CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Benefits and costs of enabling variability and traceability in source code via feature annotations

Degree Project Report in Computer Engineering

Love Rymo
Fadi Abunaj

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2023
www.chalmers.se

Degree project report 2023

Benefits and costs of enabling variability and traceability in source code via feature annotations

Love Rymo
Fadi Abunaj

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Benefits and costs of enabling variability and traceability in source code via feature annotations

Love Rympo, Fadi Abunaj

Supervisors: Thorsten Berger, Department of Computer Science and Engineering, Software Engineering Division
Wardah Mahmood, Cyber Physical Systems
Johan Martinson
Examiner: Lars Svensson, Computer Science and Engineering

Degree project report 2023
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

Benefits and costs of enabling variability and traceability in source code via feature annotations

Love Rymo, Fadi Abunaj
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg

Abstract

Features, which drive the advancement of software systems, are often obscured in modern software projects due to the lack of clear and location-specific documentation. This absence creates significant challenges in software development, notably in time and resource efficiency. This lack of explicit feature documentation often leads to time-consuming efforts in locating features. This study tackles the problem by exploring an innovative approach: embedding feature annotations directly in the source code. It investigates both the benefits and potential costs associated with this practice, filling a gap in the current body of literature. To empirically evaluate this approach, a specialized tool, that we will refer to as an 'annotation logger' was developed. The goal of the tool is to measure the time efficiency of feature annotations. Despite a few minor inaccuracies, the results of this evaluation demonstrated the tool's reliability in gathering meaningful data. The findings suggest that, while introducing some additional overhead, feature annotations could fundamentally enhance the software development process by increasing location-specific clarity and reducing search times.

Keywords: software product lines, traceability, feature location, embedded annotations.

Acknowledgements

We are profoundly grateful to our advisor, Professor Thorsten Berger. Your guidance, expertise, and feedback have been crucial for this research. We also want to express our deepest gratitude to our advisor, Wardah Mahmood. Your assistance and feedback have been indispensable for this project. We are indebted to our supervisor, Johan Martinson, for the way he devoted his time and energy whenever asked. Your technical expertise has been immensely impactful for this project.

Love Rymo, Fadi Abunaj, Gothenburg, June 2023

List of Acronyms

Below is the list of acronyms that have been used throughout this project degree report, listed in alphabetical order:

API	Application Programming Interface
BSON	Binary JavaScript Object Notation
FDD	Feature-driven development
FOSD	Feature-oriented software development
HAnS	Helping Annotate Software
IDE	Integrated development environment
JSON	JavaScript Object Notation
PSI	Program Structure Interface
SDLC	Systems development life cycle
SPLE	Software product line engineering

Nomenclature

Below is the nomenclature of variables that have been used throughout this project.

Variables

T_a	The total annotation time for a development session
T_d	The total development time for a development session
P_a	The percentage of time spent on annotations during a development session

Contents

List of Figures

List of Tables

1
Introduction

Features are an essential part of software development and are often used to drive the advancement of software systems [1]. Features can be used as a common language for people involved in the development of a system, ranging from developers to domain and business experts. They can describe the different functionalities of such systems and create an overview to make it easier to understand complex systems [2]. Even though features can be advantageous in the development, maintenance, and evolution of software systems, many modern software projects have little to no clear documentation of where in the source code these features are located [3]. When there is no explicit documentation of features, it can be time-consuming to find the different features in projects; this is in reality one of the most common activities of a developer [2]. A developer may need to know the location of a feature due to several reasons. These reasons include when needing to clone a feature, when extending or enhancing a feature, or when maintaining features. A cheap and effective way to solve these problems is to embed annotations in the source code where the features are located [2]. This way, it is possible to quickly locate and maintain features, and the annotations remain even if the project gets cloned or forked. This way of embedding annotations enables feature-oriented software development and the visualization of features, which can give a clearer image of how a project is built [4]. The importance of maintaining feature traceability with embedded annotations in source code has been highlighted [3], [2], emphasizing the potential for substantial benefits to software projects, including improved development efficiency and system understandability.

1.1 Context

Despite previous studies on this topic, a significant gap remains: the justification and comparative efficiency of manually adding annotations using tools like HAnS remains understudied. To efficiently evolve and reuse features, the locations of those features need to be known and documented. W. Wenbin et al. [4] argue that a good way to do this is to manually annotate the features in the source code. They investigated this by creating a lightweight annotation tool and stimulating the development of an open-source project. The result showed that the cost to create and maintain annotations was small compared to the saved cost of locating features for cloning them or propagating changes between them.

In another study reported on this subject, H. Jansson and J. Martinson [3] de-

veloped a plugin for the integrated development environment (IDE) IntelliJ. The plugin is known as "Helping Annotate Software" (HAnS) and is a tool to support writing annotations in the code. They then performed user studies to find out the effectiveness of the plugin. H. Jansson and J. Martinson drew the conclusion that HAnS was effective for its purpose but needed to be researched further with more profound studies that include more data to be able to draw a better conclusion about the effectiveness of such a plugin [1][3].

It is unclear if the time and effort developers spend adding these annotations using tools like HAnS is justified in terms of the benefits they provide, such as improved code quality and maintainability. It is also unclear how the time required for manually adding annotations using HAnS compares to the time it takes to write code without annotations, and how developers subjectively perceive this process. In this project report, we want to conduct further research in this field, specifically in HAnS. To be able to draw a more concrete conclusion about the plugin, we believe we need more specific data on the cost of using it. The cost we will research is the time the developer needs to spend annotating features.

1.2 Purpose of the study

The purpose of the project is to extend the research already performed on embedded feature annotations in source code during the development of software. We want to investigate the cost and challenges of writing these annotations and investigate if the benefits outweigh them. We want to extend a logging tool to track the time spent annotating during software development. The tool needs to be able to identify when the annotating starts and stops and measure the time that was spent. We can then compare this time with the time spent on coding, which the tool will also log, to draw conclusions on whether this is an effective way to annotate features in code. The tool needs to take into consideration and be able to measure the time spent when creating and modifying annotations and their assets, as well as the time spent when coding.

To further research if HAnS is an efficient, cheap, and user-friendly plugin to use for embedded annotations, we are also going to perform an evaluation study. The study will be able to provide data on the experience of using HAnS and its challenges. With this information, it is possible to draw conclusions about embedded feature annotations and how to continue with their development in the field.

There are going to be challenges when researching these questions. Some of those challenges that we will face are deciding what the requirements for logging are and how to design an effective logging tool. Another challenge is actually developing a functional logging tool and deciding what to log in the project and how to log it. The final challenge will be collecting the data and analyzing it. We need to choose how the data collection should be done and where to be stored, and how we should analyze it.

1.3 Goals of the study

This project aims to address the following key research questions:

1. How can we design and develop a tool that can measure time spent in an IDE annotating with HAnS and time spent coding?

2. What is the time invested in annotating code, and how does it compare to the time it takes for ordinary development?

3. What are the users' qualitative experiences and challenges with annotating code?

1.4 Limitations and delimitations

This research extends existing work on embedded software annotations by focusing on the HAnS plugin for IntelliJ IDEA IDE. The scope of the study is confined to evaluating the time efficiency of this annotation method. The study is not going to discuss the design or syntax choices of HAnS. The study will instead evaluate if this method of annotation is time-efficient.

The logging tool is going to be limited to working with the HAnS plugin in the IntelliJ IDEA IDE. Logging of other plugins or IDEs will not be researched or discussed.

Studies will be made where developers will annotate using HAnS and the logging tool. Data will be gathered from both the logging tool and the developer's own experience with using HAnS. The developers that will take part in the study will be chosen from a discussion with our supervisors.

2
Theory and technical background

2.1 Feature

Features are often used and needed in many software development approaches. Approaches such as feature-oriented software development (FOSD), feature-driven development (FDD), and software product line engineering (SPLE) involve identifying, declaring, and utilizing features for maintenance and reuse, as well as a unit for comparing different software versions or variants [4].

It can be difficult to clearly define what a feature is. Although definitions seem to vary greatly, a good and clear definition 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." [5]*

This definition encompasses the idea that a feature is a visible behavior that a user can see or use, and also recognizes that a feature is not limited to configurable behavior of the system [3]. Moreover, it also acknowledges that features may be defined at various levels of granularity and used to communicate the product [3]. It is also the definition which HAnS is built upon, and will also be used for the purpose of this study [3].

2.2 Embedded feature annotations

In large or complex software it can be a difficult and error-prone task to locate features, especially if they are dispersed across several files [2]. Embedded feature annotations are a strategy to solve these problems. This is done by embedding the information about a feature in the source code [2]. This way, the location of features will be easy to access and updated during development [2].

2.3 HAnS

HAnS is a plugin for the IntelliJ IDE which was developed by following the notation for embedded feature annotations described by Schwarz et al. [3]. It implements

features and structure which facilitate feature annotations in code. Since the logging tool which is the purpose of this study is an extension of HAnS it is important to have an understanding of the mechanics of HAnS.

2.3.1 .feature-model

The feature model is one of the core components of the annotation system [3]. A feature model is created by creating a file in the root directory of a project with the ending ".feature-model". All features should be listed in this file in a hierarchical structure. The file's syntax is that each file should begin with the project name followed by feature names in new lines with varying indention based on their hierarchical structure [3].

2.3.2 .feature-to-folder

Files that end with ".feature-to-folder" contains feature references [3]. They are used to map entire folders, including its sub-folders and files, to features.

2.3.3 .feature-to-file

Files that end with ".feature-to-file" contains file- and feature-names. This file map in which files features are contained in [3]. The file maps entire files to one or more features.

2.3.4 Code annotations

In difference to the previously mentioned annotation types, instead of using its own file, code annotations are located in the same file as other programming languages in the source code. By using code annotation syntax in a comment within a coding file, an annotation is created [3].

