Participant 8 used different techniques to complete the development tasks, such as copying and pasting the code, which affected the total development time.

5.7 User Perception and Qualitative Insights

Despite these discrepancies, the study collected substantial qualitative and quantitative data on user perception. Participants generally found that the embedded feature traceability annotations improved their understanding by allowing them to browse the source code, highlight significant code fragments, and minimize development time. The HAnS tool was also useful for creating embedded feature annotations. The HAnS tool enabled better code organization, easier navigation, and a better understanding of the code base.

6 Threats to Validity

The potential threats to the validity of our study can be classified into four main categories: external validity, internal validity, construct validity, and conclusion validity. Each category deals with specific concerns related to the study's design, execution, and interpretation. Understanding these threats is crucial for critically evaluating the reliability and applicability of research results.

External Validity. This refers to the extent to which the results of this study can be generalized to other contexts. Our experiment focused on evaluating the HAnS tool for documenting embedded annotations using the Logger tool in the IntelliJ IDE. The subject system used for the user study, DARTPlus, is relatively simple, consisting of 18 features and approximately 1500 lines of code. This simplicity may limit the generalizability of our findings to more complex systems with larger codebases and more complicated feature interactions. However, the choice of a simple subject system was intended to minimize misunderstandings and facilitate the tasks, considering the time constraints. The subject system was carefully chosen to mimic real-world software systems.

Internal Validity. This study employs a single-subject design, which provides numerous data points to support statistically valid conclusions. This design allows all participants to experience the same conditions and tasks, thereby facilitating direct comparisons within the group. Screencasts and log files were recorded throughout the study to gain insights and ensure that any differences observed were due to the functionalities of the tools rather than other variables.

However, many participants' unfamiliarity with the HAnS tool, the embedded feature annotations, and the subject system may limit the experiment's internal validity. The introduction of the tool and annotations may have affected participants' performance due to the learning curve associated with these new elements. Despite these limitations, the consistent use of the Logger tool to capture detailed data about participants' interactions with the HAnS tool helped maintain the integrity of the study.

The accuracy of time measurements depends on the precision of video analysis. Manual data extraction from recording videos is subject to human error, which could affect the reported times. To mitigate this, we conducted multiple reviews of the recording videos to cross-verify the time measurements.

Construct Validity. The study was evaluated to determine whether the methodology accurately measured the intended concept, namely the usability and effectiveness of the HAnS tool for documenting embedded feature annotations. To operationalize these constructs, specific tasks and questions were designed to elicit participants' experiences with the HAnS tool. The tasks were informed by a pilot study that balanced the need for comprehensive data. It was conducted to avoid any misunderstandings that participants might encounter during the experiment and to validate the clarity of the tasks. The tasks included adding, browsing, and understanding annotations, which represent typical activities developers engage in. While the simplicity of the subject system, DARTPlus was necessary for the experiment, it may not fully capture the challenges of using the HAnS tool in more complex environments.

The time spent on navigation and annotation was measured in milliseconds, but the cognitive load and the mental effort required were not directly assessed. Future studies could incorporate subjective measures, such as surveys or interviews, to gain insights into the perceived difficulty of using the tool.

Our setup included using the HAnS tool, recording the annotations with the Logger tool, and screen-casting in the IntelliJ IDE to provide a realistic environment. However, the participants' familiarity with the HAnS tool could influence their performance. Lastly, the tool's interface and the tasks helped establish construct validity, ensuring participants' experiences reflected their understanding of embedded feature annotations.

Conclusion Validity. To support the conclusion's validity, we employed well-accepted statistical tests to validate the results.

7 Conclusion

This thesis investigated the usability of embedded annotations for documenting feature locations in code. The goal of the study was to measure the effort required to write embedded feature annotations using the HAnS tool during development in terms of time. HAnS is a Java-based plugin that facilitates the annotation process by allowing the user to specify feature mappings to assets of different types, including files, folders, and code fragments [22].

Due to the significant time developers spend nowadays understanding the code base and locating features during the software development process, we conducted a user study for writing embedded feature annotations using the HAnS tool. This was achieved by recording timestamps for completing embedded feature annotations as well as the total development time for these annotations within the code while performing development tasks. The first goal was to verify the accuracy and efficiency of the Logger tool and evaluate the efficiency and usability of the HAnS tool. Additionally, the time required for navigation was measured to assess the total effort required for navigating and annotating during development. We collected a combination of quantitative and qualitative data through this experimental study and the questionnaire. A total of 13 participants took part in the experiment investigating the use of embedded feature annotations with the HAnS tool.

The study results provide valuable insights into the performance of the Logger tool in real-world scenarios. While the tool demonstrated accuracy in most cases, statistical tests revealed discrepancies in the p-values associated with the block and line annotation functionality. Potential differences in the p-values according to the statistical analysis are presented in Table 4.4 of the interpretations in the Results chapter.

Moreover, the study indicates that the average time spent on *recording* feature annotations across all 13 participants during the development process is approximately 3.2% of the total mean time for all development periods, as demonstrated in Table 4.6. The study also demonstrates that, on average, 5.69% of the total development time was spent on *both* navigation and annotation using the HAnS tool. This indicates that although minimal additional effort was required to use the HAnS tool for documenting embedded feature annotations, it was relatively small compared to the total development time.

Integrating embedded feature annotations into the development process requires minimal additional effort and can be incorporated without significant disruptions.

The participants from our controlled experiments, representing different levels of programming expertise, provided positive feedback on the HAnS tool and embedded annotations. Furthermore, HAnS appears to be a viable option for integrating feature annotations into developers' daily development workflow without imposing a significant additional effort on the developers.

Although the Logger tool provided satisfactory performance in several functional areas, further development is necessary in certain areas, such as the recording of block and line annotations. The results of our controlled experiment indicated the potential for discrepancies in certain cases. Consequently, we suggest that the Logger tool should be compatible with all releases of IntelliJ IDEA, including the latest versions. Similarly, we recommend that the HAnS tool be compatible not only with the IntelliJ IDEA but also with other integrated development environments (IDEs). Therefore, further development is necessary to enhance the tool's functionality and accuracy.

The study's findings indicate that the incorporation of embedded feature annotations can enhance feature traceability and management, potentially increasing productivity and improving code maintainability. This could subsequently reduce developers' efforts. Future research investigations into the usability and impact of feature annotations on long-term code maintenance and collaboration within larger groups would provide deeper insights.

We hope that this user study will provide valuable insight to practitioners aiming to document feature locations within the code.

List of Figures

List of Tables

Bibliography

[1] Hasanain Al-Aassi. *hasanainq/HAnS-experiment-*. `https://github.com/hasanainq/HAnS-experiment-`. [Accessed: July 20, 2024]. 2024.

[2] B. Andam, A. Burger, T. Berger, and M. R. Chaudron. "FLOrIDA: Feature LOcatIon DAshboard for Extracting and Visualizing Feature Traces". In: *Proc. Eleventh Int. Workshop Var. Modeling Softw.-Intensive Syst., VAMOS '17*. Eindhoven, Netherlands: Association for Computing Machinery, 2017, pp. 100–107. ISBN: 9781450348119. DOI: `10.1145/3023956.3023967`. URL: `https://doi.org/10.1145/3023956.3023967`.

[3] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. "Recovering Traceability Links Between Code and Documentation". In: *IEEE Transactions on Software Engineering* 28.10 (2002), pp. 970–983. DOI: `10.1109/TSE.2002.1041053`.

[4] S. Apel and C. Kästner. "An overview of feature-oriented software development". In: *Journal of Object Technology* 8.5 (2009), pp. 49–84.

[5] T. Berger, M. Chechik, T. Kehrer, and M. Wimmer. "Software evolution in time and space: Unifying version and variability management (Dagstuhl seminar 19191)". In: *Dagstuhl Reports* (2019).

[6] T. Berger, D. Lettner, J. Rubin, P. Grünbücher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki. "What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines". In: *SPLC*. 2015.

[7] K. Chen, W. Zhang, H. Zhao, and H. Mei. "An approach to constructing feature models based on requirements clustering". In: *Proceedings of the International Conference on Requirements Engineering*. 2005, pp. 31–40.

[8] *Clipchamp Video Editor for Windows*. `https://clipchamp.com/en/windows-video-editor/`. [Accessed: May 19, 2024].

[9] G. Divine, H. J. Norton, R. Hunt, and J. Dienemann. "A review of analysis and sample size calculation considerations for Wilcoxon tests". In: *Anesthesia Analgesia* 117.3 (2013), pp. 699–710.

[10] A. L. Egel, C. H. Barthold, J. L. Kouo, and F. S. Maajeeny. "Single-subject design and analysis". In: *The Reviewer's Guide to Quantitative Methods in the Social Sciences*. Routledge, 2018, pp. 417–433.

[11] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. “Enhancing clone-and-own with systematic reuse for developing software variants”. In: *ICSME*. 2014.

[12] M. W. Godfrey and D. M. German. “The past, present, and future of software evolution”. In: *Frontiers of Software Maintenance*. 2008, pp. 129–138.

[13] E. González-Estrada and W. Cosmes. “Shapiro–Wilk test for skew normal distributions based on data transformations”. In: *Journal of Statistical Computation and Simulation* 89.17 (2019), pp. 3258–3272.

[14] J. Hayes and A. Dekhtyar. “Humans in the Traceability Loop: Can’t Live With ’Em, Can’t Live Without ’Em”. In: *Proceedings of the 3rd Workshop on Traceability in Emerging Forms of Software Engineering*. 2005.

[15] H. Jansson and J. Martinson. *HAnS: IDE-based editing support for embedded feature annotations*. `https://hdl.handle.net/20.500.12380/302926`. Master’s Thesis. [Accessed: July 20, 2024]. 2021.

[16] *JetBrains Plugins*. `https://plugins.jetbrains.com/`. [Accessed: July 20, 2024].

[17] H. Jordan, J. Rosik, S. Herold, G. Botterweck, and J. Buckley. “Manually Locating Features in Industrial Source Code: The Search Actions of Software Nomads”. In: *International Conference on Program Comprehension (ICPC)*. IEEE, 2015, pp. 174–177. DOI: `10.1109/ICPC.2015.26`.

[18] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. CMU/SEI-90-TR-21. Software Engineering Institute, 1990.

[19] J. Krüger, T. Berger, and T. Leich. “Features and How to Find Them: A Survey of Manual Feature Location”. In: *Software Engineering for Variability Intensive Systems: Foundations and Applications*. Ed. by M. Galster I. Mistrik and B. Maxim. `https://www.cse.chalmers.se/~bergert/paper/2018-sevis-manual_fl.pdf` [Accessed: July 21, 2024]. Taylor & Francis Group, LLC/CRC Press, 2018.

[20] J. Krüger, M. Pinnecke, A. Kenner, C. Kruczek, F. Benduhn, T. Leich, and G. Saake. “Composing Annotations Without Regret”. In: *Software: Practice and Experience* (2017).

[21] J. Martinez, T. Ziadi, T. F. Bissyande, J. Klein, and Y. L. Traon. “Bottom-up technologies for reuse: Automated extractive adoption of software product lines”. In: *ICSE-C*. 2017.