2.4 Software development

The concept of software development originated in the 1970s, marked by the introduction of formal methodologies for creating computer software [6]. These methods often used multiphase design process models named systems development life cycle (SDLC) models. SDLC models in the early stages of the software development industry had a primary focus on producing code rather than using extensively preplanning or post-completion testing [6]. In the 1980's more advanced SDLC models had been developed which used a broader scope and placed a greater emphasis on quality control and other elements of the software development process outside of basic code production. SDLC have since continued to evolve. The term software development therefore refers to more than producing code, it refers to the process of creating software as a whole. In the article "Software development.", J. Lasky writes:

> *"The term software development refers to the task of producing functional computer software through a specialized process that typically involves a sequence of successive phases. These phases often include identification, design, programming, testing, and maintenance. The overall process of creating a particular piece of software is generally carried out according to one of several possible systems development life cycle (SDLC) models. Depending on which model is chosen, the order of developmental stages and the basic methodology of the software development process will vary." [6]*

This definition covers that software development refers to the process of creating software, not only the code-writing itself. For the purpose of this study, we have taken this into consideration and defined developing as active time spent in an integrated development environment (IDE).

2.5 The IntelliJ Platform [7]

The IntelliJ Platform is not a product in itself, but instead a foundation for building integrated development environments (IDEs). It is utilized to power various JetBrains products, such as IntelliJ IDEA which is an IDE primary for development in the programming language Java. It is also open source and is used by third parties such as Google to build the Android Studio IDE. In this study, the primary aim is to build a tool with support for the IntelliJ IDEA, but because it is built upon the IntelliJ Platform it should be modular and also work with other IDEs built on this platform such as PyCharm [8] or CLion [9] without the need for bigger changes.

2.5.1 IntelliJ plugin support

Plugins in IntelliJ may extend the platform in various ways, ranging from basic menu items to comprehensive language support, build systems, and debuggers [7]. Both HAnS and the time tracker utilize this to support embedded feature annotations and track the developer's behavior in the IDE.

2.6 JavaScript Object Notation

JavaScript Object Notation (JSON) is a lightweight data-interchange format, valued for its readability for humans and ease of parsing and generation for machines [10]. JSON is a schemaless format, which means that it does not require a schema. A JSON document or file can be altered or updated individually without affecting other JSON files. JSON also has the benefit of optimizing performance by keeping all the related data in one place [12].

2.7 MongoDB

MongoDB is a NoSQL database that is open-source and well-documented. It is written in the programming language C++. It offers automatic scaling, high performance, and high availability [11]. MongoDB works well with JSON, as the MongoDB team has developed an open data format called BSON, which is short for binary JSON [12]. BSON does not change how you use JSON, but offers performance benefits by making it easier for computers to process and search documents [12].

3
Methods

3.1 Design science research

In order to structure the study systematically, the research process was ordered into a series of progressive steps. Parts 3.1.3 to 3.1.7 were iteratively executed twice, thereby allowing the methodology to be refined. The first iteration focused on stating the initial requirements and developing a functional tool. After the first iteration, there was time for critical reflection and expanding on ideas to further develop the logging tool.

3.1.1 Problem identification and motivation

The first step involved identifying the problem to be researched and explaining its significance. The motivation behind the research was also addressed in this step.

3.1.2 Background research

After identifying the problem, conducting background research was necessary. Relevant studies and articles were researched and reviewed to gain a better understanding of the subject.

3.1.3 Define the logging tool and the evaluation study

Before the development phase, the annotation logging tool and the evaluation study were defined. The logging mechanism was decided to be integrated into the HAnS plugin and measure the time spent annotating features and compare it to the time spent developing. Annotations in code and mapping files were decided to all be counted as time spent annotating. These annotations were to be measured separately but considered together as part of the overall annotation time.

It was decided that the annotation logger would evaluate each keystroke in the IntelliJ IDE to determine if it was an annotation and categorize it accordingly. The categories, further explained in 4.1, included different types of annotations. Whenever an annotation was created, edited, or deleted, the duration of the change was to be logged in a JSON file. The duration was to be measured in milliseconds and logged as a JSON object along with the annotation type.

Before initiating the studies, it was essential to communicate to the participants

what data would be collected. An evaluation study were to later be conducted. Participants were assigned programming tasks on a software project, wherein they were instructed to annotate their code using feature annotations during the completion of these tasks.

3.1.4 Specify requirements

In this step, we outlined the criteria that the project would need to meet to be considered successful. This included defining a minimum viable product based on previous literature in the field and discussions with supervisors. The requirements for the annotation logging tool were that it could calculate the time spent annotating versus developing in a software project. The tool would focus on measuring the time taken to create, delete, or edit annotations in code and HAnS mapping files. As it was decided to be developed as an extension of the HAnS plugin, it was designed to be used when developers had the HAnS plugin downloaded and activated. The tool was to measure annotation time by analyzing the use of HAnS syntax in a software project. It would record the start and end times of annotations and calculated the duration of a certain annotation. Total time spent on each annotation type would also be calculated. By aggregating the results for each annotation, statistics for different annotation types and total annotation time were to be gathered. The results were then analyzed and compared with the time taken for development, providing valuable insights into the effectiveness of embedded software annotations.

3.1.5 Design and develop

With the knowledge gained from the previous steps, the tool was designed and developed. The tool was designed to collect data from the use of the HAnS plugin, enabling the determination of time spent annotating and coding.

3.1.6 Design of evaluation study

In this step, the evaluation study was designed and constructed. The data to be collected from the users was defined, and the study questionnaire was designed accordingly. The questionnaire consisted of both agree-disagree scale questions and open-ended questions, allowing users to express their experiences freely.

3.1.7 Evaluation study

The evaluation study was conducted by distributing instructions, the tool, and the questionnaire to a group of software developers. The study aimed to gather both quantitative and qualitative data from the participants. The annotation logger collected quantitative data as participants performed tasks, providing empirical evidence. Qualitative data was collected through a questionnaire filled out by the participants after completing the tasks.

3.1.8 Evaluation

After developing the annotation logger and conducting the evaluation study, the collected data was evaluated. This step involved analyzing the empirical data gathered from tool usage and the qualitative data obtained from the questionnaires. The analysis of this data enabled drawing conclusions about embedded software annotations, HAnS, and addressing the research questions. To find bugs or inaccuracies, the screen recordings from the evaluation study were reviewed. The recordings were created by using the "record" feature in the videotelephony software program "Zoom". The annotations captured during the screen recordings were manually measured and compared to the logs made by the annotation logger. The evaluation involved measuring the annotations in a manner consistent with the expected behavior of the annotation logger, as detailed in chapter 4. Both authors of this project degree report independently performed the measurements, which were subsequently cross-checked for consistency. The manual measurements were made by using a timer application with the precision of a centisecond. However, it is worth noting that human error may be present in these measurements, as it is challenging to precisely initiate and stop the timer at the exact intended moment.

The evaluation process also included discussions with supervisors to discuss conclusions, assess if the initial requirements were met, identify successful aspects, reflect on areas for improvement, and determine potential subjects for future research in future studies.

3.1.9 Communication and result

The final step of the study involved presenting the results obtained from the previous steps in the form of a project report. This step served of writing a comprehensive summary of the research process, highlighting the implemented methods and the outcomes derived from them.

4
Implementation

The decision was made to implement the time measuring tool, named "annotation logger", as a feature within the pre-existing IntelliJ plugin for IntelliJ "HAnS."

4.1 Categories

In order to create an efficient logging tool, it was essential to determine the specific information it should log and how to effectively categorize different components within a codebase. Consequently, whenever a user performs an action in IntelliJ that pertains to feature annotations, the action is logged. To gain insights into the time allocation for feature annotations, a categorization system was implemented. These categories are derived from the structure and syntax of HAnS, which is further explained in 2.3.

&line When a feature is annotated by using the &line[] syntax.
&block When a feature is annotated by using the &begin[] or &end[] syntax.
.feature-model When annotating inside a .feature-model file.
.feature-to-folder When annotating inside a .feature-to-folder file.
.feature-to-file When annotating inside a .feature-to-file file.

This data can be instrumental in identifying costs and challenges associated with code annotation, and can be used to lay the foundation for future improvements and advancements in embedded feature annotation development. Utilizing this categorized data enables the creation of a plugin that effectively addresses these challenges, streamlining the annotation process for enhanced productivity and efficiency.

4.2 Annotations

For the annotation logger, it is essential to track all activities related to code annotation. This involves logging any additions, deletions, or modifications made to the code using HAnS syntax, all of which are recorded as annotations. Files with extensions such as ".feature-model," ".feature-to-folder", or ".feature-to-file" exclusively consist of feature annotations, meaning any actions performed within these files are considered as part of the annotation process. Each logged annotation includes its corresponding category, as described in 4.1, along with the duration measured in milliseconds, representing the time taken for the annotation.

The annotation logger measures the duration of each annotation by capturing the timestamp of the first and last character modified within the annotation. If there is no subsequent change to the same annotation within a ten-second interval, the logger considers the annotation as completed. To calculate the duration, the logger measures the time difference between the timestamp of the first change and the timestamp of the last change. Every timestamp and duration is recorded with millisecond precision, to ensure accurate and precise measurement.

To accurately track and categorize all changes made in an IntelliJ project, Program Structure Interface (PSI) files were utilized. Each code segment in an IntelliJ project can be represented as a PSI-element, and a PSI file enables a hierarchical representation of these PSI-elements. By listening for changes in the PSI files, the plugin was able to identify if any modifications were applied to a PSI-element containing HAnS syntax, and based on what syntax was used categorize it into the categories mentioned in 4.1.

4.3 Right-click annotating

HAnS offers a feature that lets the user select code and apply the "Surround with Feature Annotation" option by right-clicking on the selection. This action triggers the plugin to automatically surround the chosen code with feature annotation syntax. The user then only needs to input and specify the desired feature name. In the first iteration of the evaluation study, we measured the duration of this annotation as described in 4.2. Through the analysis of the quantitative data obtained from the first iteration, we determined an alternative approach for measuring the time. Instead of tracking the time from the timestamp of the first change of a character belonging to an annotation, the annotation logger began measuring from the moment a user initiates the code selection process that will subsequently be enveloped with feature annotations.

4.4 Development sessions

To compare time spent annotating time with time spent coding, *development sessions* were implemented. A development session begins when opening a software project in IntelliJ and ends when closing IntelliJ. The time for the active session is counted and is logged when a session ends. If a user is idle for 30 seconds, the session gets paused and no time is measured until the session is active again. Being idle is defined as not moving the mouse or pressing any keys while being in IntelliJ. This is implemented in code by using listeners that wait for events, in this case, mouse movement or key presses in IntelliJ, and processes these by updating the last active user time.

4.5 Logging

The logging was made by writing data to a JavaScript Object Notation (JSON) file. The JSON file contains all the logged elements. Each time something is done in the software project, it is logged into the JSON file. The JSON file contains an array of all logged annotations, the total time for each annotation category, the total time for annotating and the total time for the session.


```
[[{"duration (ms)":525,"type":"&line"}]
{
  "annotationType": "&line",
  "totalTime": "525 ms"
}
{
  "annotationType": "&block",
  "totalTime": "0 ms"
}
{
  "annotationType": ".feature-to-file",
  "totalTime": "0 ms"
}
{
  "annotationType": ".feature-to-folder",
  "totalTime": "0 ms"
}
{
  "annotationType": ".feature-model",
  "totalTime": "0 ms"
}
{
  "annotationType": "Total annotation time",
  "totalTime": "525 ms"
}
{
  "annotationType": "Developing time",
  "totalTime": "19735 ms"
}
"2023-05-03 13:35:09.319"
]
```


Figure 4.1: A JSON document with logged feature annotations

4.6 Remote database

To store the development sessions, a remote database was implemented by using MongoDB Atlas. When IntelliJ closes, and therefore a development session is finished, it sends the locally stored JSON file to the remote MongoDB. All the sessions can then easily be accessed in the database. The locally stored JSON file gets deleted so when a new session is started, a new clean JSON file is created.

```
_id: ObjectId('6457e80bae61fb48f5980333')
project_name: "Snake"
log_contents: "[[{"duration (ms)":1757,"type":".feature-to-folder"},{"duration (ms)":…"
```


Figure 4.2: Screenshot of a database entry

4.6.1 Remote database API

Letting the application directly communicate with the database can be vulnerable to security risks. Therefore, an Application Programming Interface (API) was implemented. The application sends requests to the database via the API, which allows the API to validate the requests. Since the API needs to be hosted on a server and the users of the application for the studies in this project could be trusted, the API was not further developed and left as a feature that can be more developed in the future.

4.7 Preference page

As different users may have different preferences while using the plugin and annotation logger, a preference page was implemented. In the preference page, an option was provided that lets the user choose to enable the annotation logger, and therefore save logs, or have it disabled. If the user chose to have the logger enabled, there was also an option on where to store the logs. A user could save the logs locally on the current device, or save them to the remote database.

4.8 Evaluation study

An evaluation study was conducted in two iterations. Each iteration had two people participating. The first iteration focused on confirming the design of the developed annotation logger, and find bugs or other problems with it. The iteration had the intention to validate that the tool worked as expected and that the collected data were correct. After the first iteration, a feature for measuring Right-click annotations more accurately was added, as described in section 4.3. Both iterations had the intention of collecting data that assisted in answering the research questions stated in section 1.2:

- What is the time invested in annotating code, and how does it compare to the time it takes for ordinary development?
- What are the users' qualitative experiences and challenges with annotating code?

In each iteration of the study, two developers were selected to undertake programming tasks, which also involved annotating features. Following the completion of the tasks, the developers were asked to fill out a questionnaire to provide subjective

feedback on their experience. This feedback proved to be valuable in understanding the benefits and challenges associated with embedded feature annotations, as well as identifying areas for further improvement. For this study, we chose to fork the "Snake" repository by Johmara from GitHub, as it had been researched and shown to be useful in prior literature [13] [3]. Some modifications were made to tailor the project and tasks to better align with the objectives of this project, and the adapted version was uploaded to a separate GitHub repository [14]. Following the completion of the second iteration, a manual review of the screen recordings was conducted to assess the accuracy of the annotation logger. The objective was to validate if the annotations and durations logged matched with the video captured in the screen recordings.

4.8.1 Snake game

The evaluation study used a simple implementation of the classic game "Snake". In this game, the primary goal is to maneuver a snake within a playing field and enable it to grow in size by consuming spawned food items. The playing field is a rectangular grid, while the snake itself is represented by a series of interconnected blocks that traverse the grid. To control the snake's movement, the player utilizes arrow keys to change its direction. The game finishes when the snake encounters its own body, resulting in a collision.

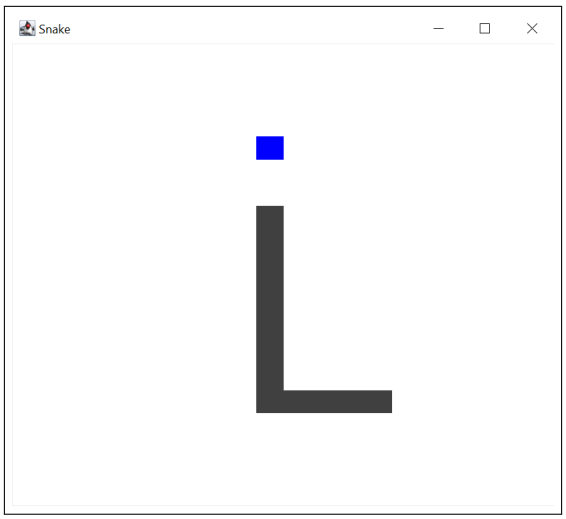


Figure 4.3: Screenshot of the snake game

The snake game was constructed appropriately for feature annotations, and also some annotations were already implemented in order to help give inspiration to the participants.

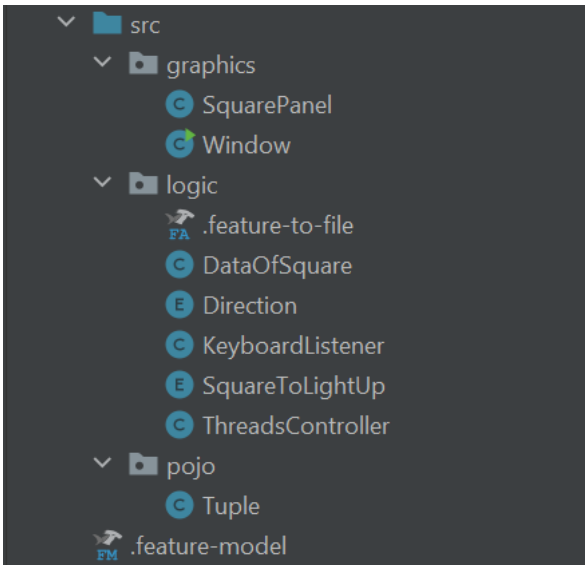


Figure 4.4: The file structure of the snake game

4.8.2 Tasks

The developers were asked to further develop by completing certain tasks in the snake project described in section 4.8.1. There were two warm-up tasks and six tasks with varying difficulty. In order to verify that the different parts of the time tool worked as expected, as well as to get feedback on the plugin, the tasks were designed to include the different annotation categories. The warm-up tasks, Task 2 and Task 5, were related to mapping features with the help of the feature-model, .feature-to-folder, or .feature-to-file files. Task 1 and Task 4 were related to implementing and appropriately annotating new features. Task 3 and Task 6 were related to maintaining features by renaming, refactoring, or editing already existing features. If a participant did not know how to solve a task, the participant was given some guidance. If the participant still could not progress, one was instructed to continue to the next task instead. All the tasks are listed below:

Warm-up task 1 Add a file with the extension ".feature-to-folder" to the graphics package. Verify that the feature "Playing_Area" is defined in the Feature Model via the Feature Model View tab.

Task 1 Implement and annotate a feature (choose a fitting name) that adds a red poison tile that if eaten shrinks the snake by three tiles. If the length of the snake is less than or equal to three, the snake dies. Hint: The poison would follow similar implementation as the feature Food.

Reminder: Make sure you annotate the code you write!

Task 2 Add a file with the extension .feature-to-file to the pojo package. Verify that the feature Tile is defined in the Feature Model. Map the feature Tile to the file Tuple.java.

Task 3 Rename (refactor) the Position feature to the new name Head, including all references to it.

Warm-up task 2 Add a file with the extension ".feature-to-folder" to the pojo package. Verify that the feature DataTypes is defined in the Feature Model via the .feature-model file. Map the feature DataTypes to the new ".feature-to-folder" file by writing it into the file.

Task 4 Implement and annotate a feature that raises the difficulty of the game by increasing its speed by one every time the snake crosses the borders of the playing area. The feature should be defined as a child feature of GameState in the Feature Model. The current difficulty should be displayed as the title of the window. Find and annotate the provided method for updating the title.

Hint 1: The GameState contains the functionality for setting the speed.
Hint 2: The difficulty may never be equal to or exceed the delay variable.
Hint 3: To check if the snake passes the bottom border, check if the head is equal to 0.
Hint 4: Look at the new Head feature.
Reminder: Make sure you annotate the code you write!

Task 5 Verify that the feature Controls are defined in the Feature Model. Map the feature Controls to the file KeyboardListener.java in .feature-to-file in the logic package.

Task 6 Rename (refactor) the Blank feature to the new name Background, including all references to it.

4.8.3 Questionnaire

Following the completion of the tasks, the study participants were engaged in a questionnaire designed to address the research question:

- What are the users' qualitative experiences and challenges with annotating code?

The questionnaire also served as a means to gather insights for improving embedded feature annotations and HAnS in the future. The questionnaire consisted of a series of questions that participants answer on a scale from 1 to 5, allowing them to express their level of agreement or disagreement. Additionally, there is a set of statements for which participants can indicate their degree of agreement on a five-point scale: (i) fully agree, (ii) agree, (iii) neutral, (iv) disagree, and (v) strongly disagree. To prevent any influence on the participants, the questions and statements were neutrally phrased. Furthermore, participants were given the opportunity to provide further elaboration after each question or statement.