[22] J. Martinson, H. Jansson, M. Mukelabai, T. Berger, A. Bergel, and T. Ho-Quang. “HAnS: IDE-Based Editing Support for Embedded Feature Annotations”. In: *25th ACM International Systems and Software Product Line Conference - Volume B (SPLC ’21)*. 2021.

[23] A. von Mayrhauser, A. M. Vans, and A. E. Howe. “Program Understanding Behavior During Enhancement of Large-Scale Software”. In: *Journal of Software Maintenance: Research and Practice* 9.5 (1997), pp. 299–327.

[24] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia. “On the Equivalence of Information Retrieval Methods for Automated Traceability Link Recovery”. In: *International Conference on Program Comprehension (ICPC)*. IEEE, 2010, pp. 68–71. DOI: 10.1109/ICPC.2010.20.

[25] L. Passos, K. Czarnecki, S. Apel, A. Wąsowski, C. Kästner, and J. Guo. “Feature-oriented software evolution”. In: *Proceedings of the 7th International Workshop on Variability Modelling of Software-Intensive Systems*. 2013, pp. 1–8.

[26] T. Pfofe. *Automating the Synchronization of Software Variants*. https://wwwiti.cs.uni-magdeburg.de/iti_db/publikationen/ps/auto/thesisPfofe16.pdf. Master’s Thesis. [Accessed: July 20, 2024]. Germany, 2016.

[27] C. Prehofer. “Feature-oriented programming: a fresh look at objects”. In: *Proc. Europ. Conf. Object-Oriented Programming, ECOOP*. Berlin, Heidelberg, New York, London: Springer, 1997, pp. 419–443.

[28] M. Revelle, T. Broadbent, and D. Coppit. “Understanding Concerns in Software: Insights Gained from Two Case Studies”. In: *International Workshop on Program Comprehension (IWPC)*. IEEE, 2005, pp. 23–32. DOI: 10.1109/WPC.2005.43.

[29] P. Royston. “Approximating the Shapiro-Wilk W-test for non-normality”. In: *Statistics and Computing* 2 (1992), pp. 117–119.

[30] J. Rubin, K. Czarnecki, and M. Chechik. “Cloned product variants: from ad-hoc to managed software product lines”. In: *International Journal on Software Tools for Technology Transfer* 17.5 (2015), pp. 627–646.

[31] J. Rubin, K. Czarnecki, and M. Chechik. “Managing cloned variants: A framework and experience”. In: *SPLC*. 2013.

[32] Julia Rubin and Marsha Chechik. “A Survey of Feature Location Techniques”. In: *Domain Engineering*. Springer, 2013, pp. 29–58.

[33] Love Rymo and Fadi Abunaj. *Benefits and Costs of Enabling Variability and Traceability in Source Code via Feature Annotations*. https://odr.chalmers.se/items/95e309fe-695f-41da-b421-8d5cbb5f8ab0/full. Bachelor’s Thesis. [Accessed: May 18, 2024]. 2023.

[34] T. Schwarz. *Design and assessment of an engine for embedded feature annotations*. https://gupea.ub.gu.se/bitstream/handle/2077/67948/gupea_2077_67948_1.pdf?sequence=1&isAllowed=y. Master’s Thesis. [Accessed: July 20, 2024]. Sweden, 2021.

[35] 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*. New York, NY, USA: ACM, 2020.

[36] D. Strüber, M. Mukelabai, J. Krüger, S. Fischer, L. Linsbauer, J. Martinez, and T. Berger. “Facing the truth: benchmarking the techniques for the evolution of variant-rich systems”. In: *SPLC*. 2019.

[37] *The IntelliJ Platform*. `https://plugins.jetbrains.com/docs/intellij/intellij-platform.html`.

[38] T. Thüm, T. Kästner, C. Benduhn, F. Meinicke, J. Saake, and T. Leich. “FeatureIDE: An extensible framework for feature-oriented software development”. In: *Science of Computer Programming* 79 (2014), pp. 70–85.

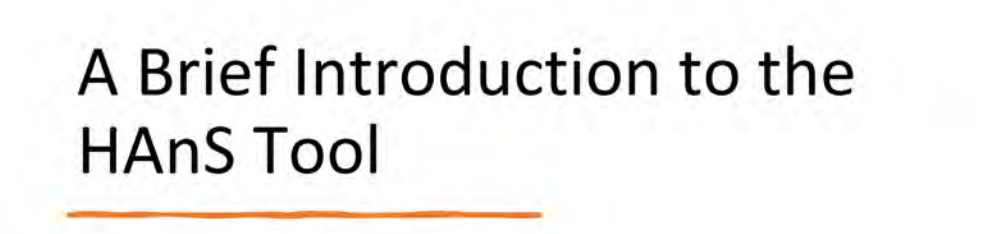
[39] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. “How Developers Perform Feature Location Tasks: A Human-Centric and Process-Oriented Exploratory Study”. In: *Journal of Software: Evolution and Process* 25.11 (2013), pp. 1193–1224.

[40] N. Wilde, M. Buckellew, H. Page, V. Rajlich, and L. Treva Pounds. “A Comparison of Methods for Locating Features in Legacy Software”. In: *Journal of Systems and Software* 65.2 (2003), pp. 105–114. DOI: `10.1016/S0164-1212(02)00052-3`.

[41] R. F. Woolson. “Wilcoxon signed-rank test”. In: *Encyclopedia of Biostatistics* 8 (2005). [Online]. Available: `https://doi.org/10.1002/0470011815.b2a15177`.

A Experiment Materials

A.1 HAnS Tutorial Introduction



HAnS Overview

- HAnS Tool (IDE-Based Editing Support for Embedded Feature Annotations).
- Java-based plugin for recording feature traceability in code assets.
- Recording feature traceability explicitly eliminates the need for feature location.

What is a Feature?

- In software engineering, a feature refers to a specific quality or observable behaviour of a software system that users can experience. It is any functionality meaningful to an end-user (e.g., feature ``Calling'' for mobile phone users).
- Features are used as a means of describing the functionality and characteristics of the software.
- Developers often need to locate features in order to maintain them, enhance them, and even reuse them.
- Without explicit documentation, feature location can take significant time and efforts, especially in very large software systems.

HAnS Tool (IDE-Based Editing Support for Embedded Feature Annotations)

- Allows explicitly documenting features inside software assets.
- Simplifies the process of documenting and eliminates the need for locating features in software assets.
- Allows developers to write embedded feature annotations directly within their code assets, such as files, folders, code fragments, and individual lines of code.

HAnS Tool (IDE-Based Editing Support for Embedded Feature Annotations)

HAnS-text provides support for:

1. Embedded Feature Annotations
2. Mapping code fragments to features
3. Mapping files or directories to features
4. Completion aid when annotating
5. Feature Model View
6. Feature Referencing
7. Renaming features
8. Quick fixes
9. Live templates

Types of embedded annotations

- Feature model
- Feature to folder mapping
- Feature to file mapping
- Feature to code-block mapping
- Feature to line mapping

Embedded feature annotations

Feature Model

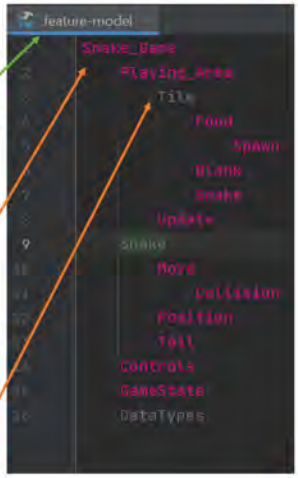
With the help of the HAns plugin, developers can create feature models, add and remove features, rename features, and add mappings between features and the assets that implement them (explained shortly)

Features are specified one per line, where the indentation shows hierarchy, i.e., features one tab later than their previous features are sub-features of those features (e.g., Tile is a sub-feature of Playing_Area)

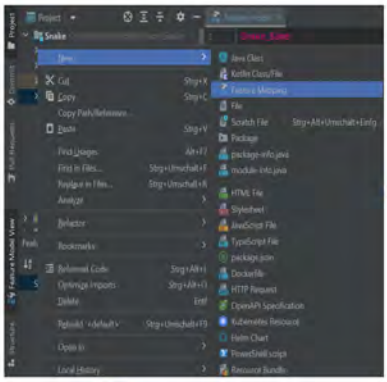
.feature-model

Feature models are used to describe the features, their relationships, and constraints between them.

Abstract features are grey in color (they do not have any mapped assets)



Adding **a feature model**

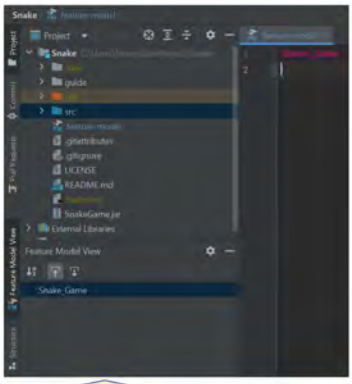


Add a new feature model by right-clicking ``New'' in the Project/Folder and selecting ``Feature Mapping''

OR

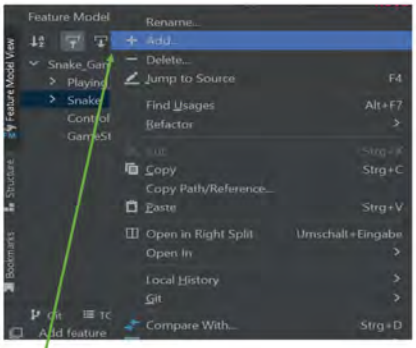
Click ``Create feature-model'' in the Feature Model View

=



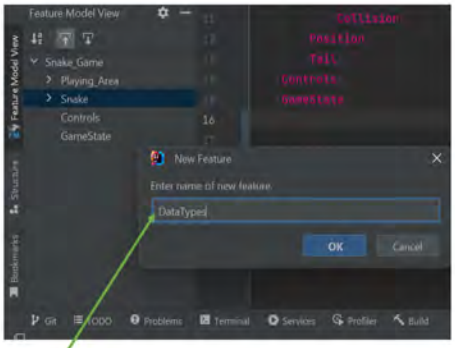
The feature model is created and added under the "src" folder of the Snake project

Adding **a feature: Method 1**



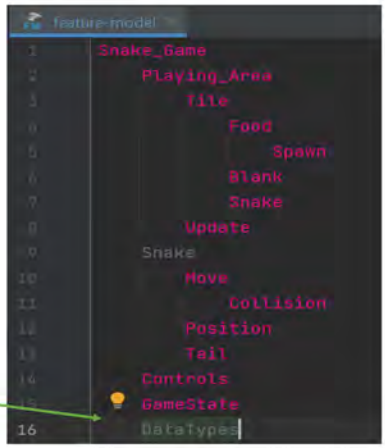
By right-clicking New the "Feature Model View" and clicking "+ Add "

=



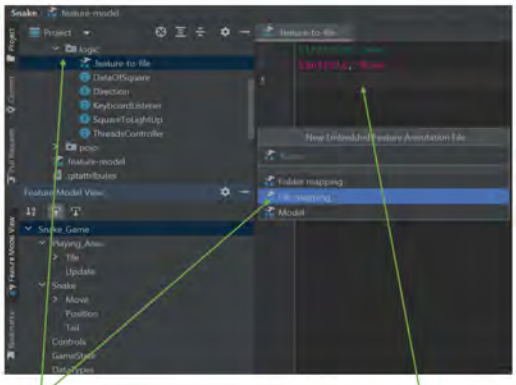
After Clicking "+ Add " you will get a new window then you can write the name of the Feature as below " DataTypes"