5
Results

5.1 Time invested in annotating code

To answer the question of how the time invested in annotating code compares to the time invested in software development, an evaluation study was conducted in two iterations. Each iteration was conducted by letting two developers complete different tasks in a snake game project. While they completed different tasks, they were also instructed to use embedded feature annotations with the help of the IntelliJ plugin HAnS. In the first iteration, some annotations with small durations were noticed. Some of these durations could be attributed to the HAnS feature "right-click annotation". To measure this feature more accurately, a measurement improvement was implemented, which is described in 4.3, for the second iteration.

In the following sections, 5.1.1 - 5.1.2, the data and statistics gathered from the evaluation study will be presented. The annotations that are presented in the results are every annotation logged with a duration longer than zero milliseconds. Annotations with zero duration can arise in two situations. One of them is if a user only changes one character in an annotation. The other way is if you let IntelliJ automatically change annotations, for example in an automatic refactor. This will lead to either a concise duration or zero duration.

In figures 5.1, 5.2, 5.3 and 5.4, the data collected during the evaluation study is presented as box plot graphs. The graphs display the recorded times for each participant across various categories, including the median time for each category. The blue rectangle above each category represents the average duration the user spent on that specific category, while the lines above and below the rectangle indicate the outliers in time spent on that category. Below is an equation that was used to help investigate the time invested in annotating code:

$$\frac{T_a}{T_d} = P_a$$

T_a is the total annotation time for a development session, and T_d is the total development time for a development session. The equation yields P_a, which is the percentage of time spent on annotating compared to the overall development session. This approach allowed us to quantitatively evaluate how much of the development time was allocated to activities related to annotating. The average percentage for the participants was 8.58%.

5.1.1 First iteration of the evaluation study

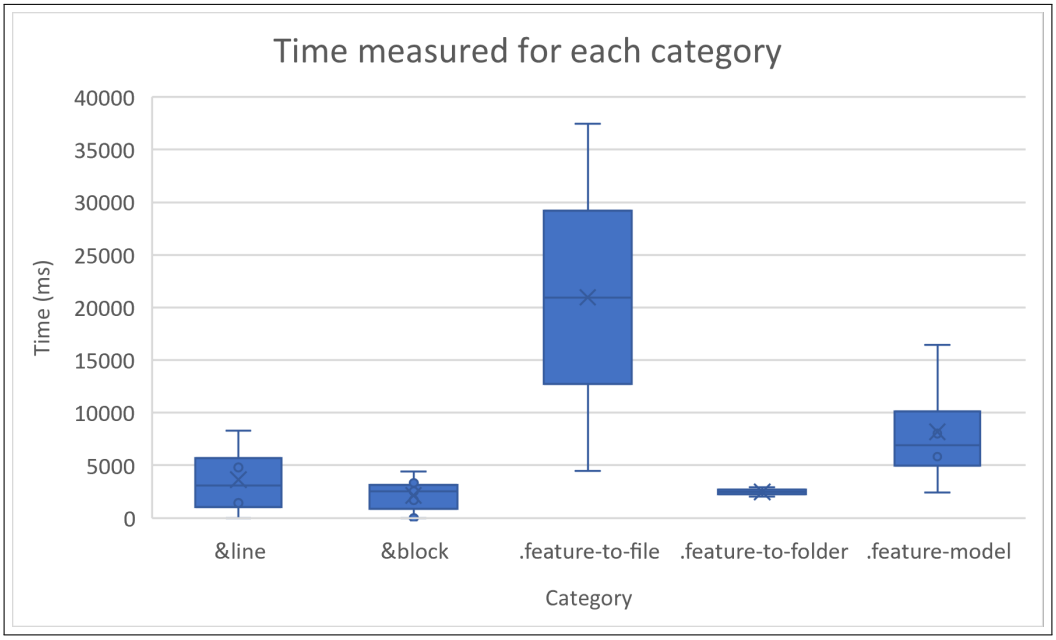


Figure 5.1: Measured times for each annotation category for participant 1

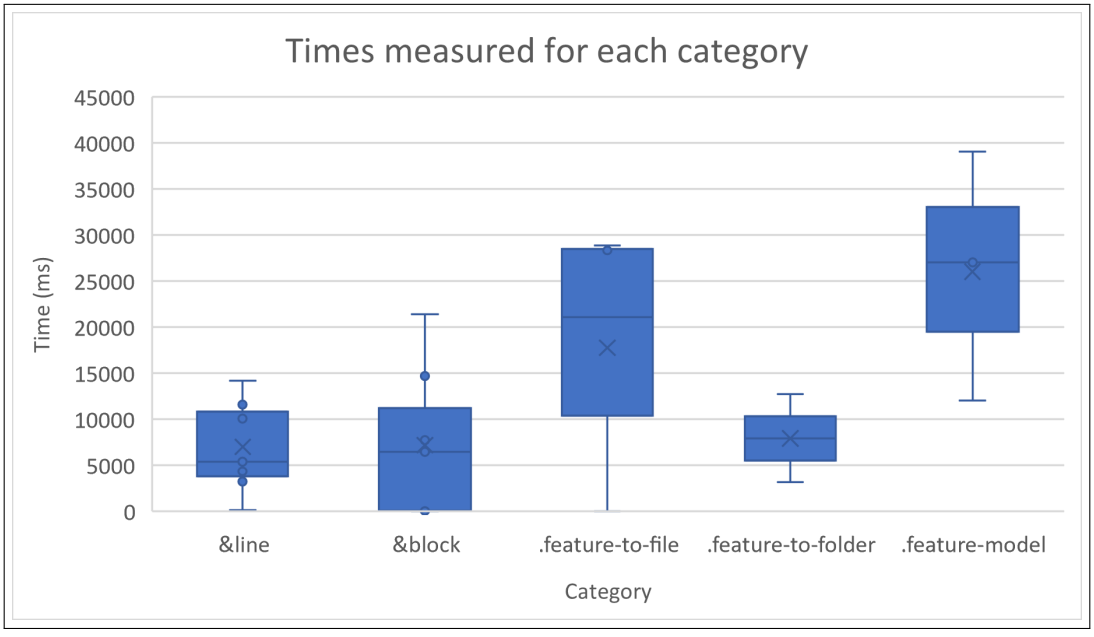


Figure 5.2: Measured times for each annotation category for participant 2

5.1.2 Second iteration of the evaluation study

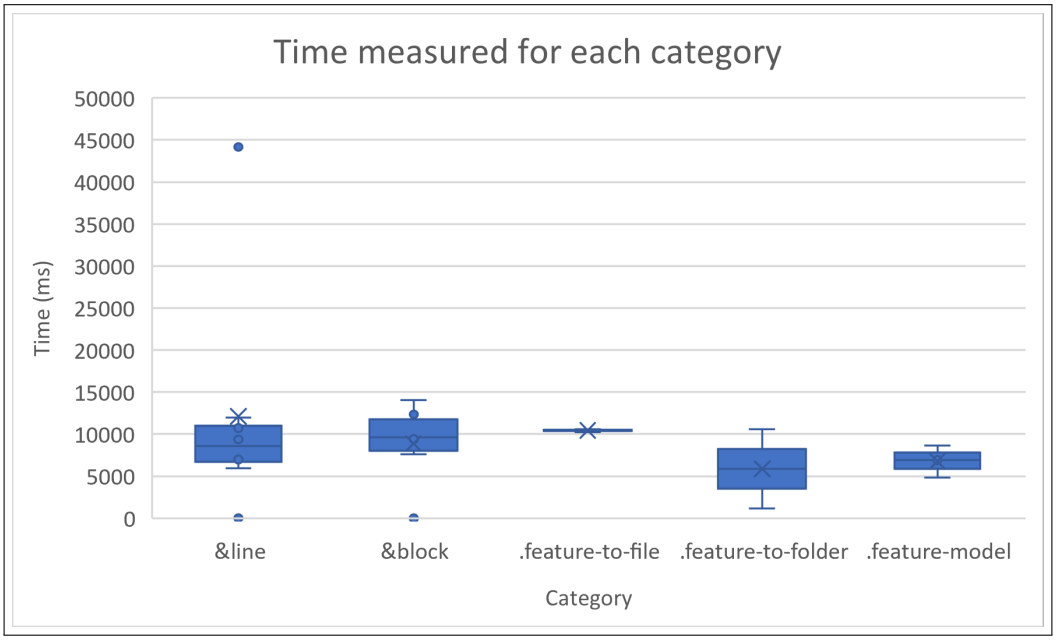


Figure 5.3: Measured times for each annotation category for participant 3

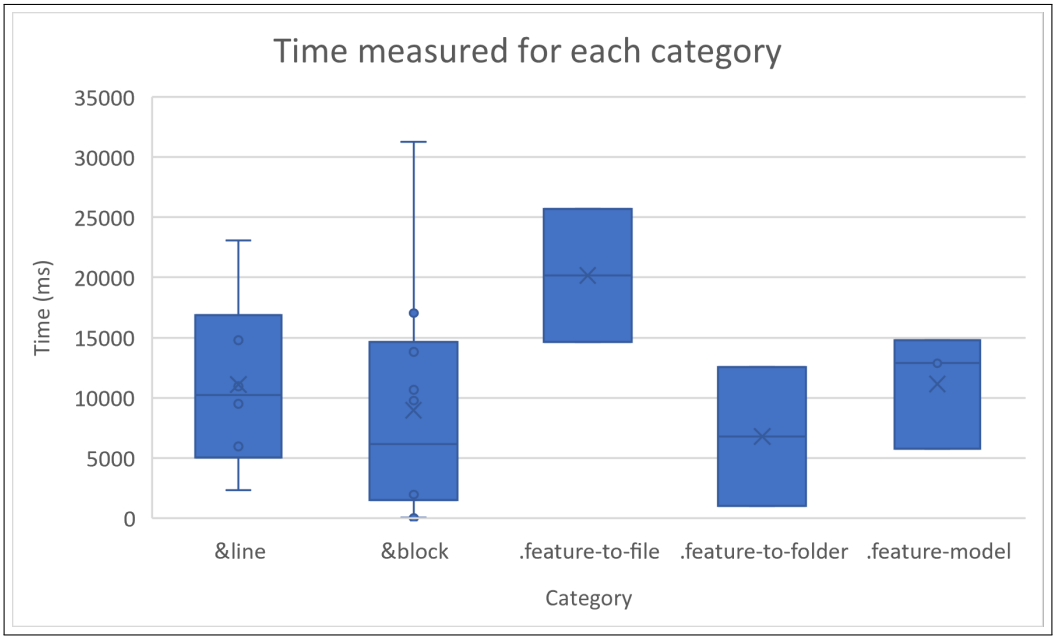


Figure 5.4: Measured times for each annotation category for participant 4

5.1.2.1 Manual review of screen recordings

During the manual review of the screen recording for participant 3, we determined that the total annotation time was 162689 milliseconds, which corresponds to 84.1% of the value recorded by the annotation logger. In the case of participant 4, the manual review revealed a total annotation time of 229803 milliseconds, accounting for 95.4% of the value obtained by the annotation logger. Tables A.9 and A.10 presents the manually measured annotation durations. These tables exclude small durations logged by the annotation logger, which resulted from utilizing the refactor feature. Since these measurements cannot be manually determined, they were included in the manual total annotation time to maintain consistency.

In general, the manually measured times for both participants were mostly accurate compared to the logged times, however, there were some inconsistencies. for Participant 3, there was one manually measured &line annotation that took 20830 milliseconds, whereas the logged time was 44160 milliseconds. Additionally, a .feature-to.folder annotation was logged as taking 10609 milliseconds but was manually measured to be 1050 milliseconds. Moreover, during the review of the footage, one logged &block annotation with a duration of 14038 milliseconds was not found. For Participant 4, three annotations with a total time of 6779 milliseconds were not found in the recording. Also, one annotation logged as taking 13819 milliseconds was manually reviewed to be 6772 milliseconds.

5.2 Qualitative experiences with annotating code

To research the users' qualitative experiences and challenges with annotating code, questionnaires were used.

5.2.1 Questionnaire results

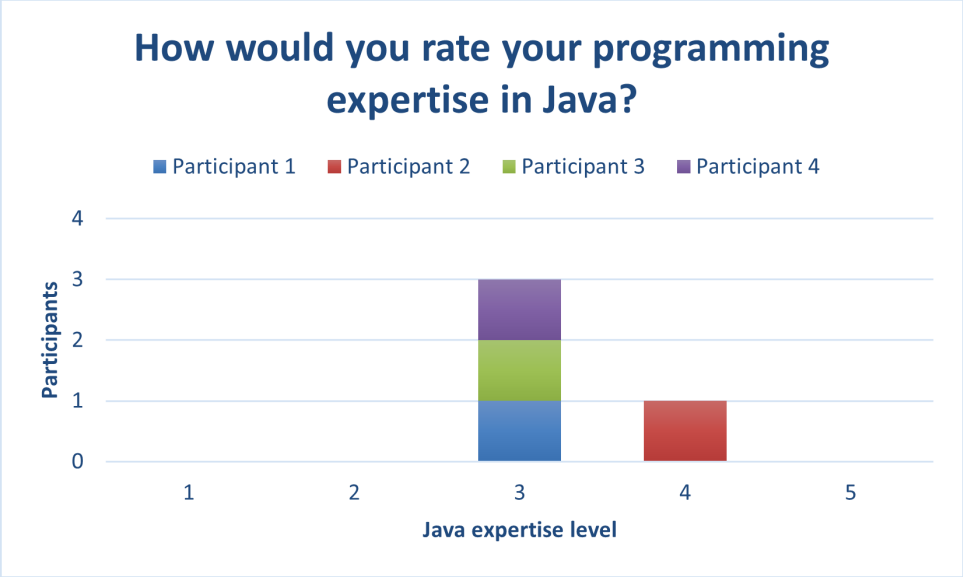


Figure 5.5: How would you rate your programming expertise in Java?

All the participants had used Java in their education and in their spare time. They got to choose their programming expertise in Java on a scale that ranged from 1 (very poor) to 5 (very good). Three of the participants rated themselves a three on the scale, and one of them rated themselves a four.

In the questionnaire, the participants were asked a set of questions on the difficulty of mapping features within the HAnS mapping files. They could answer on a scale of 1 (very easy) to 5 (very difficult). The results are presented in table 5.1.

Question	P-1	P-2	P-3	P-4
What was the difficulty of mapping a feature to a folder?	1	1	1	1
What was the difficulty of mapping a feature to a file?	1	2	1	1
What was the difficulty of mapping a feature in the feature model?	1	1	1	1

Table 5.1: Questionnaire - Mapping features

The participants felt it was very easy to map features, except for participant 2, who chose a 2 on the difficulty of mapping a feature to a file.

To evaluate the cost of using embedded feature annotations, a set of questions

regarding the difficulty of using them were asked. As in the previous questions, in this set of questions, they could answer within a range of 1 (very easy) to 5 (very difficult).

Question	**P-1**	**P-2**	**P-3**	**P-4**
How difficult was it to annotate features?	2	1	2	1
How difficult was it to refactor features?	1	1	1	1
How difficult was it to maintain features? (rename, refactor, edit existing features)	1	1	1	1
How difficult was it to locate features?	1	1	2	1
How difficult was it to determine what code to annotate?	3	1	1	2

Table 5.2: Questionnaire - Annotating features

Most of the parts of using embedded feature annotations were very easy for the participants, with some exceptions. Participant 1 motivated his answer on "How difficult was it to determine what code to annotate?", which was a 3 (neither easy nor difficult), with:

> *"Sometimes it was a little harder to understand which code to annotate and how this should be done."*

Participant 4 elaborated his answer to the question "How difficult was it to annotate features?" with:

> *"Once it's clear it's surprisingly easy"*

Else, the participants agreed that most parts of feature annotations were easy with the help of HAnS. On the question on how difficult it was to locate features, participant 1 stated:

> *"With the help of the plugin, it was very easy to locate within the code files."*

Next in the questionnaire was a set of statements that were to be answered on a scale of 1 (fully agree) to 5 (fully disagree).

Statement	**P-1**	**P-2**	**P-3**	**P-4**
It was challenging to avoid making mistakes while annotating	5	3	5	5
The time spent to make annotations compared to coding is negligible?	1	1	1	1

Table 5.3: Questionnaire - Annotating features

Participant 2 explained why he thought it was neither easy nor difficult to not make mistakes during annotating by stating:

> *"In the beginning, I made some mistakes, but with time I got better and made fewer mistakes."*

All the participants fully agreed with the statement that the time spent on making annotations compared to coding is negligible. Participant 1 explained it by writing:

> *"Compared to coding the functionality, the time spent to annotate parts of the code was almost instant."*

Participant 3 wrote:

> *"It's just that you have to remember to actually annotate, other than that it is pretty straight forward."*

And participant 4 stated to the same question:

> *"Takes literally a second"*

The next set of statements was regarding the usefulness of HAnS, and how user-friendly it was. The participants could reply in a range from 1 (fully agree) to 5 (fully disagree) on the statements:

Statement	**P-1**	**P-2**	**P-3**	**P-4**
The plugin (HAnS) was intuitive	1	1	1	1
The plugin (HAnS) was helpful	1	1	1	1
It was difficult to find the necessary features and functionalities in the plugin (HAnS)	5	4	4	5
I found that the plugin helped with accomplishing the tasks more efficiently	1	1	1	1
30 seconds of idle time is a reasonable amount to assume the developer is idle during developing	3	1	1	4

Table 5.4: Questionnaire - HAnS

The results suggest that the participants found HAnS and its functionalities useful. Regarding the usefulness of HAnS, some statements were:

> *"It's quite helpful and I can see a huge market for it"*

> *"It was pretty straight forward and the learning curve was minimal. I would definitely use this plugin in my daily work."*

> *"In larger projects, a plugin like this would bring great value to the work of a developer."*

> *"I liked the simplicity and also the efficiency of the plugin. The value it would bring to the lifecycle of a development process is something that I look forward to."*

Participant 2 found the text- and color highlighting of the code helpful. When asked about challenges, participant 1 communicated that when first using the plugin, it was a bit challenging, but with time it got easier. This is also something that Participant 2 conveyed when elaborating on the answer to the statement, "It was difficult to not make mistakes during annotating". The answers indicated that there is a learning curve when using HAnS and embedded feature annotations.

Since when measuring developing time we use 30 seconds of idle time as a threshold, before marking a developer as inactive, the participants were asked if they think 30 seconds reasonable. Participant 2 fully agreed that it is a reasonable amount of time, while Participant 1 thought it may be too short. The participant stated that the threshold could be increased to three minutes because the developer could be doing other technical stuff which should count as developing. Participant 4 also thought that the time should be increased, they stated:

> *"Takes time to get an overview of the code and what exactly implement, essentially visioning the problem and the solution in your head first. Good time would be around 1 min, perhaps a little more"*

6
Discussion

6.1 Functionality and accuracy

Functionality was implemented in the HAnS plugin to be able to measure the duration of using feature annotations and compare it to the duration of development. The time that was measured in feature annotations was when creating an annotation, modifying an annotation, deleting an annotation, and changes in the HAnS feature mapping files. How these durations were measured is explained in more detail in 4, but to summarize the timestamp of the first and last key press related to a certain annotation is stored, and the time between those are measured. The time that is spent evaluating, planning, or solving problems related to annotations, is not measured. These activities that are related to embedded feature annotations, but do not directly result in key presses, could in the practical world of software engineering add more time spent on annotations.

To assess the cost of using annotations, time spent developing was also measured. The approach undertaken in order to achieve this is explained to a greater extent in 4.4, but to condense, the active time a user spent in IntelliJ was measured as developing. If a user was idle in IntelliJ for 30 seconds or more, the time was not measured until the developer became active again. 30 seconds were chosen as it takes into consideration if a user is not actively writing code, but still developing software. Developing can also have a broader meaning. As Participant 1 of the evaluation study expressed in 5.2.1, a developer could spend time working on technical stuff related to the software development process but is not active in IntelliJ. This time was not measured in this project, but it could change the perspective of the cost of using embedded feature annotations.