Adding **a feature: Method 2**



By clicking the ".feature-model" and writing the New feature under the ".feature-model" as below : , " DataTypes"

Adding a **.feature-to-file** mapping



By right-clicking New e.g. under the "logic" and selcting the file mapping

Mapping a .feature-to-file in this case the Direction.java

Adding a **.feature-to-file** mapping

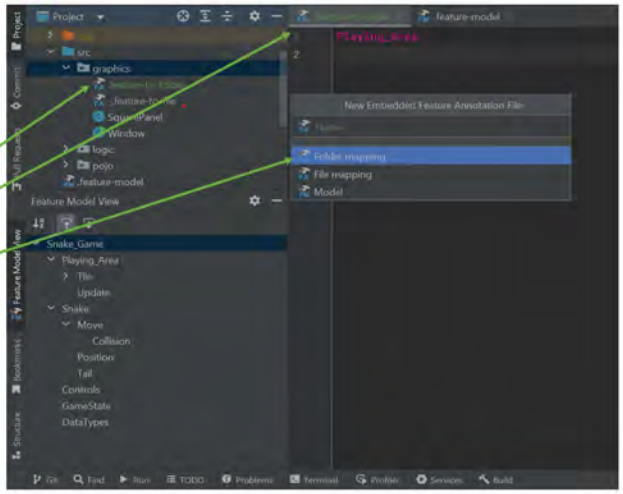
Feature-to-file mapping is the mapping of entire files to one or many features. This allows specific features to be associated with the entire files. The mapping of features to files is stored in a dedicated file identified by the **.feature-to-file** extension.

Feature-to-file mappings imply that the entire file is a complete or partial implementation of a feature. File names and features they map to are written in alternate lines (e.g., SquarePanel.java contains the implementation of feature ``Tile'')

Should look like the following

Adding a **.feature-to-folder** mapping

By right-clicking New under the "graphics" and selecting the Folder mapping



Adding a **.feature-to-folder** mapping

The mapping of features to folder is stored in a dedicated file identified by the **.feature-to-folder** extension.

Feature-to-folder mapping is the mapping of entire folders and their contents to one or more features. This allows specific features to be associated with the entire folder, including the containing files and sub-folders.

Should look like the following

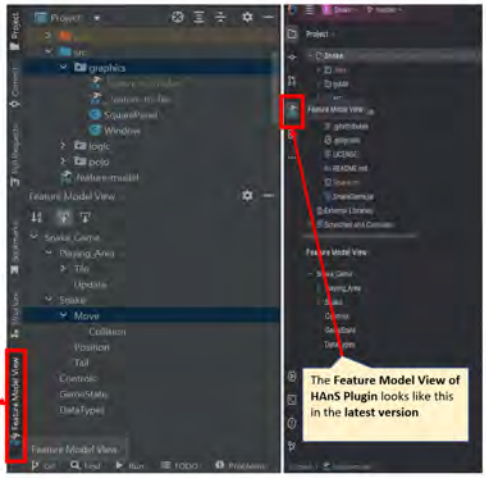
Adding **a feature to code mapping**

A feature to code mapping is used to map specfic blocks of the code to one or more features. This part of code should be encapsulated with **// &begin** and **// &end annotations.** In our example as we see the feature **[Food]** is mapped to the code block by **begin-end** annotations.

Line annotations are used to map specfic lines of code to one or more features. The code lines to be mapped should be ended by **//&line[ ]** annotations. In our example, we see the feature **[Spawn]** is mapped to the Line 35 using the **//&line[ ]** annotation.

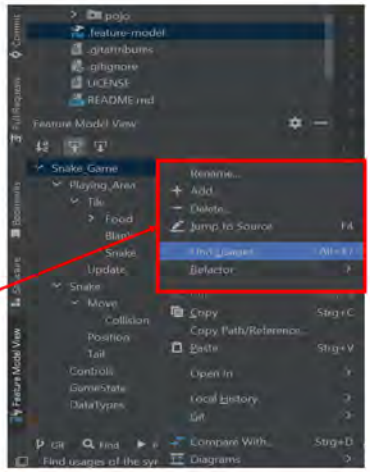
Feature Model View

- The **Feature Model View of HAnS Plugin** offers a special perspective on the software system.
- It displays a visual representation of the feature model, showing the relationships, hierarchies, and dependencies among different features.



Navigating Feature Models

- In the **Feature Model View window** at the bottom left, as you can see, by right-clicking on any feature, you can view a list of functionality.
- **HAnS** provides powerful support for **adding**, **deleting**, and **renaming** through (**refactoring**) features.
- Developers can also use the "**Find Usages**" button to see exactly where these features have been annotated in the source code.



A.2 Installation Guidelines for the HAnS Plugin and Logger Tool.

Installation Guidelines for the HAnS Tool:

This section offers comprehensive installation instructions for the HAnS (Helping Annotate Software) tool. By following these guidelines, you will be able to configure the HAnS tool within your Integrated Development Environment (IDE).

Before you start installing HAnS, make sure your system meets the following requirements:

- You have a Windows, IOS, or Linux operating system.
- You have one of these IDEs (IntelliJ IDEA, PyCharm and Android Studio) installed on your system to proceed with the installation.

Follow the steps below to download and install the HAnS tool:

Downloading HAnS:

To initiate the download process, first visit the official JetBrains Marketplace for

Here are the steps to download HAnS:

- Using built-in plugin system in your IDE, navigate to Settings/Preferences > Plugins > Marketplace.
- Perform a search for "HAnS", and proceed to install the plugin.

If you encounter difficulties finding it through the built-in system, you also have the option for manual installation:

- Download the latest release from this link: (**HAnS Latest Release)**
- Install the plugin manually by accessing Settings/Preferences > Plugins > ⚙> Install plugin from disk...

To ensure a successful installation of the HAnS tool, you need to follow a few steps:

- Launch the IDE. Create a new software project or open an existing one.
- You should then be able to access the features of the HAnS tool within your IDE, including code completion, syntax highlighting and feature annotation support.

Troubleshooting:

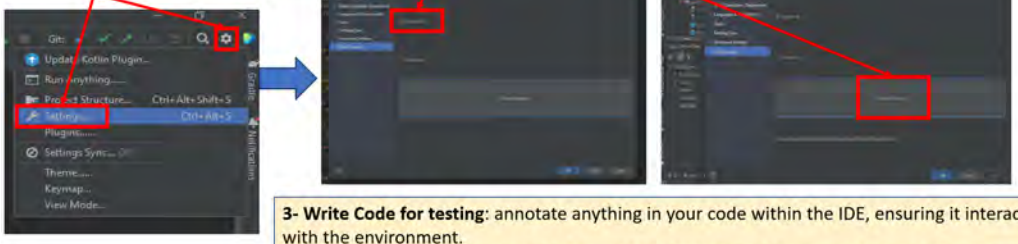
If you encounter any problems while installing or configuring the HAnS tool, you can refer to the official documentation available on the (**HAnS GitHub & HAnS-Bitbucket** or contact the developers for assistance. In this chapter we have provided comprehensive installation guidelines for the HAnS tool. By following these steps, you will be able to integrate HAnS with your chosen integrated development environment to streamline feature annotation and management during your software development tasks.

Guidelines for HAnS Logger(annotationLogger)

Configuration

1- Access Settings: In the sandbox, click on the **Settings/Preferences** icon at the top right corner.

2- Local Logging: Choose in the HAnS Logger "**Log Locally**" and select a directory.



3- Write Code for testing: annotate anything in your code within the IDE, ensuring it interacts with the environment.
4- Logging Process: Execute your code, and the Annotation Logger will capture relevant data according to your settings.
5- Session End: When you close the IDE, logging stops automatically.
6- Access Data: Find the logged JSON file "**annotationLogger**" in the chosen local directory.

A.3 DARTPlus Subject System Features

Search: All users can browse through the rentable items. Search allows users to find a particular item given an ID. Users can also find items based on their titles, genre, and year of release. Lastly, users can filter items by item type (video game, song). Location: Customer.java, Employee.java, Manager.java

IO: Short for "Input Output", is an abstract feature comprising sub-features providing user interaction. Comprises all the menus for different users of the system. Presents a list of all options of activities a user can perform, as well as the option to exit in case the user has completed the interaction. Also verifies that users only enter valid options and prompts users to enter a valid option if they provide an invalid value. Responsible for delegating the responsibilities to the right class based on the provided input. Also responsible for handling user input. User input can be of different data types (number, string, character), and used for different purposes (entering password, selecting from the menu options). Checks if the input follows the required format and conforms to other criterion. Re-asks the user for input in case a wrong value is added, or otherwise throws an exception. Location: IO

Data: Deals with data management in the system. Managers use the feature to manage the data about employees (registering and removing employees). Employees use it to manage customers (registering and removing them), and to manage rentable items (adding and removing). Location: Employee.java, Manager.java

Membership: Allows customers to request upgrades in their memberships (silver, gold, platinum). Customers receive more benefits with each increasing membership level. The feature also stores all pending membership requests and allows employees to accept or refuse them. Location: Employee.java, Customer.java

View: Abstract feature allowing users to view different things in the system.

ViewStats: Intended for Managers and Employees, the feature calculates and shows different types of stats related to the system. A few examples are average ratings of a rentable item (video game or song), renting frequency of an item, and most profitable item from the catalogue to display to the manager and employees. The feature is also used by employees to view all registered customers, and for managers to view all registered employees. Location: Manager.java, Employee.java

ViewCatalogue: Intended for all users. Shows all registered video games and songs to customers, managers, and employees. The views can be generated after filtering as well (e.g., showing songs released in a specific year). Location: Employee.java, Customer.java

RentItem: Allows customers to rent video games and songs. Responsible for checking if the item is in stock, as well as if the customer has exceeded his/her maximum limit of rentable items. For customers with different priority levels (regular customer, silver member, gold member, platinum member), the maximum limit of items rented at one point varies. Location: Customer.java

Payment: Allows customers to return and pay for the rented items. As a way to reward the loyal members, the store gives credits, which can be used to rent items in the future. Customers of different priority levels (silver, gold, platinum) get different amounts of credits after returning every item (1,2, and 3 respectively). Customers can use the accumulated credits to pay for items, 5 credits each. The feature also allows customers to redeem discounts (10%, 15%, 25%) depending on their priority level (silver, gold, platinum).Location: Customer.java

Feedback: Abstract feature allows customers to provide critique on their rented items.

Ratings: Customers can rate games and songs they rented. Ratings range between 1 and 5, the higher the better. Items can only be reviewed by customers upon returning them. Location: Customer.java

Review: Allows customers to provide textual feedback on their rented items. Reviewing is optional. Reviews can only be provided by customers upon returning a rented item. Location: Customer.java

ExceptionHandling: Adds exceptions in all the methods dealing with user input, and throws exceptions against invalid inputs, for example, NameEmptyException (customer name not provided during registration) and NegativeSalaryException (salary of less than 0 added for an employee by the manager). Location: Utilities

Messaging: Allows customers to send messages to each other. Sets an empty inbox against each customer, which is filled by the messages sent by other customers. Customers can read the messages, which changes their status from unread to read. They can also respond to messages and remove messages from their inbox. Location: Message.java, Controller.java, Customer.java, CustomerSilver.java, CustomerGold.java, CustomerPlatinum.java, Employee.java

Security: Allows employees to add passwords for customers when registering them. Also responsible for checking if the entered passwords at the time of logging in are correct but enhanced to automatically generate unique passwords for each new registered customer. Location: Tools.java

DataLogging: Abstract feature allowing managers to read data from files and write data back to files.