During the manual review of screen recordings from the evaluation study, discrepancies were observed when comparing them to the logs generated by the annotation logger. In the case of Participant 3, our manual measurements indicated a 15.9% lower total annotation time compared to what was recorded by the annotation logger. Similarly, for Participant 4, we found that our manual measurements accounted for a 4.6% lower total annotation time as recorded by the annotation logger. We attribute some of these inaccuracies to bugs in the implementation of the "right-click" annotation support, which was introduced between the first and second iteration of the evaluation study. In certain instances, the annotation logger associated a right-click event with an annotation, when it should not have. Consequently, this led

to an incorrect start time being recorded for the annotation. It is evident that this particular feature required further testing before its inclusion in the second iteration of the evaluation study. Furthermore, there were some discrepancies which we were unable to identify the underlying cause. Additionally, there were a few instances where annotations logged by the annotation logger could not be found in the screen recordings, and the reason for this discrepancy remains unknown.

6.2 Evaluation study

The evaluation study was conducted with four participants. The study gave valuable data and insight into the use of feature annotations in the source code. The study provided both quantitative and qualitative results, but the size of the study was small. There were only four participants, and the software project "Snake" that was chosen is also very small compared to complex software systems. Therefore, the results and experiences could diverge from using embedded feature annotations in bigger and more advanced software source code.

The work and tasks software developers encounter in their profession can vary to a great degree. The tasks in this study were not chosen to give a precise representation of software development. Instead, the selection of tasks aimed to provide a deepened understanding of the cost and challenges with the different parts of embedded feature annotations while using HAnS. Thus, it is important to acknowledge that the quantitative data collected in the evaluation study may have discrepancies when compared to real-world implementation scenarios.

All developers participating in the study had never encountered or used HAnS and embedded feature annotations. This resulted in an experienced learning curve, according to the participants' qualitative experiences presented in 5.2.1. This indicates that developers with practice get more proficient and effective in using HAnS and embedded feature annotations, which can lead to less time spent per annotation.

6.3 Ethical aspects

The software which was developed and researched in this project has the potential to streamline time and effort for software developers. It may offer the ability to enhance the structure of software systems, rendering them more comprehensible and amenable to expansion. The benefits of using embedded feature annotations during a software development process may have the capacity to accelerate technological advancements globally, thereby benefiting societies and individuals alike.

The ethical dimension of this project is also observable in the use and development of open-source software. As the use of an open-source approach allows for public access to the source code and results of the project, it could aid and promote advancements in technology and software for the betterment of society.

6.4 Evaluation of the methods and process

The methods and time plan established at the beginning of the project were mostly followed. Despite encountering unexpected challenges and bugs along the way, we allocated sufficient time to address and resolve them. By adopting an iterative approach, as detailed in chapter 3, we initially built the foundation for an annotation logger and then developed it further based on new requirements. However, it would have been beneficial to more clearly define all the requirements at the beginning of each iteration, as during development additional desired features emerged. This led to not all features being fully tested, causing bugs during the second iteration of the evaluation study, as further described in section 6.1. Furthermore, considerable time was required to reach an agreement on scheduling the evaluation study. It would have been advantageous to discuss the scheduling earlier in the process. Nevertheless, the methods and processes used during the project were significant to accomplish the purpose and goals of the project.

6.5 Future work

6.5.1 Bug fixes

As previously mentioned in 6.1, there were some bugs and inaccuracies in the annotation logger. These bugs need to be addressed for future work. Specifically, the bugs related to right-click annotations, which are further explained in 6.2 needs to be fixed. Research also needs to be conducted to identify the reasons behind the logged annotations that were not found in the recordings. Furthermore, it is important to investigate and resolve the source of the inaccuracies that have not yet been identified. By solving these issues, the annotation logger will have more accurate and reliable functionality.

6.5.2 Tests

The functionality and precision of the annotation logger underwent manual testing during both the development phase and the evaluation study. To assure that there are no inaccuracies or bugs, it is important to conduct additional testing. This could be achieved by implementing automated tests and conducting a detailed review of the source code and the time measurements. By employing these measures, any potential inaccuracies or bugs can be identified and addressed, ensuring the software's reliability and precision.

6.5.3 Longitudinal collection of data

In this project, we conducted an evaluation study to gather data and opinions on the use of embedded feature annotation, but it is a topic that could use more research. A way to do this is to over a longer time period collect data from a lot of software developers from their ordinary work. This could give valuable insight into the cost for developers using feature annotations in their software projects. We suggest that

a good way to do this is to upload the HAnS plugin with the time tool feature to the IntelliJ plugin marketplace, which lets IntelliJ users use the plugin. Before releasing the plugin on the marketplace, there needs to be clear documentation of what data is collected and how it is collected.

6.5.4 Database

For this project, a temporary database was created in MongoDB. A simple API to the database was also created to let applications with the HAnS plugin communicate with the database. In this API, no validation or security was built because the participants of the studies were trusted. But for future work and data collection, there should be data validation and security implemented to protect the database from wrong data or database injections. A permanent database should also be implemented to continue to store data from the use of HAnS.

7
Conclusion

This project has researched the benefits and costs of enabling variability and traceability in source code via feature annotations. To accomplish this, the HAnS plugin was extended with an annotation logger feature. It enabled measuring the duration of embedded feature annotations in source code, as well as the duration of development time in software projects. To validate the accuracy and effectiveness of the annotation logger, an evaluation study was conducted. This study also aimed to gather quantitative and qualitative data from the use of feature annotations.

The evaluation study revealed that, overall, the annotation logger demonstrated accuracy. However, it also identified certain inaccuracies that must be addressed to obtain more precise results. Additionally, the study indicated that the average overhead associated with utilizing embedded feature annotations amounted to 8.58% of the total development time. It is worth noting that this figure cannot be directly generalized to software development as a whole, as the study specifically focused on a particular software project and its associated tasks.

To gather qualitative data, participants in the evaluation study were given a questionnaire to document their experiences. Overall, the participants had positive views regarding feature annotations and HAnS. The results indicated that the participants saw the benefits of utilizing embedded feature annotations in software development. Furthermore, the participants expressed that HAnS was a valuable tool in supporting feature annotations.

As previously mentioned, the measurements obtained from the annotation logger had certain inaccuracies and bugs that need to be addressed before conducting further studies. Nonetheless, we hope that this project has provided valuable insights into the benefits and costs associated with enabling variability and traceability in source code via feature annotations. Furthermore, we hope that the project has laid the foundation for an annotation logger that can accurately measure the duration of feature annotations.

Bibliography

[1] J. Martinson, H. Jansson, M. Mukelabai, T. Berger, A. Bergel, and T. Ho-Quang, “Hans: Ide-based editing support for embedded feature annotations”, in Proceedings of the 25th ACM International Systems and Software Product Line Conference-Volume B, 2021, pp. 28–31.

[2] T. Schwarz, W. Mahmood, and T. Berger, “A common notation and tool support for embedded feature annotations”, in SPLC ‘20: Proceedings of the 24th ACM International Systems and Software Product Line Conference - Volume B, 2020, pp. 5-8.

[3] H. Jansson and J. Martinson, "HAnS: IDE-based editing support for embedded feature annotations", thesis, 2021.

[4] W. Ji, T. Berger, M. Antkiewicz, and K. Czarnecki, “Maintaining feature traceability with embedded annotations”, in SPLC ‘15: Proceedings of the 19th International Conference on Software Product Line, 2015, pp. 61-70.

[5] S. Apel, D. Batory, C. Kästner, and G. Saake. (2013). "Software product lines", in Feature-oriented software product lines (pp. 3–15).

[6] J. Lasky, “Software development”, Salem Press Encyclopedia, 2022, [Online]. Available: `https://search.ebscohost.com/login.aspx?direct=true&db=ers&AN=100259309&site=eds-live&scope=site`

[7] “The Intellij Platform”. [Online]. Available: `https://plugins.jetbrains.com/docs/intellij/intellij-platform.html`. [Accessed: 01-May-2023].

[8] “PyCharm”, Available: `https://www.jetbrains.com/pycharm/`, [Accessed on: June 15, 2023]

[9] “CLion”, Available: `https://www.jetbrains.com/clion/`, [Accessed on: June 15, 2023]

[10] JSON, “Introducing JSON”. [Online]. Available: `https://www.json.org/json-en.html`. [Accessed: 01-May-2023].

[11] S. Chellappan and D. Ganesan, MongoDB Recipes: With Data Modeling and Query Building Strategies. Berkeley, California, Apress, 2020.

[12] D. Hows, P. Membrey, and E. Plugge, MongoDB Basics. Berkeley: California, 2014.

[13] Johmara, “Snake,” GitHub. [Online]. Available: `https://github.com/johmara/Snake`. [Accessed: May 15, 2023]

[14] Love-Ry1, “Snake,” GitHub. [Online]. Available: `https://github.com/Love-Ry1/Snake`. [Accessed: May 15, 2023]

A

Quantitative data from the evaluation study

A.0.1 Participant 1

$$P_a = \frac{T_a}{T_d} = \frac{109001 \text{ ms}}{1872661 \text{ ms}} = 5.82\%$$

Category	Time (ms)	Time (SS:ms)
&line	8297	08:297
&line	1410	01:410
&line	4803	04:803
&line	2	00:002
&block	1686	01:686
&block	3341	03:341
&block	2545	02:545
&block	9	00:009
&block	4432	04:432
&block	2892	02:892
&block	8	00:008
.feature-to-file	4482	04:482
.feature-to-file	37443	37:443
.feature-to-folder	2937	02:937
.feature-to-folder	2030	02:030
.feature-model	8024	08:024
.feature-model	2414	02:414
.feature-model	16437	16:437
.feature-model	5833	05:833

Table A.1: Measured times for each annotation for participant 1

Category	Time (ms)	Time (MM:SS:ms)
&line	14512	00:14:512
&block	14913	00:14:913
.feature-to-file	41901	00:41:901
.feature-to-folder	4967	00:04:967
.feature-model	32708	00:32:708
Total annotation time	109001	01:49:001
Total developing time	1872661	31:12:661

Table A.2: Measured times for each category for participant 1

A.0.2 Participant 2

$$P_a = \frac{T_a}{T_d} = \frac{264715 \text{ ms}}{2483200 \text{ ms}} = 10.66\%$$

Category	Time (ms)	Time (SS:ms)
&line	11608	11:608
&line	4364	04:364
&line	14226	14:226
&line	5390	05:390
&line	152	00:152
&line	10087	10:087
&line	3272	03:272
&block	21442	21:442
&block	6496	06:496
&block	14682	14:682
&block	59	00:059
&block	88	00:088
&block	7742	07:742
&block	6	00:006
.feature-to-file	1	00:001
.feature-to-file	13839	13:839
.feature-to-file	28345	28:345
.feature-to-file	28887	28:887
.feature-to-folder	3156	03:156
.feature-to-folder	12729	12:729
.feature-model	39065	39:065
.feature-model	27026	27:026
.feature-model	12053	12:053

Table A.3: Measured times for each annotation for participant 2

Category	Time (ms)	Time (MM:SS:ms)
&line	49099	00:49:099
&block	50515	00:50:515
.feature-to-file	71072	01:11:072
.feature-to-folder	15885	00:15:885
.feature-model	78144	01:18:144
Total annotation time	264715	04:24:715
Total developing time	2483200	41:23:200

Table A.4: Measured times for each category for participant 2

A.0.3 Participant 3

$$P_a = \frac{T_a}{T_d} = \frac{193335 \text{ ms}}{2541586 \text{ ms}} = 7.61\%$$

Category	Time (ms)	Time (SS:ms)
&line	11943	11:943
&line	44160	44:160
&line	9375	09:375
&line	10721	10:721
&line	17	00:017
&line	5952	05:952
&line	7737	07:737
&line	6968	06:968
&block	12352	12:352
&block	9896	09:896
&block	7586	07:586
&block	14038	14:038
&block	9379	09:379
&block	2	00:002
.feature-to-file	10609	10:609
.feature-to-file	10268	10:268
.feature-to-folder	1138	01:138
.feature-to-folder	10609	10:609
.feature-model	4801	04:801
.feature-model	8669	08:669
.feature-model	6884	06:884

Table A.5: Measured times for each annotation for participant 3

Category	**Time (ms)**	**Time (MM:SS:ms)**
&line	96873	01:36:873
&block	53226	00:53:226
.feature-to-file	20877	00:20:877
.feature-to-folder	2005	00:02:005
.feature-model	20354	00:20:354
Total annotation time	193335	03:13:335
Total developing time	2541586	42:21:586

Table A.6: Measured times for each category for participant 3

A.0.4 Participant 4

$$P_a = \frac{T_a}{T_d} = \frac{241002 \text{ ms}}{2360398 \text{ ms}} = 10.21\%$$

Category	Time (ms)	Time (SS:ms)
&line	9511	09:511
&line	10954	10:954
&line	2319	02:319
&line	5964	05:964
&line	14775	14:775
&line	23077	23:077
&block	13819	13:819
&block	9758	09:758
&block	2547	02:547
&block	13	00:013
&block	2509	02:509
&block	31298	31:298
&block	10640	10:640
&block	1951	01:951
&block	17033	17:033
&block	8	00:008
.feature-to-file	25709	25:709
.feature-to-file	14644	14:644
.feature-to-folder	12551	12:551
.feature-to-folder	1002	01:002
.feature-model	5775	05:775
.feature-model	12868	12:868
.feature-model	14786	14:786

Table A.7: Measured times for each annotation for participant 4

Category	Time (ms)	Time (MM:SS:ms)
&line	66600	01:06:600
&block	89576	01:29:576
.feature-to-file	40353	00:40:353
.feature-to-folder	13553	00:13:553
.feature-model	33429	00:33:429
Total annotation time	241002	04:01:002
Total developing time	2360398	39:20:398

Table A.8: Measured times for each category for participant 4

A.0.5 Manual review

Category	Time (ms)	Time (SS:ms)
&line	12770	12:770
&line	20830	20:830
&line	10120	10:120
&line	10870	10:870
&line	6030	06:030
&line	8110	08:110
&line	7250	07:250
&block	12190	12:190
&block	10280	10:280
&block	8120	08:120
&block	10600	10:600
.feature-to-file	10160	10:160
.feature-to-file	10960	10:960
.feature-to-folder	1210	01:210
.feature-to-folder	1050	01:050
.feature-model	5130	05:130
.feature-model	9080	09:080
.feature-model	7910	07:910

Table A.9: Manually measured times for each annotation for participant 3

Category	Time (ms)	Time (SS:ms)
&line	9710	09:710
&line	11500	11:500
&line	6390	06:390
&line	13780	13:780
&line	21710	21:710
&block	6772	06:770
&block	10680	10:680
&block	2410	02:410
&block	31110	31:110
&block	12210	12:210
&block	16470	16:470
.feature-to-file	25450	25:450
.feature-to-file	14410	14:410
.feature-to-folder	12720	12:720
.feature-to-folder	1070	01:070
.feature-model	5770	05:770
.feature-model	12920	12:920
.feature-model	14700	14:700

Table A.10: Manually measured times for each annotation for participant 4

B

Instructions for the evaluation study

Purpose:

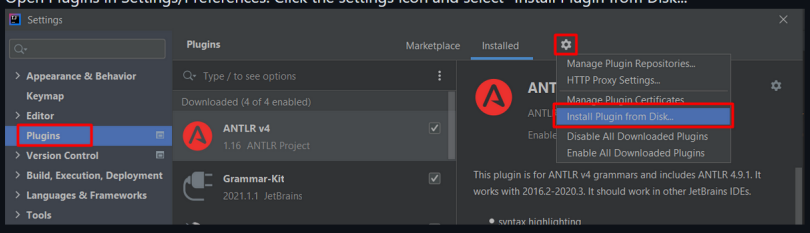
The purpose of this repo is to evaluate the benefits and costs of enabling variability and traceability in source code via feature annotations while using the plugin HAnS for IntelliJ.

Requirements:

- IntelliJ installed
- Recommended JDK 14
- Installed HAnS-text plugin

Installation of the HAnS plugin:

- Open Plugins in Settings/Preferences. Click the settings icon and select "Install Plugin from Disk..."

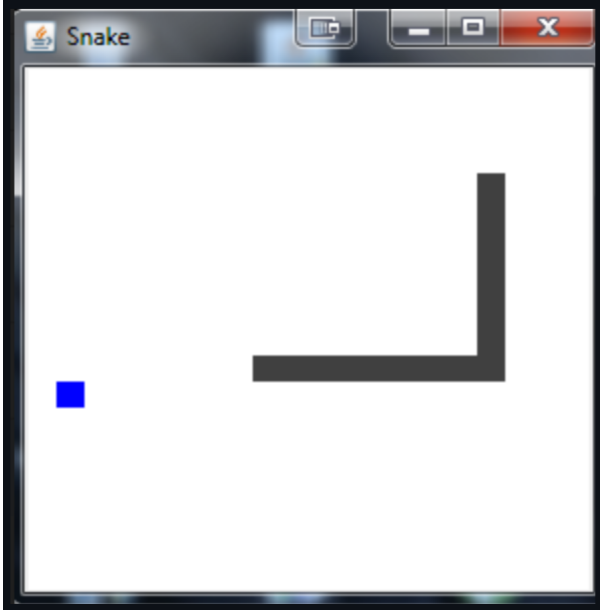


- Choose the path to the zip file of the plugin.

The Snake

A simple snake game in java, forked from @hexadeciman, using Threads and Java Swing to display the game.

How it looks:



How it works:

The aim of the game is to make the snake grow as big as possible by moving across the playing area and eating food. The snake is controlled with the arrow keys of your keyboard. No walls are present in the game so when the snake crosses the edge of the playing area it appears at the opposite side. Food is represented by blue squares that increases the size of the snake by one square when eaten. After being eaten, the food respawns at a random location not occupied by the snake. The game is finished only when the snake collides with itself in any way.

The playing area is represented by a grid of tiles where each tile has a color that signals what type of tile is located there. The background of the game is filled with white tiles where the snake may move freely. The snake itself is made up of grey tiles that move according to the directions given by the player. Food are blue tiles that have logic to enlarge the snake by one and then respawn.

Background

As software projects keeps getting bigger and bigger developers often find navigating the code to be a difficult task. Using features is a common way to talk about code and the product, but it is often difficult to find the features in code. The use of embedded feature annotations is a way to leave traces of where features are implemented.

Definition of a feature

> A feature is a characteristic or end-user-visible behaviour 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. - Apel, Batory, Kästner, and Saake (2013)

The analogy to the snake game is for example the feature `Controls` which is the collective code and behaviour concerned with pressing the arrow keys.