ImportData: Managers can import data into the system. Specifically, they can add employees, games, songs, and customers from text files into the system. Location: Manager.java, Controller.java

ExportData: Managers can export data to files. Specifically, they can write the data of the registered employees, games, songs, and customers in text files. Location: Manager.java, Controller.java

A.4 Scenarios - Tasks

Complete the Tasks Make sure you take notes as you code while completing the tasks. These notes will help you to answer the questionnaire once you have completed all the tasks.

Warm-up scenario:

1. Add a new `Feature Mapping` named `.feature-to-folder` file within the `Items` directory.
   - Ensure that the feature *`Data`* is defined in the Feature Model using the `Feature Model View Window` found at the bottom left.
   - Map the feature *`Data`* into the new `.feature-to-folder` file to create its mapping.

   By completing these steps, you've successfully mapped the feature *`Data`* to the `Items` directory.
2. Add a new `Feature Mapping` named `.feature-to-file file` within the `IO` directory.
   - Ensure that the feature *`Membership`* is defined in the Feature Model using the `Feature Model View Window` found at the bottom left.
   - Map the feature *`Membership`* to the file `Controller.java` into the new `.feature-to-file` file to create its mapping.

   By completing these steps, you've successfully mapped the feature *`Membership`* to the file within the `IO` package.
3. Navigate all the usages of the feature *`Search`* using the Feature Model.
   - To ensure exactly where the features have been annotated in the source code. Use the `Feature Model View Window` at the bottom left, by right-clicking on the feature *`Search`* and selecting the `Find Usages`.

   By completing these steps, you've successfully found the feature *`Search`* used in all places 6 `results`.

4. Adding a feature-to-code mapping.

   - The feature *Review* is implemented in the `Rentable()` class. Your task is to identify the corresponding code fragment and annotate it with the feature *Review*.

     Hint: *You will need a begin and end annotation for the feature annotation.*

Scenario 1: Task: Importing Data from Text Files

Task: Implementing ImportData Feature

Objective:

Your task is to add a new feature called `DataLogging` to the DARTPlus system, specifically implementing the `ImportData` functionality. This feature will enable the system to read data from the text file `dartData.txt` containing information about employees, songs, games, and customers, and register them accordingly. Please note that employees and customers have dedicated array lists (employees and customers), whereas rentable items (songs and games) can be added to the same array list (i.e., items).

Integration Steps:

1. Feature Addition:
   - Open the feature model of the DARTPlus system.
   - Add a new feature `DataLogging` to the `feature model`.
   - Create two sub-features under `DataLogging` named `ImportData` and `ExportData`.
2. Code Implementation and Annotation:
   - Navigate to the class `Manager` in the directory Persons in the `DARTPlus` source code.
   - Implement a method named `readFile ()` within the `Manager` class.
   - Design the method to read data from a txt file (called `dartData.txt`) and parse each line to extract relevant information about employees, songs, games, and customers.

     Note: The `dartData.txt` you can find in GitHub repository.
   - Register the Employees in array list **employees of type employee**, Songs and Games in **Items of type Rentable**, and Customers in the **customers of type Customer**.

     Help:
     For file reading, you can use BufferedReader:

```java
public void readFile(ArrayList<Employee> employees,
↪  ArrayList<Customer> customers, ArrayList<Rentable> items,
↪  Scanner input) {
    try {
        File dartData = new File("dartData.txt");
        FileReader fr = new FileReader(dartData);
        BufferedReader br = new BufferedReader(fr);
        String line;

        while((line = br.readLine()) != null) {
            String[] dartInfo = line.split(";"); // parse data
            //register employees, songs, and games based on the
            ↪  first word of the line (employee, game, song)

            //For registering, use methods this.registerEmployee(),
            ↪  items.add(new Game or Song), and customers.add()
        }
    } catch (Exception var12) {
        System.out.println("File does not exist");
    }
}
```

- You are supposed to **annotate** the **implemented code** with **embedded feature annotations**.

3. Menu Option Addition:
   - Navigate the Controller class.
   - Add a menu option for data importing in the manager's menu options within the Controller Class. For that, go to the `managerLoggedIn()` method in the Controller class, and add a print statement adding a new option `System.out.println("b. Import data from text file");`
   - You are supposed to **annotate** the **implemented code** with **embedded feature annotations**.
   - Lastly, put a new case in the body of the `managerLoggedIn()` method that calls the `readFile method()`.
   - You are supposed to **annotate** the **implemented code** with **embedded feature annotations**.

Expected Outcome: By completing this task, you will enable the DARTPlus system to import data from the text file, enhancing its functionality and data management capabilities. Remember to annotate the code appropriately to maintain consistency between the feature model and codebase.

Scenario 2: Task: Allowing Customers to also Rent Movies

Objective:

Clone the `Song` class to create a `Movie` class in the DARTPlus system. Add a `feature-to-file mapping` for `RentItem` to `Movie.java` and implement the functionality of renting movies. Details written below.

Tasks:

1. Clone a `Song` Class:
   - Locate the Song `class` in the DARTPlus codebase.
   - Clone the `Song` class to create a new class called `Movie` (in the same folder, i.e., `Items`
   - Within the Movie class, replace the `artist` with the `director`
2. Map the rentable items `Games` and `Song`, including the cloned file **RentItem**
3. Implement Movie Renting Functionality:
   - Add a menu option for renting Item (movie) in the `customerLoggedIn` options within the `Controller` Class.
   - Ensure that the implementation aligns with the existing rental functionality for `Games` and `Songs`. By finding the usages of the *`RentItem`* feature in the `feature model View Window` and updating the existing feature implementation to accommodate `Movie` rentals.

Expected Outcome:

- Feature-to-file mapping **created** and **updated** to include Movie.java for the RentItem feature.
- Implementation of movie rental functionality within the Movie class, allowing users to rent movies.

A.5 dartData.txt

For dartData text reading, the following data text was provided as a reference:

```
Employee;Dwight Schrute;1989;Abc Street;2000;2500
Song;Waka Waka;Shakira;2;2020
Game;Mario Bros;Platform;3;2018
Customer;Creed Bratton;abc123
Employee;Michael Scott;1990;Def Street;2100;2700
Song;Shape of You;Ed Sheeran;4;2017
Game;The Legend of Zelda;Adventure;5;2017
Customer;Angela Martin;password123
Employee;Pam Beesly;1992;Ghi Street;2200;2800
Song;Despacito;Luis Fonsi;6;2017
Game;Red Dead Redemption 2;Action-Adventure;7;2018
Employee;Jim Halpert;1995;Jkl Street;2300;2900
Customer;Kevin Malone;theoffice
Song;Old Town Road;Lil Nas X;8;2019
Game;The Witcher 3: Wild Hunt;RPG;9;2015
Employee;Stanley Hudson;1998;Mno Street;2400;3000
Song;Uptown Funk;Mark Ronson;10;2014
Customer;Kelly Kapoor;qwerty
Game;Overwatch;First-Person Shooter;11;2016
```

B Times for validating the accuracy of the Logger tool.

To validate the accuracy of the Logger tool, it was necessary to extract the timestamps from each participant. These were then compared with the recorded videos in order to present the initial results.

B.1 Participant 1:

$$\textit{Percentage of total spent time in Logger tool} = \frac{46161}{1487297} \times 100 = 3.1\%$$

Type	Logger Time in (ms)	ScreenCast Time in (ms)
.feature-model	5101	7000
.feature-to-file	1460	1000
.feature-to-file	6520	6000
.feature-to-file	13168	13000
.feature-to-folder	-	1500
&block	8051	8000
Deleted &end	1260	1500
&block	8090	6000
Deleted &end	1361	1500
&line	3771	4000
Total Annotation time	46161	46500

Table B.1: Measured times for each of the annotations from the Logger and the screencast for Participant 1.

Type	Logger Time in (ms)	ScreenCast Time in (ms)	Difference in (ms)
.feature-model	5101	7000	1899
.feature-to-file	21148	20000	1148
.feature-to-folder	-	1500	1500
&block	18762	14000	4762
&line	3771	4000	229
Total Annotation time	46161	46500	339

Table B.2: Measured times for each annotation type from the Logger and the screencast for Participant 1.

B.2 Participant 2:

$$\textit{Percentage of total spent time in Logger tool} = \frac{75387}{1767893} \times 100 = 4.3\%$$

Type	Logger Time in (ms)	ScreenCast Time in (ms)
.feature-model	19656	20000
.feature-to-file	17609	17000
.feature-to-file	5174	5000
.feature-to-file	5222	5000
.feature-to-folder	293	300
.feature-to-folder	601	500
&block	9913	10000
&block	8082	8000
&line	8837	10000
Total Annotation time	75387	75800

Table B.3: Measured times for each of the annotations from the Logger and the screencast for Participant 2.

Type	Logger Time in (ms)	ScreenCast Time in (ms)	Difference in (ms)
.feature-model	19656	20000	344
.feature-to-file	28005	27000	1005
.feature-to-folder	894	800	94
&block	17995	18000	5
&line	8837	10000	1163
Total Annotation time	75387	75800	413

Table B.4: Total of the measured times for each type of annotation from the Logger and the screencast, including the difference time for Participant 2.

B.3 Participant 3:

$$\textit{Percentage of total spent time in Logger tool} = \frac{102708}{3773677} \times 100 = 2.7\%$$

Type	Logger Time in (ms)	ScreenCast Time in (ms)
.feature-model	2179	3000
.feature-model	13307	13000
.feature-model	19368	23000
.feature-model	14182	15000
.feature-to-file	7685	7000
.feature-to-file	14175	13000
.feature-to-file	29599	27000
.feature-to-file	1634	1000
.feature-to-folder	216	300
.feature-to-folder	363	500
Deleted &block	-	10000
Deleted &block	-	1000
Deleted &block	-	8000
Deleted &block	-	1000
Deleted &block	-	3000
Deleted &block	-	2000
Deleted &line	-	2000
Total Annotation time	102708	102800

Table B.5: Measured times for each of the annotations from the Logger and the screencast for Participant 3.

Type	Logger Time in (ms)	ScreenCast Time in (ms)	Difference in (ms)
.feature-model	49036	54000	4964
.feature-to-file	53093	48000	5093
.feature-to-folder	579	800	221
&block	20271	25000	4729
&line	-	14000	14000
Total Annotation time	102708	102800	92

Table B.6: Total of the measured times for each type of annotation from the Logger and the screencast, including the difference time for Participant 3.

B.4 Participant 4:

$$\textit{Percentage of total spent time in Logger tool} = \frac{132678}{1285628} \times 100 = 10.3\%$$

Type	Logger Time in (ms)	ScreenCast Time in (ms)
.feature-model	11234	11000
.feature-to-file	2330	3000
.feature-to-file	745	500
.feature-to-file	3475	4000
.feature-to-file	22967	18000
.feature-to-folder	503	900
&block	8105	9000
Deleted &begin	3789	4000
&block	5271	10000
&block	2203	2000
&block	21652	15000
Deleted &end	1064	1500
&block	13678	12000
&block	14165	-
&block	5568	6000
&line	12671	12500
&line	8111	8500
Total Annotation time	132678	112400

Table B.7: Measured times for each of the annotations from the Logger and the screencast for Participant 4.

Type	Logger Time in (ms)	ScreenCast Time in (ms)	Difference in (ms)
.feature-model	11234	11000	234
.feature-to-file	29517	25500	4017
.feature-to-folder	503	900	397
&block	75375	54000	21375
&line	20782	21000	218
Total Annotation time	132678	112400	20278

Table B.8: Total of the measured times for each type of annotation from the Logger and the screencast, including the difference time for Participant 4.

B.5 Participant 5:

$$\textit{Percentage of total spent time in Logger tool} = \frac{81510}{3812627} \times 100 = 2.1\%$$

Type	Logger Time in (ms)	ScreenCast Time in (ms)
.feature-model	4885	4000
.feature-model	10465	9000
.feature-to-file	5658	6000
.feature-to-file	31236	32000
.feature-to-folder	1526	2000
Deleted &begin	1809	2000
&block	12098	12000
&block	9465	4000
&block	6177	6000
Deleted &end	1237	2000
Total Annotation time	81510	75000

Table B.9: Measured times for each of the annotations from the Logger and the screencast for Participant 5.

Type	Logger Time in (ms)	ScreenCast Time in (ms)	Difference in (ms)
.feature-model	15350	13000	2350
.feature-to-file	36894	38000	1106
.feature-to-folder	1526	2000	474
&block	30786	22000	8786
&line	-	-	-
Total Annotation time	81510	75000	6510

Table B.10: Total of the measured times for each type of annotation from the Logger and the screencast, including the difference time for Participant 5.

B.6 Participant 6:

$$\textit{Percentage of total spent time in Logger tool} = \frac{110945}{3110786} \times 100 = 3.6\%$$

Type	Logger Time in (ms)	ScreenCast Time in (ms)
.feature-model	987	1000
.feature-model	24816	26000
.feature-to-file	9167	8000
.feature-to-file	11955	6000
.feature-to-file	5286	6500
.feature-to-file	8102	10000
.feature-to-file	9338	5000
.feature-to-file	4431	5000
.feature-to-file	4966	6000
.feature-to-file	19941	20000
.feature-to-folder	358	800
&block	-	-
Deleted &begin	772	700
&block	11598	10000
&line	-	4000
Total Annotation time	110945	108300

Table B.11: Measured times for each of the annotations from the Logger and the screencast for Participant 6.

Type	Logger Time in (ms)	ScreenCast Time in (ms)	Difference in (ms)
.feature-model	25803	27000	1197
.feature-to-file	73186	66500	6686
.feature-to-folder	358	800	442
&block	12370	10000	2370
&line	-	4000	4000
Total Annotation time	110945	108300	2645

Table B.12: Total of the measured times for each type of annotation from the Logger and the screencast, including the difference time for Participant 6.

B.7 Participant 7:

$$\textit{Percentage of total spent time in Logger tool} = \frac{63820}{3115269} \times 100 = 2.04\%$$

Type	Logger Time in (ms)	ScreenCast Time in (ms)
.feature-model	16017	16000
.feature-to-file	4267	4500
.feature-to-file	3634	4300
.feature-to-file	3540	4000
.feature-to-file	3831	3500
.feature-to-folder	334	500
.feature-to-folder	21535	22000
&block	10662	8500
Total Annotation time	63820	63300

Table B.13: Measured times for each of the annotations from the Logger and the screencast for Participant 7.

Type	Logger Time in (ms)	ScreenCast Time in (ms)	Difference in (ms)
.feature-model	16017	16000	17
.feature-to-file	15272	16300	928
.feature-to-folder	21869	22500	631
&block	10662	8500	2162
&line	-	-	-
Total Annotation time	63820	63300	520

Table B.14: Total of the measured times for each type of annotation from the Logger and the screencast, including the difference time for Participant 7.

B.8 Participant 8:

$$\textit{Percentage of total spent time in Logger tool} = \frac{58522}{1378505} \times 100 = 4.2\%$$

Type	Logger Time in (ms)	ScreenCast Time in (ms)
.feature-model	8845	11000
.feature-to-file	8333	9500
.feature-to-file	24135	23500
.feature-to-folder	546	700
Deleted &end	2206	2000
&block	16625	16000
&block	-	-
&block	38	300
&line	-	-
&line	-	-
Total Annotation time	58522	61000

Table B.15: Measured times for each of the annotations from the Logger and the screencast for Participant 8.

Type	Logger Time in (ms)	ScreenCast Time in (ms)	Difference in (ms)
.feature-model	8845	11000	2155
.feature-to-file	32468	33000	532
.feature-to-folder	546	700	154
&block	18869	16300	2569
&line	-	-	-
Total Annotation time	58522	61000	2478

Table B.16: Total of the measured times for each type of annotation from the Logger and the screencast, including the difference time for Participant 8.

B.9 Participant 9:

$$\textit{Percentage of total spent time in Logger tool} = \frac{222101}{5787350} \times 100 = 3.8\%$$

Type	Logger Time in (ms)	ScreenCast Time in (ms)
.feature-model	14184	18000
.feature-to-file	8953	8000
.feature-to-file	20772	20000
.feature-to-file	15568	15000
.feature-to-file	6429	7000
.feature-to-file	19680	11000
.feature-to-file	36747	16000
.feature-to-folder	6358	5500
&block	2017	2000
&block	9992	10000
&block	2593	2000
Deleted &end	1233	1000
&block	17168	14000
Deleted &begin	8257	10000
Deleted &end	302	-
&block	28286	15000
Deleted &begin	4581	-
Deleted &begin	15294	-
&block	442	-
&block	10743	12000
Deleted &begin	1018	1000
Deleted &end	424	-
&block	5538	2000
Deleted &end	999	500
&block	16631	1000
&block	-	1000
Total Annotation time	254209	132500

Table B.17: Measured times for each of the annotations from the Logger and the screencast for Participant 9.

Type	Logger Time in (ms)	ScreenCast Time in (ms)	Difference in (ms)
.feature-model	14184	18000	3816
.feature-to-file	108149	50000	58149
.feature-to-folder	6358	5500	858
&block	125518	59000	66518
&line	-	-	-
Total Annotation time	254209	132500	121709

Table B.18: Total of the measured times for each type of annotation from the Logger and the screencast, including the difference time for Participant 9.

B.10 Participant 10:

$$\textit{Percentage of total spent time in Logger tool} = \frac{69375}{3594069} \times 100 = 1.9\%$$

Type	Logger Time in (ms)	ScreenCast Time in (ms)
.feature-model	-	-
.feature-model	-	-
.feature-model	-	-
.feature-to-file	7778	6000
.feature-to-file	23569	23500
.feature-to-folder	3482	1500
.feature-to-folder	2844	3000
&block	10346	9500
&block	10628	12000
&block	10728	11000
&line	-	5000
&line	-	4800
&line	-	4000
Total Annotation time	69375	80300

Table B.19: Measured times for each of the annotations from the Logger and the screencast for Participant 10.

Type	Logger Time in (ms)	ScreenCast Time in (ms)	Difference in (ms)
.feature-model	-	-	0
.feature-to-file	31347	29500	1847
.feature-to-folder	6326	4500	1826
&block	31702	32500	798
&line	-	13800	13800
Total Annotation time	69375	80300	10925

Table B.20: Total of the measured times for each type of annotation from the Logger and the screencast, including the difference time for Participant 10.

B.11 Participant 11:

$$\textit{Percentage of total spent time in Logger tool} = \frac{77796}{1856612} \times 100 = 4.2\%$$

Type	Logger Time in (ms)	ScreenCast Time in (ms)
.feature-model	24069	23000
.feature-to-file	3753	4000
.feature-to-file	18054	17200
.feature-to-folder	2756	2200
&block	7759	10000
&block	5866	9000
&block	7014	7500
Deleted &begin	2248	3000
&block	8525	-
&line	-	5000
Deleted &line	6314	-
Total Annotation time	86358	77900

Table B.21: Measured times for each of the annotations from the Logger and the screencast for Participant 11.

Type	Logger Time in (ms)	ScreenCast Time in (ms)	Difference in (ms)
.feature-model	24069	23000	1069
.feature-to-file	21807	21200	607
.feature-to-folder	2756	2200	556
&block	31412	26500	4912
&line	6314	5000	1314
Total Annotation time	77796	77900	104

Table B.22: Total of the measured times for each type of annotation from the Logger and the screencast, including the difference time for Participant 11.

B.12 Participant 12:

$$\textit{Percentage of total spent time in Logger tool} = \frac{85826}{1706352} \times 100 = 5.02\%$$

Type	Logger Time in (ms)	ScreenCast Time in (ms)
.feature-model	25263	13000
.feature-to-file	4402	5000
.feature-to-file	10919	11250
.feature-to-folder	3709	4000
.feature-to-folder	7419	7000
&block	9337	10000
Deleted &end	2907	1000
&block	3948	4500
&line	5216	5000
&line	6941	7400
&line	4220	4750
&line	2775	4000
Deleted &end	-	1000
&line	1677	2750
Total Annotation time	85826	78650

Table B.23: Measured times for each of the annotations from the Logger and the screencast for Participant 12.

Type	Logger Time in (ms)	ScreenCast Time in (ms)	Difference in (ms)
.feature-model	25263	13000	12263
.feature-to-file	15321	16250	929
.feature-to-folder	11128	11000	128
&block	16192	14500	1692
&line	20829	23900	821
Total Annotation time	85826	78650	7176

Table B.24: Total of the measured times for each type of annotation from the Logger and the screencast, including the difference time for Participant 12.

B.13 Participant 13:

$$\textit{Percentage of total spent time in Logger tool} = \frac{80401}{2427531} \times 100 = 3.3\%$$

Type	Logger Time in (ms)	ScreenCast Time in (ms)
.feature-model	13672	13400
.feature-to-file	3032	3400
.feature-to-file	14933	15300
.feature-to-folder	451	465
Deleted &end	3903	4000
&block	12826	13000
Deleted &end	1786	3000
&block	15944	16600
&block	4548	4700
&line	14995	15500
Total Annotation time	80401	82365

Table B.25: Measured times for each of the annotations from the Logger and the screencast for Participant 13.

Type	Logger Time in (ms)	ScreenCast Time in (ms)	Difference in (ms)
.feature-model	13672	13400	272
.feature-to-file	17965	18700	735
.feature-to-folder	451	465	14
&block	39007	34300	4707
&line	14995	15500	505
Total Annotation time	80401	82365	1964

Table B.26: Total of the measured times for each type of annotation from the Logger and the screencast, including the difference time for Participant 13.

C Effort for annotating including the navigating time.

C.1 Participant 1:

Type	Annotation Time in (ms)	Navigation Time in (ms)	Total Time (ms)
.feature-model	7000	2500	9500
.feature-to-file	1000	4000	5000
.feature-to-file	6000	2000	8000
.feature-to-file	13000	2300	15300
.feature-to-folder	1500	2000	3500
&block	8000	6500	14500
&block	6000	1700	7700
&line	4000	3500	7500
Total Time	46500	24500	71000

Table C.1: Measured times for each annotation in the screencast and navigation time for Participant 1.

Type	Annotation Time in (ms)	Navigation Time in (ms)	Total A+N Time in (ms)
.feature-model	7000	2500	9500
.feature-to-file	20000	8300	28300
.feature-to-folder	1500	2000	3500
&block	14000	8200	22200
&line	4000	3500	7500
Total Time	46500	24500	71000

Table C.2: Total of measured times for each type of annotation performed on the screencast, including navigation time, with total annotation and navigation time counted for Participant 1.

C.2 Participant 2:

Type	Annotation Time in (ms)	Navigation Time in (ms)	Total A+N Time in (ms)
.feature-model	20000	2000	22000
.feature-to-file	17000	1500	18500
.feature-to-file	5000	1600	6600
.feature-to-file	5000	2200	7200
.feature-to-folder	300	500	800
.feature-to-folder	500	1000	1500
&block	10000	21800	31800
&block	8000	4000	12000
&line	10000	7200	17200
Total time	75800	41800	117600

Table C.3: Measured times for each annotation in the screencast and navigation time for Participant 2.

Type	Annotation Time in (ms)	Navigation Time in (ms)	Total A+N Time in (ms)
.feature-model	20000	2000	22000
.feature-to-file	27000	5300	32300
.feature-to-folder	800	1500	2300
&block	18000	25800	43800
&line	10000	7200	17200
Total Time	75800	41800	117600

Table C.4: Total of measured times for each type of annotation performed on the screencast, including navigation time, with total annotation and navigation time counted for Participant 2.

C.3 Participant 3:

Type	Annotation Time in (ms)	Navigation Time in (ms)	Total A+N Time in (ms)
.feature-model	3000	4600	7600
.feature-model	13000	2200	15200
.feature-model	23000	2600	25600
.feature-model	15000	500	15500
.feature-to-file	7000	2000	9000
.feature-to-file	13000	1100	14100
.feature-to-file	27000	1900	28900
.feature-to-file	1000	2500	3500
.feature-to-folder	300	1800	2100
.feature-to-folder	500	1600	2100
Total time	102800	20800	123600

Table C.5: Measured times for each annotation in the screencast and navigation time for Participant 3.

Type	Annotation Time in (ms)	Navigation Time in (ms)	Total A+N Time in (ms)
.feature-model	54000	9900	63900
.feature-to-file	48000	7500	55500
.feature-to-folder	800	3400	4200
&block	-	-	-
&line	-	-	-
Total Time	102800	20800	123600

Table C.6: Total of measured times for each type of annotation performed on the screencast, including navigation time, with total annotation and navigation time counted for Participant 3.

C.4 Participant 4:

Type	Annotation Time in (ms)	Navigation Time in (ms)	Total A+N Time in (ms)
.feature-model	11000	5000	16000
.feature-to-file	3000	1700	4700
.feature-to-file	500	2500	3000
.feature-to-file	4000	1000	5000
.feature-to-file	18000	9000	27000
.feature-to-folder	900	3000	3900
&block	9000	16200	25200
&block	10000	3000	13000
&block	2000	17600	19600
&block	15000	4600	19600
&block	12000	3500	15500
&block	6000	500	6500
&line	12500	7500	20000
&line	8500	1000	9500
Total time	112400	76100	188500

Table C.7: Measured times for each annotation in the screencast and navigation time for Participant 4.

Type	Annotation Time in (ms)	Navigation Time in (ms)	Total A+N Time in (ms)
.feature-model	11000	5000	16000
.feature-to-file	25500	14200	39700
.feature-to-folder	900	3000	3900
&block	54000	45400	99400
&line	21000	8500	29500
Total Time	112400	76100	188500

Table C.8: Total of measured times for each type of annotation performed on the screencast, including navigation time, with total annotation and navigation time counted for Participant 4.

C.5 Participant 5:

Type	Annotation Time in (ms)	Navigation Time in (ms)	Total A+N Time in (ms)
.feature-model	4000	6400	10400
.feature-model	9000	2000	11000
.feature-to-file	6000	3600	9600
.feature-to-file	32000	9300	41300
.feature-to-folder	2000	2600	4600
&block	12000	12500	24500
&block	4000	3500	7500
&block	6000	3700	9700
Total time	75000	43600	118600

Table C.9: Measured times for each annotation in the screencast and navigation time for Participant 5..

Type	Annotation Time in (ms)	Navigation Time in (ms)	Total A+N Time in (ms)
.feature-model	13000	8400	21400
.feature-to-file	38000	12900	50900
.feature-to-folder	2000	2600	4600
&block	22000	19700	41700
&line	-	-	-
Total Time	75000	43600	118600

Table C.10: Total of measured times for each type of annotation performed on the screencast, including navigation time, with total annotation and navigation time counted for Participant 5.

C.6 Participant 6:

Type	Annotation Time in (ms)	Navigation Time in (ms)	Total A+N Time in (ms)
.feature-model	1000	6900	7900
.feature-model	26000	500	26500
.feature-to-file	8000	4250	12250
.feature-to-file	6000	8300	14300
.feature-to-file	6500	12000	18500
.feature-to-file	10000	500	10500
.feature-to-file	5000	1600	6600
.feature-to-file	5000	4700	9700
.feature-to-file	6000	500	6500
.feature-to-file	20000	400	20400
.feature-to-folder	800	4200	5000
&block	10000	21900	31900
&line	4000	7500	11500
Total time	108300	73250	181550

Table C.11: Measured times for each annotation in the screencast and navigation time for Participant 6

Type	Annotation Time in (ms)	Navigation Time in (ms)	Total A+N Time in (ms)
.feature-model	27000	7400	34400
.feature-to-file	66500	32250	98750
.feature-to-folder	800	4200	5000
&block	10000	21900	31900
&line	4000	7500	11500
Total Time	108300	73250	181550

Table C.12: Total of measured times for each type of annotation performed on the screencast, including navigation time, with total annotation and navigation time counted for Participant 6

C.7 Participant 7:

Type	Annotation Time in (ms)	Navigation Time in (ms)	Total A+N Time in (ms)
.feature-model	16000	15500	31500
.feature-to-file	4500	2300	6800
.feature-to-file	4300	300	4600
.feature-to-file	4000	1000	5000
.feature-to-file	3500	5200	8700
.feature-to-folder	500	4000	4500
.feature-to-folder	22000	2300	24300
&block	8500	45000	53500
Total Time	63300	75600	138900

Table C.13: Measured times for each annotation in the screencast and navigation time for Participant 7.

Type	Annotation Time in (ms)	Navigation Time in (ms)	Total A+N Time in (ms)
.feature-model	16000	15500	31500
.feature-to-file	16300	8800	25100
.feature-to-folder	22500	6300	28800
&block	8500	45000	53500
&line	-	-	-
Total Time	63300	75600	138900

Table C.14: Total of measured times for each type of annotation performed on the screencast, including navigation time, with total annotation and navigation time counted for Participant 7.

C.8 Participant 8:

Type	Annotation Time in (ms)	Navigation Time in (ms)	Total A+N Time in (ms)
.feature-model	11000	64700	75700
.feature-to-file	9500	3900	13400
.feature-to-file	23500	21200	44700
.feature-to-folder	700	2000	2700
&block	16000	42500	58500
&block	300	5500	5800
Total Time	61000	139800	200800

Table C.15: Measured times for each annotation in the screencast and navigation time for Participant 8.

Type	Annotation Time in (ms)	Navigation Time in (ms)	Total A+N Time in (ms)
.feature-model	11000	64700	75700
.feature-to-file	33000	25100	58100
.feature-to-folder	700	2000	2700
&block	16300	48000	64300
&line	-	-	-
Total Time	61000	139800	200800

Table C.16: Total of measured times for each type of annotation performed on the screencast, including navigation time, with total annotation and navigation time counted for Participant 8.

C.9 Participant 9:

Type	Annotation Time in (ms)	Navigation Time in (ms)	Total A+N Time in (ms)
.feature-model	18000	2000	20000
.feature-to-file	8000	5300	13300
.feature-to-file	20000	7000	27000
.feature-to-file	15000	2500	17500
.feature-to-file	7000	4300	11300
.feature-to-folder	5500	2800	8300
&block	2000	13400	15400
&block	10000	15700	25700
&block	2000	3500	5500
&block	14000	32100	46100
&block	15000	5300	20300
&block	12000	2400	14400
&block	2000	5200	7200
&block	1000	6300	7300
&block	1000	3000	4000
Total time	132500	110800	243300

Table C.17: Measured times for each annotation in the screencast and navigation time for Participant 9.

Type	Annotation Time in (ms)	Navigation Time in (ms)	Total A+N Time in (ms)
.feature-model	18000	2000	20000
.feature-to-file	50000	17100	67100
.feature-to-folder	5500	2800	8300
&block	59000	86900	145900
&line	-	-	-
Total Time	132500	108800	241300

Table C.18: Total of measured times for each type of annotation performed on the screencast, including navigation time, with total annotation and navigation time counted for Participant 9.

C.10 Participant 10:

Type	Annotation Time in (ms)	Navigation Time in (ms)	Total A+N Time in (ms)
.feature-to-file	6000	3500	9500
.feature-to-file	23500	5500	29000
.feature-to-folder	1500	5800	7300
.feature-to-folder	3000	2000	5000
&block	9500	28000	37500
&block	12000	5500	17500
&block	11000	2700	13700
&line	-	-	-
&line	5000	7000	12000
&line	4800	5900	10700
&line	4000	1400	5400
Total time	80300	67300	147600

Table C.19: Measured times for each annotation in the screencast and navigation time for Participant 10.

Type	Annotation Time in (ms)	Navigation Time in (ms)	Total A+N Time in (ms)
.feature-model	-	-	-
.feature-to-file	29500	9000	38500
.feature-to-folder	4500	7800	12300
&block	32500	36200	68700
&line	13800	14300	28100
Total Time	80300	67300	147600

Table C.20: Total of measured times for each type of annotation performed on the screencast, including navigation time, with total annotation and navigation time counted for Participant 10.

C.11 Participant 11:

Type	Annotation Time in (ms)	Navigation Time in (ms)	Total A+N Time in (ms)
.feature-model	23000	6000	29000
.feature-to-file	4000	7600	11600
.feature-to-file	17200	4900	22100
.feature-to-folder	2200	6000	8200
&block	10000	23600	33600
&block	9000	1700	10700
&block	7500	4700	12200
&line	5000	1300	6300
Total Time	77900	55800	133700

Table C.21: Measured times for each annotation in the screencast and navigation time for Participant 11.

Type	Annotation Time in (ms)	Navigation Time in (ms)	Total A+N Time in (ms)
.feature-model	23000	6000	29000
.feature-to-file	21200	12500	33700
.feature-to-folder	2200	6000	8200
&block	26500	30000	56500
&line	5000	1300	6300
Total Time	77900	55800	133700

Table C.22: Total of measured times for each type of annotation performed on the screencast, including navigation time, with total annotation and navigation time counted for Participant 11.

C.12 Participant 12:

Type	Annotation Time in (ms)	Navigation Time in (ms)	Total A+N Time in (ms)
.feature-model	13000	5500	18500
.feature-to-file	5000	2000	7000
.feature-to-file	11250	9700	20950
.feature-to-folder	4000	3700	7700
.feature-to-folder	7000	1000	8000
&block	10000	34500	44500
&block	4500	7000	11500
&line	5000	2300	7300
&line	7400	7900	15300
&line	4750	1200	5950
&line	4000	1400	5400
&line	2750	1000	3750
Total Time	78650	77200	155850

Table C.23: Measured times for each annotation in the screencast and navigation time for Participant 12

Type	Annotation Time in (ms)	Navigation Time in (ms)	Total A+N Time in (ms)
.feature-model	13000	5500	18500
.feature-to-file	16250	11700	27950
.feature-to-folder	11000	4700	15700
&block	14500	41500	56000
&line	23900	13800	37700
Total Time	78650	77200	155850

Table C.24: Total of measured times for each type of annotation performed on the screencast, including navigation time, with total annotation and navigation time counted for Participant 12.

C.13 Participant 13:

Type	Annotation Time in (ms)	Navigation Time in (ms)	Total A+N Time in (ms)
.feature-model	13400	6000	19400
.feature-to-file	3400	600	4000
.feature-to-file	15300	10000	25300
.feature-to-folder	465	1000	1465
&block	13000	33700	46700
&block	16600	4900	21500
&block	4700	2700	7400
&line	15500	6500	22000
Total Time	82365	65400	147765

Table C.25: Measured times for each annotation in the screencast and navigation time for Participant 13.

Type	Annotation Time in (ms)	Navigation Time in (ms)	Total A+N Time in (ms)
.feature-model	13400	6000	19400
.feature-to-file	18700	10600	29300
.feature-to-folder	465	1000	1465
&block	34300	41300	75600
&line	15500	6500	22000
Total Time	82365	65400	147765

Table C.26: Total of measured times for each type of annotation performed on the screencast, including navigation time, with total annotation and navigation time counted for Participant 13.

D Questionnaire Form

User Study of Using the HAnS Tool for Writing Embedded Feature Annotations

Section 1 - Instructions and Disclaimer
This questionnaire is part of a scientific experiment conducted by researchers at Ruhr University Bochum and Chalmers University of Gothenburg. All the survey data is stored and published in anonymized form.

Participation:
Participation in the experiment is entirely voluntary.

Experiment goal:
The goal of this experiment is to measure the time and effort required to write embedded feature annotations using the HAnS (Helping Annotate Software) tool. HAnS tool is a Java-based plugin that facilitates the annotation process by allowing the user to specify feature mappings to assets of different types such as folders, files and code fragments this includes features such as code completion for feature names and syntax highlighting.

Experiment Structure:
The experiment comprises tow parts. The first part involves getting an introduction of the HAnS tool and familiarizing yourself with the subject system. For this, please view the file "HAnS Tutorial Introduction.pdf" in slideshow mode before starting the tasks. Then, read the handouts titled "Subject Systems' Introduction". The second part is to fill this questionnaire by performing the given tasks and responding to the questions (about those tasks).

Please note that you are also provided with a hard-copy of this questionnaire, but it is only for facilitating you to perform the required tasks without the need to switch between multiple screens repeatedly.
You still need to fill the online questionnaire with task responses once you have finished performing a task.

---> Please read the instruction and the task materials before you begin the questionnaire. You are allowed to consult these materials at any time during the experiment.

--> Feel free to use a word file (.docx) to temporarily write the output of a task elsewhere before putting in the form.

Preparation steps for the experiment are as follows.

1. Download or clone the GitHub repository at https://github.com/hasanainq/HAnS-experiment-.
2. Launch InelliJ IDEA "the minimum version must be 2023.1.1 and the maximum version should be 2023.2.6"
3. Please view the file "HAnS Tutorial Introdction.pdf" in slideshow within the downloaded GitHub repository.
4. Familiarise yourself with the subject system used in the experiment by quickly reading the handouts titled "Subject Systems' Introduction".
5. Open the project "**DartPlus**" from the cloned repository in IntelliJ IDEA.
6. Go to IntelliJ Settings --> select "Plugins" --> Click on > ⚙ >"Install plugin from Disk..." -->Navigate to the folder of the downloaded repository then select and install "**HAnS-0.0.4.zip**".
7. After installing the plugin, apply and restart the IDE.

Alternatively, refer to the "Installation Guidelines for the HAnS plugin" document for detailed instructions.

Informed Consent *

- [ ] I hereby consent to participate in this experiment. I acknowledge that participation is voluntary and I can leave at any time throughout the experiment.
- [ ] I have read the instruction materials
- [ ] I have read the task materials

This is a required question

Next Clear form

Section 2 - Participant Demographics and Technical Background

Please provide some information about yourself and your technical background.

What is your occupation? *

- ◯ Student (BSc or BEng)
- ◯ Student (MSc)
- ◯ Student (PhD)
- ◯ Academic/Scientist
- ◯ Practitioner

How do you rate your programming expertise? *

An expert programmer has exceptional problem-solving skills, extensive knowledge about data structures and algorithms, as well as practical experience with most of the popular programming paradigms (e.g. procedural, object-oriented, and logic programming etc.).

	1	2	3	4	5	
I can't program at all	◯	◯	◯	◯	◯	I'm an expert

How do you rate your knowledge of feature annotations? *

The scale for knowledge of feature annotations ranges from 1 to 5. A rating of 1 indicates a beginner with little to no knowledge, while a rating of 5 indicates an expert with extensive knowledge and experience.

	1	2	3	4	5	
I'm a beginner	○	○	○	○	○	I'm an expert

Back Next Clear form

Section 3 - Subject System : DartPlus(viDeo gAme and song RenTal System)

DARTPlus was created as an extension of DARTBasic, is a software for a video game and music rental system. The company has been previously using the old paper and pen to record the data about games, personnel, and transactions, etc. With rentable items being added to the system every day and an increasing number of customers, the company decided to digitalize their system by creating a simple Java-based console application.

What you'll need:

- A Java-capable IDE, such as IntelliJ, for running the variant's code.
- Ensure that you have installed the HAnS plugin "HAnS-0.0.4.zip".
- Installation Guidelines for the HAnS tool and the HAnS Introduction slides description handout.
- Prepare your preferred screen recording software, such as the **Snapping Tool** in Windows, or any other recording software.

Checkpoint 0:

Check the HAnS Logger configuration in IntelliJ IDEA

Ensure that the logger tool works by choosing the right directory in IntelliJ IDEA : here you can find the HAnS annotationLogger Guidelines.pdf : HAnS annotationLogger Guidelines.pdf

Checkpoint 1:

Start recording the screen before proceeding with the tasks outlined in "Scenarios_Tasks.pdf" to capture the entire process in the IntelliJ IDEA.

Please review the document "**Scenarios_Tasks.pdf**" in the **GitHub repository**.

Once you have completed the tasks outlined in the document, then you can continue with the questionnaire.

Checkpoint 2:
Stop recording, save the video file, and save the JSON file named "annotationLogger" for future reference or documentation.

How difficult was it to perform the task "Adding a feature to a feature model" using the HAnS Tool? *

Remember: You added the feature "DataLogging" with two sub-features "ImportData" and "ExportData".

- ◯ Very Easy
- ◯ Easy
- ◯ Neutral
- ◯ Difficult
- ◯ Very Difficult

How difficult was it to perform the task "Adding a feature to code mapping" using the HAnS Tool? *

Remember: You added a feature annotation for the feature "ImportData" to the implemented code using begin and end annotations.

- ◯ Very Easy
- ◯ Easy
- ◯ Neutral
- ◯ Difficult
- ◯ Very Difficult

How difficult was it to perform the task "Adding a feature to code mapping" using the HAnS Tool? *

Remember: You added a line Annotation at the end of a line a for the feature "ImportData" in the manager's menu options within the class "Controller".

- ◯ Very Easy
- ◯ Easy
- ◯ Neutral
- ◯ Difficult
- ◯ Very Difficult

How difficult was it to perform the task "Adding a feature to file mapping" using HAnS the Tool? *

Remember: You added a new feature "RentItem" to the .feature-to-file file to include Movie.java, Song.java and Game.java.

- ◯ Very Easy
- ◯ Easy
- ◯ Neutral
- ◯ Difficult
- ◯ Very Difficult

How difficult was it to perform the task "Find usages of a feature" using the HAnS Tool? *

Remember: You used the HAnS tool to find the usages of the feature "RentItem" in the feature model.

- ◯ Very Easy
- ◯ Easy
- ◯ Neutral
- ◯ Difficult
- ◯ Very Difficult

How difficult was it to decide when to add annotations? *

Please elaborate:

Your answer

How difficult was it to decide where to add annotations? *

Please elaborate:

Your answer

Back Next Clear form

Embedded feature traceability annotations

What do you think are the benefits of embedded feature traceability annotations? *

Please elaborate:

Your answer

This is a required question

In your opinion, what are the advantages of browsing embedded feature traceability annotations? *

Please elaborate:

Your answer

How intuitive is it to browse embedded feature traceability annotations? *

- ◯ Not intuitive at all
- ◯ Not intuitive
- ◯ Neutral
- ◯ intuitive
- ◯ Very intuitive

How would you rate the usability of using embedded feature traceability annotations? *

- ◯ Very low
- ◯ Low
- ◯ Neutral
- ◯ High
- ◯ Very High

The process of adding annotations enhances my understanding of the codebase. *

- ◯ Strongly Disagree
- ◯ Disagree
- ◯ Neutral
- ◯ Agree
- ◯ Strongly Agree

How well did the experiment match your understanding of the feature traceability annotations? *

- ◯ Does not match at all
- ◯ Does not match
- ◯ Neutral
- ◯ Does match
- ◯ Does match very well

Back Next Clear form

Experience and benefits

What do you think are the benefits of the HAnS tool? *

Please elaborate:

Your answer

Have you ever used the HAnS tool for feature annotation before participating in this study? *

- ◯ Yes
- ◯ No

How difficult is it to learn how to use the HAnS tool? *

- ◯ Very Easy
- ◯ Easy
- ◯ Neutral
- ◯ Difficult
- ◯ Very Difficult

How did you find the user interface of the HAnS tool in terms of usability and user-friendliness? *

- ◯ Very User-Friendly
- ◯ User-Friendly
- ◯ Neutral
- ◯ Not Very User-Friendly
- ◯ Not User-Friendly at All

How useful is HAnS based on the tasks you performed in the experiment? *

- ◯ Very Useful
- ◯ Useful
- ◯ Neutral
- ◯ Not Useful
- ◯ Not Very Useful

While using the HAnS tool for feature annotation, did you encounter any technical issues or bugs?
If yes, please describe:

Your answer

What suggestions do you have for improving the HAnS tool?
Feel free to elaborate:

Your answer

Feel free to share your experiences with the HAnS tool.

Your answer

Back Next Clear form

Thank you for taking the time to complete this questionnaire. Your input is essential to our research. If you have any further comments or insights you'd like to share, please feel free to do so below.

Thank you for your participation!

Please upload the annotationLogger.json file after exiting intelliJ IDEA.
(if you have a share link, otherwise a usb is fine as well)

Your answer

Link to video (if you have a share link, else usb is fine as well)

Your answer

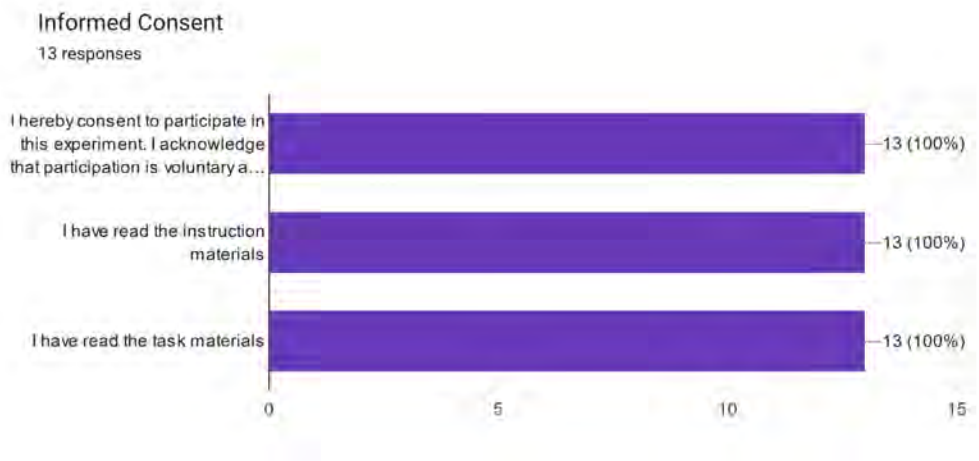
Back Submit Clear form

E Questionnaire Responses

User Study of Using the HAnS Tool for Writing Embedded Feature Annotations

In summary, 13 participants provided responses to the questionnaire.

These responses are provided below, as they could not be included in the results section due to the scope of the questionnaire.



How difficult was it to decide when to add annotations?

13 responses

easy.

I was not always sure if I need to add an annotation

Since I wan't that familiar with code it was pretty hard

When working on a feature, that was easy. However, there was quite some guidance on what steps to perform.

I needed to understand the code first and read the introduction, which took me some time

I just added a mapping for every line I wrote

As someone how has little to no knowledge about annotating code, when was quite difficult but a bit easier than where. I just guided myself with, every time some new functionality is added (e.g., rent Import Data), one should annotate the code.

Since I was not sure about the current annotation style it was overall neutral but without respect to best practices it was rather very easy.

Deciding on what to annotate is always a bit difficult in my opinion, because if it's overdone the code can become cluttered. In this case it was only a few so I think it can be very useful when searching for features.

was straightforward

It was easy but you should be careful to annotate correctly

It was not always clear for me if I should add a feature annotation. I would add them normally for code snippets that have more complex logic to indicate the functionality of the code implemented.

It was difficult to decide if the lines needed a additional annotation

How difficult was it to decide where to add annotations?

13 responses

neutral.
the specifics of how the annotations will be used isn't 100% clear so it's tricky to know where sometimes

This was intuitive

Since I wan't that familiar with code, again, it was kinda hard

That was easy, too. I only annotate code I touched. Also there was no interference with other code features.

Not that difficult

I just added a mapping for every line I wrote

As someone how has little to no knowledge about annotating code, where was quite difficult. I never knew if annotating the feature implementation only was enough. For example, adding the movie rent functionality, everything was annotated with ...[RentItem].... Is a finer grain required or not? However, the tool makes it easy to add annotations for feature implementations.

In this case it was easy if done correctly. But I can imagine more complicated projects where features are intertwined or not that clearly separable.

I think this is more or less the same question as above? I don't understand the difference.

was very simple

but simply look for the right anotat

After I decided when to annotate the decision of where to set the feature annotation was not difficult at all.

I was unsure when to add the full functions or lines to annotate the code

Embedded feature traceability annotations

What do you think are the benefits of embedded feature traceability annotations?

13 responses

makes managing constantly changing databases easier

It helps developers a lot in understanding the project system and features

Improved Understanding of the codebase, easier to find features and their implementations, easier to understand how features are interconnected

Easy to find out why code exists and find code that belongs to each other.

They save time and effort in understanding how the code is related

Improved feature traceability

Easier documentation, especially for later documentation. The annotation is directly in the source code and not somewhere else (word documentation, etc.) not related to a "person's knowledge".

Easy navigation, Easier refactoring, Improved security, implements the principle of 'divide and conquer'

I think it makes finding entire features in a project easier.

better documentation and for the project, new people are able to read the code and understand it better

it was good functionality for quickly searching where Annotate is exactly

Due to this feature traceability, it is now much easier to find the implementation of a feature in several software assets. I find it very useful.

It is fast to find the code needed to do modification to parts of the codebase

In your opinion, what are the advantages of browsing embedded feature traceability annotations?

13 responses

makes organizing new projects and understanding existing ones easier

gives a good overview of the system

better understanding, improved decision making, efficient troubleshooting

Work like comments in the code and have similiar streghst. They also help to find related code.

That makes coding easier when there are many contributors to the same project, helps avoiding errors as well

Finding features easier

Browsing makes it easier to find features in code, instead of manually searching 1000+ lines of code.

more intuitive to me as programmer than navigating like inside of a UML class-diagram. Also faster in the develoment process. Projects are overall easier to read.

Browsing the annotations helps getting an overview over the code and to find parts of a feature. I think it will be especially useful on larger projects, even more so with code written by someone else.

the project is easier to understand

it fits well for other programmers to know how methods in Implimntaion work

Browsing feature annotations keeps me as a developer connected with the features implemented in the project.

It makes the process of writing code faster, by reducing the search time with only adding minimal time consumption to add the annotations

Experience and benefits

What do you think are the benefits of the HAnS tool?

13 responses

makes the coding process easier, especially for new coders

helps developers annotate features in code so it is still clear for other developers working on the same project

feature traceability, better code understanding

Codecompletion and visualisation of the feature model making it easier to add the annotations

Allows developers to explicitly document features and their relationships, and helps locate the code

Improved feature traceability

Easy feature traceability and annotation over a whole project (also bigger scale).

all of the previously mentioned. Security, Readability, Navigation, Intuitive

It makes navigating the code easier.

it makes it easier to organize, manage and add annotations

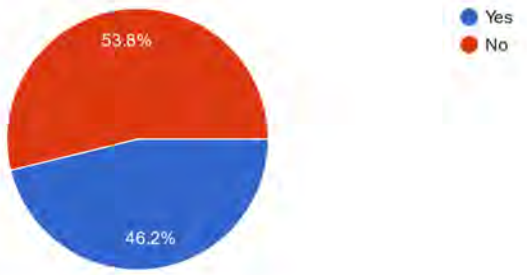
good for future processing and rewriting implication

HAnS helps me as a developer to keep the locations of features implemented in code clear for later usages. Especially for other developers who will start working on the same project. It will be much easier for them to understand the codebase.

It increases the velocity by enabling the developer to find the code faster

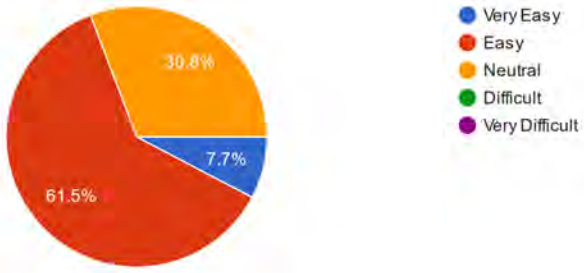
Have you ever used the HAnS tool for feature annotation before participating in this study?

13 responses



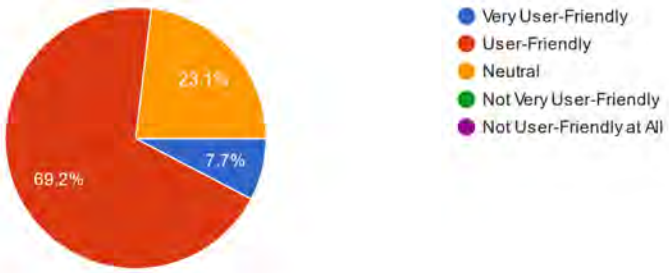
How difficult is it to learn how to use the HAnS tool?

13 responses



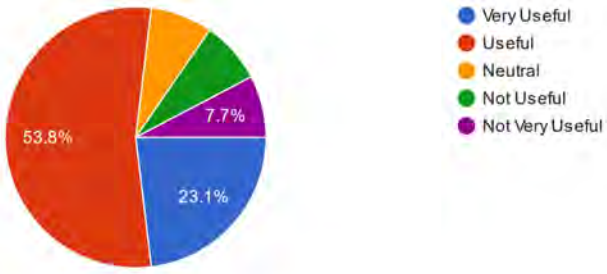
How did you find the user interface of the HAnS tool in terms of usability and user-friendliness?

13 responses



How useful is HAnS based on the tasks you performed in the experiment?

13 responses



While using the HAnS tool for feature annotation, did you encounter any technical issues or bugs?

11 responses

no

the model view was not on by default which made it confusing

-

I had to name files even though that does not seem to be necessary. When adding the annotations manually, the closing is not automatically updated

No

Not with HAnS.

Github Co-Pilot was annyoing but also helping with annotations

No I don't think I did.

No, it runs effortlessly

No I did not

What suggestions do you have for improving the HAnS tool?

6 responses

-

none

Make visualisation better for adding and deleting features with their annotations

syntax highlighting when selecting begin/end annotation

none specific but needs to be better clarified

Automating functionality for tasks such as adding, removing, or moving features annotations.

Feel free to share your experiences with the HAnS tool.

7 responses

generally positive and can see the potential

good experience

writing bachelor thesis on HAnS

It was tricky at first to figure out what help is included in the tool. Especially marking code and then adding an annotation was pretty useful. There are no options to go to the next or prevous occurence of code with the same feature

the gui wasnt too intuitive since i am rarely programming in IntellijIDE and did not know what was new (part of the plugin) in the menus and guis

no comments

I find HAnS really helpful and can imagine myself using it in daily software.

Thank you for taking the time to complete this questionnaire. Your input is essential to our research. If you have any further comments or insights you'd like to share, please feel free to do so below.

Please upload the annotationLogger.json file after exiting intelliJ IDEA.

5 responses

ur stick folder jan ;)

https://www.dropbox.com/scl/fi/dfrb0ekw5xs1fj74agyoe/annotationLogger.json?rlkey=4ox926kobccx5n95uw8vserli&st=pqbwrznl&dl=0

Okay, I'm done and I'll share it

USB

https://mega.nz/file/wdRH1IqS#wYc5aFM-5Rf82Ew1YkZ-00GK0yzWS8w5-s0Ub48o7ng

Link to video (if you have a share link, else usb is fine as well)

4 responses

ur stick folder jan ;)

https://www.dropbox.com/scl/fi/r6fsj94o5pfa4mvwhq2je/Aufzeichnung-2024-04-24-174358.mp4?rlkey=p29478daudti747hf25apejc4&st=sbfrvz7o&dl=0

USB

https://mega.nz/file/ER4mzIxC#wSSYk0HgaCmwM1yOWnCu1IqR6-OTKjSOu9HCraacCRA