Feature location

A large part of the work of a developer consists of finding the implementation of a feature in code. This is necessary in order to extend, maintain and remove features, and it often requires substantial effort. This activity is known as feature location. A definition of feature location reads:

> Feature location is the task of finding the source code in a system that implements a feature. - Krüger, Berger, et al. (2018)

Embedded Feature Annotations

The usage of feature annotations is to map sections of code to functionality of the software. The intent is that this can help developers with feature location. The system of annotations that the HAnS plugin uses is able to map features to any file type and programming language (except languages that do not have support for comments). The central part of this annotation system is a file with the extension `.feature-model`. This is a feature hierarchy model, describing feature names, and their hierarchy in textual form. These are all the features present in the system, and they may then be referenced by mapping them to code by using the annotations described below. The feature model is where you define a feature. The feature model below is contains all features present in the Snake game.

```
Snake_Game
    Playing_Area
        Tile
            Food
                Spawn
            Blank
            Snake
        Update
    Snake
        Move
            Collision
        Position
        Tail
    Controls
    GameState
    DataTypes
```

Feature Reference Names

Inside the feature hierarchy model, features with the same name may appear twice or more often. To reference features uniquely the individual feature is pre-extended with its ancestor until the combined feature reference is unique (separated by "::"). This technique is called Least-Partially-Qualified name, short LPQ. The feature `Snake` is mentioned twice above and must therefore be referenced uniquely in the manner below.

```
Tile::Snake
Snake_Game::Snake
```

Feature-to-code mapping

The feature-to-code mapping serves to link specific blocks and lines of code to one or more features. The parts of the source code which are mapped to a certain feature are called annotation scopes. An annotation scope is surrounded by annotation markers and contains at least one feature reference. In the example below the feature `Move` is mapped to the block encapsulated by the `&begin` and `&end` statements. The feature `Collision` is mapped to the single line where it lies.

```
// &begin[Move]
private void example() {
    getAnotherExample(); // &line[Collision]
}
// &end[Move]
```

Feature-to-file mapping

The feature-to-file mapping is a specialized file with the extension `.feature-to-file` and is used to map one or more file(s) and its/their content to one or more features. All content of the linked file is considered fully to be part of the given feature references. The mapping file must be stored in the same folder as the source code files and covers only the file in this folder. In the example below each feature is mapped to the file listed above. Additional mappings can be added beneath existing mappings in the file.

```
Direction.java
Controls, Move
```

Feature-to-folder mapping

The purpose of this file is to map complete folders and their content to one or more feature references. The mapping of feature references to folders allows linking specific features to the folder, including all its sub-folders and files. With this, the mapping of complete folder structures to features is possible and may substitute the feature-to-file mapping. The mapping file is located on the top level inside the to be annotated folder. Let's say, for example, that a feature relates to all code in a folder, then it could be mapped by writing the feature name in a file with the extension `.feature-to-folder` as below. Features must be separated by either spaces or new lines

```
1  Snake_Game
```

HAnS: Helping Annotate Software

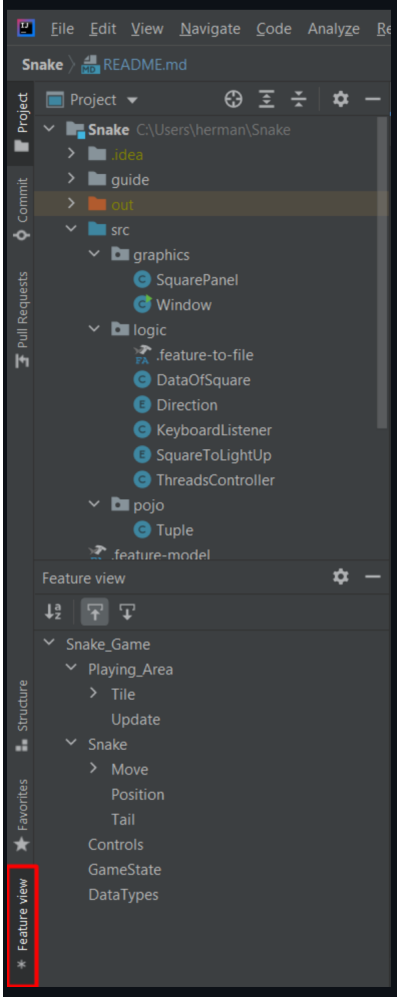
The purpose of HAnS is to enable recording and editing support for feature annotations.

HAnS-text supports:

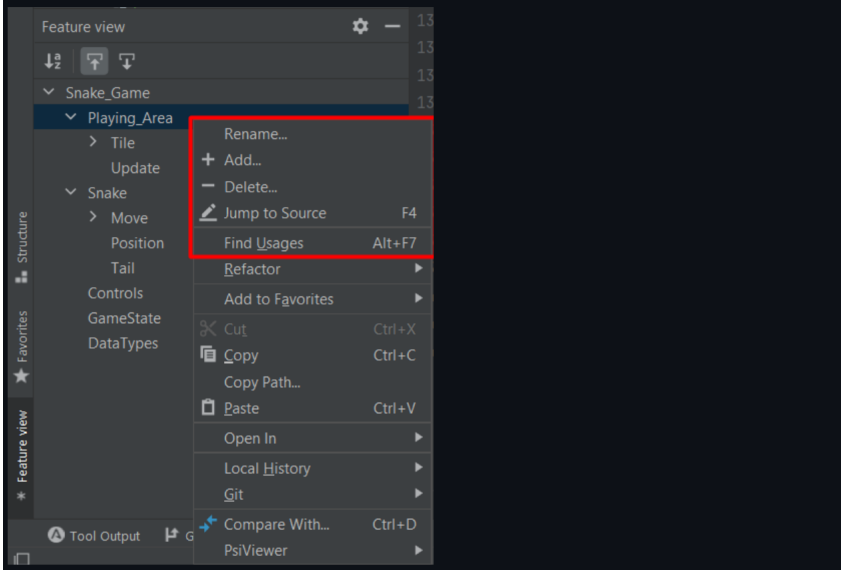
- Embedded Feature Annotations
- Syntax Highlighting
- Code Completion
- Referencing
- Refactoring
- Feature Model View
- Live Templates

Feature Model View

The Feature Model View, from HAnS, is a representation of the feature model. It is available as a tool window at the bottom left of the display (see below).



The Feature Model View supports adding, removing and renaming of features (via refactoring) as well as finding the usages of features and opening the Feature Model file by jumping to source. These actions make the corresponding changes to the Feature Model file automatically. If the Feature Model file is changed manually, the Feature Model View is updated automatically.



Live Templates

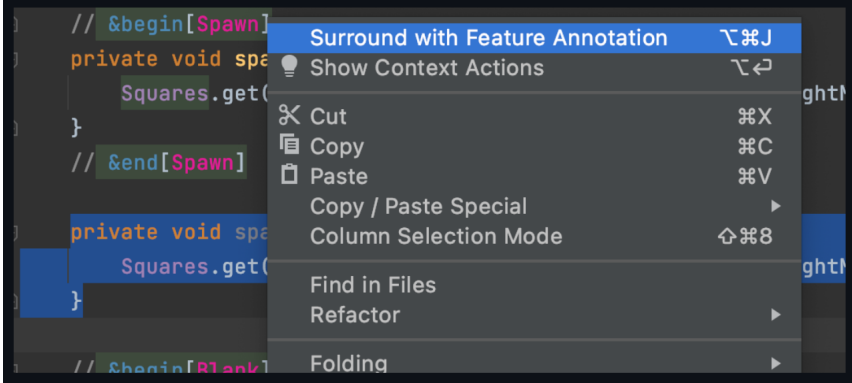
Live templates are used to insert common constructs into your code. HAnS supports a couple of live templates:

- `&begin` *This template creates a begin tag with a matching end tag.*
- `&line` *This template creates a line tag.*
- `&end` *This template creates an end tag*

To invoke a live template simply start typing the name of it and press `enter` or `tab` on your keyboard to complete the invocation.

Surround with Live Templates (Feature Annotation)

Live templates can also be used to surround marked code. To use this mark the code you want to surround press `ctrl-alt-J` (`cmd-option-J`) and choose the template you want to surround the code with. It can also be found via the right-click menu in the editor.



HAnS Demo Video

https://www.youtube.com/watch?v=cx_-ZshHLgA

C

Questionnaire

Questionnaire on benefits and costs of enabling variability and traceability in source code via feature annotations					
Participant name:					
Email:					
Role in the company:					
Education level:					
On a scale of 1 (very poor) to 5 (very good), how would you rate your programming expertise in Java?					
	fully agree	***agree***	***neutral***	***disagree***	***fully disagree***
Instructions					
The instructions were clear and easy to understand					
Please elaborate					
I know what embedded annotations are and how they are specified after going through the instruction material					
	1	***2***	***3***	***4***	***5***
Mapping features (Tasks: Warmup, 2, 5)					
On a scale 1 (very easy) to 5 (very difficult), what was the difficulty of mapping a feature to a folder?					
On a scale of 1 (very easy) to 5 (very difficult), what was the difficulty of mapping a feature to a file?					
On a scale 1 (very easy) to 5 (very difficult), what was the difficulty of mapping a feature in the feature model?					
Annotating features (Tasks: 1 and 4)					
On a scale of 1 (very easy) to 5 (very difficult), how difficult was it to annotate features?					
Please elaborate					
Refactor features (Tasks: 3 and 6)					
On a scale of 1 (very easy) to 5 (very difficult), how difficult was it to refactor features?					
Please elaborate					
On a scale of 1 (very easy) to 5 (very difficult), how difficult was it to maintain features? (rename, refactor, edit existing features)					
Embedded feature annotations					
On a scale of 1 (very easy) to 5 (very difficult), how difficult was it to locate features?					
Please elaborate					
On a scale of 1 (very easy) to 5 (very difficult), how difficult was it to determine what code to annotate?					
Please elaborate					
On a scale of 1 (fully agree) to 5 (fully disagree), it was challenging to avoid making mistakes while annotating					
Please elaborate					
On a scale of 1 (fully agree) to 5 (fully disagree) , the time spent to make annotations compared to coding is negligible					
Please elaborate					
HAnS					
On a scale of 1 (very intuitive) to 5 (not intuitive), how intuitive was the plugin (HAnS)?					
Please elaborate					
On a scale of 1 (very helpful) to 5 (not helpful), how helpful was the plugin (HAnS)?					
Please elaborate					
On a scale of 1 (fully agree) to 5 (fully disagree), it was difficult to find the necessary features and functionalities in the plugin (HAnS)					
Please elaborate					
On a scale of 1 (very efficently) to 5 (not efficently), I found that the plugin helped with accomplish the tasks more efficently					
Please elaborate					
On a scale of 1 (fully agree) to 5 (fully disagree), 30 seconds of idle time is a reasonable amount to asume the developer is idle during developing					
Please elaborate					
What I liked about the plugin					
Challenges I faced					
Suggestions for improvement					
Other comments					

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY