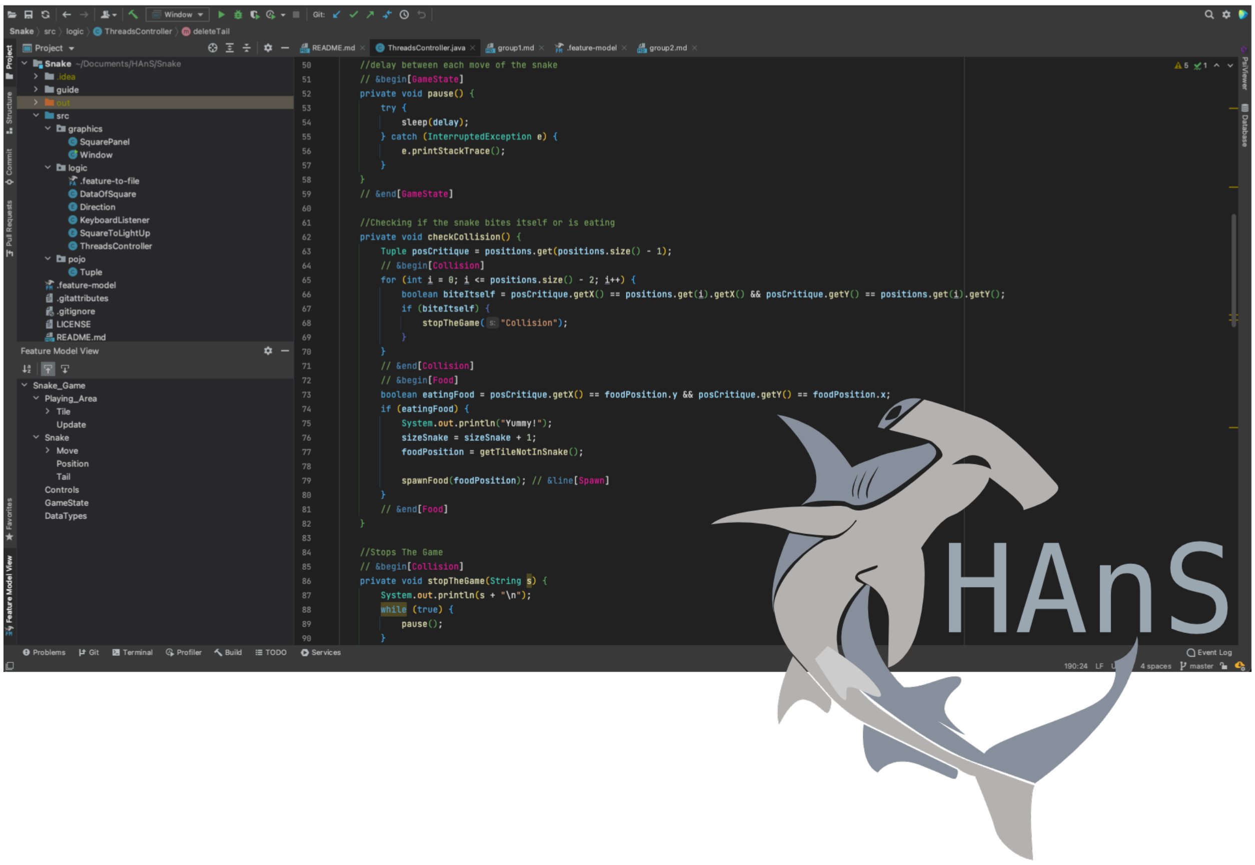
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



```
    //delay between each move of the snake
    // &begin[GameState]
    private void pause() {
        try {
            sleep(delay);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    // &end[GameState]

    //Checking if the snake bites itself or is eating
    private void checkCollision() {
        Tuple posCritique = positions.get(positions.size() - 1);
        // &begin[Collision]
        for (int i = 0; i <= positions.size() - 2; i++) {
            boolean biteItself = posCritique.getX() == positions.get(i).getX() && posCritique.getY() == positions.get(i).getY();
            if (biteItself) {
                stopTheGame(s: "Collision");
            }
        }
        // &end[Collision]
        // &begin[Food]
        boolean eatingFood = posCritique.getX() == foodPosition.y && posCritique.getY() == foodPosition.x;
        if (eatingFood) {
            System.out.println("Yummy!");
            sizeSnake = sizeSnake + 1;
            foodPosition = getTileNotInSnake();

            spawnFood(foodPosition); // &line[Spawn]
        }
        // &end[Food]
    }

    //Stops The Game
    // &begin[Collision]
    private void stopTheGame(String s) {
        System.out.println(s + "\n");
        while (true) {
            pause();
        }
```

HAnS: IDE-based editing support for embedded feature annotations

Master's thesis in Software Engineering & Technology

Herman Jansson

Johan Martinson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

HAnS: IDE-based editing support for embedded feature annotations

HERMAN JANSSON

JOHAN MARTINSON

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

HAnS: IDE-based editing support for embedded feature annotations

HERMAN JANSSON AND JOHAN MARTINSON

Supervisor: Mukelabai Mukelabai, Computer Science and Engineering
Advisor: Thorsten Berger, Computer Science and Engineering
Examiner: Christian Berger, Computer Science and Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Screenshot of IntelliJ with HAnS enabled overlapped with the logo of HAnS.

Typeset in LaTeX
Gothenburg, Sweden 2021

HAnS: IDE-based editing support for embedded feature annotations

HERMAN JANSSON AND JOHAN MARTINSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

One of the most common and widespread activities among software developers is locating features (Krüger, Berger, & Leich, 2019). However, feature location is a costly activity that is challenging and takes considerable effort to perform, as records of a features' explicit location are rarely kept. In this thesis, HAnS is presented, a plugin for IntelliJ with editing support for embedding feature annotations in source code during development, as embedding feature annotations is shown to be a cheap and reliable way to keep track of feature locations. To implement the plugin, a design science methodology was adopted where we performed user studies to find what features would make HAnS effective. The results of the user studies show that the most important functionality of HAnS is its support for writing feature annotations through convenient code completion, refactoring and navigation of features. The user studies also found that HAnS was effective at reducing the errors created during development that require correction later on. The major drawback of HAnS is its refactoring performance which is slow in larger projects. From these results, we draw the conclusion that the plugin is effective for its purpose, but that further research is needed in the form of a longitudinal study where long term use data is gathered. We also suggest further development to extend the plugin's functionality.

Keywords: embedded feature annotations, feature location, tool support, IntelliJ, IDE, integrated development environment

Acknowledgements

First and foremost, we would like to thank Mukelabai Mukelabai for a wonderful job as our supervisor, who has gladly supported us and always left us in a good mood! We also want to express our appreciation to Thorsten Berger and Alexandre Bergel for their sincere interest in our thesis and their valuable input.
Also, we want to extend our gratitude to everyone who participated in our study and review process by taking the time to contribute to our project and providing valuable feedback. Additionally, we want to thank the people closest to us for the support and sometimes invaluable help during the project.
Lastly, we want to thank our examiner Christian Berger for providing excellent feedback at crucial times and giving us the boost needed to finish on time.

Herman Jansson and Johan Martinson, Gothenburg, June 2021

Contents

List of Figures

List of Listings

List of Tables

1

Introduction

Many modern software projects specify their functionality as features, but have little to no explicit documentation of where the features are located in source code (Krüger et al., 2018). This entails that when developers evolve or maintain a software system, there is a need to locate where implementation of a feature begins and where it ends. Feature location, therefore, is a very common and widespread task (Krüger et al., 2019) that takes considerable effort to perform (Krüger et al., 2018). There are automated tools for this but unfortunately they generally have low accuracy and require significant adjustments (Entekhabi, Solback, Steghöfer, & Berger, 2019; Krüger et al., 2019, 2018; Schwarz, Mahmood, & Berger, 2020).

In order to effectively and efficiently locate features, developers may map features to source code during development, while the memory of the features is still fresh in their minds. One cheap way to achieve this is to embed feature annotations within the source code itself (Ji, Berger, Antkiewicz, & Czarnecki, 2015). These annotations indicate a feature's location by specifying where the feature begins and ends in code, or by specifying which files and folders the feature is mapped to. These embedded annotations can then co-evolve with the code base making them more robust compared to traces kept externally (Ji et al., 2015).

Annotations suffer largely the same problems as code comments i.e., they are often not prioritised by developers (Shmerlin, Hadar, Kliger, & Makabee, 2015). Therefore, tool support for making the addition of annotations require less effort is essential. One approach to make the tool as non-intrusive as possible is integrating it into an Integrated Development Environment (IDE). Such a tool can perform IDE based editing support for efficiently managing these feature annotations.

In this thesis we present HAnS: Helping Annotate Software, a plugin for IntelliJ that aims to aid developers in recording and managing feature annotations during development. While previous tools have focused on visualising existing features or retroactively locating features, HAnS actively focuses on adding annotations during development. There is therefore, a need to investigate how embedding feature annotations may be facilitated and how effectively this can be performed.

1.1 Problem statement

Feature location is a subject that has been analysed from various perspectives. It is the practice in which a developer locates a certain feature in a system. Much of the research has focused on the contrasts between automated feature location and manual feature location. Automated tools are convenient when working with preexisting code bases, but unfortunately, they are generally inaccurate in locating features and often require manual editing (Entekhabi et al., 2019; Krüger et al., 2019, 2018; Schwarz et al., 2020). Manual feature location on the other hand, while being more accurate, requires substantial effort to perform.

Previous research and tools have provided valuable insight into how features may be located and visualised, but there is a need for additional research into how to annotate features during development. The idea of this study has therefore been to build upon previous work and develop a plugin for IntelliJ, which is a popular IDE for Java developers. Other IDEs like visual studio code were considered, but ultimately IntelliJ was chosen as it has good documentation and a large community in plugin development. Since, as aforementioned, developers have a tendency to not put much effort into documenting code, a focal point of the study has been to identify how to encourage manual annotation. This encouragement may be achieved by simplifying the annotation process by providing short term benefits such as convenient visualisation inside of the tool that can help during development.

1.2 Purpose of the study

Feature location is a significant research problem given several studies that address this topic. Recent research suggests that to alleviate the problem of feature location, developers must record features early during development using embedded annotations. Also, Mukelabai, Nešić, Maro, Berger, and Steghöfer (2018) show that one of the reasons many state of art techniques for code analysis are not applied in industry is because these techniques assume existence of features in code and assume formal feature models. Yet some companies, especially those not using product lines, lack feature traceability in their code. Hence there is a need to first record features in code before applying any sophisticated feature-oriented analysis techniques. However, there is currently no tool offering developers editing support for embedded annotations.

The purpose of the study was therefore to engineer and evaluate a tool that supports developers when recording feature locations. The evaluation consisted of user studies centred around finding qualitative and quantitative data regarding the effectiveness of the tool. The tool uses embedded feature annotations since they have been shown to be cheap to apply and less intrusive with regular development activity (Ji et al., 2015). Editing support is facilitated by the application's use of techniques that can be expected from a modern editing tool.

Additionally, the study should contribute to research concerning embedded feature annotations, and ultimately the findings of the study should guide future research and development.

1.3 Research questions

The research questions we aim to answer are divided into two parts that concern the design and efficiency of the developed tool, respectively.

RQ 1 How can we design effective feature-annotation editing support in a mainstream IDE?
This research question aims to find which features are required for a tool with editing support to be considered effective.

- *RQ 1.1* How can we encourage developers to use such a tool?
  This sub question aims to gain understanding about which features or design of these features that would encourage developers to use the tool. If they encourage developers to use the tool it should also be considered as a necessity for a well designed tool. To do this we conducted brainstorming sessions with our supervisors to collect initial requirements and later refined these requirements based on feedback gathered from user studies.

RQ 2 How effective is the tool?
This research question aims to find quantitative and qualitative data to determine the usability and effectiveness of the tool.

- *RQ 2.1* What are the annotation recording and editing costs?
  In order to determine if the tool is effective, this sub question aims to measure the usability by calculating the number of errors the participants make during development that require correction later on.

- *RQ 2.2* What are the users' qualitative experiences?
  This sub question aims to asses the usability of the tool from a qualitative perspective. To do this, we asked our user study participants to answer a questionnaire about their experiences.

1.4 Limitations and delimitations

The purpose of the study is not to evaluate the usefulness of embedded feature annotations, as research has to some extent already covered that topic. It is, however, of interest to evaluate the usefulness of embedded feature annotations when using a tool for editing support, as this could help determine the usefulness of the tool.

Also, this study does not make claims about the syntax of the embedded feature annotations use. The syntax used in this thesis has been discussed previously (Schwarz et al., 2020) and therefore this study only focuses on providing editing support for that syntax.

1.5 Significance of the study

Feature location is one of the most common activities of developers (Biggerstaff, Mitbander, & Webster, 1993; Poshyvanyk, Guéhéneuc, Marcus, Antoniol, & Rajlich, 2007; Rubin & Chechik, 2013; Wang, Peng, Xing, & Zhao, 2013); however, it is still challenging since existing techniques rely on retroactively locating features, often long after implementation, resulting in low accuracy and significant effort. To alleviate inadequacies of existing tools, this study proposes a tool that supports developers in recording feature locations early, during development, thus saving future location effort and improving accuracy. Furthermore, recording features early brings other benefits such as assuring code quality since features can easily be visualised in the source code. This helps with maintainability of the code. The tool could also help bridging the gap between developers and other stakeholders; like operations teams, testers and product owners, in understanding where features are located and if they have been implemented.

This study also lays ground for further research on feature location, e.g., understanding how developers reason about features during development.

1.6 Structure of the thesis

To answer the research questions, we applied design science research with two iterations. In the first iteration we (i) engineered a set of initial requirements for a minimum viable product through brainstorming sessions, (ii) developed HAnS based on the initial requirements, (iii) conducted a user study with 6 participants to evaluate the tool and gather more requirements. In the second iteration we (i) improved the tool based on feedback from the fist user study, and (ii) conducted a second and final user study with 13 participants, (iii) analysed our findings and reported them in this thesis.

The rest of this thesis is organised as follows. Chapter 2 presents related work on the subject and previous tools concerning feature location. In chapter 3, the relevant theory is described that is crucial for understanding the study. The methodology of the study is presented in chapter 4, consisting of a description of design science research and its application for this study. Chapter 5 presents HAnS and its functionality in the latest stage of this study. In chapter 6, the results of the study are presented. Chapter 7 provides a discussion of the results in relation to the research questions and the purpose of the study. Lastly, the conclusions of the study are presented in chapter 8.

2

Related Work

The following chapter provides an overview of the related work concerning the subject. Both the research on embedded feature annotations and various tools aiming to facilitate the use of annotations are described.

2.1 Maintaining feature traceability

Feature traceability in source code is a non trivial topic that has been a subject of several research studies (Rubin & Chechik, 2013). Features can be traced to source code either immediately during development (i.e. eager strategy) or sometime later (i.e. lazy strategy). These traces can be stored internally within source assets, e.g., using embedded annotations (Ji et al., 2015) or externally using traceability databases. Storing feature traces internally is shown to be cheaper since it does not give much overhead to developers, but the traces are mainly useful if recorded early while developers remember them. When the traces are not recorded early, even if embedded, the developers run into the feature location problem, where they have to either manually search through their code or use some automated tool which may yield less accurate results. Ji et al. (2015) conclude that there is a benefit of using the eager strategy in conjunction with embedding these annotations in the source code.

2.2 Related tools

As mentioned in the introduction there exist tools designed to retroactively locate features in source code (Rubin & Chechik, 2013). These tools aim to automatically identify software at various granularity that belong to specific features. Unfortunately, automated tools generally require substantial effort to set up and result in low accuracy. One such tool is LEADT which is an Eclipse plugin which provides support for feature location in legacy code (Kästner, Dreiling, & Ostermann, 2011).

In this study we instead focus on *manually* recording the location of features. Several tools exist that are designed to utilise and take advantage of the existence of features. These tools help extract and visualise features and they demonstrate the usefulness of feature information. Some of these tools are mentioned below.

2.2.1 FEAT

In 2007, Robillard and Murphy proposed a tool called FEAT designed to represent *concerns* in source code. A concern is a broader concept of a feature which also incorporates things like nonfunctional requirements. The tool is implemented as a plugin for the Eclipse IDE and can be used in Java projects. FEAT requires manual generation of concern graphs which are stored as external artefacts.

2.2.2 FeatureIDE

In an effort to facilitate feature-oriented software development, Kastner et al. (2009) built FeatureIDE. The IDE supports several programming languages and claims to integrate all phases of feature-oriented software development. As it is an entire IDE, users must migrate from their previous IDE to FeatureIDE. The tool was developed primarily for variability management for product lines and as such is rather heavy weight.

2.2.3 FLOrIDA

Andam, Burger, Berger, and Chaudron (2017) present a lightweight tool with the goal to encourage developers to use embedded feature annotations. It is available as a standalone Java program and is thus runnable on any Java-supported platform. The tool, called FLOrIDA, approaches its goal by providing both visualisations for features and support for retroactive retrieval of features in non-annotated code. Visualisations are produced by extracting feature annotations and subsequently processing these into various views and metrics, available to the user.

Feature location recovery is achieved by a built-in mechanism that searches the project for features based on user-defined descriptions. The user then has the option to reject, confirm or edit the proposed annotations.

2.2.4 FeatureDashboard

FeatureDashboard is an open-source tool presented by Entekhabi et al. (2019). Like FLOrIDA, the tool is aimed towards providing continuous benefits to users by extracting and visualising features in a system. It is available both as a plugin to the Eclipse IDE and as a standalone software. In order to visualise features and their metrics, the tool is dependent on the existence and use of embedded feature annotations. While aiming to encourage developers to use embedded feature annotations, FeatureDashboard does not provide editing support for these.

2.2.5 Feature Annotation Extraction Engine (FAXE)

FAXE is a library developed and introduced by Schwarz et al. (2020). The tool parses and extracts annotations written in a syntax proposed by the authors, introduced here in section 3.3. Benefits of FAXE includes its language independence and that the open-source library may be integrated with IDEs and other tools. Four commands are supported, namely to retrieve annotations, calculate metrics, find inconsistencies and rename features.

FAXE also supports feature-based partial commits in Git. This is presented as a way to organise commits along features such that changes related to a specific feature may be committed separately.

Most tools mentioned aim to encourage users to use embedded feature annotations and they achieve this by focusing on the usefulness of annotations. FAXE takes another step in an effort to facilitate the maintenance of annotations through its ability to rename and check inconsistencies. With HAnS, we bring the annotations closer to the user within a popular IDE in the community. We aim to achieve convenient editing support and direct utilisation of annotations during development.

3 Theory

This chapter presents knowledge required to understand and reproduce this study. An understanding of features, feature location and the annotation syntax that HAnS uses is important. Also, information about developing plugins for the IntelliJ platform is presented.

3.1 Feature

What is a feature? How is it defined? These questions are answered in Krüger et al. (2019) where the authors perform a survey of papers on the topic of manual feature location. The definitions they found are varying to a great degree, however, the definition they found from Apel, Batory, Kästner, and Saake (2013) gives a suitable representation of a collection of definitions.

> *"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."*

This definition covers the notion that a feature is some sort of behaviour that a user can see (use) but also encapsulates that a feature is not only configurable behaviour of the system (Krüger et al., 2019). It also encompasses the fact that features may be defined in a range of granularity and used to communicate the product. For the purpose of this study, we will use the definition from Apel et al. (2013).

3.2 Feature location

A large part of a developer's work consists of finding the implementation of a feature in code (Biggerstaff et al., 1993; Poshyvanyk et al., 2007; Rubin & Chechik, 2013; Wang et al., 2013). This is necessary in order to extend, maintain and remove features and it often requires substantial effort. This activity is known as feature location. For the purpose of this study, we use the definition of feature location from Krüger et al. (2019) that reads:

> *“Feature location is the task of finding the source code in a system that implements a feature.”*

3.3 Notation for embedded feature annotations

Embedded feature annotations (EFA) can be described as styled comments that create a mapping between a location in the code and a feature (Entekhabi et al., 2019; Ji et al., 2015; Schwarz et al., 2020). The expression of “*mapping*” a feature to code means that the code fragment in question is said to be part of the implementation of said feature. This section introduces the notation and system as described by Schwarz et al. (2020), Schwarz (2020) and Schwarz (2021). This annotation system allows developers to record feature locations for folders, files, code fragments and even individual lines of code. Features are defined in a feature model, which lists all features in the software. Folders and files are mapped through textual mapping files, while fragments and lines of code are mapped using inline comments that are independent of the programming language. Figure 3.1 displays an example of how the structure for a project using this notation is set up, i.e. where the different files are placed. This notation is what HAnS is based upon.

```
ProjectRoot[Snake]
|-- .feature-model
|-- src
  |-- graphics
    |-- .feature-to-folder
    |-- Window.java
    |-- ...
  |-- logic
    |-- .feature-to-file
    |-- KeyboardListener.java
    |-- ThreadsController.java
    |-- ...
  |-- pojo
    |-- .feature-to-file
    |-- Tuple.java
    |-- ...
```

Figure 3.1: An example of the project structure of a Snake game implementation with EFA.

3.3.1 Feature model

The centrepiece of the annotation system is the feature model (Schwarz, 2021). This is a file with the extension *.feature-model*, placed in the project root (figure 3.1), that defines the available features of a system and their hierarchical relations in a textual format. The first line of the file is the *root feature*, which is named after the project, and subsequent lines represent unique features. An example of a feature model is presented in listing 3.1 (Schwarz, 2021), exemplifying the above description. The feature model serves to provide an overview of all features and every feature present in the system must be defined here to be used for mapping it to code.

```
ProjectName
    FeatureA
        FeatureA1
        FeatureA2
    FeatureB
        FeatureB1
    ...
```

Listing 3.1: Syntax of a .feature-model file.

3.3.2 Reference names

Inside the feature model, features with the same name may appear twice or more often (Schwarz, 2021). Although this means that the name is not unique, its location in the hierarchical structure of the feature model is unique. To reference these features uniquely in annotations, the individual feature is pre-extended with its ancestor until the combined feature reference is unique (separated by "::"). The resulting string is called the feature's Least-Partially-Qualified name, short LPQ.

An example is given in listing 3.2. In the example both a car and motorcycle are listed, both having *Wheels* as child features. The resulting LPQs for the distinct features become ***Car::Wheels*** and ***Motorcycle::Wheels***, respectively.

```
Vehicle
    Car
        Steering_Wheel
        Wheels
        ...
    Motorcycle
        Handlebar
        Wheels
        ...
```

Listing 3.2: Example of a .feature-model file with features of vehicles.

3.3.3 Feature-to-code mapping

The feature-to-code enables mapping of specific blocks and lines of code to one or more features (Schwarz, 2021). There are two choices of mapping, depending on if a single line or a block of code is to be linked. To map a single line an ampersand

(&) followed by the keyword *line* is added to the end of the line, followed by at least one feature reference. The feature references must be written as LPQs and are separated by commas. These references may be enclosed in hard brackets though it is not required.

Mapping of a block of code is achieved by wrapping the block with two annotation lines. These annotations follow the same syntax as a line annotation but with the keywords *begin* and *end* instead. Both block and line annotations are shown in listing 3.3.

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

Listing 3.3: Syntax of a code mapping.

3.3.4 Feature-to-file mapping

The feature-to-file mapping is used to map one or more files with their contents to one or more features (Schwarz, 2021). The file has the extension *.feature-to-file* and is placed in a directory, as can be seen in figure 3.1. The mapping file can then contain mappings only to files in this directory. All content of the linked file is considered fully to be part of the given feature references, which can be especially useful for files that do not contain any source code. Listing 3.4 (Schwarz, 2021) displays the syntax where one or more features are mapped to the file(s) listed above. Additional mappings can be added beneath existing mappings in the file.

```
File_a (File_b ...)
Feature_1 (Feature_2 ...)

File_x (File_y ...)
Feature_n (Feature_m ...)
```

Listing 3.4: Syntax of a feature-to-file mapping.

3.3.5 Feature-to-folder mapping

The purpose of a file, with the extension *.feature-to-folder*, is to map complete folders and their content to one or more feature references (Schwarz, 2021). This allows linking specific features to the folder where the *.feature-to-folder* file resides (figure 3.1), including all its sub-folders and files. The mapping file is located on the top level inside the target folder. This way, entire folder structures can be mapped to features which, in turn, may substitute the feature-to-file mapping. 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 listing 3.5 shows. Feature references must be separated by either spaces or new lines.

```
Feature_1 (Feature_2 ...)
...
Feature_n
```

Listing 3.5: Syntax of a feature-to-folder mapping.

3.3.6 Overlapping of features

There are special cases of annotations which arise due to overlapping annotation scopes. One such case is nesting of feature annotations, i.e. the existence of feature annotations within other feature annotations, shown in listing 3.6 (Schwarz, 2021). Another case is interleaving feature annotations such that one annotation begins before the previous one has ended, as shown in listing 3.7 (Schwarz, 2021). Interleaving features may not be a good idea, as stated by Rugaber, Stirewalt, and Wills (1996), as it complicates comprehension. However, interleaving features are common in complex code. Listing 3.8 shows that interleaving features without using the supported interleaving annotations needs more annotations and could, in a more complex example, increase the effort to maintain. Meanwhile, in listing 3.9 the interleaving annotations are used which reduces the amount of annotation tags needed to trace the feature. Therefore, using interleaving annotations is not required but the syntax is flexible so that it can be used if preferred.

```
// &begin[Feature_1]
... (code Feature_1)
// &begin[Feature_2]
... (code Feature_1, Feature_2)
// &end[Feature_2]
... (code Feature_1)
// &end[Feature_1]
```

Listing 3.6: Example of nested features.

```
// &begin[Feature_1]
... (code Feature_1)
// &begin[Feature_2]
... (code Feature_1, Feature_2)
// &end[Feature_1]
... (code Feature_2)
// &end[Feature_2]
```

Listing 3.7: Example of interleaving features.

```
// &begin[Food]
private void createFood() {
    foodPosition = new Tuple(15, 15);
    spawnTile(foodPosition, SquareToLightUp.FOOD);
}
// &end[Food]

// &begin[Food, Poison]
private void spawnTile(Tuple position, SquareToLightUp square){
    Squares.get(position.x).get(position.y).lightMeUp(square);
}
// &end[Food, Poison]

// &begin[Poison]
private void createPoison() {
    poisonPosition = new Tuple(5, 5);
    spawnTile(poisonPosition, SquareToLightUp.POISON);
}
```

```
// &end[Poison]
```

Listing 3.8: Interleaving features with non-interleaving annotations.

```
// &begin[Food]
private void createFood() {
    foodPosition = new Tuple(15, 15);
    spawnTile(foodPosition, SquareToLightUp.FOOD);
}

// &begin[Poison]
private void spawnTile(Tuple position, SquareToLightUp square){
    Squares.get(position.x).get(position.y).lightMeUp(square);
}
// &end[Food]

private void createPoison() {
    poisonPosition = new Tuple(5, 5);
    spawnTile(poisonPosition, SquareToLightUp.POISON);
}
// &end[Poison]
```

Listing 3.9: Interleaving features with interleaving annotations.

The third and final case is if a begin contains several feature references that end at different locations or vice versa (Schwarz, 2021). Both these scenarios are shown in listing 3.10 and 3.11, respectively. All these situations are supported by the syntax.

```
// &begin[Feature_1, Feature_2]
... (code)
// &end[Feature_1]
... (code)
// &end[Feature_2]
```

Listing 3.10: Example of features beginning at the same location but ending at different.

```
// &begin[Feature_1]
... (code)
// &begin[Feature_2]
... (code)
// &end[Feature_1, Feature_2]
```

Listing 3.11: Example of features beginning at different locations but ending at the same location.

3.4 The IntelliJ platform[1]

IntelliJ is an IDE (Integrated Development Environment) from JetBrains. The IntelliJ IDE has several different types of plugins such as themes, tools and languages. The IDE is primarily made for programming in Java but has extensive support for other languages, for example Javascript, C, PHP, HTML and SQL. Plugins developed with the IntelliJ platform can, as long as no dependencies are on Java, also be used by any of the other JetBrains IDEs, for example PyCharm or CLion.

Plugin development is a large part of the IntelliJ community and JetBrains consequently provides documentation called the IntelliJ Platform SDK which provides

[1]The Official Documentation of the IntelliJ Platform:
https://plugins.jetbrains.com/docs/intellij/welcome.html

detailed information on how to create plugins and custom language support. From the documentation, we have found that plugin development for the IntelliJ platform is rather modular. This modularity is realised by extension points that define available ways to extend functionality. By using these extension points it is possible to integrate a plugin closely into IntelliJ and utilise the inherent behaviour and performance of the IDE. There is also support for creating so called actions for a plugin that can be used to perform specific tasks in IntelliJ.

HAnS utilises 15 distinct extension points and registers five new actions. Many of the extension points were used multiple times, such as the *filetype* and *parserDefinition* extensions that were used for defining the annotation languages. Other extension points were used for example to add a tab view by extending *toolWindow* and validating feature names by extending *renameInputValidator*.

The actions that were created performed partly renaming, addition and deletion of features. Other actions were defined to create new annotation files and to surround a code block with annotations. Once registered, these actions can be accessed in the plugin where suitable, e.g. in the right click context menu.

4
Methodology

The study presented in this paper was performed according to the design science methodology as described by Peffers, Tuunanen, Rothenberger, and Chatterjee (2007). The idea behind this is to iteratively develop an artefact in order to get feedback in-between each iteration. This chapter describes the steps taken for each iteration and how data was collected.

4.1 Design science research

In order to answer the established research questions, the study involved two iterations. The iterations consisted of, development of the tool and subsequently an evaluation by conducting user studies. During development of the plugin, the authors annotated its source code, thereby gaining insights on challenges developers might face and how they could be mitigated. The steps involved in design science are described below.

The iterations can be seen in figure 4.1 which describes the steps taken. The first iteration of the study consisted of steps 4.1.1-4.1.5 after which the second iteration proceeded with steps 4.1.2-4.1.6 until the final results were obtained and reported at the end of the second iteration.

4.1.1 Problem identification and motivation

The first step was covered at the beginning of the project and is described in Chapter 1 of this report. This was the foundation for conducting the study and describes the problem and a motivation for why this project was important to pursue.

4.1.2 Definition of objectives for a solution

With a clear understanding of the problem to be addressed, there was a need for defining the initial requirements for a solution as well as what should be considered a success. This was discussed with the supervisor and was also based upon the literature of previous studies. These requirements can be found in section 4.3. The feedback from the first user study was used to improve the tool in the second

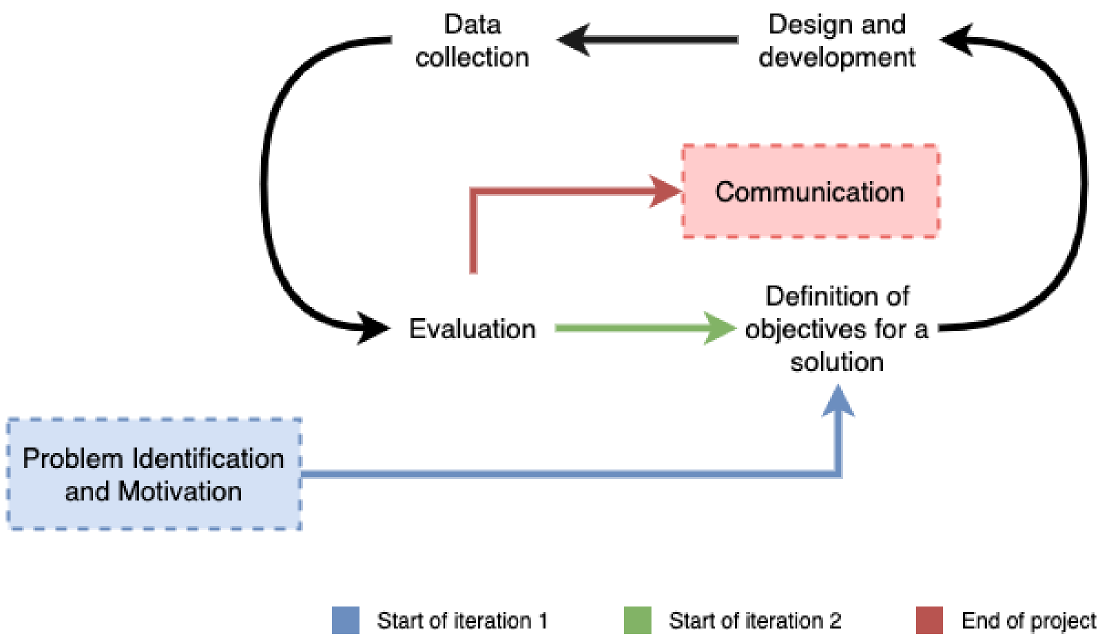


Figure 4.1: Design science approach indicating the content of both iterations.

iteration. As the results of the first iteration help answer what features are needed for effective editing support (RQ1), the second iteration is influenced by the features regarded, as required by the participants of the user study.

4.1.3 Design and development

This step consisted of the actual implementation of the tool as an IntelliJ plugin. The process was adaptive to encompass the knowledge gained about challenges of feature annotation alongside the implementation. During the first iteration, this stage also included a thorough investigation of how plugins are created for IntelliJ.

4.1.4 Data collection

In order to assess the usability and effectiveness of HAnS we conducted user studies with groups of both students and practitioners, who all studied or worked as software or computer engineers. Six people participated in the first study of which one had professional coding experience and 13 participated in the second, where seven people had professional coding experience. These participants were tasked with completing a set of tasks described in further detail under section 4.2. The participants' screens were recorded for further analysis. After performing the tasks, the participants were asked to answer a set of questions with the aim to gain qualitative and some quantitative data. The answers were then used to answer the research questions.

4.1.5 Evaluation

Following the collection of data, an evaluation in order to interpret responses from participants was conducted. The results from the evaluation were used to answer the research questions and to update the requirements of the tool, in order to enable improvement of the plugin for the second iteration. The evaluation was carried out by discussion between the authors and the supervisors in order to determine what parts of the tool that were considered a success and what needed improvement. This was evaluated in relation to the requirements that were specified in step 4.1.2. Other comments and suggestions from the user studies were also considered. The evaluation is later used to discuss the research questions. The functionality that was found to answer what features are needed for effective tool support (RQ1) were used for the second iteration of the study in order to provide answers to how effective the tool is (RQ2).

4.1.6 Communication

The final step of the study was to communicate the findings and their importance. This report serves as that communication as well as having the project open source for possible further development and research.

4.2 User studies

In design science research it is recommended that every iteration should provide data for the research questions, although the emphasis may differ between iterations (Knauss, 2021). User studies were performed in both iterations where the user study for the first iteration was aimed at answering RQ1 as well as gaining feedback on the features of the tool. The study performed for the second iteration was primarily aimed at answering RQ2 but also to validate the answer from RQ1. Together the two iterations provide answers to all research questions.

4.2.1 Design

The user studies were designed with focus on gaining an understanding about the plugin and the editing support it provides and not the EFA notation. A crossover design was chosen to eliminate the risk that the order in which the plugin was used would affect the result of the study. The tasks were split into two task-sets and the participants into two groups so that one group performed the first task-set with the plugin enabled and the second task-set with the plugin disabled. The other group performed the same tasks but had the plugin enabled in the reverse order. The structure of the two groups is presented in table 4.1.

Table 4.1: Visualisation of the crossover design.

	Group 1	Group 2
Task-set 1	Plugin enabled	Plugin disabled
Task-set 2	Plugin disabled	Plugin enabled

In order to get the participants' opinion on the plugin the users needed to use the tool in the context of an existing project. Therefore, tasks were constructed for an example system that was simple enough for the users to understand, such that the tasks could be executed in a maximum of 90 minutes. In order to use a crossover design, the example system also had to be large enough to include at least two different features with similar complexity, as to avoid learning bias between the task-sets. An open source project of appropriate size was found in an implementation of a Snake game[1]. To prepare the project we refactored it to an appropriate structure and annotated some parts of the code with feature names.[2] Annotations were added to help the participants with the syntax of the annotations and to get the full use of the plugin while locating as well as recording new locations for a feature. In the following sections, we describe the Snake game followed by a summary of the tasks we created for the user studies and a description of the participants of the study.

All participants participated through remote video calls. Initially, the participants were informed of the structure of the study as well as given instructions for installing IntelliJ, downloading the repository, installing the appropriate JDK and the HAnS plugin. During the introduction of the user study, both an oral introduction and a textual introduction of the embedded feature annotation notation was given.

During the study, the participants were asked to record their screens such that these recordings could be used in a later stage to see how they navigated in IntelliJ. The recordings were later used to check for errors made by the users and as confirmation that the correct tasks were performed. During the sessions, we were also available for clarifications of the tasks or the syntax of EFA.

The Snake game

The example project that was used consisted of a simple implementation of a snake game. The game consists of a grid of tiles that the snake may move freely across. No walls are present in the game, so when the snake crosses the edge of the playing area it appears on the opposite side. The snake is controlled with the arrow keys and the goal is to grow as large as possible by eating "food" that pops up on the grid, represented by blue tiles. Figure 4.2 shows the running game as it is before performing the implementation tasks of the study. After being eaten, the food re-

[1]Snake game implementation by @hexadeciman at GitHub:
`https://github.com/hexadeciman/Snake`

[2]The refactored Snake game implementation:
`https://github.com/johmara/Snake`

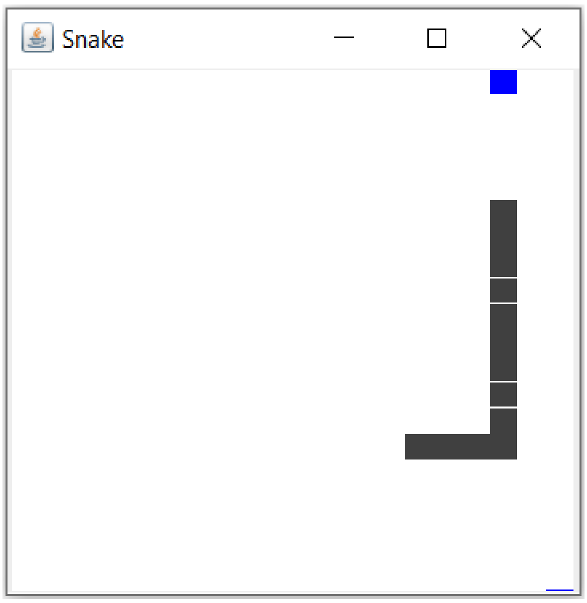


Figure 4.2: Screenshot of the Snake game before implementation tasks.

spawns at a random location not occupied by the snake. The game is finished only when the snake collides with itself in any way.

The source code is split into three packages that are all subject to be annotated according to the tasks performed by the participants of the study. In total, the project consists of 15 features across eight Java files.

Overview of the tasks

The sets of tasks we created had two aims: (i) finding what features are needed for effective editing support (RQ1) and; (ii) measuring the recording and editing costs (RQ2).

The first task was to implement a new feature that was similar to another feature present in the example system. This task aids in finding the answer to RQ2, as the participants get to experience what it would be like to record a new feature's location in a system. Defining features in a feature specification, in our case the feature-model, and mapping them in code is relevant but needs to be adopted by companies that want to trace features in code since most do not. However, these are standard activities for companies that develop variant rich systems such as product lines or highly configurable systems, for instance the Linux kernel. Implementing a new feature also forces the participants to think about the different granularities of the EFA notation, described in section 3.3, and when to use them. By defining the implementation of the new feature to be similar to another we also guide the participants to locate a feature. This is an important part of the task as research shows that feature location is one of the most common development activities. It also makes it easier for the participants to implement as they could use the existing feature as a guide to implement the new. The goal is to target feature traceability in the Snake game so that the participants can reason about the features in the source code.

The second task was to rename a feature, which is also important in finding an answer to the editing costs. Mukelabai et al. (2018) studied industrial needs for analysing configurable systems and revealed that practitioners would like to know consequences of renaming and even be able to rename features or split features. As such, code often gets refactored, e.g. a big feature might be split, in which case it might be renamed. The tasks were performed two times, so that a participant would perform the tasks both with and without the plugin enabled, which is explained in section 4.2. We provide a more detailed description of the tasks in appendix A.1 for the first study and in appendix B.1 for the second.

After the programming tasks the participants were asked to fill out a form where we defined questions to help answer RQ1 and RQ2. The idea was that by having used the tool during the programming tasks they would have opinions on what was missing in the design for the tool. The questions posed in the respective iterations can be found in appendix A.2 for the first study and in appendix B.2. Through verbal communication, the participants understood that we were conducting a research-based study and that their responses were relevant to improve the tool and guide further research. The questionnaire was anonymous so that we could use participant responses in our report without revealing personal details of the participants.

Participants

The participants in the user studies had either professional or educational experience. Their average years of experience with Java can be seen in table 4.2. We selected participants by sending out messages to different forums (a Facebook group for students in the division and various groups of former and current students at Chalmers University of Technology) with an approximate reach of 150 people. The chosen forums had a general population of individuals with some sort of experience with coding. The reasoning behind choosing participants in this way was driven by RQ1.1 (how to encourage developers to use the tool) and RQ2.2 (qualitative experiences of using the tool). In order to get different opinions and find the answer to RQ1.1 collecting participants with an array of different coding experience was important, as that helps determine what is important for all developers independent of experience. This spectrum of participants was fulfilled by reaching out to groups that were known to include this variability and also asking about experience in the questionnaire. The participants' experience with IntelliJ was not explicitly investigated although it became clear that there was a variety in levels of experience. This was desired as both experienced and inexperienced users should be able to use the plugin.

4.2.2 Verification of design

To verify the study design, we conducted a pilot study with one participant, who did not take part in the final experiment. During the pilot study several semantic

Table 4.2: Number of participants and their average experience.

Iteration	Type of experience	#	avg experience with Java
First	Professional	1	4-5 years
	Only educational	5	2-3 years
	Total:	6	3-4 years
Second	Professional	7	2-3 years
	Only educational	6	2-3 years
	Total:	13	2-3 years

mistakes were observed and corrected for the final design. The difficulty of the tasks were adjusted to be suitable for the scope of the study by performing the tasks ourselves. Lastly, the introductory information of the study was elaborated with clearer descriptions of feature location and embedded feature annotations.

4.2.3 First iteration

The user study of the first iteration focused on getting answers to RQ1. Since initial requirements had already been elicited as described in section 4.1.2 the iteration had the specific emphasis on evaluating the existing functionality of HAnS and finding which features are most important to implement for the second iteration.

The study for the first iteration were conducted over the span of three days with six participants divided into the two groups shown in table 4.1. The questionnaire was split in to two parts: (i) asking about the features and tasks completed with the plugin and; (ii) asking about the tasks completed without the plugin. We decided that participants should answer each part of the questionnaire immediately after performing tasks corresponding to the task-set with or without the plugin. The reason behind asking the questions after each task-set and not at the end, was so that the participants would have their thoughts on the task-set fresh in their mind and to not introduce any bias against or for the plugin while executing the second task-set of the study.

4.2.4 Second iteration

The second iteration was aimed at answering RQ2 and to confirm that the functionality that was concluded to be necessary, for a well designed editing tool (RQ1), was justified.

The study for the second iteration was conducted over the span of two days with 13 participants divided into the groups described by table 4.1. After observing the participants in the first iteration it was clear that a more thorough oral description of the EFA notation and the plugin should be held before the tasks were started. We decided that this was justified as we are not testing the participants proficiency with either Java or the EFA notation and because the participants should understand what the plugin can do for them so that they do not disregard any functionality. We also decided that the survey for the second iteration only should be answered after all tasks were completed. This way the participants could compare the experience of using annotations with the plugin to without it.

The design of the survey was changed from being divided into two parts to instead only have one, where the questions mostly conformed to statements and participants answering to which degree they agreed with the statement. The statements were largely inspired by the System Usability Scale (SUS)(Brooke, 1996).

System Usability Scale

The SUS presents a set of statements, to which participants respond in a five degree agreement scale: (i) *strongly disagree*; (ii) *disagree*; (iii) *neither agree or disagree*; (iv) *agree* and; (v) *strongly agree*. This has been shown to be a robust method for evaluating the usability of a product (Bangor, Kortum, & Miller, 2008). The SUS statements are both positive, *"I would like to use this system frequently."*, and negative, *"I found the system unnecessarily complex."*. The positive statements are scored with *Strongly disagree* corresponding to a score of zero and *Strongly agree* corresponding to a score of four. For the negative statements these scores are reversed, i.e. *Strongly disagree* corresponding to a score of four and *Strongly agree* corresponding to zero. All participants' scores for the statements are added together and then divided by the number of statements to create a mean score. The mean score from the SUS statements is multiplied by 2.5 to construct a score of 0-100 where a score above 70 is considered passable. Scores below 70 are candidates for increased scrutiny and improvements and scores below 50 are cause for significant concern. One part of the evaluation was inspired by the SUS but did not include all statements. Removing statements from the SUS structure has been shown to still provide reliable results given that the score is adjusted accordingly (Lewis & Sauro, 2017). By using the SUS, we are able to quantify the qualitative experiences regarding the usability of the tool.

4.3 Minimum Viable Product

The minimum viable product (MVP) was determined by discussing features with the supervisors and researchers in the field. The MVP was used to prioritise work and have a measure of when the user studies could be carried out. Table 4.3 lists the functionality of the MVP paired with their acceptance criteria.

This initial list of features comes from years of experience working with practitioners who expressed interest in a tool to annotate and visualise features in code. It was relevant to have an initial version of the system so that our user study participants had something to work with to make their feedback more meaningful rather than conducting interviews with people who would have to imagine the situation. This way, the feedback was more relevant since the participants worked with the prototype tool and suggested features that they need but were missing while working with the tool.

Table 4.3: Minimum viable product

Feature	**User Story**	**Acceptance Criteria**
Code Completion	As a user I want to be provided options for code completion based on features in the feature model.	• When within brackets after a keyword, completion options are shown that correspond to the features listed in the feature model. • When pressing $ctrl + space$ while within the brackets completion options are shown.
Syntax Highlighting	As a user, I want the plugin to provide syntax highlighting so that annotations may be easily identified.	• The keyword in an in-code annotation should be highlighted with a colour. • A feature in the feature model, feature-to-folder, feature-to-file and the in-code annotation should be highlighted. • The colour for features should be the same for all contexts.
Refactoring	As a user I want to be able to change names of features via refactoring.	• Refactoring a feature in any context propagates to all its uses.
Feature Model View	As a user I want to have a visualisation of the structure of the feature model.	• The view visualises the feature model and changes as the model changes.

Quick fix new feature	As a user I want to be able to add a feature to the feature model via a quick fix.	• If a feature in an annotation is not in the feature model it should be added by pressing *alt* + *enter*. • An option should be provided such that the feature can be added to a feature-to-file file. • An option should be provided such that the feature can be added to a feature-to-folder file.
Live Template	As a user I want to be provided options for completing an annotation after the first character has been entered so that making annotations becomes a quicker task.	• User is provided with begin, end and line option when typing an '&'.
Warning of undefined features	As a user I want to be warned if an embedded annotation contains a feature which is not in the feature model and if a feature in the feature model is never used.	• An error in the form of a red marker is visible if a feature is not yet in the model. • The annotation is underlined with a red squiggly line. • If a feature defined in the feature model is not used it should be highlighted with grey.

4.4 Other functionality

Some of the functionality that was discussed was not considered crucial for the success of the project. These features may still provide value and are this listed in table 4.4 with a priority rated medium (**M**) or low (**L**).

Table 4.4: Other Functionality

Feature	**User Story**	**Acceptance Criteria**	

Highlighting Settings	As a user, I want to be able to change syntax highlighting settings.	• Highlighting can be changed for the annotation and custom files.	**M**
Feature Referencing	As a user, I want to go to where a feature is defined by *ctrl* + *click* on a feature name so that I can get quick overview of the hierarchy.	• The Feature model file is opened and brought to focus. • The feature referenced is highlighted in the file. • The feature referenced is scrolled to when the file is brought to focus.	**M**
Surrounding Live Templates	As a user I want to be provided with options to add annotations from a context menu when marking lines of code.	• When selecting the block option the marked code should be annotated as a block. • When selecting the line option all marked lines should be annotated with line annotations.	**M**
Find usages	As a user, I want to be able to find usages of features so that I can get a quick overview of where annotations of features are located.	• When using the built in function in IntelliJ to find usages, all occurrences of the selected feature are displayed.	**M**
Line Markers	As a user, I want line markers on annotated lines so that annotations may be spotted easily.	• An icon is displayed next to annotated lines.	**L**
Commenting a feature	As a user, I want to be able to comment out a feature or add a comment in the feature model by using the built in short cuts in IntelliJ.	• Built in shortcut for commenting should work in the new custom file types.	**L**

5

Implementation of HAnS[1]

The implementation of the plugin was realised by following the notation for embedded feature annotations described by Schwarz et al. (2020). Functionality was implemented as a result from focus groups with the supervisors and from the user study in the first iteration.

5.1 Languages

In order to set the foundation of the tool there was a need to define the languages that are used by the plugin. An effort was made to define these as similar and compliant as possible to the grammar described in Schwarz et al. (2020), summarised in section 3.3. The IntelliJ Plugin Development SDK provided helpful documentation for implementing custom languages. Each language is generated from a definition of the syntax and a custom lexical processor (lexer), for the specific grammar, is also generated to parse the languages. The parsing uses the elements in the syntax that the lexer can understand and categorise them into usable elements. These elements can conveniently be used for navigating the files and defining logic based on the categorisation of the elements.

5.1.1 Feature model

The feature model is the central part of the annotation system and is where features are listed and structured hierarchically. Part of the feature model file from the Snake code base is presented in listing 5.1. The language conforms to the syntax of the EFA notation which dictates that the file begins with a project name and is followed by feature names of varying indentation depending on their hierarchical structure.

```
Snake_Game
    Playing_Area
        Tile
            Poison
                Spawn
            Food
                Spawn
            Background
```

[1]Source code: `https://bitbucket.org/easelab/hans-text/src/master/`

```
            Snake
        Update
```

Listing 5.1: Part of the .feature-model file in the Snake game with HAnS enabled

The elements, categorised as feature names, in this language have some defined logic that enables renaming, adding and removal of elements (feature names). This logic can be used for other functionality.

5.1.2 Folder annotations

The feature to folder annotations is the most simple file type. As defined in the EFA notation, the file simply consists of any number of feature references separated by spaces or new lines as seen in listing 5.2.

```
Playing_Area
Tile::Snake
```

Listing 5.2: Example of a .feature-to-folder file

5.1.3 File annotations

File annotation files are separate files that map features to certain files. They follow the syntax from the EFA notation where two kinds of elements are categorised by the lexer: (i) filenames and; (ii) feature names. The filenames, written on the first line with the extension, gets mapped to the feature name, written on the second line in listing 5.3.

```
Direction.java
Controls, Move

KeyboardListener.java
Controls
```

Listing 5.3: Example of a .feature-to-file file

5.1.4 Code annotations

Code annotations differ from the previously mentioned types in that they reside within files of other languages. To make this possible, a separate language was defined which subsequently is injected into files that have code annotations within comments. This means that, when files are indexed, code annotations are recognised and the language is injected into the comment. Using comments for annotations enables utilising the built-in structure of the IDE.

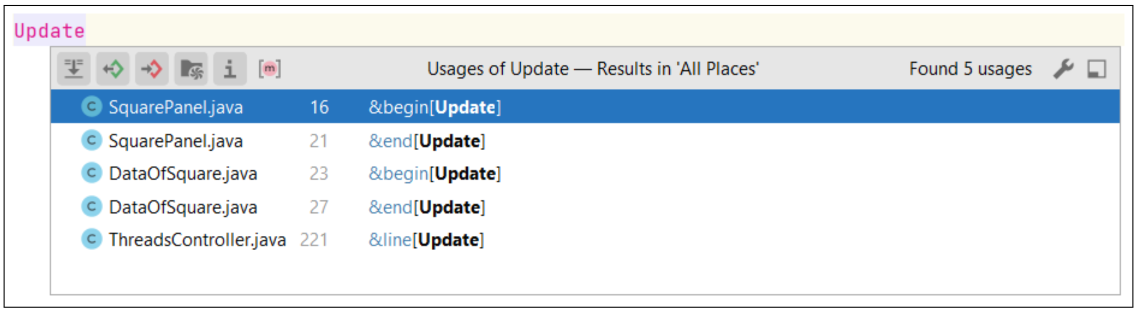
```
// &begin[Food]
foodPosition = new Tuple(Window.getWindowHeight() - 1 ,
                        Window.getWindowWidth() - 1);
spawnFood(foodPosition); // &line[Food::Spawn]
// &end[Food]
```

Listing 5.4: Example of both block and line code annotations

As the EFA notation describes, code annotations are of either block type or line type, which both are supported by HAnS. Examples of both types are visualised in listing 5.4.

5.2 Referencing

A crucial part of the plugin is the ability to recognise where features are referenced in code. Referencing fulfils this need by linking all references of a feature to the definition of that feature in the feature model, such that they all represent the entire feature. This enables convenient navigation of features that gives the user easy access to where feature are located in code.



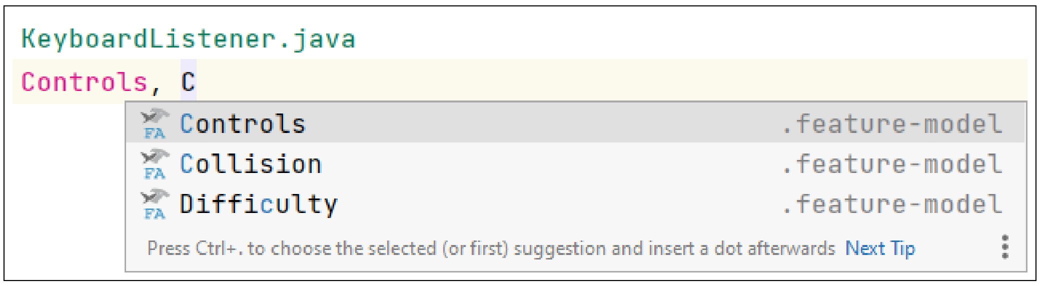
```
Update
Usages of Update — Results in 'All Places'          Found 5 usages
SquarePanel.java        16   &begin[Update]
SquarePanel.java        21   &end[Update]
DataOfSquare.java       23   &begin[Update]
DataOfSquare.java       27   &end[Update]
ThreadsController.java  221  &line[Update]
```

Figure 5.1: Finding usages of “Update” from the .feature-model file.

Referencing is also the infrastructure that enables other functionality, e.g. refactoring. For a reference to work it must be using the LPQ of the feature that is referenced, such that it is uniquely identified as a certain feature defined in the feature model.

5.3 Code completion

HAnS supports code completion for all of the EFA languages, which syntax is described in section 3.3, and provides completion based on the context. If the context of the caret is a feature name, then the user is presented with a list of all feature names stated in the feature model, which is visualised in figure 5.2.



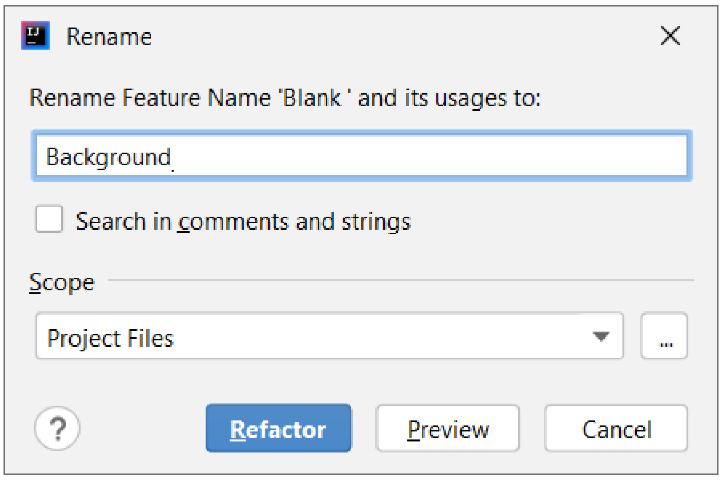
```
KeyboardListener.java
Controls, C
    Controls                 .feature-model
    Collision                .feature-model
    Difficulty               .feature-model
    Press Ctrl+. to choose the selected (or first) suggestion and insert a dot afterwards  Next Tip
```

Figure 5.2: Code completion suggestion in HAnS.

Another case of code completion is the list of suggestions that the user is presented with when writing specifically in a feature-to-file. If the caret is in the context of writing file names the user gets a list of all files within the directory of the file.

5.4 Refactoring

Refactoring provides the users of HAnS with an option to change the name of a given feature from anywhere that it is referenced. HAnS then searches for all references to that particular feature and changes the name given by the user in a dialog window, exemplified in figure 5.3.


Rename

Rename Feature Name 'Blank ' and its usages to:

Background

Search in comments and strings

Scope

Project Files

Figure 5.3: Refactoring dialog in HAnS.

Special consideration is taken when the addition of the new name or the removal of the old name conflicts with other feature names. For example, in figure 5.3, if there already exists a feature called *Background*, then these features are no longer uniquely referenced by their names. HAnS handles this by recognising all affected features and updating all references to their new LPQ. This ensures that no links are broken due to refactoring.

5.5 Feature model view

In order to provide quick and convenient access to the feature model, there is an implementation of a feature model view, as seen in figure 5.4. This view is accessed as a tool window at the bottom left of the interface and is collapsible if needed. The tree structure visualised is a direct representation of the feature hierarchy in the feature model file.

By right-clicking on features in the tree the user is presented with a context menu with the options to rename, add or remove a feature, as figure 5.5 shows. These changes affect the feature model and as the view is a direct representation of the

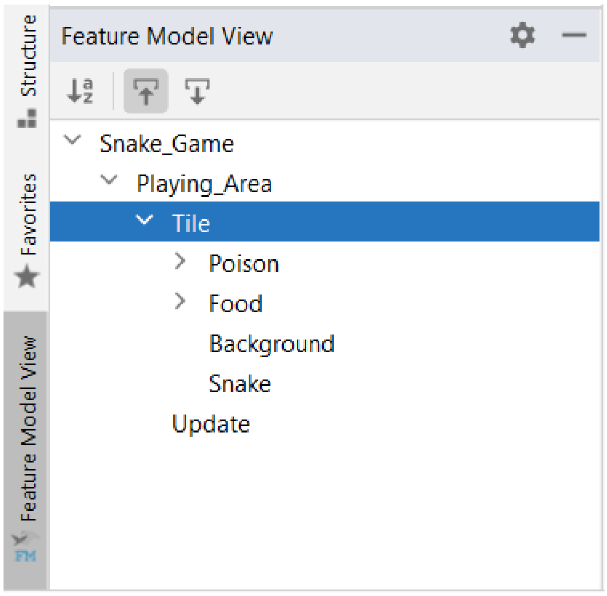


Figure 5.4: Feature Model View in HAnS.

feature model it is updated accordingly. The view also provides an option to "Find Usages" which displays all occurrences of the feature.

5.6 Quality of life features

The following features are referred to as quality-of-life features as they help with the smaller things that a programmer could want but are not crucial for functionality.

5.6.1 Syntax highlighting

All of the mentioned languages support syntax highlighting for the convenience of the user. The idea with syntax highlighting is to give the user quick visual information to help readability. The functionality for this is implemented with different colours for various element types as well as error markers for code that does not conform to the grammar. The syntax highlighting can be seen in the listings 5.1, 5.2, 5.3 and 5.4.

To be able to know whether a feature is defined in the feature model or not an additional annotator is implemented. The annotator looks into the feature model for the feature that the user wants to use in the annotations. If it is defined, the annotation is provided with a defined colour. However, if it is not, the annotation is coloured red as in figure 5.6. The file also shows the problems in the "Problems" tab in IntelliJ, as shown in figure 5.7.

An annotator was also implemented for the feature model, with the responsibility to make sure that defined features that are never used get coloured grey, as can be seen in figure 5.8. This also produces a weak warning, "Feature is never used", in

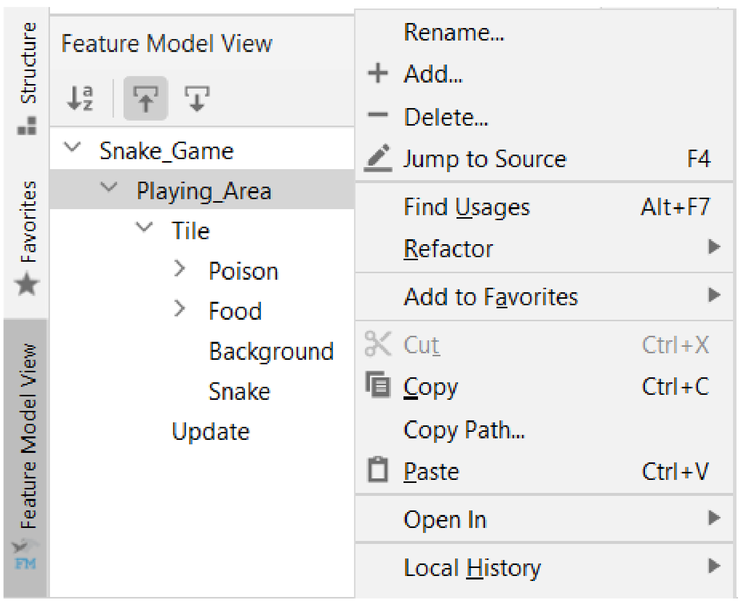
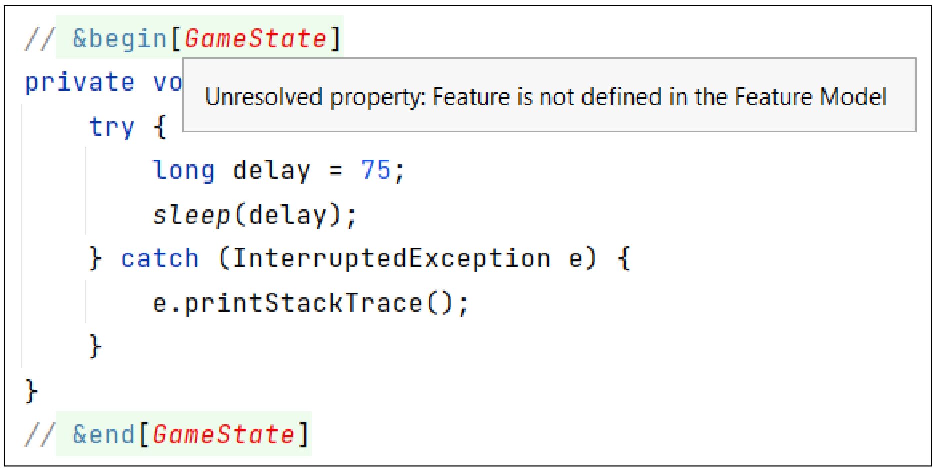


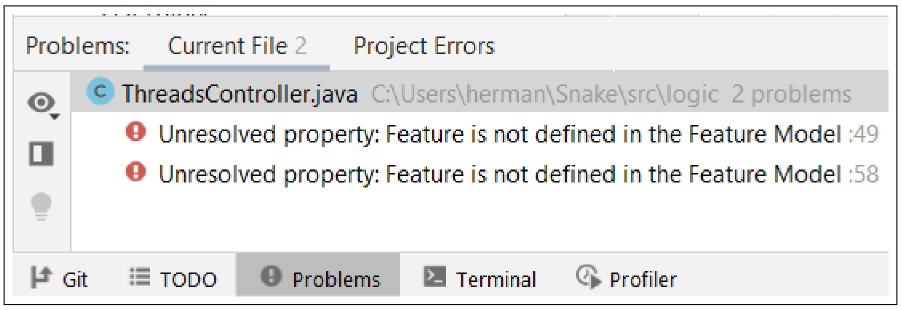
Figure 5.5: Feature Model View in HAnS.



```
// &begin[GameState]
private vo
    try {
        long delay = 75;
        sleep(delay);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
// &end[GameState]
```

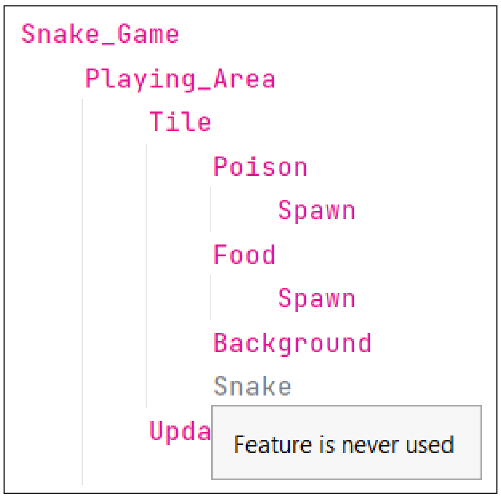
Unresolved property: Feature is not defined in the Feature Model

Figure 5.6: Feature not defined in the feature model.



```
ThreadsController.java C:\Users\herman\Snake\src\logic 2 problems
    Unresolved property: Feature is not defined in the Feature Model :49
    Unresolved property: Feature is not defined in the Feature Model :58
```

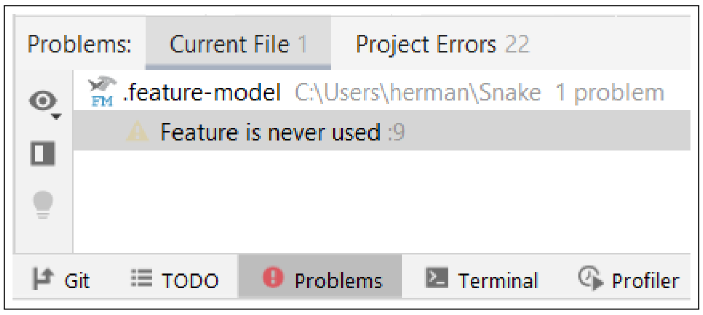
Figure 5.7: Problems for current file stating that features are not defined.



```
Snake_Game
    Playing_Area
        Tile
            Poison
                Spawn
            Food
                Spawn
            Background
            Snake
        Upda
```

Feature is never used

Figure 5.8: Feature not used.



.feature-model C:\Users\herman\Snake 1 problem

Feature is never used :9

Figure 5.9: Problems for the *.feature-model* file stating that features are not used.

the problems tab displayed in figure 5.7.

5.6.2 Live templates

Live templates are used to insert common constructs into code. They can be added manually using any JetBrains IDE. Live templates work similarly to code completion with the exception that live templates usually are acronyms for the construct they create. One example of a Live Template is typing *sout* in IntelliJ which resolves to *System.out.println*(|). To improve quality of life, HAnS has three templates shown in the listings below. The templates are activated by typing the start of a code annotation. Typing *&begin* will produce what is shown in listing 5.5 and typing *&line* and *&end* will produce the respective results in listing 5.6. The '|' character indicates where the text marker ends up after completing the live template.

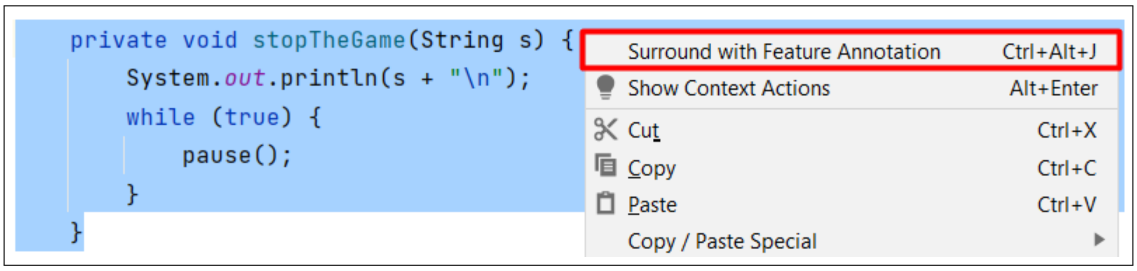
```
// &begin[|]

// &end[|]
```

Listing 5.5: Code produced in java with the &begin live template

```
// &line[|]
```

```
// &end[|]
```

Listing 5.6: Code produced in java with the &line or the &end live template



```
private void stopTheGame(String s) {
    System.out.println(s + "\n");
    while (true) {
        pause();
    }
}
```

Surround with Feature Annotation Ctrl+Alt+J
Show Context Actions Alt+Enter
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Copy / Paste Special

Figure 5.10: Surround with live template

To increase the recording efficiency for the developers even more, live templates can be made to support surrounding code with a template. The *&begin* template supports this kind of action when the programmer uses the standard command $ctrl + alt + J$. To use this action a programmer marks some code and press $ctrl + alt + J$ as displayed in figure 5.10.

5.6.3 New file

Creating new embedded feature annotation files is simplified by the existence of templates for the different file types. The templates are accessed as an option in the new file menu where other new files can be created from, like Java and Kotlin files, as shown in figure 5.11.

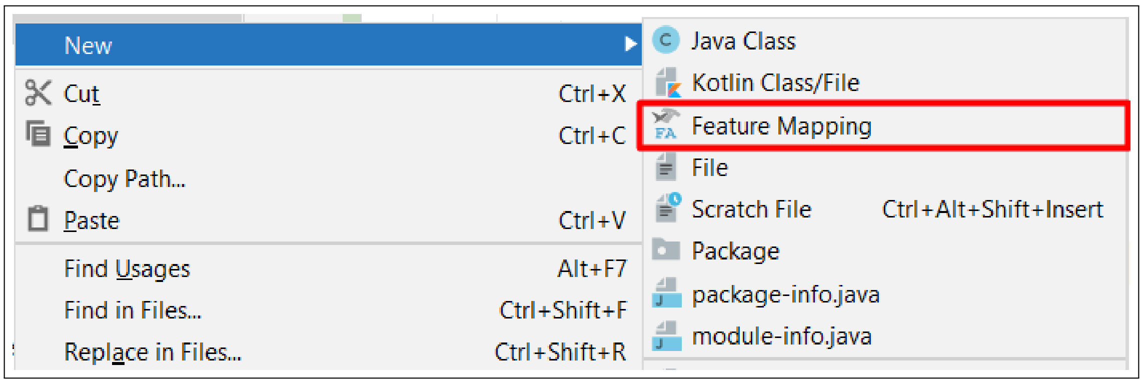


Figure 5.11: New file menu option

Selecting "Feature Mapping" in the menu presents the user with the dialog shown in figure 5.12. From this dialog the user can select which type of file to create and a suitable name.

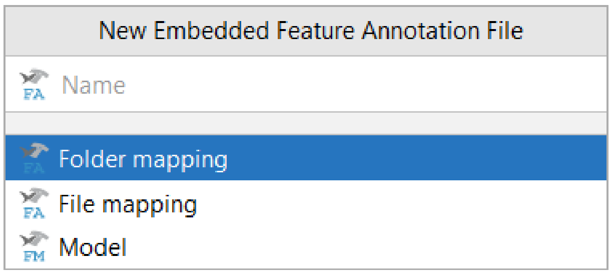


Figure 5.12: New Embedded Feature Annotation File dialog

6
Results

The results of the study that answer the research questions are presented here in three parts. The first one presents the achieved functionality of the plugin and the following two present the results from the user studies.

6.1 The achieved functionality of HAnS (RQ1)

To answer RQ1 (requirements for editing support), an initial version of HAnS was developed, which we used in a first user study to obtain feedback on more requirements. The final version of HAnS achieved the functionality described in section 4.3, with the exception of the quick fix feature. Some additional functionality described in section 4.4 has also been implemented. Additionally, some features that were not defined after the initial elicitation of requirements have been added. The functionality is listed in table 6.1 below.

Table 6.1: Achieved functionality in HAnS

Achieved functionality in the MVP
Code Completion
Syntax Highlighting
Refactoring
Feature Model View
Live Template
Warning of undefined features
Other achieved functionality
Feature Referencing
Surrounding Live Templates
Find Usages
Functionality that was not elicited
New file → A template so that adding the different feature mapping files can be done through the usual new file menu.

6.2 Evaluation of the first iteration (RQ1)

The first user study provided results that help answer RQ1: which functionality are required for a plugin to be well designed and; RQ1.1 what functionality could encourage developers to use the plugin. In the user study there were six participants who where tasked to (i) implement a new feature with and without the plugin; (ii) manually rename an existing feature, i.e. refactoring without the plugin; and (iii) answer the questionnaire after each set of tasks. HAnS supported earlier versions of the following three features; code completion, syntax highlighting (without annotators) and feature model view (only showed the feature model structure). The results from the questionnaire are presented as tables and quotes down below.

The participants where asked to rate the difficulty of tasks on a five degree scale; *easy* (1), *somewhat easy* (2), *neither easy or difficult* (3), *somewhat difficult* (4) and *difficult* (5). Every task also had a correlated question asking the participants to rate the perceived time on a five degree scale; *significantly less time than expected* (1), *less time than expected* (2), *as much time as expected* (3), *more time than expected* (4) and *significantly more time than expected* (5). The mean opinion of each group with and without the plugin is calculated by correlating an opinion to a number and finding the mean value and then translating the number back to an opinion.

6.2.1 Preamble results to define required functionality

The task of implementing a new feature entailed locating a previously implemented feature and in table 6.2 the mean opinion of the different groups with and without the plugin can be seen for the location part of that task. The table also presents the results of the participants' estimated time for locating a feature. As can be seen, both groups found locating features easier with the plugin enabled, however the differences are minor. This can be explained by also looking at the results regarding the time estimation where both groups found that using the plugin took them longer than they expected.

The participants accredited the size of the project and the annotations to be the reasons for the close ratings with the plugin enabled or disabled. The primary reason for the plugin edging in front in regards to difficulty was the syntax highlighting, as supported by the following quote.

> *"I got used to the files in the project but I missed the highlighting of the annotations."*

The time estimation is similar in closeness and the participants stated that the functionality that was needed to make the plugin useful was some sort of overview and interface of the feature locations. The following two quotes support this.

> *"Going through files to find the necessary features was much easier than usual, but something like a feature explorer where you can click through*

Table 6.2: Location of features and the estimated time it took to locate.

Plugin	**Difficulty locating of features**		**Estimated time of location**	
	Score	Mean opinion	Score	Mean opinion
Enabled *Group*$_1$	2	Somewhat Easy	3.67	A little more time than expected
Enabled *Group*$_2$	3.33	Neither Easy or Difficult	3.67	A little more time than expected
Disabled *Group*$_1$	2.33	Somewhat Easy	3.33	As much time as expected
Disabled *Group*$_2$	3.67	Somewhat Difficult	4.33	A little more time than expected

every occurrence of a feature would be nice."

"To minimise the time spent locating features some overview or find all occurrences should be in the tool"

The implementation of a new feature also entailed defining a new feature in the feature model and mapping it to where it was implemented. For the defining task the participants had similar opinions which can be seen in table 6.3 as well as their perception regarding time. For the mapping of a feature the participants also had similar opinions. As seen in table 6.4 the scores vary a bit for the difficulty, while the results for the time estimation are consistent between the groups.

The results indicate that group 1 experienced some differences regarding defining a new feature and mapping it with the plugin compared to without. However, group 2 saw no substantial differences. The estimated time and the feature that was most important in lowering the score for defining a new feature was the Feature Model View, which is supported by the following quote.

"I had a moment where I didn't know how to define a feature as a child without the feature model view"

The estimated time for mapping of a feature can be credited to the code completion functionality. Even though, it was not something that the participants pointed out themselves it was clear, from the observations, that most of them used the feature. However, some participants typed code so quickly that the code completion did not show up before the line was finished.

Table 6.3: Defining a feature and the estimated time it took to define it.

Plugin	**Difficulty defining a feature**		**Estimated time of defining**	
	Score	Mean opinion	Score	Mean opinion
Enabled *Group*$_1$	1.33	Easy	1.33	Significantly less time than expected
Enabled *Group*$_2$	2.33	Somewhat Easy	2.67	As much time as expected
Disabled *Group*$_1$	2.33	Somewhat Easy	2.33	A little less time than expected
Disabled *Group*$_2$	2	Somewhat Easy	3.33	As much time as expected

Table 6.4: Mapping a feature and the estimated time it took to map it.

Plugin	**Difficulty mapping a feature**		**Estimated time of mapping**	
	Score	Mean opinion	Score	Mean opinion
Enabled *Group*$_1$	1.22	Easy	2.22	Significantly less time than expected
Enabled *Group*$_2$	2.67	Neither Easy or Difficult	2.33	A little less time than expected
Disabled *Group*$_1$	2.33	Somewhat Easy	2.67	As much time as expected
Disabled *Group*$_2$	1.44	Easy	3	As much time as expected

6.2.2 Refactoring

To rate the refactoring task two open ended questions were posed to the participants; "How did you perform the refactoring?" and "What alternative method would you expect from a well designed plugin?". The participants used the find-and-replace method in IntelliJ to perform the task and stated that they would expect the refactoring functionality that comes with IntelliJ to work for features as well. The time estimation for both groups was "As much time as expected" with an average score of 3 and $3\frac{1}{3}$. Since there was no refactoring functionality implemented at this stage this score is unexpected, it is expected to be much higher. However, the participants once again accredited the size of the project and syntax highlighting for the lower score.

6.2.3 Feature suggestions

At the end of the survey the participants provided some additional feature requests in order for the tool to be useful. They stated that it was crucial for referencing and refactoring to be implemented, as supported by the following two quotes.

"It would be nice to have more functionality like finding usages in the feature model view."

"Refactoring was very tedious without support for the built in refactoring in IntelliJ."

The Feature Model View, which was perceived as a positive feature, required improvements according to participants, as stated in the following quote.

"It was good as an overview but giving the window more functionality can make it even more useful."

The code completion was perceived well as described in section 6.2.1. However, in order to improve the functionality it was suggested, as seen in the quote below, that live templates should be implemented.

"I would have liked to see some functionality were I can mark some code and the surround it with a begin and end tag."

In summary, the participants for the first user study found that locating features with the plugin was neither easier nor more difficult than doing it without. They did, however, find it to be a time consuming task. The users found defining and mapping a feature to be marginally easier with the plugin because of the code completion feature. Since we had not yet implemented functionality to support refactoring, participants found it to be the biggest challenge, together with locating features. They recommended that HAnS should be extended with referencing (finding usages), refactoring, an updated version of the feature model view and enabling live templates in order to improve the code completion. Hence the tool was improved for the second iteration to address this feedback.

6.3 Evaluation of the second iteration (RQ2)

The second user study provides results for RQ2.1: what the are the costs of annotating with the plugin and; RQ2.2 what their qualitative experience were. The first two sections are mostly concerned with RQ2.2. The third and fourth have a split focus on both RQ2.1 and RQ2.2, as they contain both qualitative results about the usability and quantitative results about the costs of recording feature locations. The last section provides results on the participants qualitative experiences of the notation that HAnS uses.

The 13 participants in the study were, similarly to the first user study tasked to (i) implement a new feature with and without the plugin; (ii) manually rename an existing feature, i.e. refactoring without the plugin but; (iii) answer the questionnaire after both task-sets. The tasks were for this iteration performed on the final version of HAnS, which included all the achieved functionality described in section 6.1 and

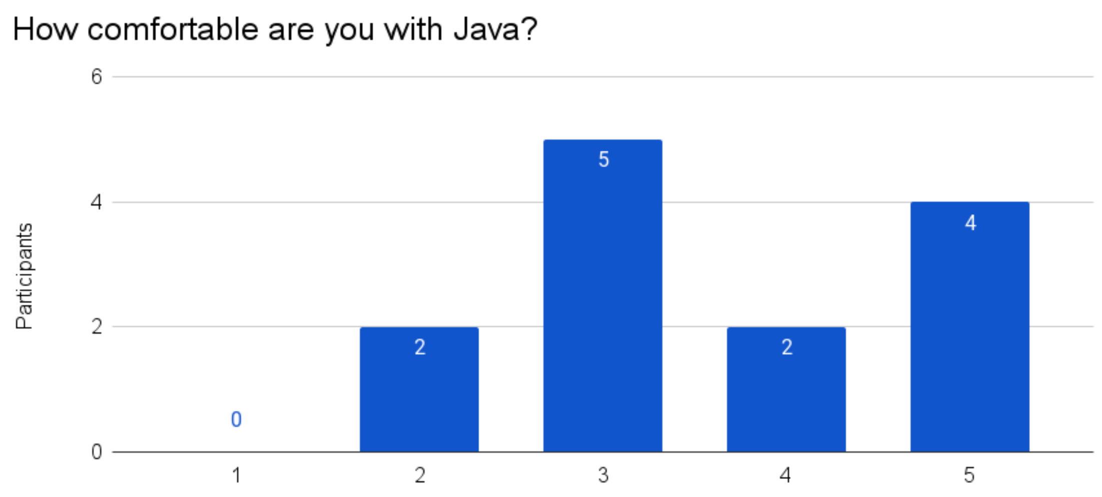


Figure 6.1: How comfortable are you with Java?

chapter 5. The results from the questionnaire are presented as figures and tables.

The participants in the study have Java experience from either education, profession or both. An overview of how comfortable the participants were with Java can be seen in figure 6.1. The scale ranged from “Not at all” to “Very” across five degrees. They have answered that they at least have some comfort and experience with Java, since the lowest number means that the participant would not be comfortable at all. The participants from the first user study indicated that the EFA notation required a more thorough explanation, therefore the oral introduction was refined.

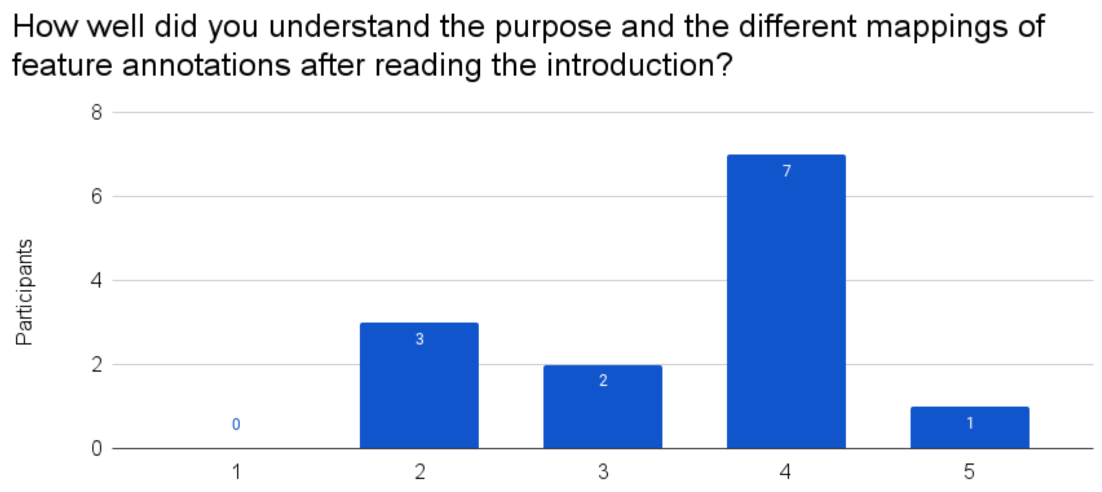


Figure 6.2: How well did you understand the purpose and the different mappings of feature annotations after reading the introduction?

After completing the introduction the participants were asked about how well they thought they understood the annotations and their purpose, 6.2. This scale was ranging from “Not at all” to “Very” confident. As with the question about Java the results shows at least some understanding regarding feature annotations. In figure 6.2 it is also evident that most participants understood the annotations well. This is an important result as too low scores in understanding the notation can introduce bias against the plugin.

6.3.1 Perceived difficulty implementing a new feature

In order to find what the qualitative experiences (RQ2.2) were, regarding the perceived effort of locating and mapping features with HAnS, the participants were tasked with two implementation assignments, one with the plugin and one without. Both assignments included hints of a feature to locate in order to help with implementation. Figure 6.3 shows how the participants felt about locating the two different features. The expected result would be that the feature that they locate with the help of the plugin is easier to find. Independent of whether the plugin was used or not, the results show that they are leaning more towards disagreeing with the statement.

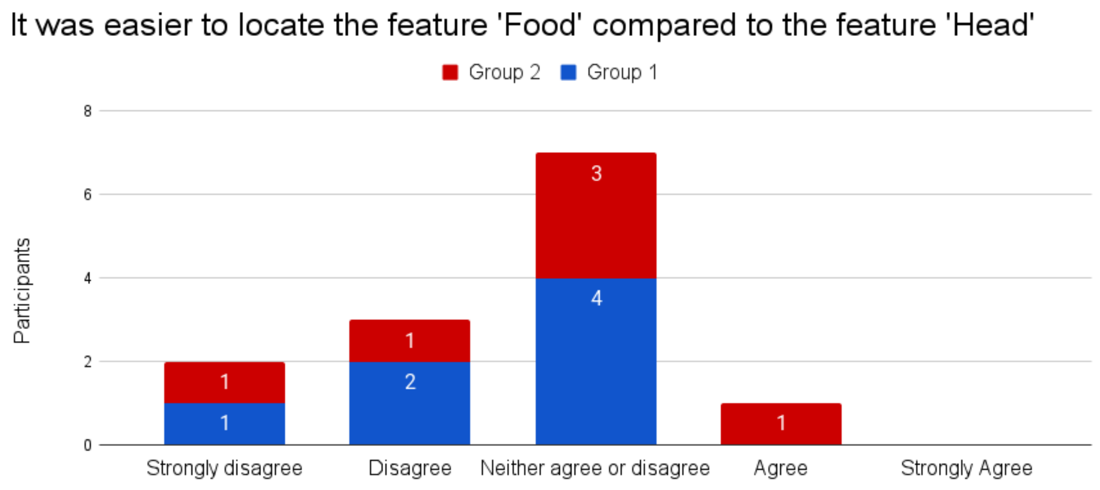


Figure 6.3: Group diagram showing the uniformity between groups

The participants motivated their answers in the quotes below. They concern both the scattering of the features and the intuition of where a feature would be as well as not finding it any easier or harder for locating one or the other. Participants from group one had the plugin enabled when locating the *food* feature while participants in group two had the plugin enabled while locating the *head* feature.

> *“The location of the Head feature made intuitive sense, so finding it was not hard. The Food feature could have been placed in different parts of the program structure which made it less obvious where to look.”* - Participant from group one.

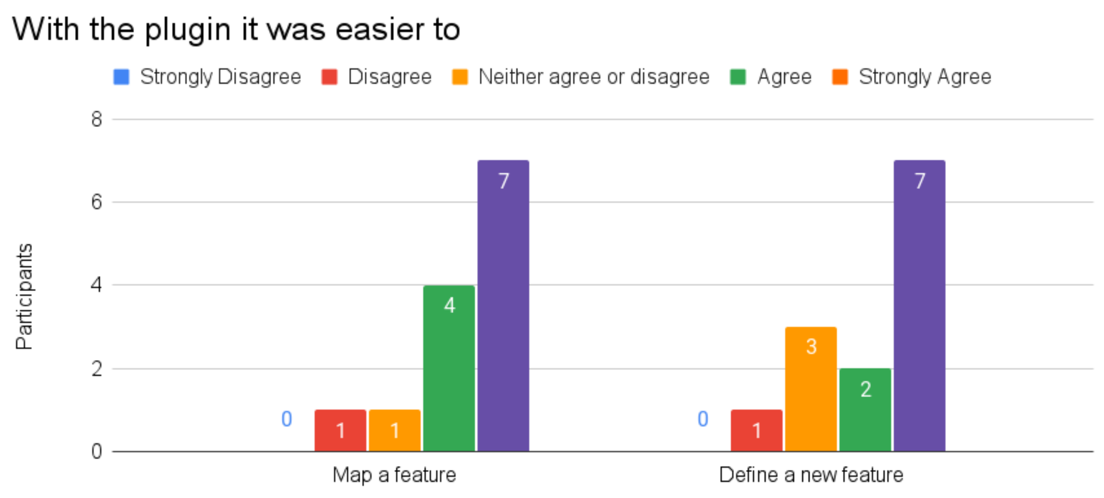


Figure 6.4: Rating of the difficulty of creating annotations

> *"The food feature was more spread out compared to the head feature. Because the food feature also had more details than the head feature, it was harder both to locate it and to understand it."* - Participant from group two.

> *"It was just as easy to find Head as it was to find Food."* - Participant from group two.

> *"I used the concept of the notation manually, which helped me both with the plugin activated and not activated."* - Participant from group two.

To find whether the implementation of HAnS helps with adding new annotations the participants were asked to rate the difficulty of mapping a feature and defining a new feature with the plugin, figure 6.4. The figure shows that the participants were quite unified in their answers where the majority either agreed or strongly agreed with the statement.

The motivations that the participants gave for their scores can be summarised by the quotes below. They explain why they find it easy and give some indication of what features of HAnS they use to make mapping and defining easier. One participant disagreed with the plugin making mapping and defining easier and stated that they where so unused to auto-completion and that it trips them up.

> *"If I were in a setting where the tool was being used, I would have a much easier time to locate a good place to create the feature in the code space. In the case of a brand new unrelated feature there would still be a certain discomfort in knowing where exactly in the code space to place said feature, but since it makes it so easy to map and find the feature, it would be easy to remedy at a later stage."*

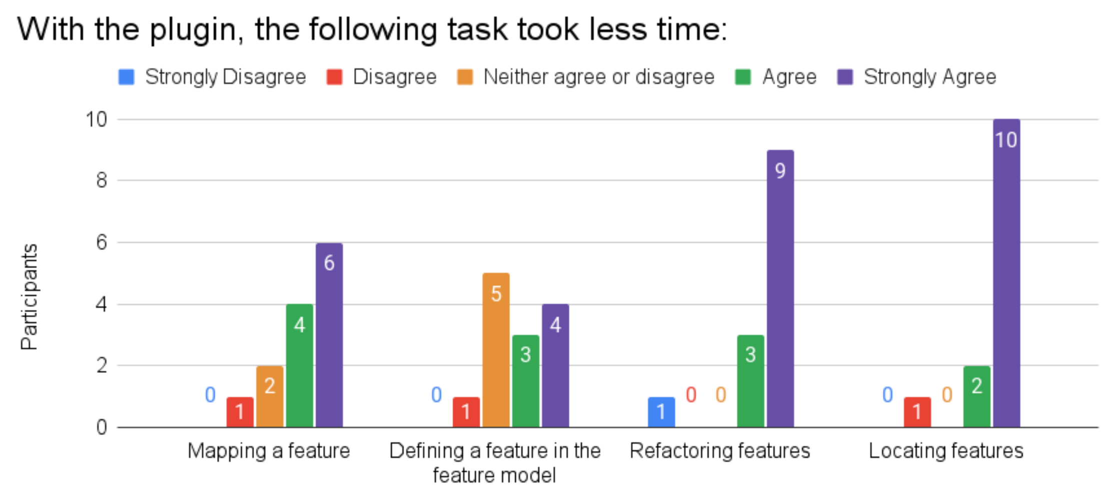


Figure 6.5: Perceived time for tasks utilising the plugin compared to without the plugin.

> *"It was easier to map the feature with the use of plugin since I could highlight a block of code and map a specific feature to the code with just 2 clicks."*
>
> *"Syntax-highlighting, and referencing makes the world go around."*

6.3.2 Perceived time effort

In order to assess the perceived time it took to perform different tasks the participants were asked to rate to what extent they agreed the plugin shortened the duration. The tasks to compare were to map, define, refactor and locate a feature and the results are shown in figure 6.5. These activities constitute the necessities for using the annotation system and common practises in industry, as described in section 4.2.1. Mapping and defining a feature were perceived to take less time with the plugin with the exception of one participant who disagreed and two participants who were indifferent. None of these participants motivated their answer to these statements.

Refactoring and locating of features were well received with all except one of the participants agreeing that the tasks took less time with the plugin. The disagreement was motivated with a comment stating that the refactoring was slow and searched too many files and that locating features was easier with the search function in IntelliJ than with the plugin. Another participant noted that it "*was much easier finding and locating the features with the plugin which made it much easier to do the tasks compared to when the plugin was turned off.*" The same participant added that "*for this plugin to be efficient you have to learn the syntax and also remember to use it since you will have to write some more lines of code than usual*". Another participant noted the following as a justification for giving high scores:

"The feature tree was an extremely nifty tool to locate, map, refactor and define a feature. As I started with the HAnS tool enabled, I quickly became dependant on it, so when I had to disable the tool, I felt completely lost, rendering all actions that had to be done in the task much harder."

In order to obtain results to answer RQ2.1 we analysed the video recordings from participants with focus on their refactoring. The motivation for this was that incorrect refactoring can lead to additional cost of required correction of feature references that are left unchanged. Without HAnS, participants used a variety of techniques from manually searching through files to using IntelliJ's infrastructure to find and replace strings. These methods took a long time and involve a risk that not all references are correctly changed. With HAnS, all participants utilised the plugin, however, not everyone used the refactoring functionality. Some participants used the plugin to find usages of a feature and subsequently changing these. Everyone that used HAnS did refactor the features correctly.

In the same manner, we attempted to measure the time it took for the participants to record feature locations of various granularities. The recordings show that there is a high variability in the approaches of writing annotations. Some participants copied a similar annotation or utilised live templates while others needed to double-check the syntax while annotating. This resulted in a wide range of time required to add an annotation, from a couple of seconds to over two minutes. This is true for all different mappings (code, file and folder) and the definition of a new feature in the feature model. Therefore it was also difficult to differentiate between time duration of using HAnS compared to without HAnS.

6.3.3 Usability of the system

The Feature Model View was created to provide an easy overview of features and as an interface for the functionality for creating and editing features. In an effort to evaluate the Feature Model View as its own component the participants were asked to rate their agreement with a selection of statements from the System Usability Scale. Since some statements were excluded, these ratings were adjusted accordingly. The results from this evaluation can be seen in table 6.5. The scale ranges from 0 to 100 with the latter representing a perfect score. Some statements are negative, that is, they express a dislike. The scoring of positive statements range from zero to four and negative statements are reversed four to zero, which is described in section 4.2.4. This way, a higher score indicates better usability independent of whether the statement is positive or negative. Bangor et al. (2008) report that scores above 70 are passable and below 70 are candidates for increased scrutiny and improvements. Scores below 50 are cause for significant concern. The resulting scores of the Feature Model View show a value of 75.55 in total. The statement "I felt very confident using the 'Feature Model View"' received the lowest of the scores at 63.46.

Table 6.6 shows the resulting scores from the System Usability Scale evaluated on the entire plugin. The score for each statement is presented followed by the total

Table 6.5: System Usability Scale - Feature Model View

Statement	SUS-score
I would like to use the ‘Feature Model View’ frequently.	71.15
I found the ‘Feature Model View’ unnecessarily complex.	80.77
The ‘Feature Model View’ is easy to use.	78.85
The various functions in the ’Feature Model View’ are well integrated	73.08
I imagine that most developers would learn to use the ‘Feature Model View’ very quickly	86.54
The ‘Feature Model View’ is cumbersome to use.	75
I feel very confident using the ‘Feature Model View’	63.46
	75.55

Table 6.6: System Usability Scale - HAnS

Statement	SUS-score
I would like to use this plugin frequently.	75.00
I found the plugin unnecessarily complex.	82.69
The plugin was easy to use.	78.85
I would need the support of a specialist to use this plugin.	82.69
The various functions in this plugin are well integrated.	73.08
There is too much inconsistency in this plugin.	71.15
I imagine that most developers would learn to use this plugin very quickly.	90.38
The plugin is very cumbersome to use.	78.85
I felt very confident using the plugin.	67.31
I needed to learn a lot of things before I could get going with this plugin.	75.00
	77.50

score at the bottom of the table. All statements received a score of more than 68 except for the statement “I felt very confident using the plugin” which got a score of 67.31.

Assessing HAnS’s effectiveness through the annotation mistakes it alleviates

In order to find the answer for RQ2.1 the number of errors (ϵ) that each participant (ρ) made were calculated from the videos that the participants recorded. Four different errors were identified: (i) LPQ (Least-Partially-Qualified name) - The participant forgot to define LPQs where it should have been defined or defined an LPQ where there was no need for one; (ii) Syntax - the participant did not follow the correct syntax for the EFA; (iii) Definition - the participant forgot to define a feature before using it and; (iv) Spelling - the participant misspelled a features name. The severity of the errors are graded depending on the eventual effort it would take a developer to correct the error. If one makes a syntax, LPQ or spelling error the effort will be high as looking for the locations where a mistake has been made will be very time consuming and these errors are therefore rated with high severity. In comparison, forgetting to define the feature before using it can have technical debt issues but may be resolved quickly by defining it in the feature-model file. There can still be a large effort if the feature that was not defined is unique and requires LPQs, as it is not likely that a user would remember to define an LPQ if the plugin doesn't help the developer with this.

The total errors are calculated for eleven participants (two of the 13 participants' recordings was corrupted), six in group 1 and five in group 2. The errors per participant is calculated with formula 6.1.

$$E_{\phi} = \frac{\epsilon}{\Sigma\rho} \tag{6.1}$$

This metric help to formulate for example two out of three participants make an LPQ error with the plugin or every participant makes two errors and one out of five makes three errors without the plugin.

As can be seen in table 6.7 when starting with the EFA notation the participants made errors independent of whether they had the plugin enabled or not. Removing the plugin from group 1 did not effect them very much, they still made errors, however, giving the plugin to group 2 after they started without the plugin lowered the amount of errors. The amount of errors with LPQs for task-set two is expected to be lower as there is no requirement to define any references with the help of LPQs in that task-set. However, comparing the groups on the same task-set shows that there is a difference regarding using the plugin and not. The results also show that the syntax is harder without the plugin, which is expected as no syntax-highlighting makes it harder to see whether the syntax is correct or not. Defining features before using them is something that many did but rectified immediately when they noticed that the syntax-highlighting showed them that the feature was not defined. However, with the plugin disabled the mistake did not get rectified.

6.3.4 Perception of the EFA notation with HAnS

In order to be able to discuss the qualitative experiences of HAnS, and answer RQ2.2, the resulting usability of the EFA notation is required. This is not to evaluate the syntax in itself but rather to give an indication of how the introduction of the annotation system may have affected the participants. To evaluate the usefulness and usability of the EFA notation with HAnS the question in the survey was posed similarly to how the SUS questions were posed. Because of the similarities the score was calculated in the same way as with SUS. In table 6.8 the scores for each individual statement is presented with the corresponding scores.

The biggest challenges with the EFA notation seen by the participants were regarding the syntax, how it is hard to know when to annotate and where as well as that annotations do not give enough information. This is explained in the quotes below.

Table 6.7: Number of errors (ϵ) divided per group with the plugin enabled compared to disabled.

			Group 1		Group 2	
		Participants	6		5	
Severity	Type of ϵ		Enabled Task-set 1	Disabled Task-set 2	Disabled Task-set 1	Enabled Task-set 2
High	LPQ	ϵ	4	1	5	0
		E_ϕ	0.67	0.17	1.00	0.00
High	Syntax	ϵ	1	4	2	1
		E_ϕ	0.17	0.67	0.40	0.20
Medium	Definition	ϵ	4	2	4	0
		E_ϕ	0.67	0.33	0.80	0.00
High	Spelling	ϵ	0	0	0	0
		E_ϕ	0.00	0.00	0.00	0.00
		Total ϵ	9	7	11	1
		E_ϕ	1.50	1.17	2.20	0.20

Table 6.8: Perception of the EFA notation

Statement	**SUS-Score**
Our motivation for using embedded feature annotations was convincing.	78.85
I see a clear benefit of using embedded feature annotations.	86.54
The syntax of annotations (in-file annotations, file/folder mapping, feature model) is easy to understand.	88.46
The effort for adding feature annotations is low during development.	75.00
It is clear to me when to annotate code with a feature.	69.23
It is easy to reason about the different granularities of features (features and sub-features in the feature model).	69.23
I was confident I annotated the right code parts with the right features.	76.92
I will use embedded feature annotations in the future.	71.15
	76.92

"Learning whether an entire file, a block of code or a single row should be annotated. It takes some time to get an intuitive feeling for that. Also, it is hard to know at what level of granularity a feature should be annotated."

"The biggest challenge was to remember the syntax from the README. If I would remember to select and use live templates to add annotations, it would have been much easier.

"The code-bloat. Usually I would be all for commenting code. If the annotations could serve that purpose as well as be used for documentation."

7

Discussion

This chapter serves to discuss the results and the methodology of the project. The discussion relates back to the research questions and the purpose of the study.

7.1 Functionality of HAnS

The first step in evaluating the plugin was establishing its required functionality, this is encompassed by RQ1. The idea was to determine the functionality required by users in order for them to use HAnS. Early elicitation resulted in a list of functionalities that are crucial for the plugin to work. These constitute the foundation of the plugin, such as IDE support for the EFA syntax, highlighting of said syntax and code completion. This makes up some sort of base line for custom language support. The outcome is essentially IDE support for the notation compared to writing annotations as text files and comments without any recognition. Although many of the participants expressed appreciation of this language support, they wanted additional functionality.

A common theme among the participants was a desire to be able to utilise the existence of feature annotations, as they see it as a clear benefit in order to maintain feature locations indicated by the high SUS-score in table 6.8. This included both refactoring, for convenient editing, and referencing, for immediate benefits. Both of these were added to HAnS and received high approval, indicating that it is indeed instrumental for the acceptance of the plugin.

The main reason for implementing HAnS into an IDE was to bring the editing closer to the developer and make it as non-intrusive as possible, this is something we hope to realise with the Feature Model View. The Feature Model View extends the previously mentioned functionality by providing a convenient interface and a hierarchical overview of the features. This was much appreciated by the participants which is apparent from the high SUS score and textual feedback.

In addition to the mentioned functionality, that was requested by many, there were additional things that were requested by a few. This relates to the purpose of the study which in part is to provide functionality that can be expected from such a tool. Even if not actively used by everyone there are things that users may assume are present in a certain software. By including these functionalities, we hope to encourage users to use the plugin. Among these were the addition of live templates and the option to create files for feature annotations.

Of course, when answering RQ1.1, a line must be drawn for what is essential since all added functionalities may be regarded as improvements to the software. Judging by the results we draw the conclusion that language support paired with referencing, refactoring and the Feature Model View constitute what is necessary for effective editing support while the remaining functionality can be regarded as enhancements for personal preference that may encourage specific individuals.

7.2 The effectiveness of HAnS

After having addressed RQ1 about functionality of HAnS we proceed to discuss its effectiveness i.e., RQ2. This research question is divided into both the costs of recording and editing annotations, and the qualitative experience of the participants.

In general, the usability of the plugin was well received as shown by the SUS scores. The plugin received a score of 77.50 which is considered to be in the higher spectrum of passable scores. The lowest score was related to the participants’ confidence in using the plugin. This is not surprising given the fact the participants had limited time to familiarise themselves with HAnS.

There is not much to evaluate, regarding the basic functionality of the plugin since it is more of a binary state where it either is implemented or not. The EFA syntax, implemented as languages, is indeed supported and syntax highlighting, and code completion were elicited requirements that were discussed early to be crucial. Most participants expressed that this functionality was expected and as such it did not significantly affect their opinion. Interestingly, one participant was confused by the code completion but stated that it was due to inexperience. A choice could be made to require manual invocation of code completion, but this can be disregarded since most participants were pleased with the behaviour.

Referencing was much appreciated as well as refactoring and the Feature Model View. Participants reported that navigating the code and finding features required less effort when using the supported functionality compared to manually locating feature annotations. Most participants also noted that refactoring was convenient and removed the cognitive load that manual refactoring has, as well as ensuring that no references are left unchecked. The major concern about the plugin, however, is its performance. Even though refactoring is automatic it takes a considerable amount of time and it is therefore in need of optimisation.

Another aspect to discuss is special consideration that the participants did not foresee. This concerns renaming, adding and removing features which can alter the hierarchical structure of features and thus change their LPQs. For example, renaming a feature to the same name as another feature implies that all related references must be updated with their new LPQ. These cases were difficult to actively include in the user studies but may have impacted responses from participants. Our reasoning is that using HAnS in practice would prove to require less effort since these considerations would be handled by the plugin and not the user.

In regards to RQ2.1, we collected data related to additional cost as a result of errors made when recording and editing features. The results from the refactoring tasks indicate that there is a greater risk of errors when performing manual refactoring compared to using HAnS' support for this. We also present results of errors made in both study groups during implementation. Both groups made more errors during the first task-set, probably due to inexperience in using EFA. Interestingly, participants in group 1 continued to make errors after disabling the plugin while group 2 reduced the errors from eleven to only one. Based on these results, we claim that HAnS reduces the need for future corrections and answer RQ2.1 by stating that the cost of annotations is lower when using HAnS than without using it. We want to complement this by emphasising that costs still exist but that they are small and mostly related to using the EFA notation and not HAnS.

When analysing recordings it proved difficult to measure the time required to record various annotations. The multitude of approaches and individual experience resulted in a wide range of results. Since the concept of feature annotations were new to the participants, we could also see that the limited experience gained during the first task-set affected the time that recording a feature's location took. Therefore, we cannot draw any conclusions regarding the efficiency of HAnS in relation to time.

Many participants noted difficulty in applying the syntax of the annotations. A couple of people also stated that they would not use feature annotations for smaller projects. This was expected since there is not much need for annotations in projects that are already easily navigated. We do not consider these issues to be evaluations of the plugin but rather evaluations of the use of feature annotations. Since we in this report aim to evaluate editing support we do not focus on the usefulness of annotations. However, it is interesting to include these results in order to be able to reason about the results concerning the plugin.

7.3 Threats to validity

External validity. This study was evaluated based on a tool for embedded annotations in the IntelliJ IDE and the findings are therefore limited to software development that uses IntelliJ. IntelliJ is a software by JetBrains which has similar IDEs for a multitude of programming languages. HAnS was tested briefly in JetBrains' Python IDE, PyCharm, and worked as expected, indicating that the plugin may work in more software of the JetBrains environment.

The example system used for the user study is quite small and only contains 285 lines of code and 15 features. This means that location of features may be just as easy with the plugin as without. However, results indicate that that was not the case despite the systems size. The size also influenced the set of tasks that were constructed for the study as they had to be similar but still be different enough to not introduce learning bias. It is worth noting that some learning bias will of course happen as the system is still the same. The choice of the system was therefore based on if it included two features with similar complexity, as this was key for the participants to be able to compare using the plugin or not which was a central part of the design. It was also based on shortening the time it took to implement some feature, as a study that takes over 90 minutes to complete would be very hard to find participants for.

The results of the study are also limited regarding what claims of the tools usefulness that can be made, as it only investigates the users qualitative experience of a system that was already annotated. Letting users participate in the study without prerecorded locations for features would have increased the time a study would take, which as stated above would not be beneficial.

Internal validity. During the study the authors have recorded feature annotations for the developed software. This has been to gather experience and insights to guide development. This may lead to some features of the plugin being biased. However, to mitigate this we have had discussions with other developers who are familiar with manual feature annotations.

Another aspect that might impact the results is the brevity of the user studies. Both the concept of embedded feature annotations and HAnS were new to most participants and therefore there was a need for a thorough introduction. After the introduction the participants needed some time to use the plugin in order to later perform the evaluation. These steps limited the amount of time that the participants had to use the plugin and as such they might not have understood it, or its usefulness, fully. This could skew the results about the usability of the tool.

Construct validity. There might be a risk that the participants in the user studies did not understand the questions asked or simply misinterpreted questions. In order to mitigate this risk, a pilot study was performed to verify the design of the tasks and questions. Also, we verified if the participants understood the study by explicitly asking them. In addition to this, we were available for any clarifications needed during the study, as mentioned in section 4.2.1.

7.4 Future work

The outcome of this project has shown that there are many ways to continue the work, both as development of the plugin but also in continued research. The data gathered from the participants indicate several improvement areas that are described below. Also, we present an idea for further research while simultaneously hoping that this project may raise the topic for other research.

7.4.1 Longitudinal study

The project we have performed was directed towards producing the tool for editing support and to evaluate in accordance to the scope of the thesis. We do see an opportunity to evaluate embedded feature annotations and the tool in a longitudinal study. This could provide insights into how the system is used in practice and how it is received after a longer time of usage.

One way to help create a longitudinal study is to add logging to HAnS that, either, get uploaded automatically to a server or uploaded manually by participants. These logs could contain data of what features the developers are using frequently, the size of the project and response times for functionality like refactoring, finding references and usages.

7.4.2 Additional functionality of HAnS

During the development of HAnS we have come up with some additional functionality that would improve the plugin. In the following sections the most important ideas are described.

Matching tags

An alteration of the IDEs interpretation of the feature to code mappings can be considered, to enable a begin tag to produce an error if there is no matching end tag. This also implies that when changing the feature name in the begin tag the corresponding end tag should change correspondingly.

Visualisations and metrics

Adding visualisations to provide graphical information about features, like FeatureDashboard (2.2.4) does, could help with the location of features. The visualisations would be based on metrics that also can be shown as a table in some sort of view in the IDE.

7.5 Migration to other IDEs

Migrating the plugin to other IDEs is an idea to, help developers with different preferences, have a tool that works with editing support. This could either be realised by making plugins for the major IDEs like Eclipse, Visual Studio and Visual Studio code or by creating a LSP (Langserver)[1] that could be used in several IDEs.

[1]`https://langserver.org/`

8

Conclusion

One of the most common and widespread activities among software developers is locating features. However, feature location is a costly activity that is challenging and takes considerable effort to perform, as records of a features' explicit location are rarely kept. Therefore, existing techniques rely on retroactively locating a feature long after the implementation, resulting in high effort and low accuracy. However, effective feature locations must be recorded during development (Ji et al., 2015).

This study proposes a tool, Help Annotating Software (HAnS), which aims to encourage developers to record feature locations early during development, by providing them with editing support through an IDE. In order to be as non-intrusive as possible, the tool was integrated in to IntelliJ, one of the most popular Integrated Development Environments (IDE). HAnS provides functionality that helps developers with editing and locating feature references. We conducted user studies with a total of 19 participants to evaluate the usability and effectiveness of HAnS. The studies found that linking of references to a feature is especially important as it provide tangible value for developers. This creates a way for them to navigate through the code with the help of these feature references. Referencing is also the base for two other important functionalities for minimising costs: (i) Code Completion and; (ii) Refactoring. HAnS was found to be effective at minimising the technical debt of feature annotations and providing functionality that make the recording of feature locations easier and less time consuming to perform. However, it was only evaluated in short user studies where the participants had limited time with the plugin and the EFA notation. One reported drawback of the tool is its performance when refactoring larger projects.

We hope that this study is able to be ground for further research on feature location, e.g., understanding how developers reason about features during development. HAnS still needs further development, and more studies need to be conducted to understand how it would be used in practice. We propose a longitudinal study to acquire data from practitioners while developing code. For future development, we suggest complementing HAnS with visualisations and metrics of features as well as matching tags for block annotations.

References

Andam, B., Burger, A., Berger, T., & Chaudron, M. R. (2017). Florida: Feature location dashboard for extracting and visualizing feature traces. In *Proceedings of the eleventh international workshop on variability modelling of software-intensive systems* (pp. 100–107).

Apel, S., Batory, D., Kästner, C., & Saake, G. (2013). Software product lines. In *Feature-oriented software product lines* (pp. 3–15). Springer.

Bangor, A., Kortum, P. T., & Miller, J. T. (2008). An empirical evaluation of the system usability scale. *Intl. Journal of Human–Computer Interaction*, *24*(6), 574–594.

Biggerstaff, T. J., Mitbander, B. G., & Webster, D. (1993). The concept assignment problem in program understanding. In *[1993] proceedings working conference on reverse engineering* (pp. 27–43).

Brooke, J. (1996). Sus: a "quick and dirty'usability. *Usability evaluation in industry*, *189*.

Entekhabi, S., Solback, A., Steghöfer, J.-P., & Berger, T. (2019). Visualization of feature locations with the tool featuredashboard. In *Proceedings of the 23rd international systems and software product line conference-volume b* (pp. 1–4).

Ji, W., Berger, T., Antkiewicz, M., & Czarnecki, K. (2015). Maintaining feature traceability with embedded annotations. In *Proceedings of the 19th international conference on software product line* (pp. 61–70).

Kästner, C., Dreiling, A., & Ostermann, K. (2011). Variability mining with leadt. *Tec. Rep., Philipps Univ. Marburg*.

Kastner, C., Thum, T., Saake, G., Feigenspan, J., Leich, T., Wielgorz, F., & Apel, S. (2009). Featureide: A tool framework for feature-oriented software development. In *2009 ieee 31st international conference on software engineering* (pp. 611–614).

Knauss, E. (2021). Constructive master's thesis work in industry: Guidelines for applying design science research. In *2021 ieee/acm 43rd international conference on software engineering: Software engineering education and training (icse-seet)* (pp. 110–121).

Krüger, J., Berger, T., & Leich, T. (2019). Features and how to find them: a survey of manual feature location. *Software Engineering for Variability Intensive Systems*, 153–172.

Krüger, J., Gu, W., Shen, H., Mukelabai, M., Hebig, R., & Berger, T. (2018). Towards a better understanding of software features and their characteristics:

a case study of marlin. In *Proceedings of the 12th international workshop on variability modelling of software-intensive systems* (pp. 105–112).

Lewis, J. R., & Sauro, J. (2017). Can i leave this one out? the effect of dropping an item from the sus. *Journal of Usability Studies*, *13*(1).

Mukelabai, M., Nešić, D., Maro, S., Berger, T., & Steghöfer, J.-P. (2018). Tackling combinatorial explosion: a study of industrial needs and practices for analyzing highly configurable systems. In *Proceedings of the 33rd acm/ieee international conference on automated software engineering* (pp. 155–166).

Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of management information systems*, *24*(3), 45–77.

Poshyvanyk, D., Guéhéneuc, Y.-G., Marcus, A., Antoniol, G., & Rajlich, V. (2007). Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, *33*(6), 420–432.

Robillard, M. P., & Murphy, G. C. (2007). Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *16*(1), 3–es.

Rubin, J., & Chechik, M. (2013). A survey of feature location techniques. In *Domain engineering* (pp. 29–58). Springer.

Rugaber, S., Stirewalt, K., & Wills, L. M. (1996). Understanding interleaved code. In *Reverse engineering* (pp. 47–76). Springer.

Schwarz, T. (2020). Embedded annotations specification. `https://bitbucket.org/easelab/faxe/src/master/specification/embedded_annotation_specification.pdf`.

Schwarz, T. (2021). *Design and assessment of an engine for embedded feature annotations* (Master's Thesis). Chalmers University of Technology, Sweden.

Schwarz, T., Mahmood, W., & Berger, T. (2020). 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* (pp. 5–8).

Shmerlin, Y., Hadar, I., Kliger, D., & Makabee, H. (2015). To document or not to document? an exploratory study on developers' motivation to document code. In *International conference on advanced information systems engineering* (pp. 100–106).

Wang, J., Peng, X., Xing, Z., & Zhao, W. (2013). How developers perform feature location tasks: a human-centric and process-oriented exploratory study. *Journal of Software: Evolution and Process*, *25*(11), 1193–1224.

A

Resources for first iteration

A.1 Introduction used in first iteration.

Purpose:

The purpose of this repo is to test and evaluate the embedded feature annotations plugin HAnS for IntelliJ.

HAnS-text supports:

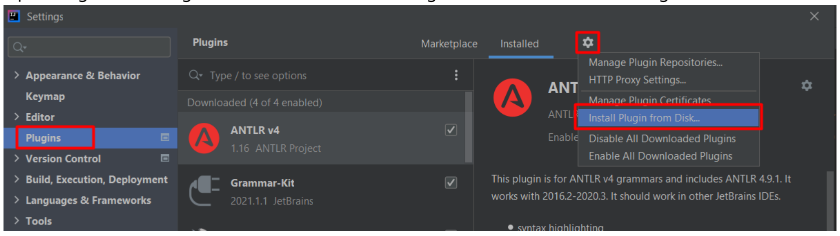
- Feature Annotation Languages
  - Feature Model
  - Feature to folder
  - Feature to file
  - Code annotations
- Syntax Highlighting
- Code Completion
- Feature View

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..."



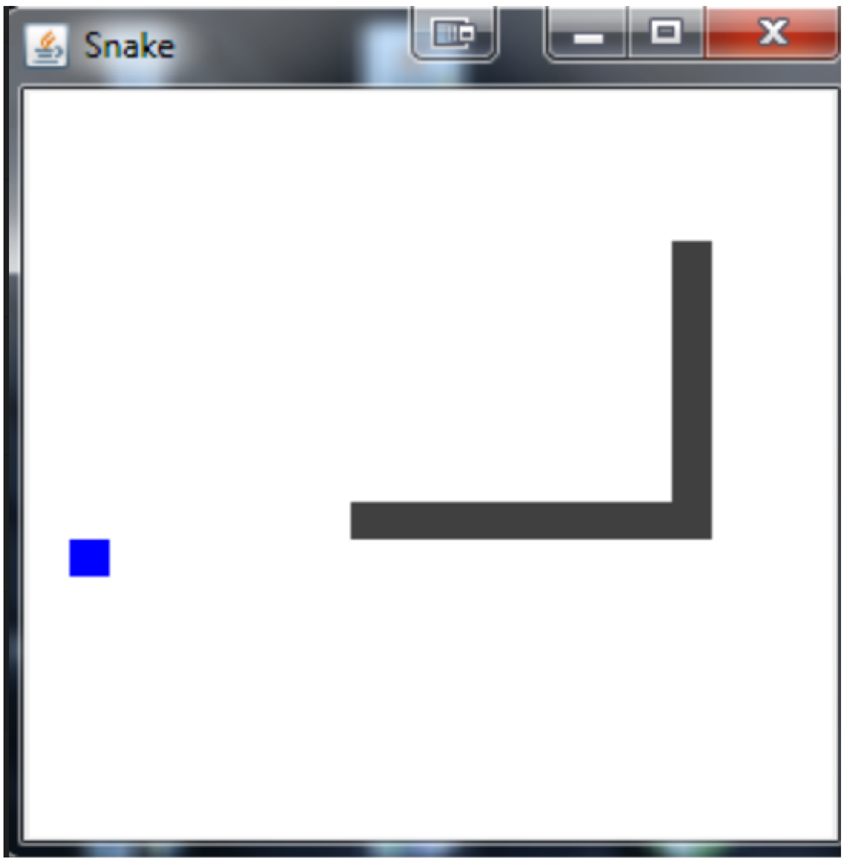
This plugin is for ANTLR v4 grammars and includes ANTLR 4.9.1. It works with 2016.2-2020.3. It should work in other JetBrains IDEs.

- 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 below is contains all features present in the Snake game.

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

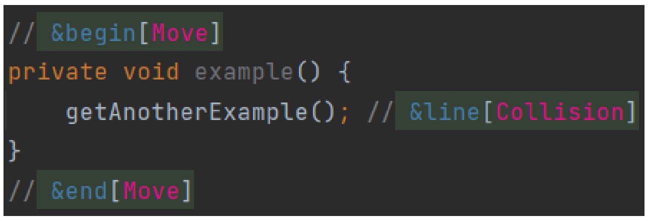
Feature Reference Names

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 feature. 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.

```
KeyboardListener.java
Controls

Direction.java
Controls, Move
```

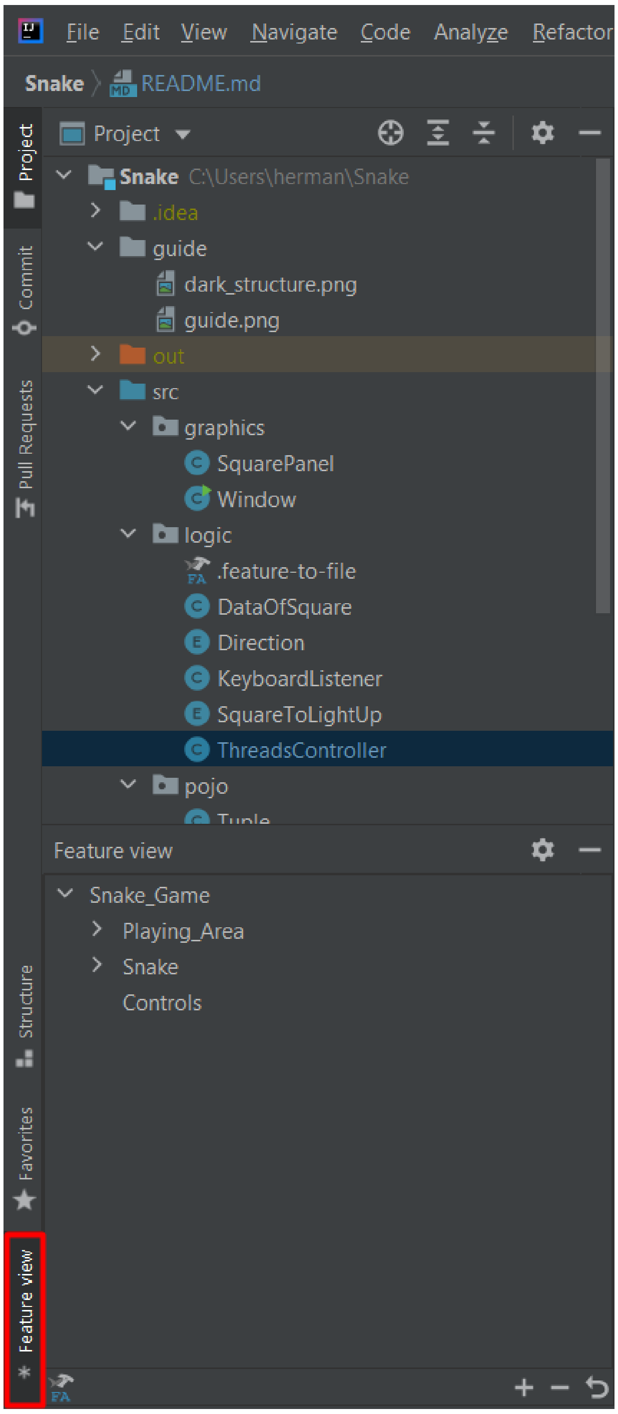
Feature-to-folder mapping

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

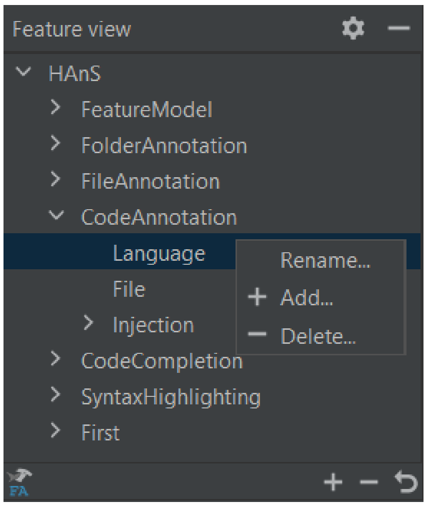
```
Example
```

Feature View

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


File
Edit
View
Navigate
Code
Analyze
Refactor
Snake
README.md
Project
Snake C:\Users\herman\Snake
.idea
guide
dark_structure.png
guide.png
out
src
graphics
SquarePanel
Window
logic
.feature-to-file
DataOfSquare
Direction
KeyboardListener
SquareToLightUp
ThreadsController
pojo
Tuple
Feature view
Snake_Game
Playing_Area
Snake
Controls
Commit
Pull Requests
Structure
Favorites
Feature view

The Feature View supports adding, removing and renaming of features in the Feature Model. These actions make the corresponding changes to the Feature Model file automatically. If the Feature Model file is changed manually the Feature View can be updated by clicking the button in the bottom right corner.



- Group 1
- Group 2

A.1.1 Task sheet for group 1 in first iteration.

Tasks to complete

During the task, take notes while coding, so you can answer the questionnaire after completing all the tasks.

Warmup task

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 View tab (bottom left).
- Map the feature `Playing_Area` to the new `.feature-to-folder` file by writing it into the file.

You have now mapped the feature *Playing_Area* to the *graphics* directory.

Tasks:

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 `Head` feature to the new name `Position`, including all references to it.

Task 4

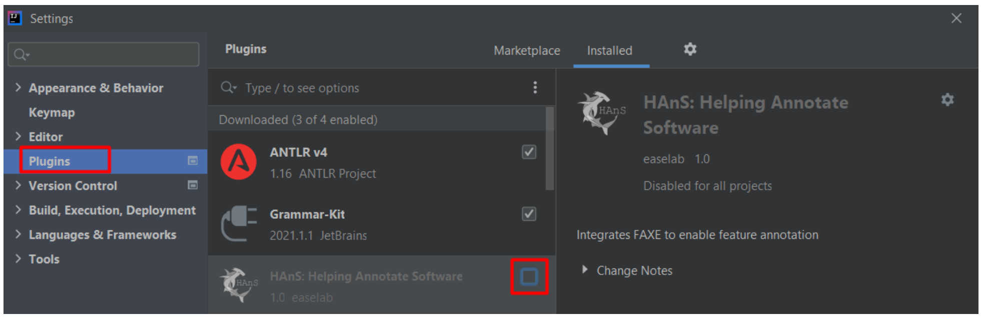
After the above tasks are completed, answer the questions here.

Second part:

Task 5

Disable the plugin.

1. Open Plugins in Settings/Preferences. Uncheck the box marked in red in the image below.
2. Click "Apply".



Task 6

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, create methods to enable this.

- *Hint 1: The speed of the game is inverted. It is implemented as a sleep call, so the shorter the sleep, the faster the game.*
- *Hint 2: The difficulty may never be equal to or exceed the speed variable.*
- *Hint 3: To check if the snake passes the bottom border check if the head is equal to 0.*

Reminder: Make sure you annotate the code you write!

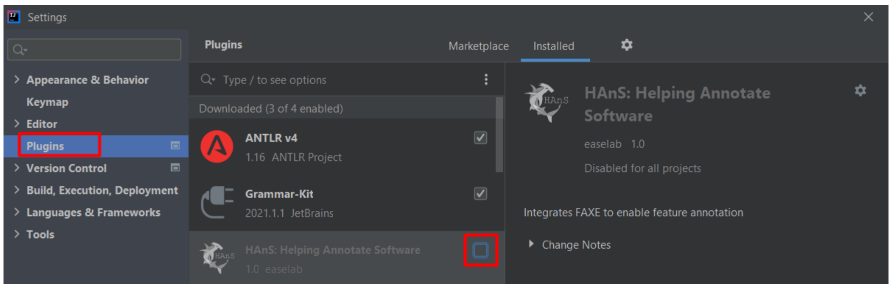
A.1.2 Task sheet for group 2 in first iteration.

Tasks to complete

During the task, take notes while coding, so you can answer the questionnaire after completing all the tasks.

Disable the plugin.

- Open Plugins in Settings/Preferences. Uncheck the box marked in red in the image below.
- Click "Apply".



Warmup task

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 View tab (bottom left).
- Map the feature `Playing_Area` to the new `.feature-to-folder` file by writing it into the file.

You have now mapped the feature *Playing_Area* to the *graphics* directory.

Tasks:

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 `Head` feature to the new name `Position`, including all references to it.

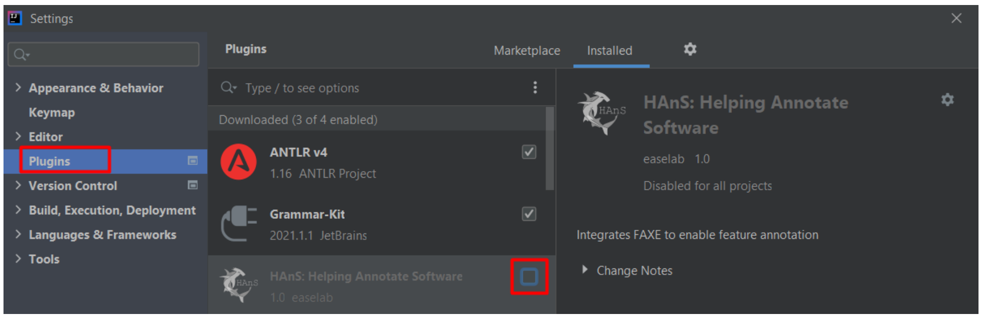
Task 4

After the above tasks are completed, answer the questions here.

Second part:

Task 5

Enable the HAnS plugin: * Open Plugins in Settings/Preferences. Check the box marked in red in the image below. * Click "Apply".



Integrates FAXE to enable feature annotation

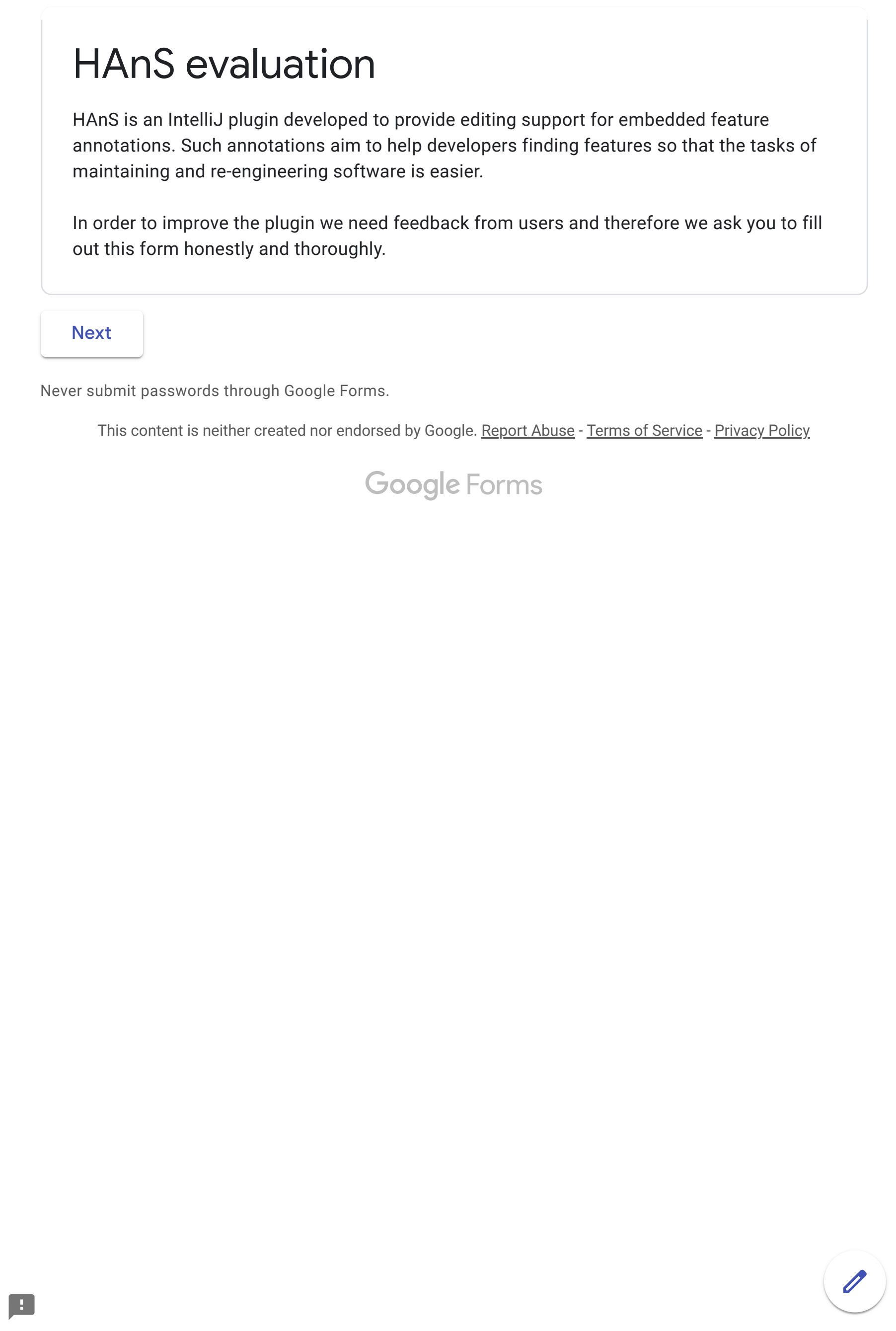
Task 6

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, create methods to enable this.

- *Hint 1: The speed of the game is inverted. It is implemented as a sleep call, so the shorter the sleep, the faster the game.*
- *Hint 2: The difficulty may never be equal to or exceed the speed variable.*
- *Hint 3: To check if the snake passes the bottom border check if the head is equal to 0.*

Reminder: Make sure you annotate the code you write!

A.2 Questionnaire for the first user study



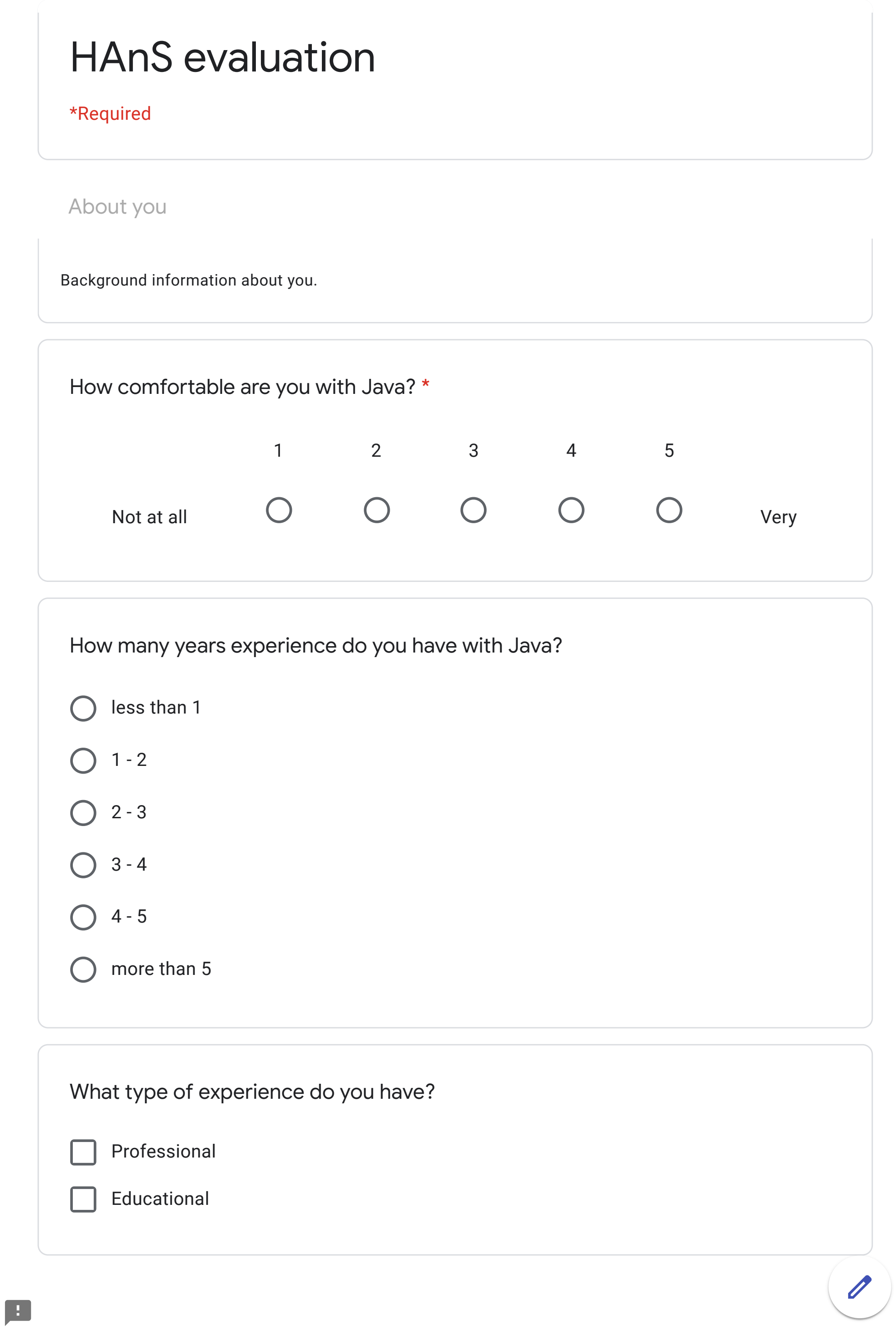
HAnS evaluation

HAnS is an IntelliJ plugin developed to provide editing support for embedded feature annotations. Such annotations aim to help developers finding features so that the tasks of maintaining and re-engineering software is easier.

In order to improve the plugin we need feedback from users and therefore we ask you to fill out this form honestly and thoroughly.

Never submit passwords through Google Forms.

This content is neither created nor endorsed by Google. Report Abuse - Terms of Service - Privacy Policy



HAnS evaluation

*Required

About you

Background information about you.

How comfortable are you with Java? *

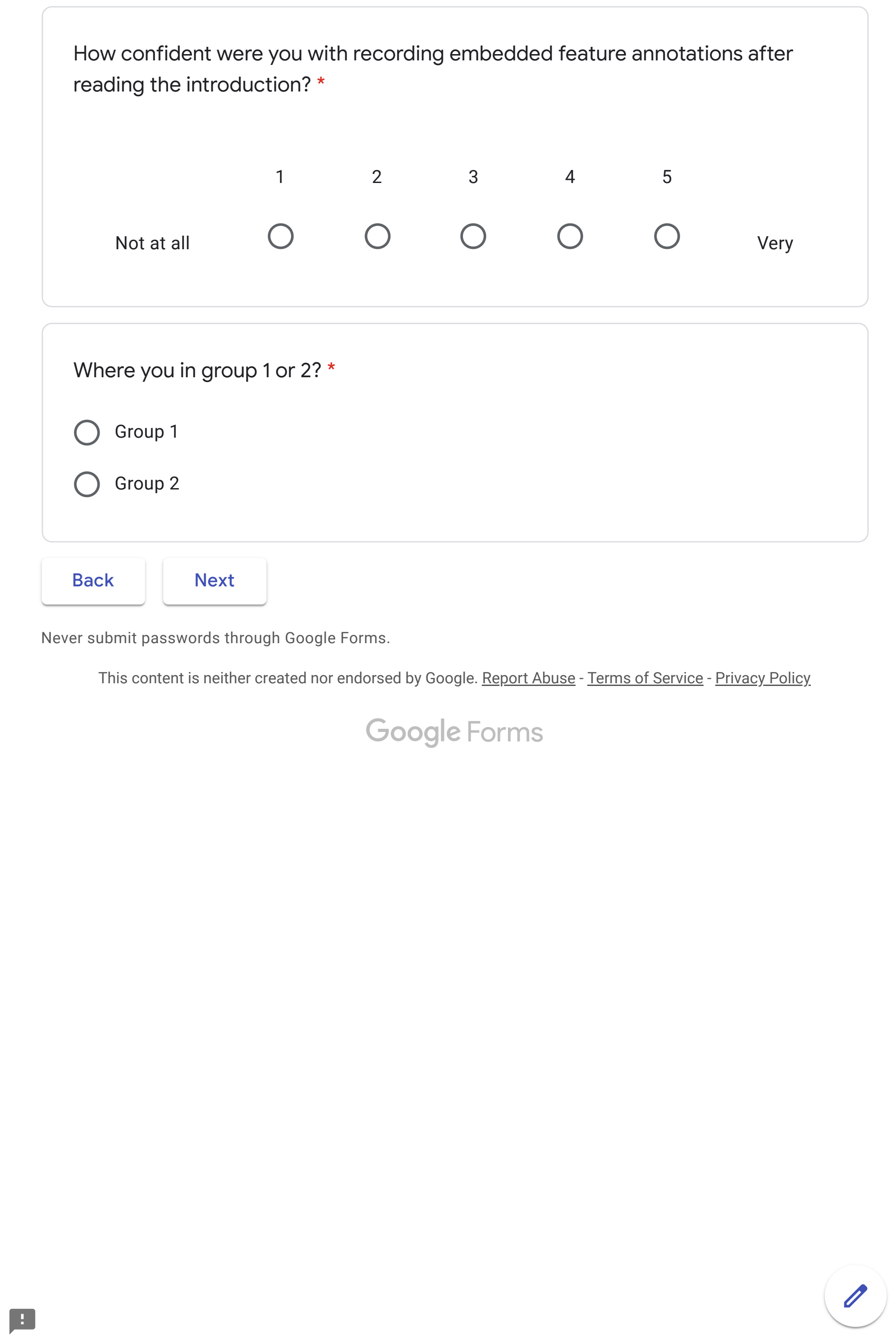
	1	2	3	4	5	
Not at all	○	○	○	○	○	Very

How many years experience do you have with Java?

- ○ less than 1
- ○ 1 - 2
- ○ 2 - 3
- ○ 3 - 4
- ○ 4 - 5
- ○ more than 5

What type of experience do you have?

- ☐ Professional
- ☐ Educational



How confident were you with recording embedded feature annotations after reading the introduction? *

	1	2	3	4	5	
Not at all	○	○	○	○	○	Very

Where you in group 1 or 2? *

○ Group 1

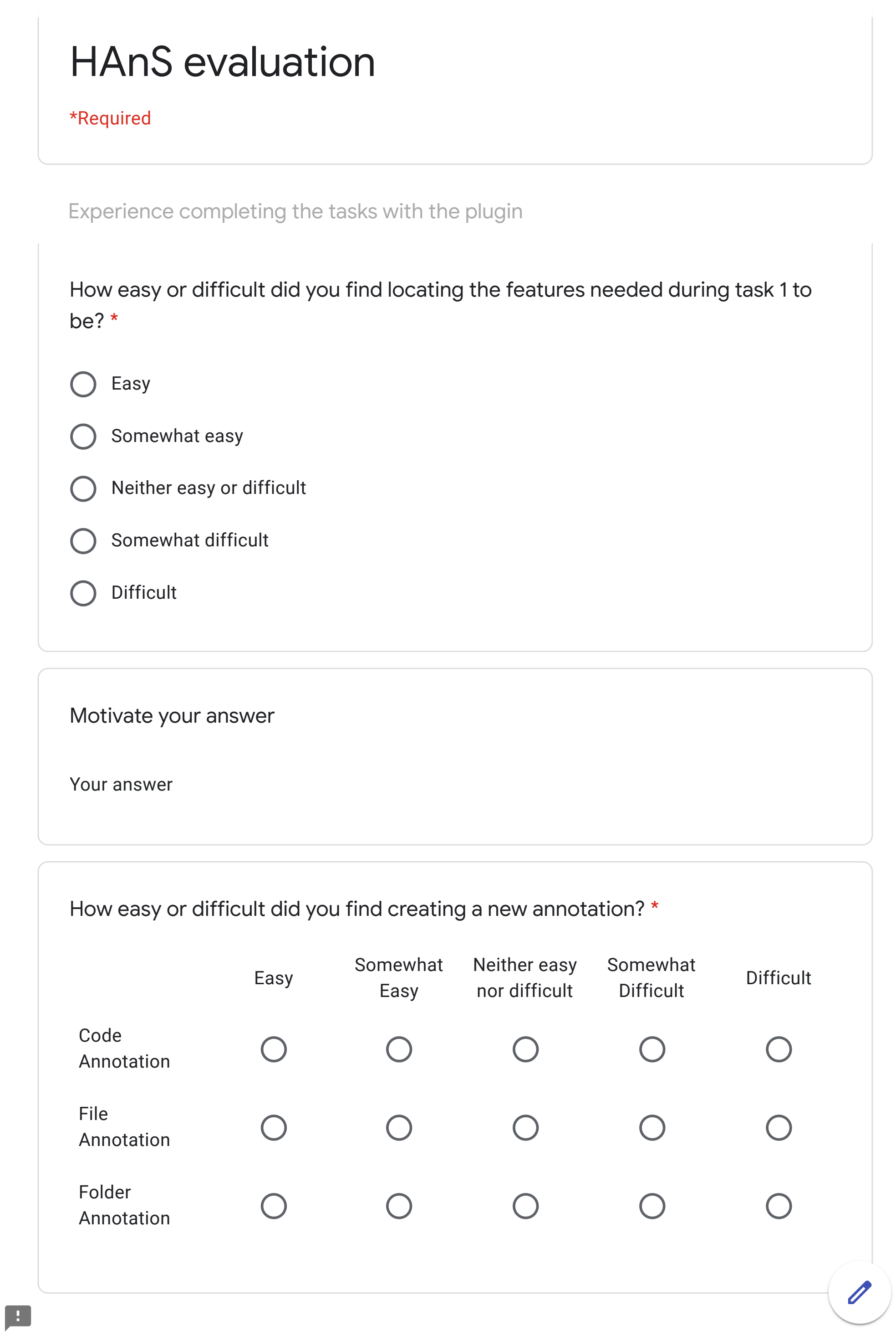
○ Group 2

Never submit passwords through Google Forms.

This content is neither created nor endorsed by Google. Report Abuse - Terms of Service - Privacy Policy

Google Forms

A.2.1 Questionnaire for group 1 in first iteration.



HAnS evaluation

*Required

Experience completing the tasks with the plugin

How easy or difficult did you find locating the features needed during task 1 to be? *

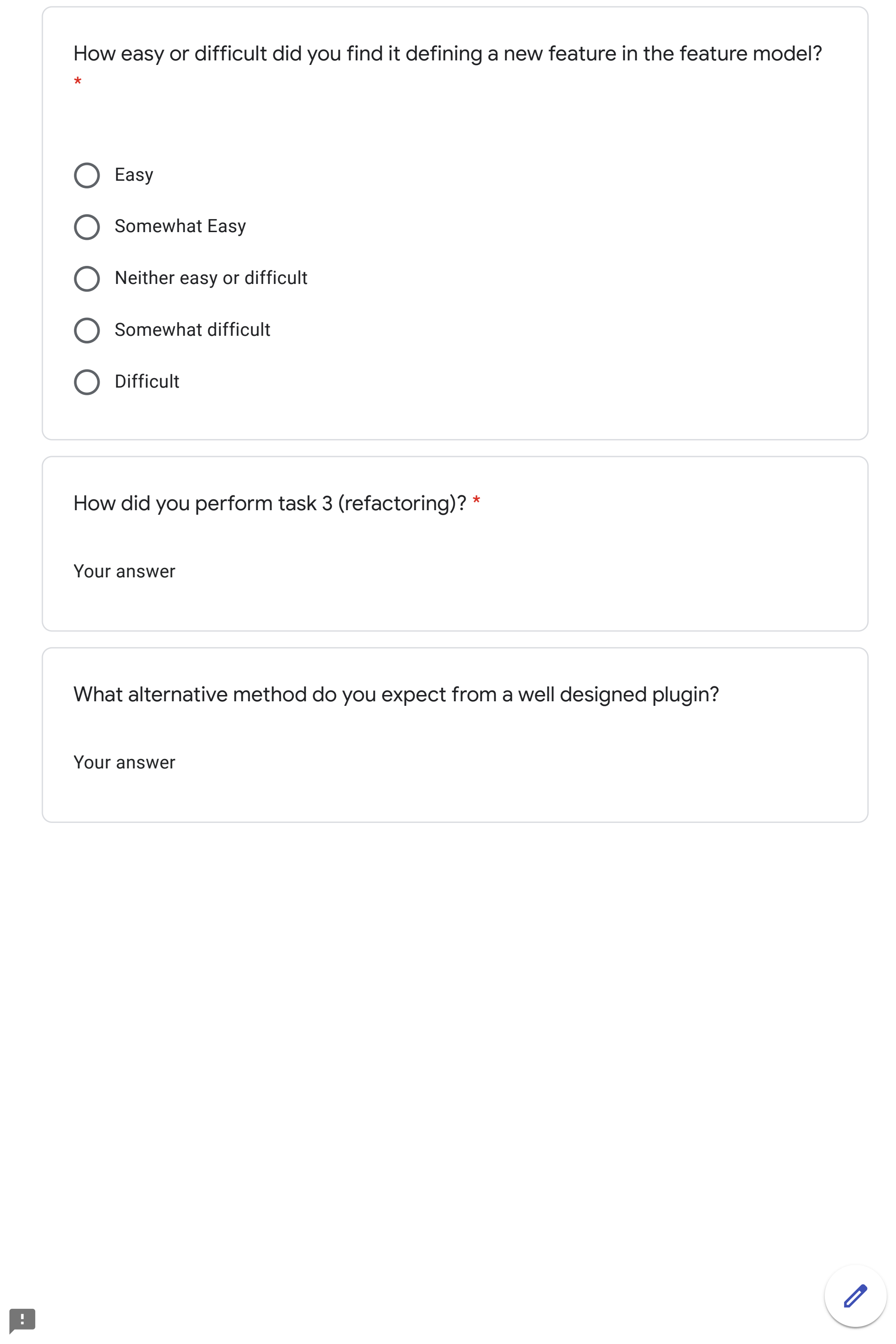
- ◯ Easy
- ◯ Somewhat easy
- ◯ Neither easy or difficult
- ◯ Somewhat difficult
- ◯ Difficult

Motivate your answer

Your answer

How easy or difficult did you find creating a new annotation? *

	Easy	Somewhat Easy	Neither easy nor difficult	Somewhat Difficult	Difficult
Code Annotation	◯	◯	◯	◯	◯
File Annotation	◯	◯	◯	◯	◯
Folder Annotation	◯	◯	◯	◯	◯



How easy or difficult did you find it defining a new feature in the feature model? *

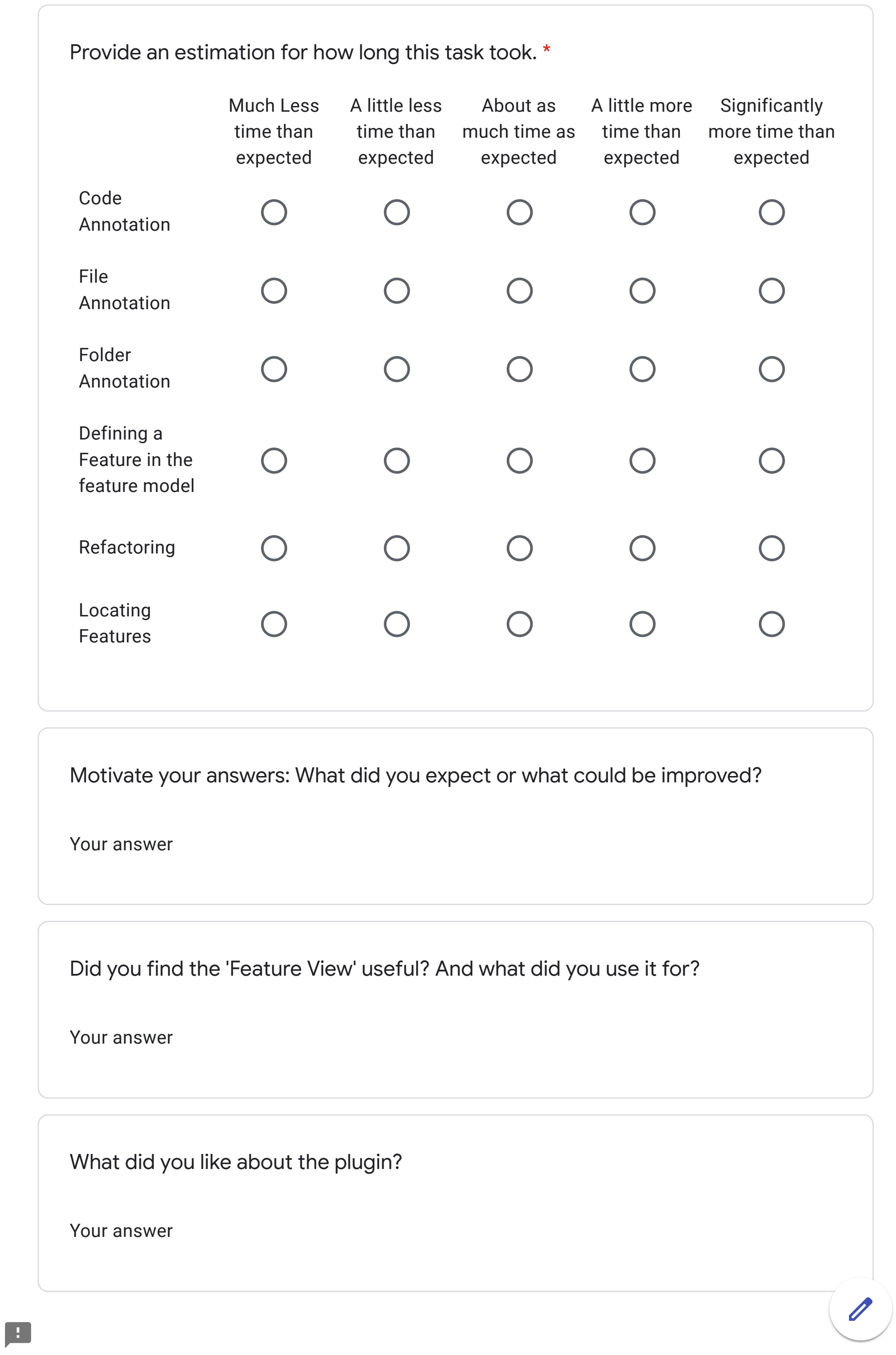
- ○ Easy
- ○ Somewhat Easy
- ○ Neither easy or difficult
- ○ Somewhat difficult
- ○ Difficult

How did you perform task 3 (refactoring)? *

Your answer

What alternative method do you expect from a well designed plugin?

Your answer



Provide an estimation for how long this task took. *

	Much Less time than expected	A little less time than expected	About as much time as expected	A little more time than expected	Significantly more time than expected
Code Annotation	○	○	○	○	○
File Annotation	○	○	○	○	○
Folder Annotation	○	○	○	○	○
Defining a Feature in the feature model	○	○	○	○	○
Refactoring	○	○	○	○	○
Locating Features	○	○	○	○	○

Motivate your answers: What did you expect or what could be improved?

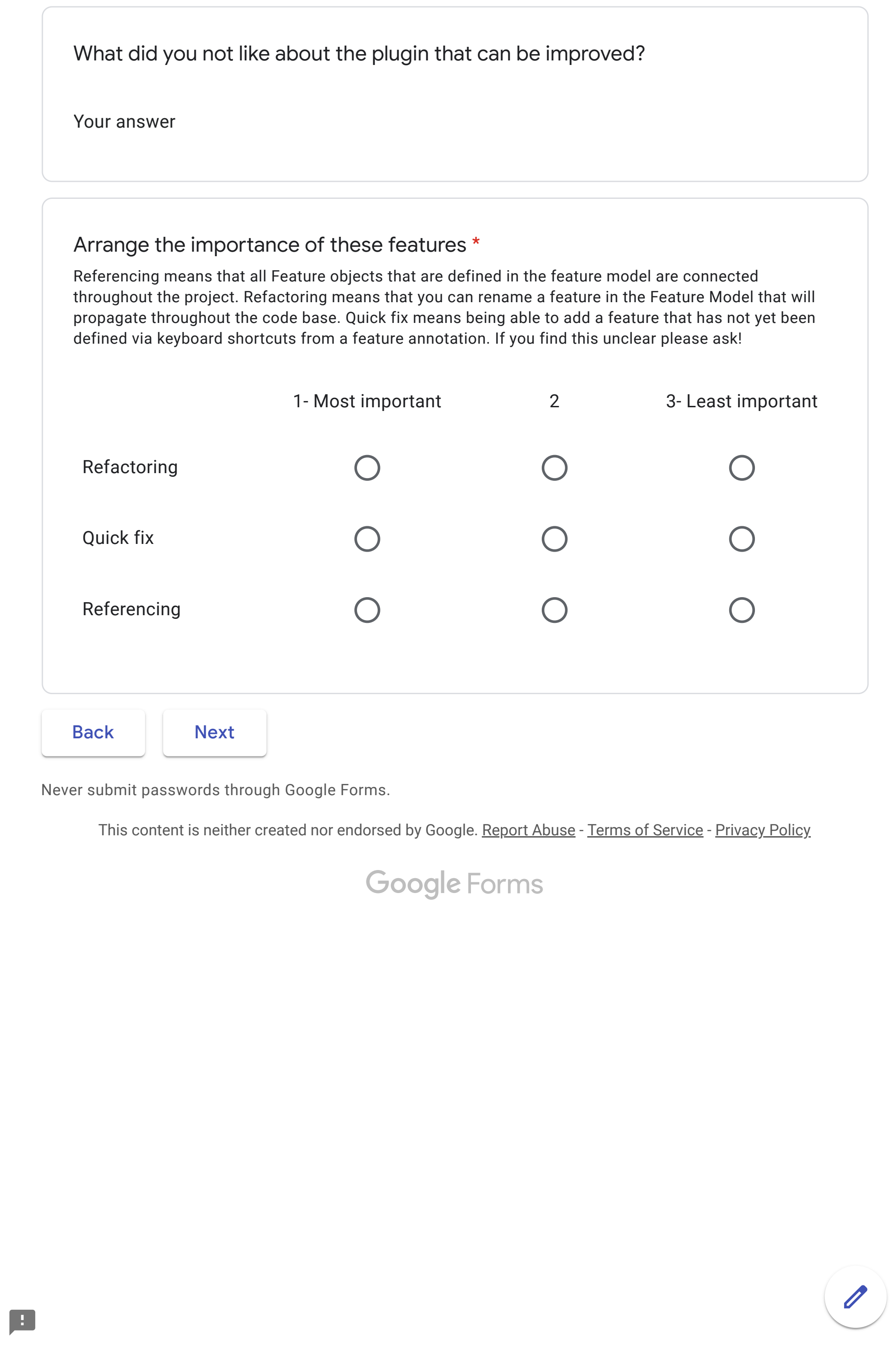
Your answer

Did you find the 'Feature View' useful? And what did you use it for?

Your answer

What did you like about the plugin?

Your answer



What did you not like about the plugin that can be improved?

Your answer

Arrange the importance of these features *

Referencing means that all Feature objects that are defined in the feature model are connected throughout the project. Refactoring means that you can rename a feature in the Feature Model that will propagate throughout the code base. Quick fix means being able to add a feature that has not yet been defined via keyboard shortcuts from a feature annotation. If you find this unclear please ask!

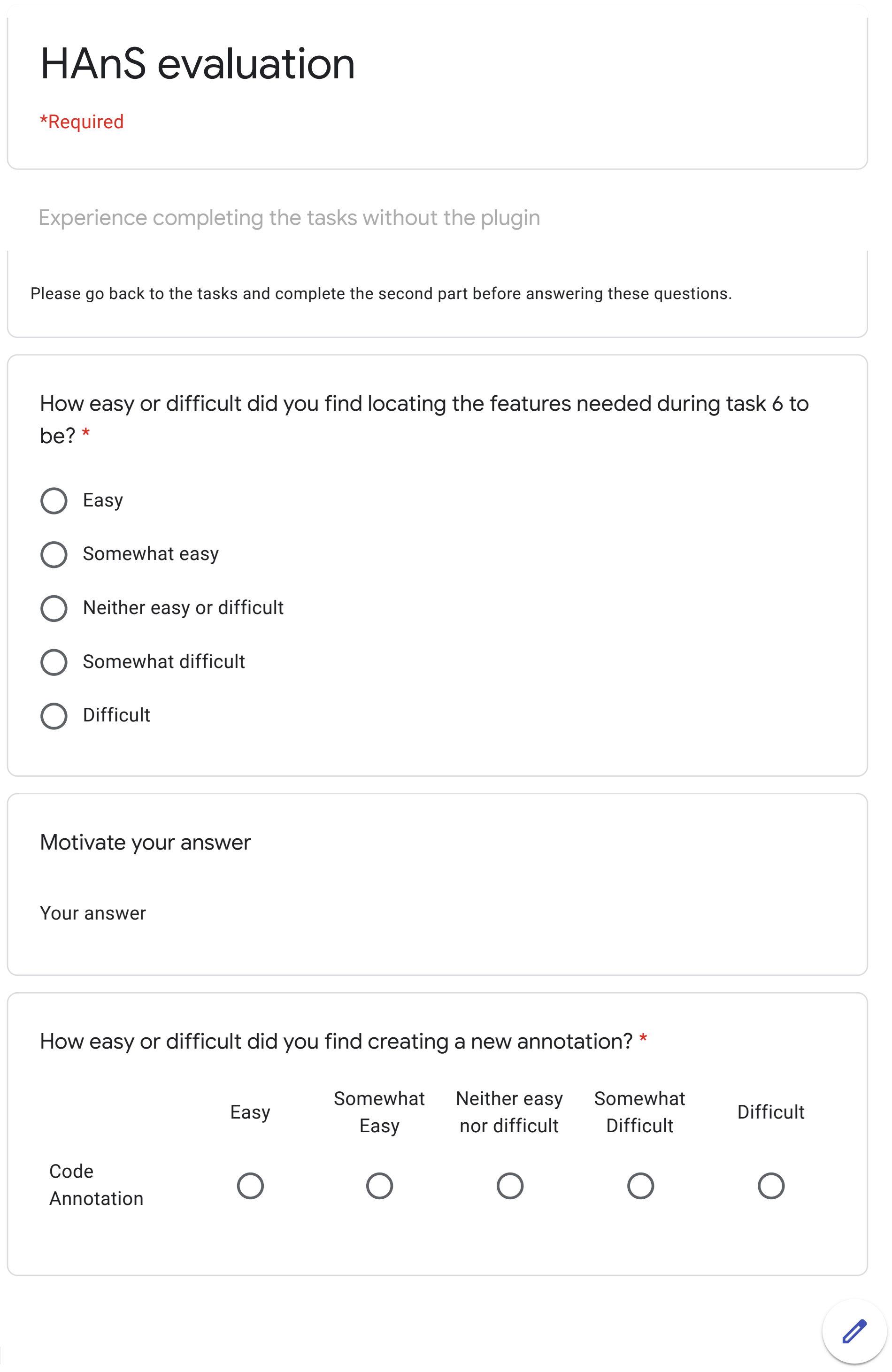
	1- Most important	2	3- Least important
Refactoring	○	○	○
Quick fix	○	○	○
Referencing	○	○	○

Back Next

Never submit passwords through Google Forms.

This content is neither created nor endorsed by Google. Report Abuse - Terms of Service - Privacy Policy

Google Forms



HAnS evaluation

*Required

Experience completing the tasks without the plugin

Please go back to the tasks and complete the second part before answering these questions.

How easy or difficult did you find locating the features needed during task 6 to be? *

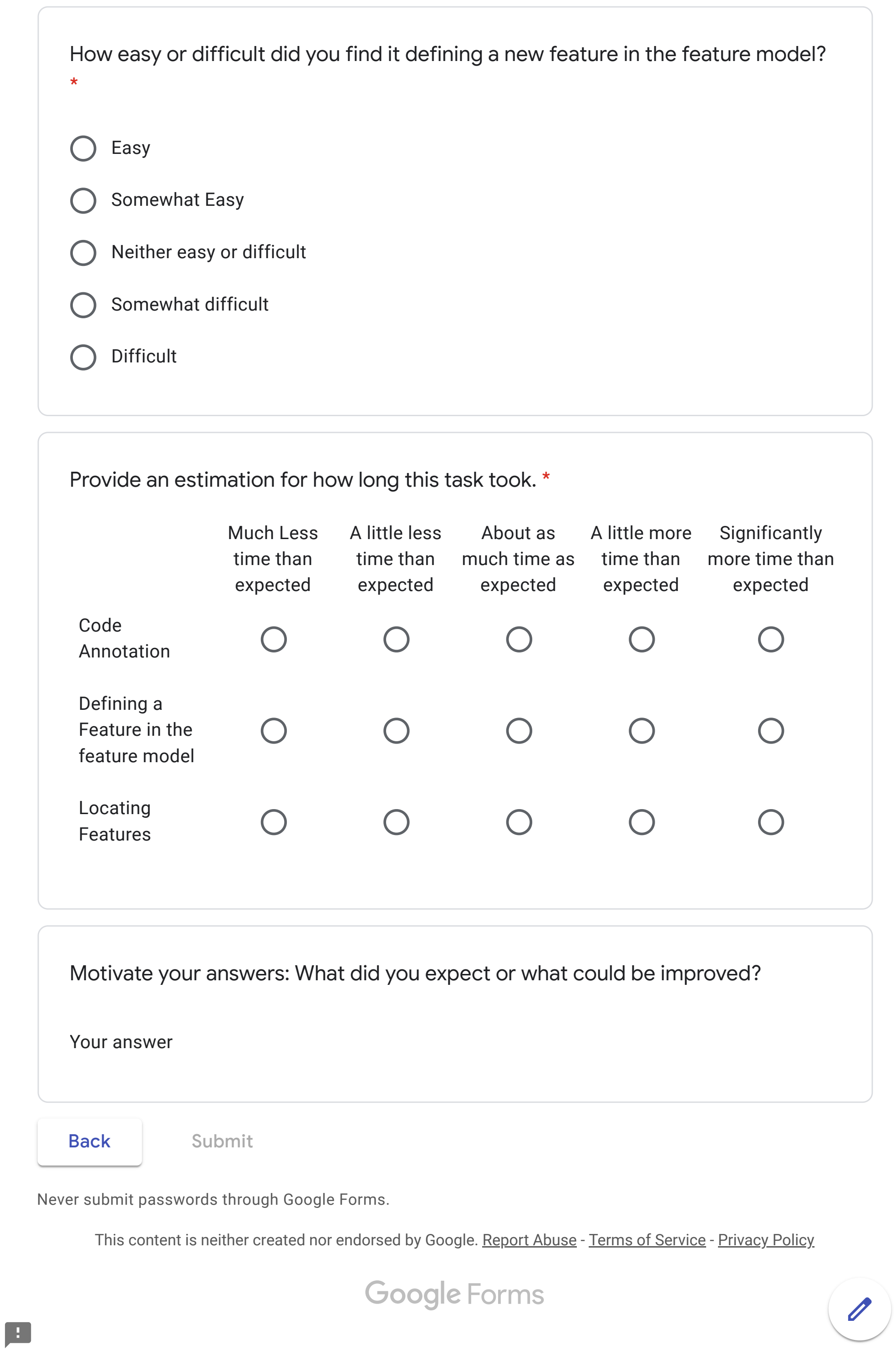
- ○ Easy
- ○ Somewhat easy
- ○ Neither easy or difficult
- ○ Somewhat difficult
- ○ Difficult

Motivate your answer

Your answer

How easy or difficult did you find creating a new annotation? *

	Easy	Somewhat Easy	Neither easy nor difficult	Somewhat Difficult	Difficult
Code Annotation	○	○	○	○	○



How easy or difficult did you find it defining a new feature in the feature model? *

- ○ Easy
- ○ Somewhat Easy
- ○ Neither easy or difficult
- ○ Somewhat difficult
- ○ Difficult

Provide an estimation for how long this task took. *

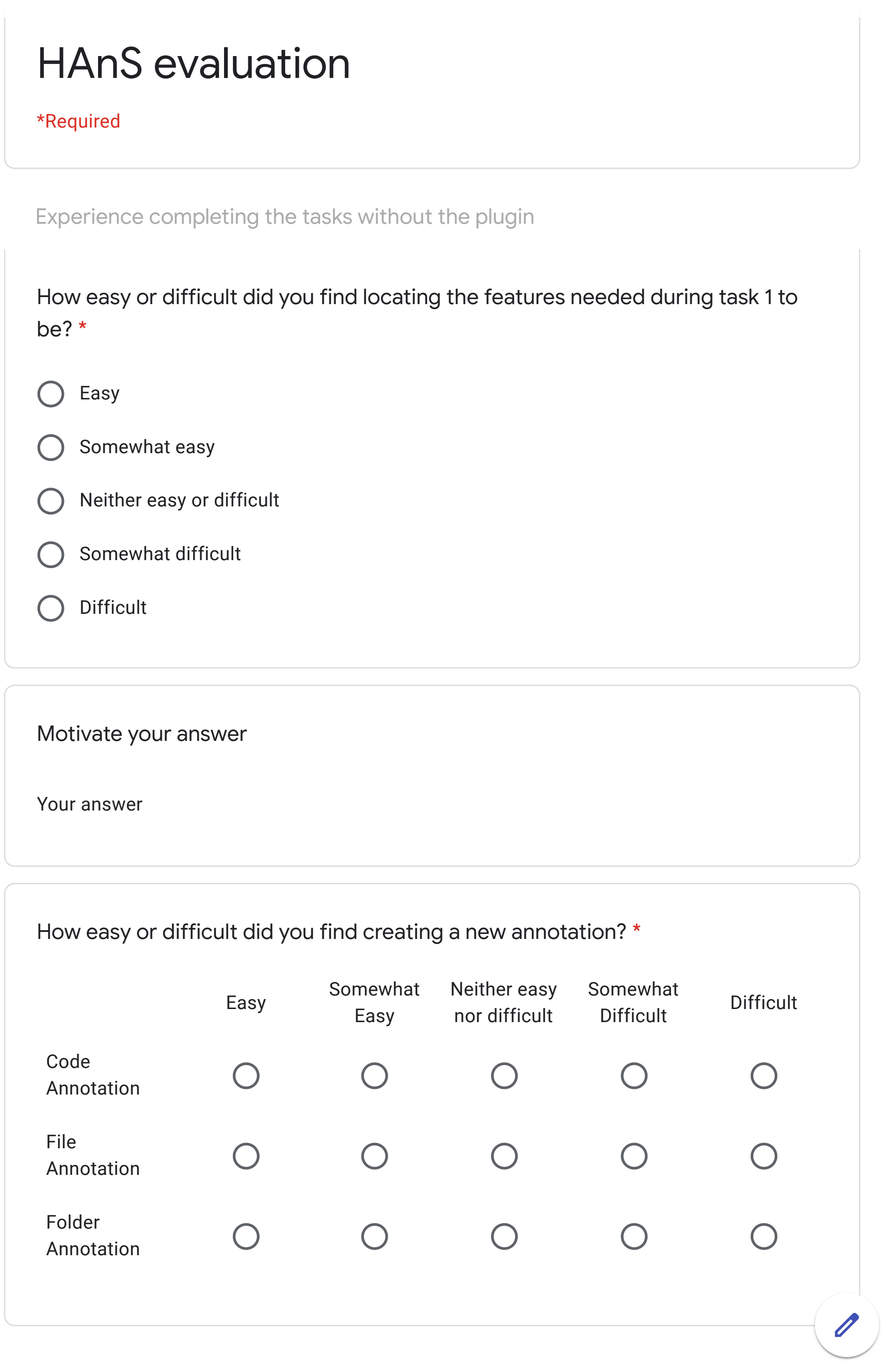
	Much Less time than expected	A little less time than expected	About as much time as expected	A little more time than expected	Significantly more time than expected
Code Annotation	○	○	○	○	○
Defining a Feature in the feature model	○	○	○	○	○
Locating Features	○	○	○	○	○

Motivate your answers: What did you expect or what could be improved?

Your answer

Never submit passwords through Google Forms.

A.2.2 Questionnaire for group 2 in first iteration.



HAnS evaluation

*Required

Experience completing the tasks without the plugin

How easy or difficult did you find locating the features needed during task 1 to be? *

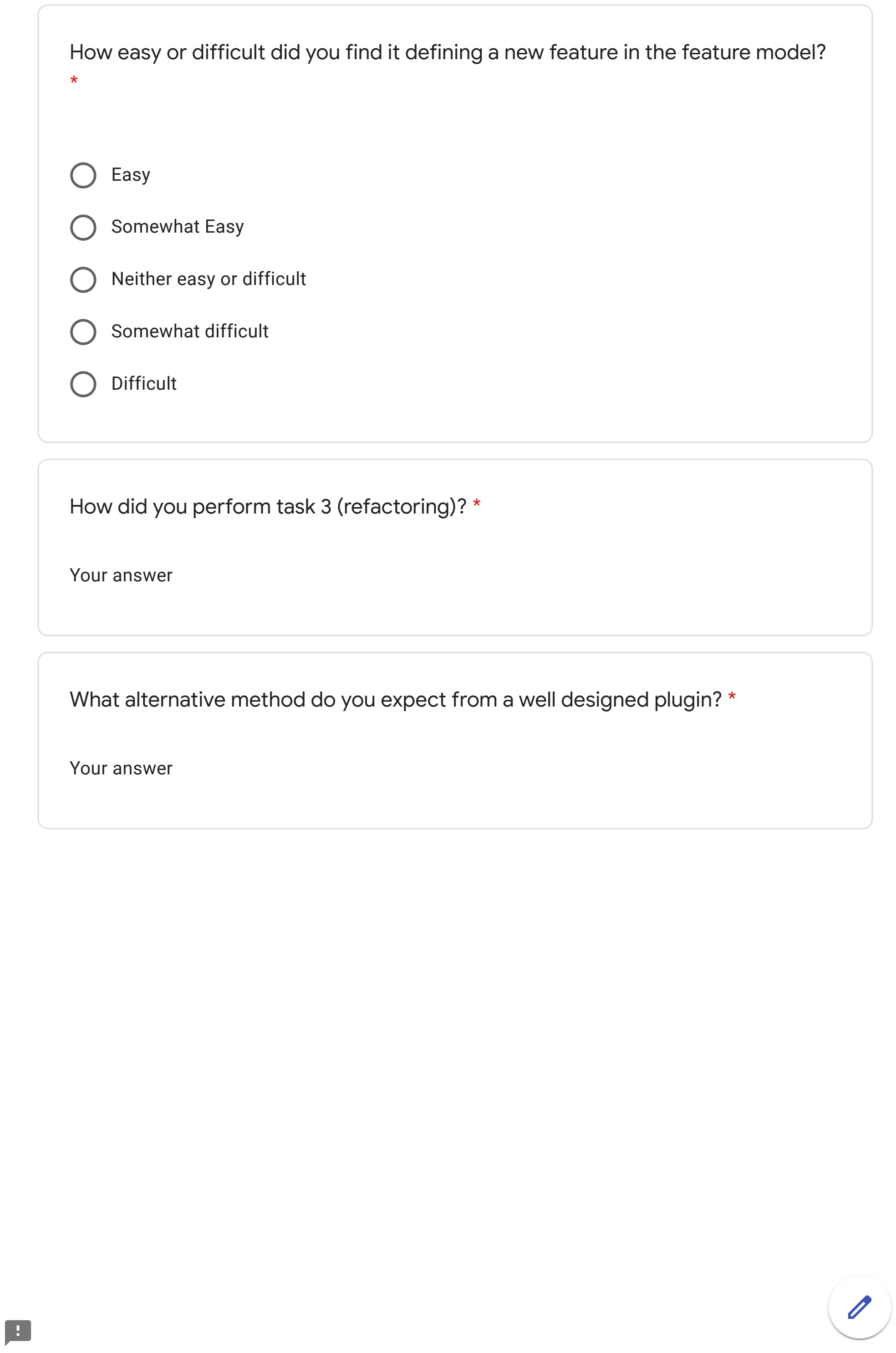
- ○ Easy
- ○ Somewhat easy
- ○ Neither easy or difficult
- ○ Somewhat difficult
- ○ Difficult

Motivate your answer

Your answer

How easy or difficult did you find creating a new annotation? *

	Easy	Somewhat Easy	Neither easy nor difficult	Somewhat Difficult	Difficult
Code Annotation	○	○	○	○	○
File Annotation	○	○	○	○	○
Folder Annotation	○	○	○	○	○



How easy or difficult did you find it defining a new feature in the feature model? *

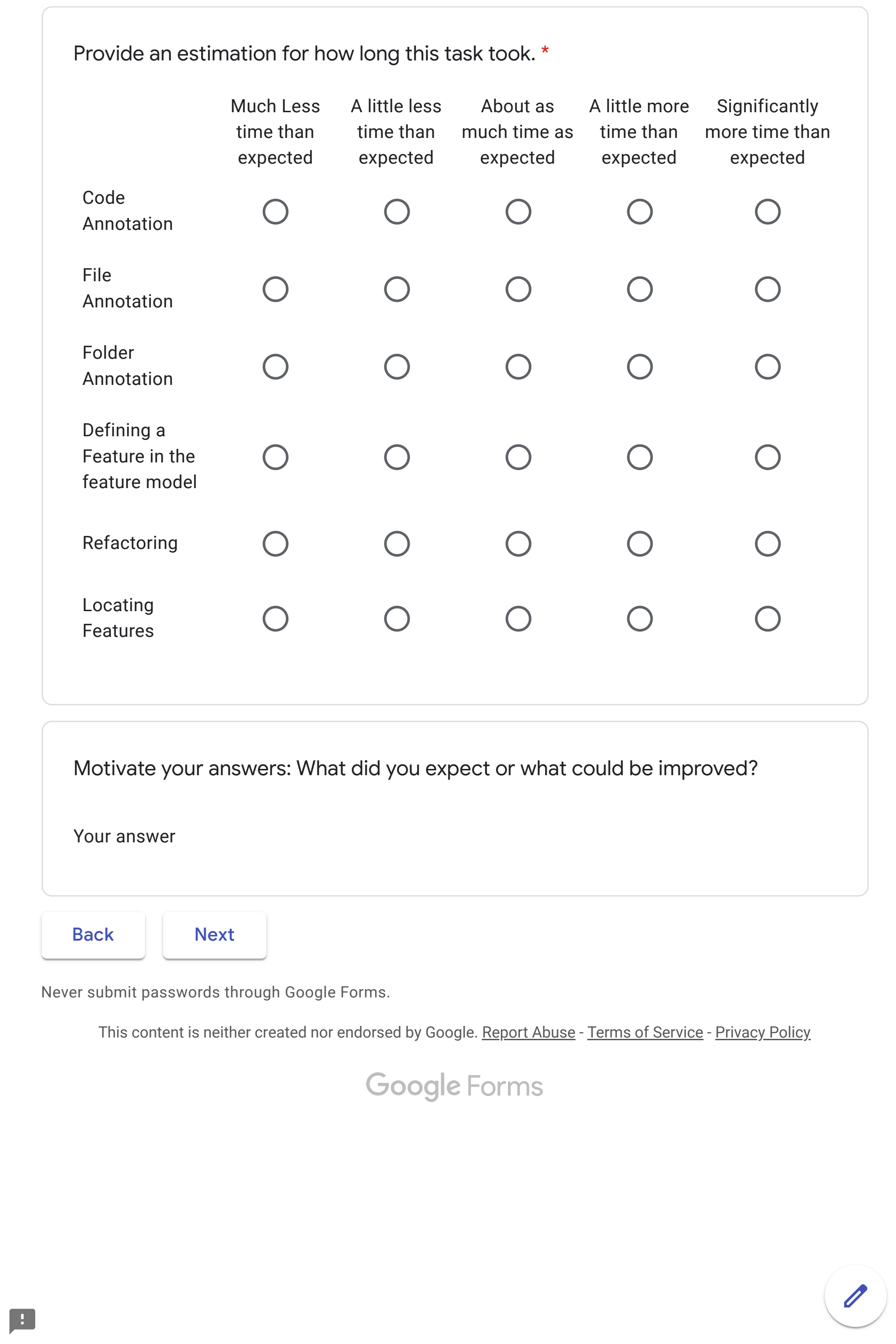
- ○ Easy
- ○ Somewhat Easy
- ○ Neither easy or difficult
- ○ Somewhat difficult
- ○ Difficult

How did you perform task 3 (refactoring)? *

Your answer

What alternative method do you expect from a well designed plugin? *

Your answer



Provide an estimation for how long this task took. *

	Much Less time than expected	A little less time than expected	About as much time as expected	A little more time than expected	Significantly more time than expected
Code Annotation	○	○	○	○	○
File Annotation	○	○	○	○	○
Folder Annotation	○	○	○	○	○
Defining a Feature in the feature model	○	○	○	○	○
Refactoring	○	○	○	○	○
Locating Features	○	○	○	○	○

Motivate your answers: What did you expect or what could be improved?

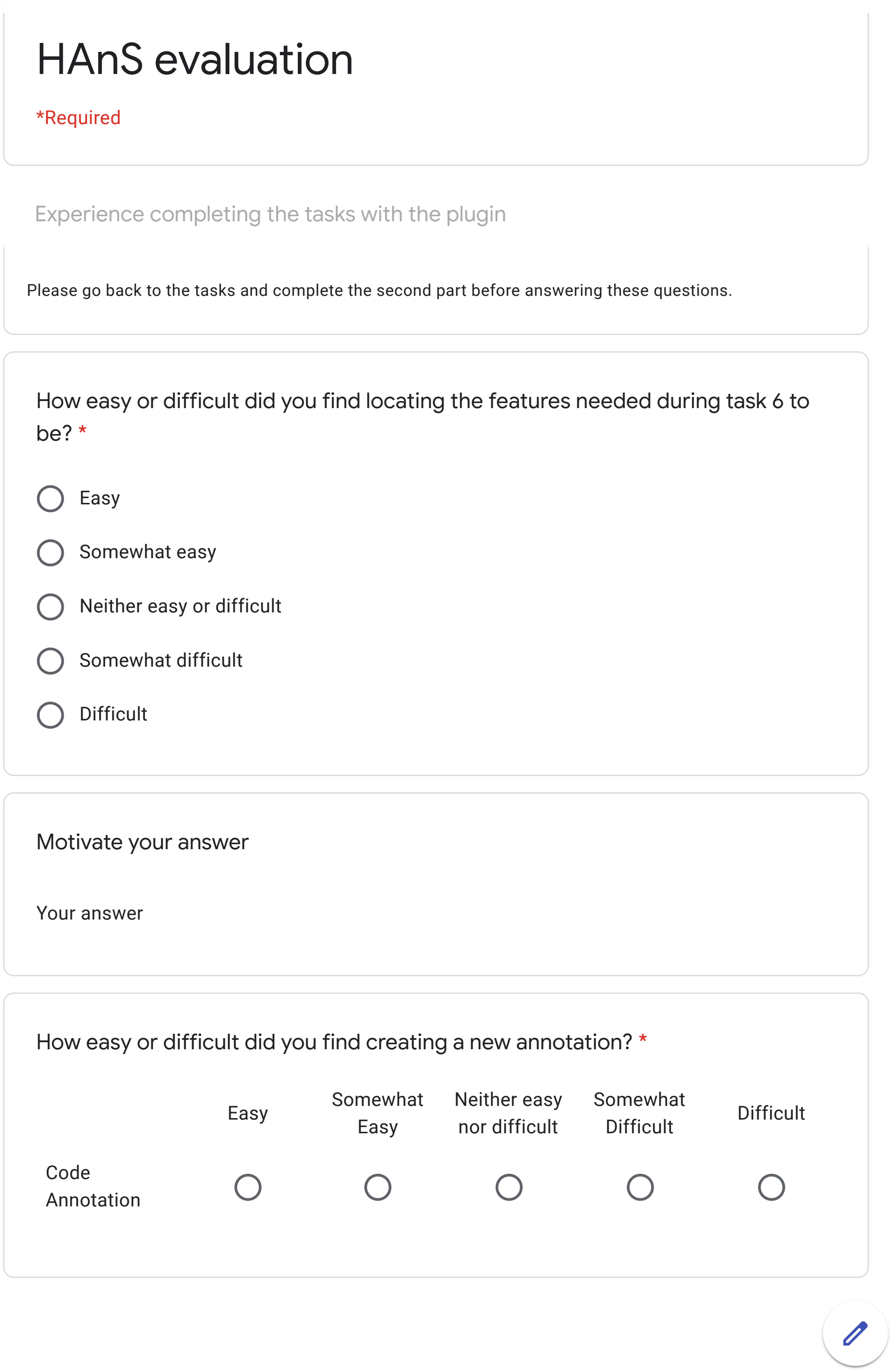
Your answer

Back Next

Never submit passwords through Google Forms.

This content is neither created nor endorsed by Google. Report Abuse - Terms of Service - Privacy Policy

Google Forms



HAnS evaluation

*Required

Experience completing the tasks with the plugin

Please go back to the tasks and complete the second part before answering these questions.

How easy or difficult did you find locating the features needed during task 6 to be? *

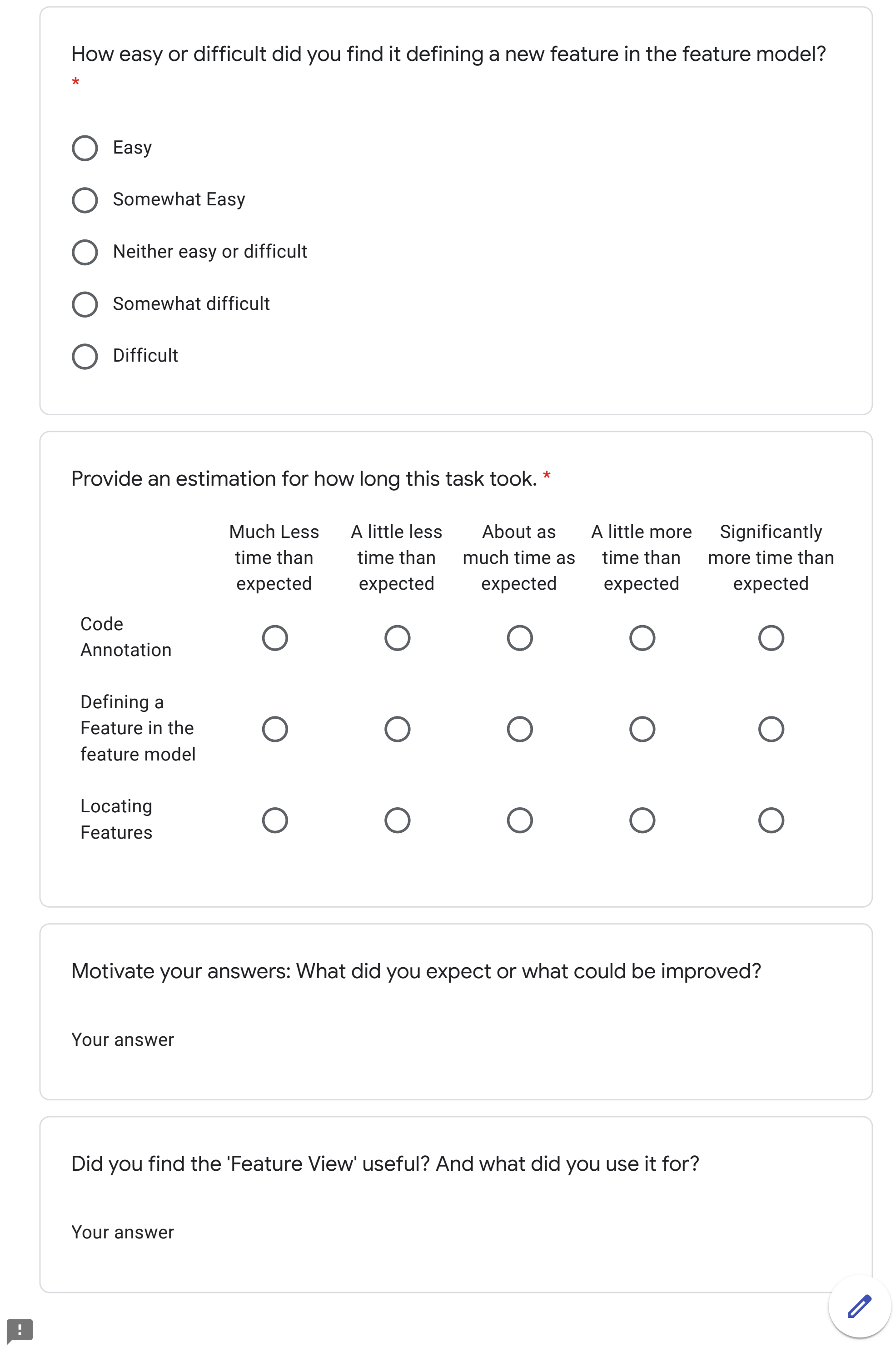
- ○ Easy
- ○ Somewhat easy
- ○ Neither easy or difficult
- ○ Somewhat difficult
- ○ Difficult

Motivate your answer

Your answer

How easy or difficult did you find creating a new annotation? *

	Easy	Somewhat Easy	Neither easy nor difficult	Somewhat Difficult	Difficult
Code Annotation	○	○	○	○	○



How easy or difficult did you find it defining a new feature in the feature model? *

- ○ Easy
- ○ Somewhat Easy
- ○ Neither easy or difficult
- ○ Somewhat difficult
- ○ Difficult

Provide an estimation for how long this task took. *

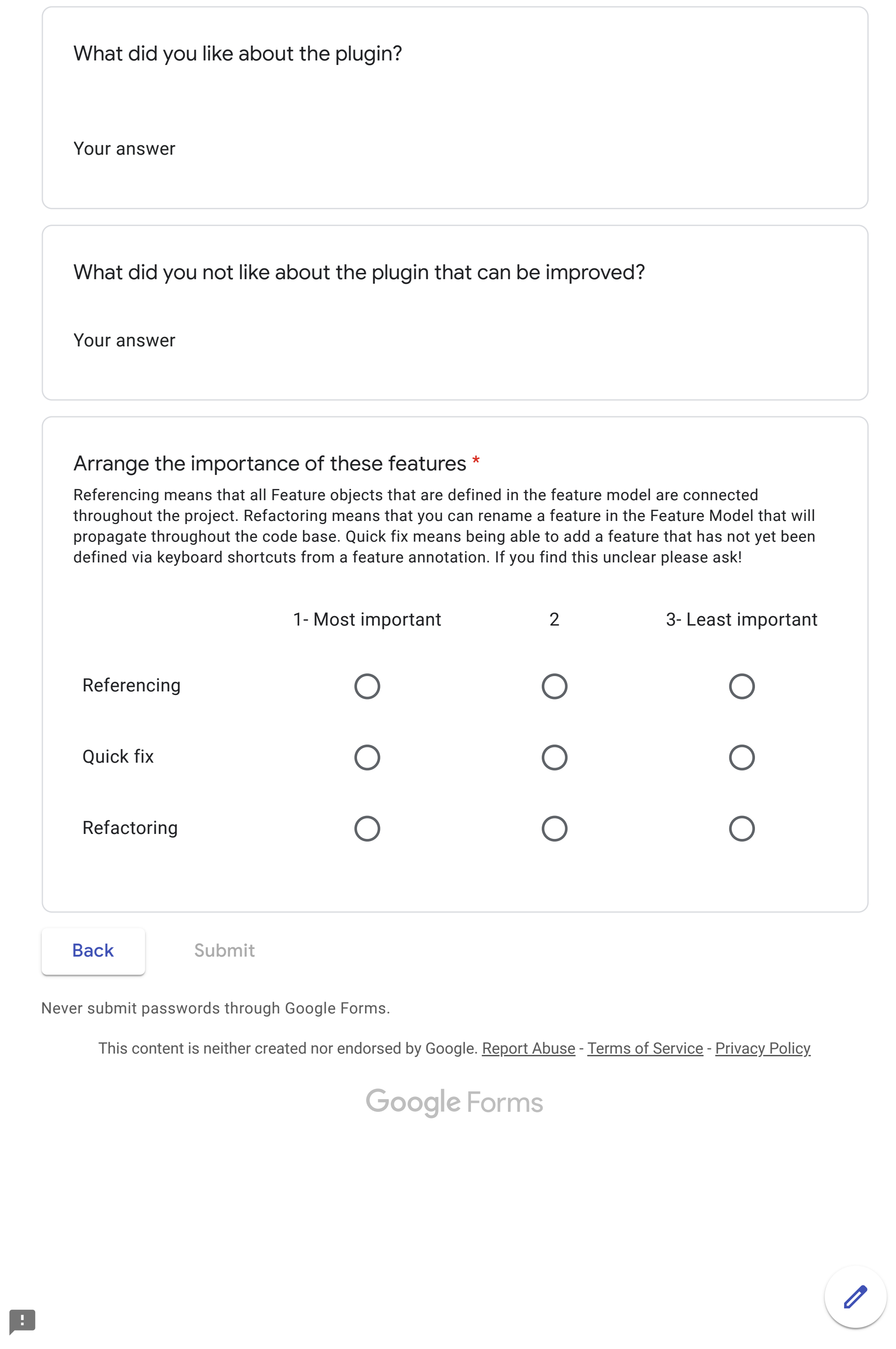
	Much Less time than expected	A little less time than expected	About as much time as expected	A little more time than expected	Significantly more time than expected
Code Annotation	○	○	○	○	○
Defining a Feature in the feature model	○	○	○	○	○
Locating Features	○	○	○	○	○

Motivate your answers: What did you expect or what could be improved?

Your answer

Did you find the 'Feature View' useful? And what did you use it for?

Your answer



What did you like about the plugin?

Your answer

What did you not like about the plugin that can be improved?

Your answer

Arrange the importance of these features *

Referencing means that all Feature objects that are defined in the feature model are connected throughout the project. Refactoring means that you can rename a feature in the Feature Model that will propagate throughout the code base. Quick fix means being able to add a feature that has not yet been defined via keyboard shortcuts from a feature annotation. If you find this unclear please ask!

	1- Most important	2	3- Least important
Referencing	○	○	○
Quick fix	○	○	○
Refactoring	○	○	○

Never submit passwords through Google Forms.

This content is neither created nor endorsed by Google. Report Abuse - Terms of Service - Privacy Policy

Google Forms

B

Resources for second iteration

B.1 Introduction used in second iteration.

Purpose:

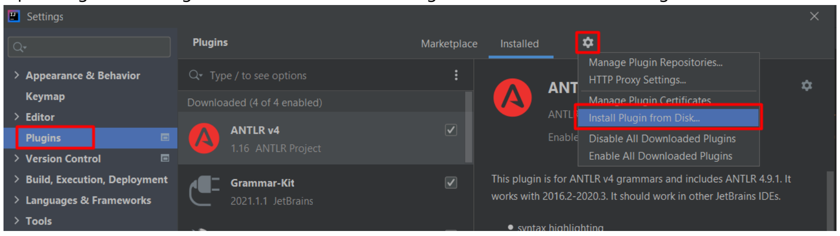
The purpose of this repo is to test and evaluate the embedded feature annotations plugin HAnS for IntelliJ.

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..."

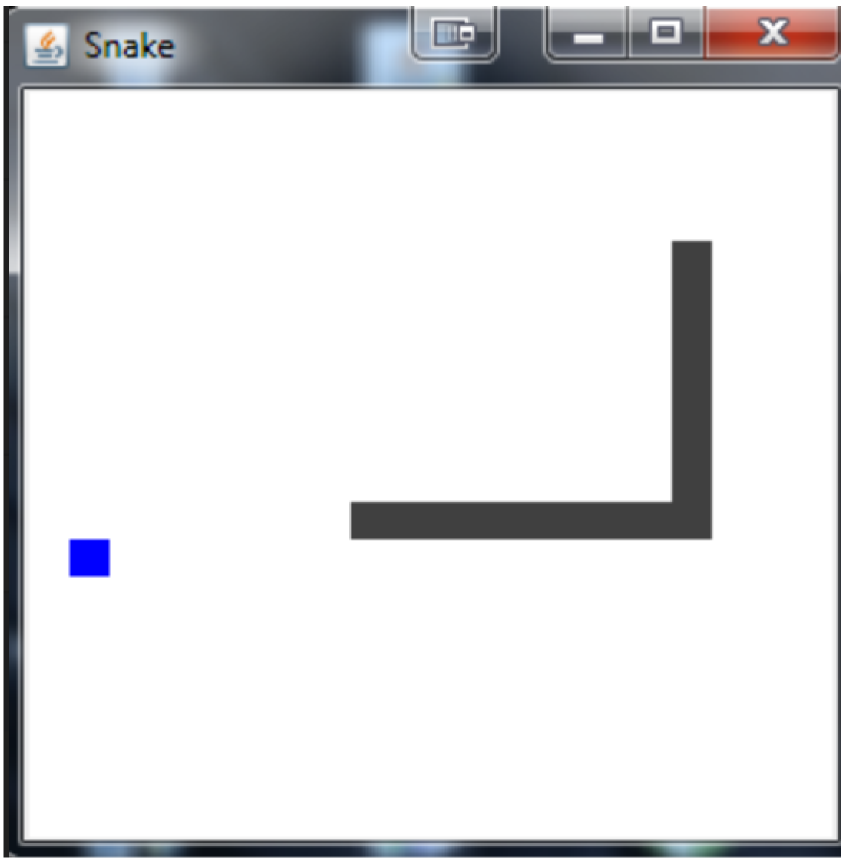

Settings
Appearance & Behavior
Keymap
Editor
Plugins
Version Control
Build, Execution, Deployment
Languages & Frameworks
Tools
Plugins
Marketplace
Installed
Type / to see options
Downloaded (4 of 4 enabled)
ANTLR v4
1.16 ANTLR Project
Grammar-Kit
2021.1.1 JetBrains
Manage Plugin Repositories...
HTTP Proxy Settings...
Manage Plugin Certificates
Install Plugin from Disk...
Disable All Downloaded Plugins
Enable All Downloaded Plugins
This plugin is for ANTLR v4 grammars and includes ANTLR 4.9.1. It works with 2016.2-2020.3. It should work in other JetBrains IDEs.
syntax highlighting

- 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.

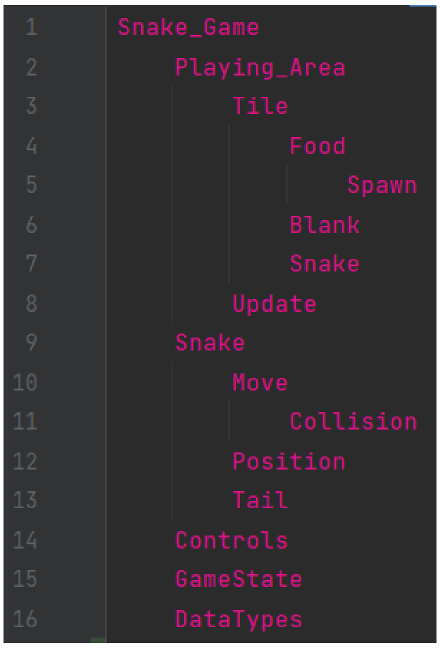
Feature location

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

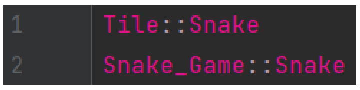
The usage of feature annotations is to map sections of code to functionality of the software. The intent is that this can help developers with feature location. The system of annotations that the HAnS plugin uses is able to map features to any file type and programming language (except languages that 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

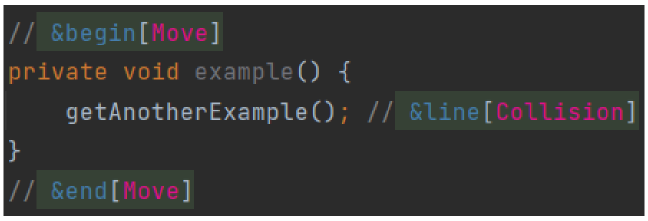
Inside the feature hierarchy model, features with the same name may appear twice or more often. To reference features uniquely the individual feature is pre-extended with its ancestor until the combined feature reference is unique (separated by "::"). This technique is called Least-Partially-Qualified 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

```
Snake_Game
```

HAnS: Helping Annotate Software

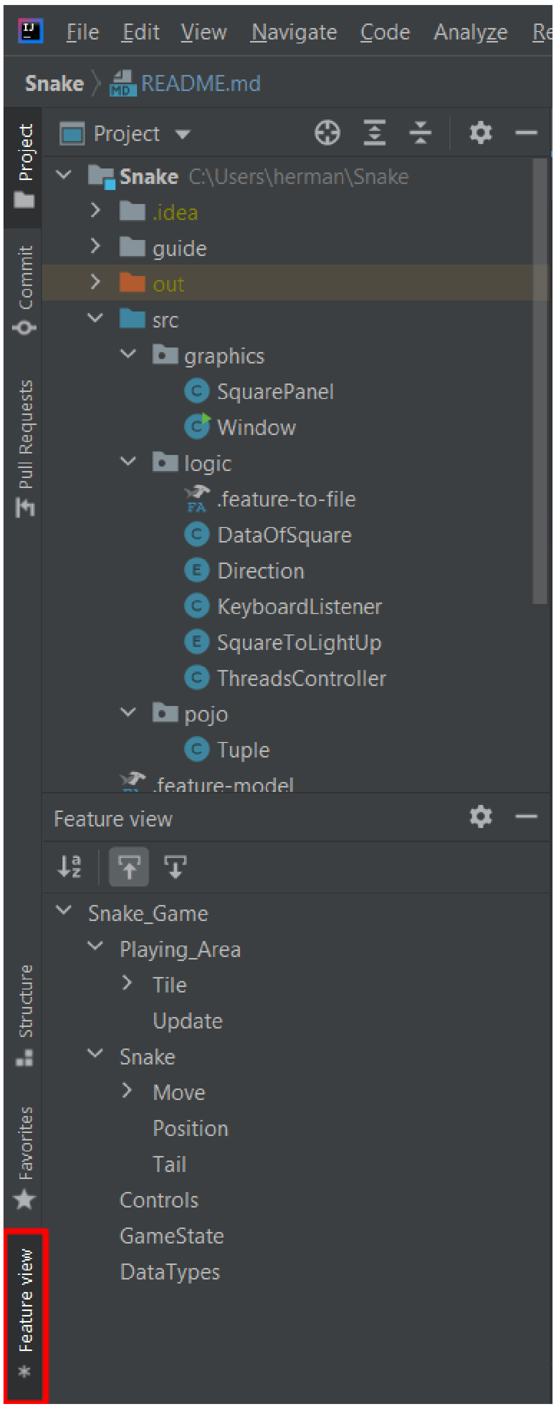
The purpose of HAnS is to enable recording and editing support for feature annotations.

HAnS-text supports:

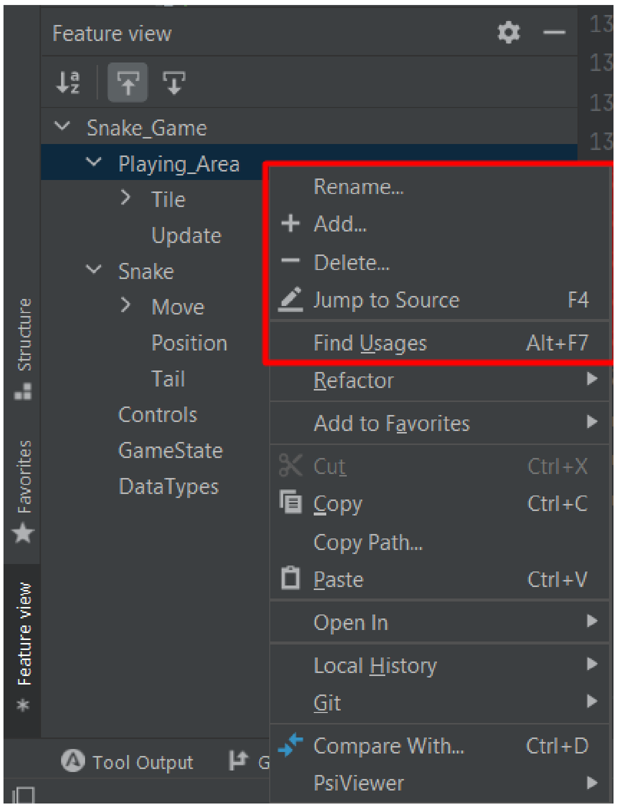
- Embedded Feature Annotations
- Syntax Highlighting
- Code Completion
- Referencing
- Refactoring
- Feature Model View
- Live Templates

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).


File Edit View Navigate Code Analyze Re
Snake README.md
Project
Snake C:\Users\herman\Snake
.idea
guide
out
src
graphics
SquarePanel
Window
logic
.feature-to-file
DataOfSquare
Direction
KeyboardListener
SquareToLightUp
ThreadsController
pojo
Tuple
.feature-model
Feature view
Snake_Game
Playing_Area
Tile
Update
Snake
Move
Position
Tail
Controls
GameState
DataTypes
Project
Commit
Pull Requests
Structure
Favorites
Feature view

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

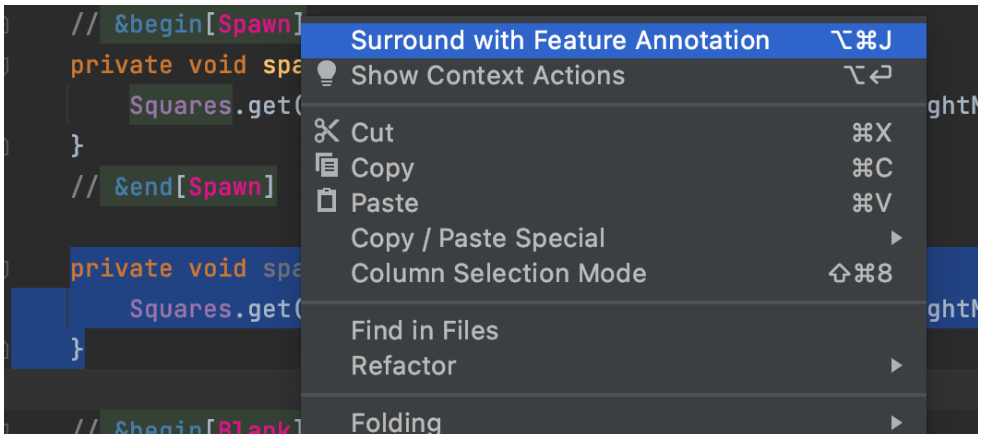
Live templates are used to insert common constructs into your code. HAnS supports a couple of live templates:

- `&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.



```
// &begin[Spawn]
private void spa
    Squares.get(                ghtN
}
// &end[Spawn]

private void spa
    Squares.get(                ghtN
}

// &begin[Blank]
```

Study Group

- Group 1
- Group 2

B.1.1 Task sheet for group 1 in second iteration.

Tasks to complete

During the task, take notes while coding, so you can answer the questionnaire after completing all the tasks.

First part:

Warmup task

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 (bottom left).
- Map the feature `Playing_Area` to the new `.feature-to-folder` file by writing it into the file.

You have now mapped the feature *Playing_Area* to the *graphics* directory.

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`.

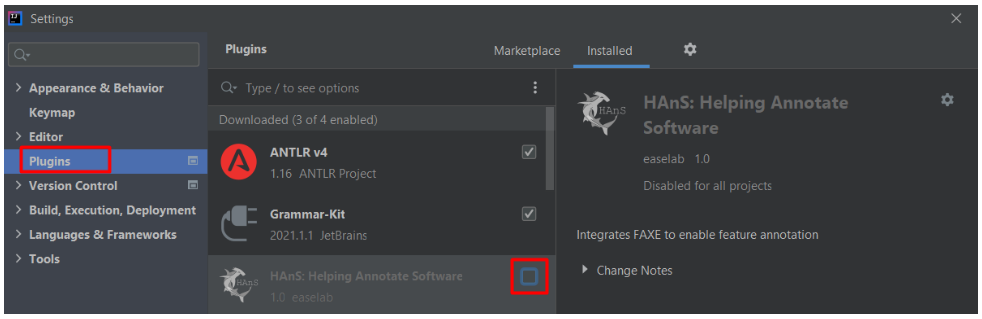
Task 3

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

Second part:

Disable the plugin.

1. Open Plugins in Settings/Preferences. Uncheck the box marked in red in the image below.
2. Click "Apply".



Warmup task

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.

You have now mapped the feature *DataTypes* to the *pojo* directory.

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` is 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.

Answer questions

After the above tasks are completed, fill out the survey.

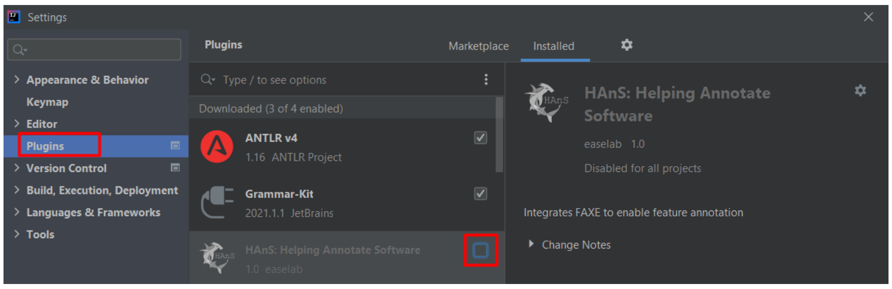
B.1.2 Task sheet for group 2 in second iteration.

Tasks to complete

During the task, take notes while coding, so you can answer the questionnaire after completing all the tasks.

Disable the plugin.

- Open Plugins in Settings/Preferences. Uncheck the box marked in red in the image below.
- Click "Apply".



ANTLR v4 ✓
1.16 ANTLR Project

Grammar-Kit ✓
2021.1.1 JetBrains

HAnS: Helping Annotate Software ☐
1.0 easelab

HAnS: Helping Annotate Software
easelab 1.0
Disabled for all projects

Integrates FAXE to enable feature annotation

First part:

Warmup task

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` File.
- Map the feature `Playing_Area` to the new `.feature-to-folder` file by writing it into the file.

You have now mapped the feature *Playing_Area* to the *graphics* directory.

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`.

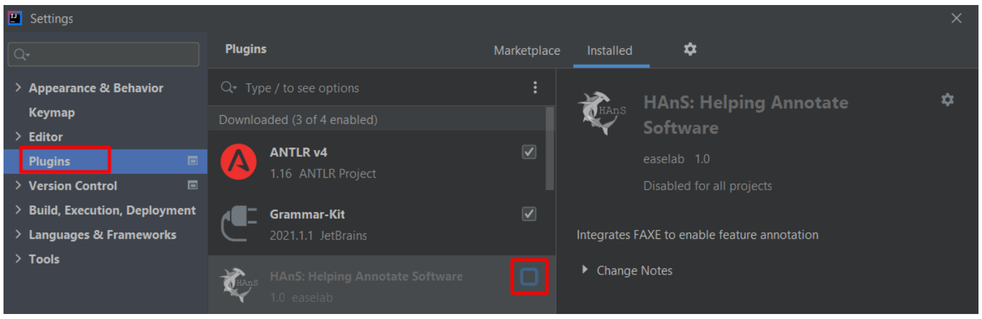
Task 3

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

Second part:

Enable the plugin

1. Open Plugins in Settings/Preferences. Check the box marked in red in the image below.
2. Click "Apply".



Warmup task

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 View tab (bottom left).
- Map the feature `DataTypes` to the new `.feature-to-folder` file by writing it into the file.

You have now mapped the feature *DataTypes* to the *pojo* directory.

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, create methods to enable this.

- *Hint 1: The speed of the game is inverted. It is implemented as a sleep call, so the shorter the sleep, the faster the game.*
- *Hint 2: The difficulty may never be equal to or exceed the speed 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` is defined in the Feature Model.
- Map the feature `Controls` to the file `KeyboardListener.java` in `.feature-to-file` in the *logic* package.

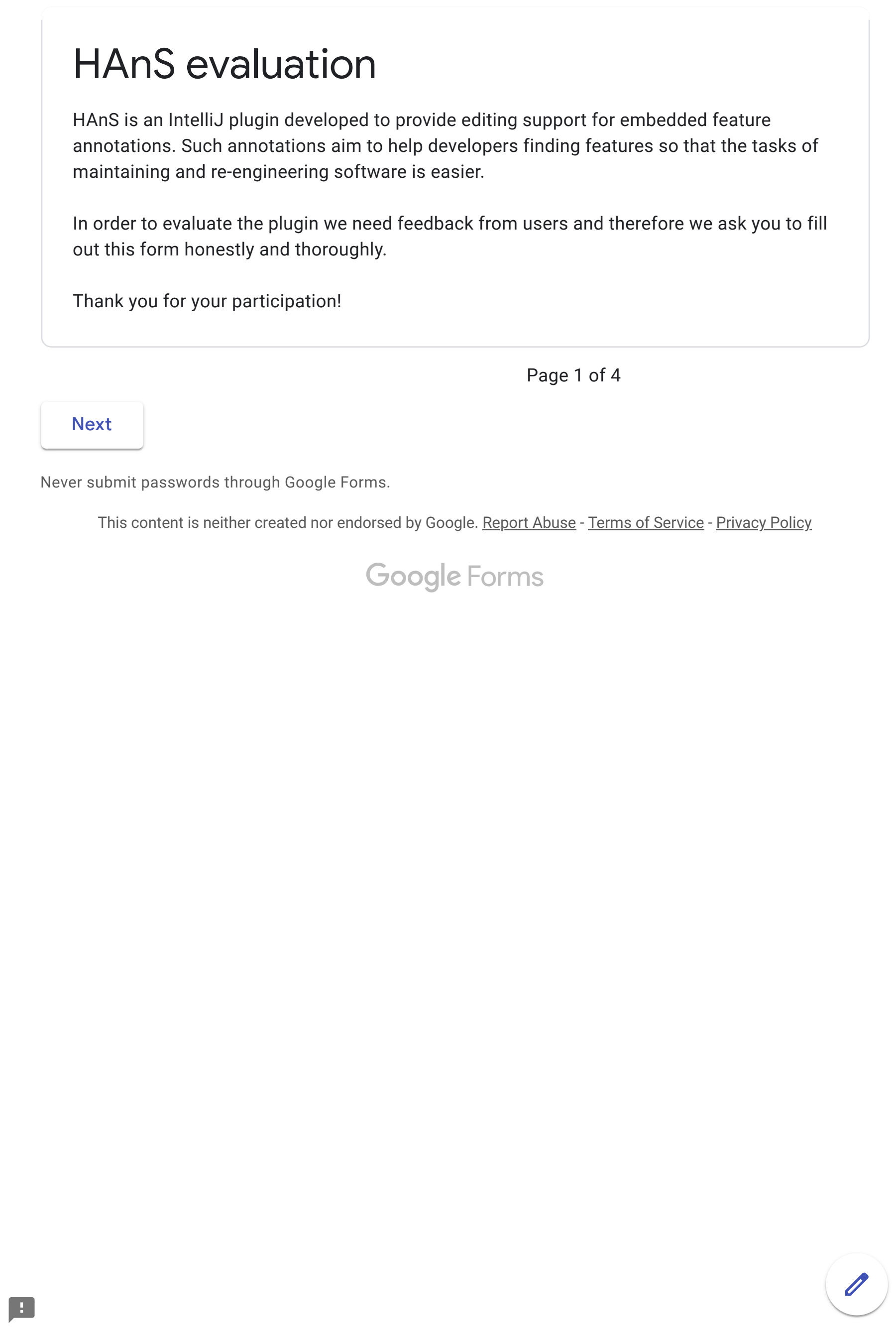
Task 6

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

Answer questions

After the above tasks are completed, fill out the survey.

B.2 Questionnaire for second iteration.



HAnS evaluation

HAnS is an IntelliJ plugin developed to provide editing support for embedded feature annotations. Such annotations aim to help developers finding features so that the tasks of maintaining and re-engineering software is easier.

In order to evaluate the plugin we need feedback from users and therefore we ask you to fill out this form honestly and thoroughly.

Thank you for your participation!

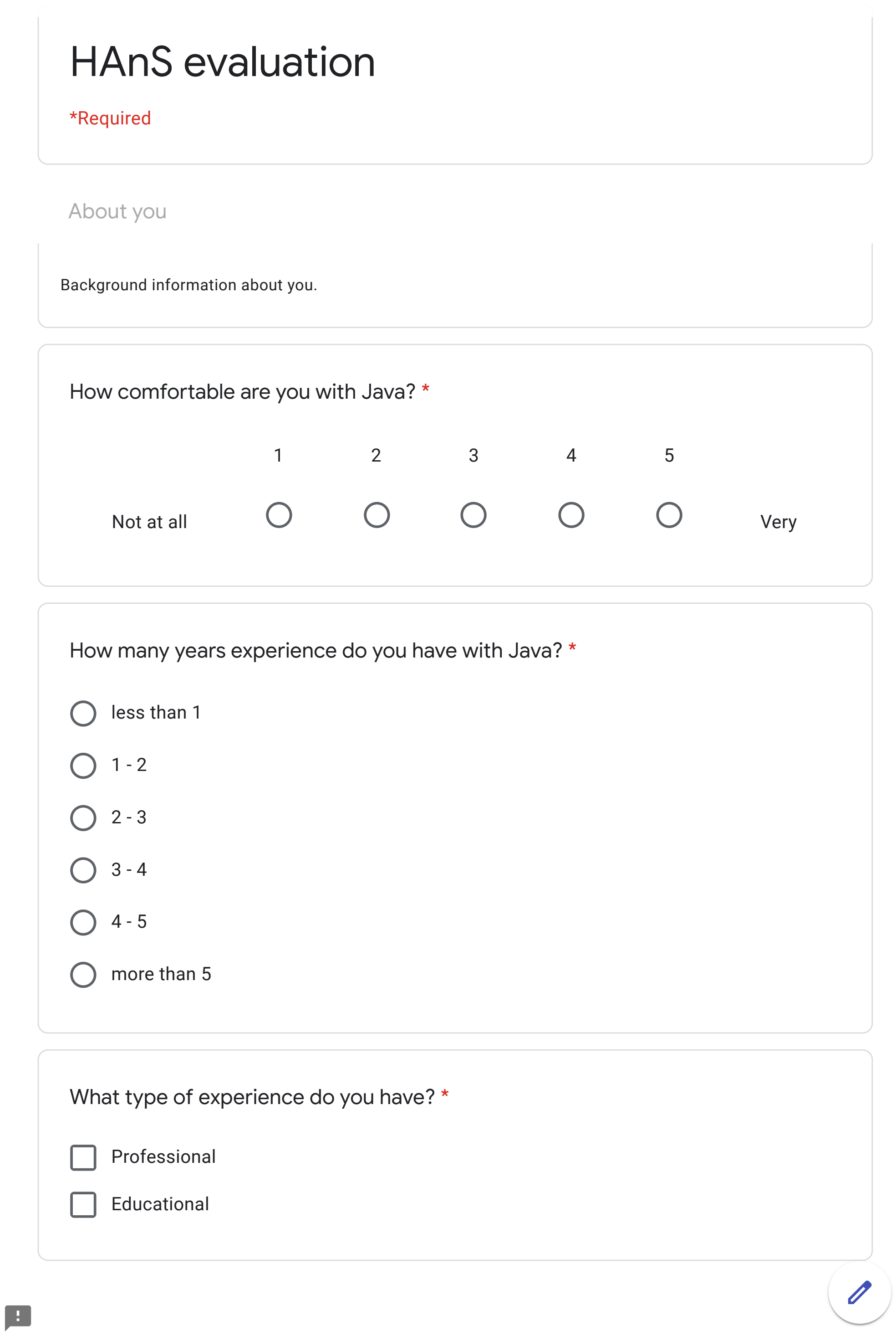
Page 1 of 4

Next

Never submit passwords through Google Forms.

This content is neither created nor endorsed by Google. Report Abuse - Terms of Service - Privacy Policy

Google Forms



HAnS evaluation

*Required

About you

Background information about you.

How comfortable are you with Java? *

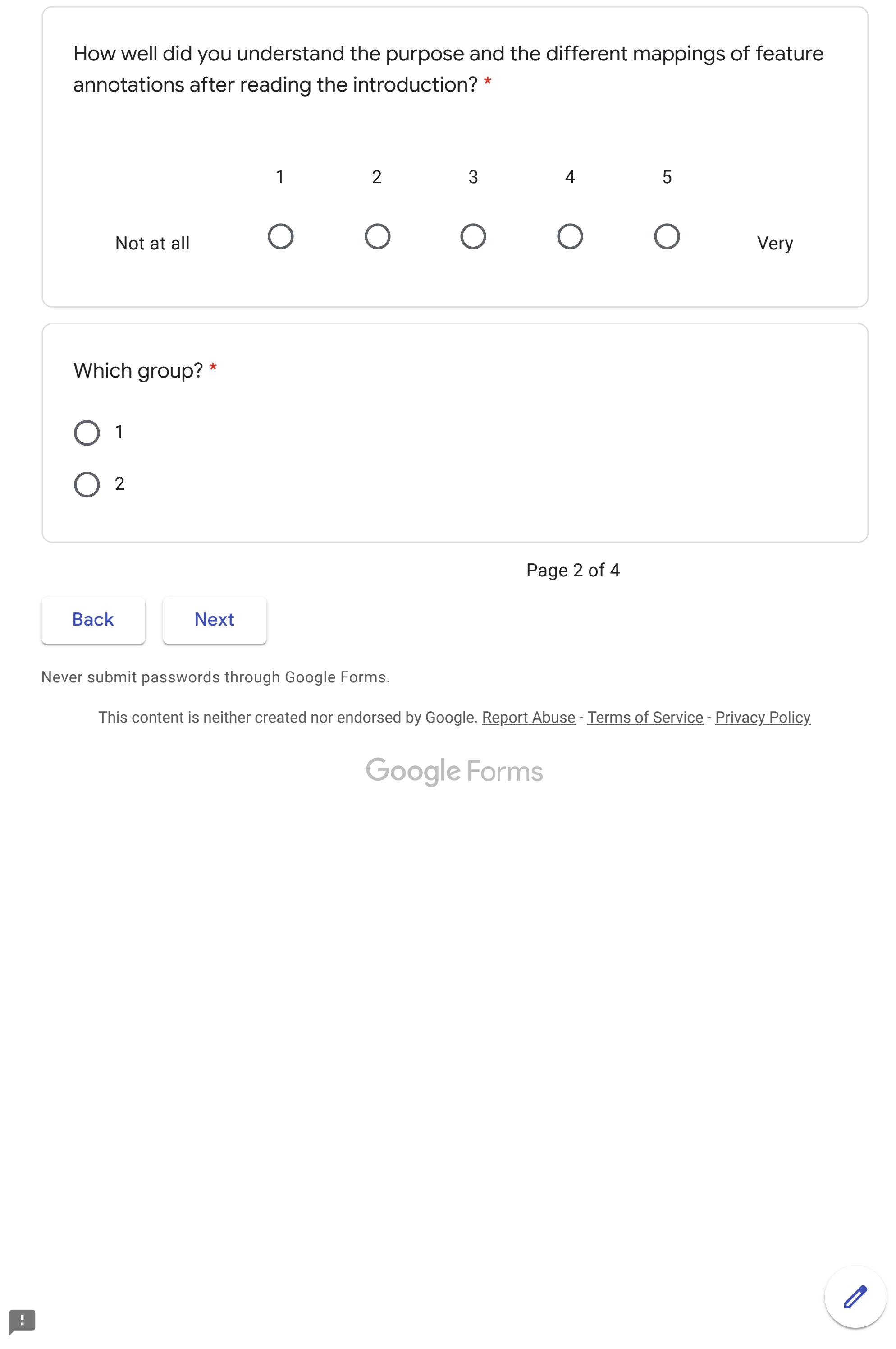
	1	2	3	4	5	
Not at all	○	○	○	○	○	Very

How many years experience do you have with Java? *

- ○ less than 1
- ○ 1 - 2
- ○ 2 - 3
- ○ 3 - 4
- ○ 4 - 5
- ○ more than 5

What type of experience do you have? *

- ☐ Professional
- ☐ Educational



How well did you understand the purpose and the different mappings of feature annotations after reading the introduction? *

	1	2	3	4	5	
Not at all	○	○	○	○	○	Very

Which group? *

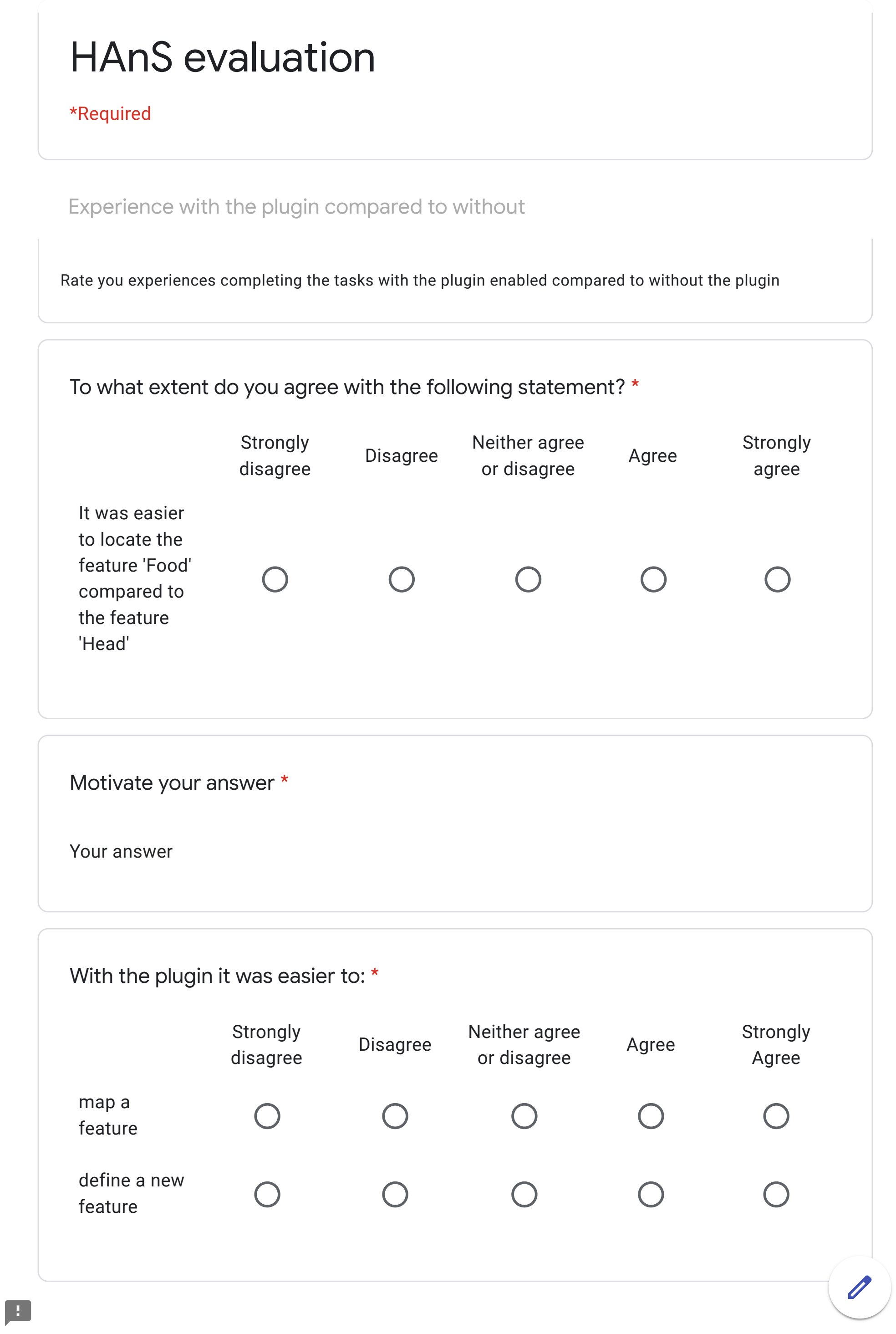
○ 1

○ 2

Page 2 of 4

Never submit passwords through Google Forms.

This content is neither created nor endorsed by Google. Report Abuse - Terms of Service - Privacy Policy



HAnS evaluation

*Required

Experience with the plugin compared to without

Rate you experiences completing the tasks with the plugin enabled compared to without the plugin

To what extent do you agree with the following statement? *

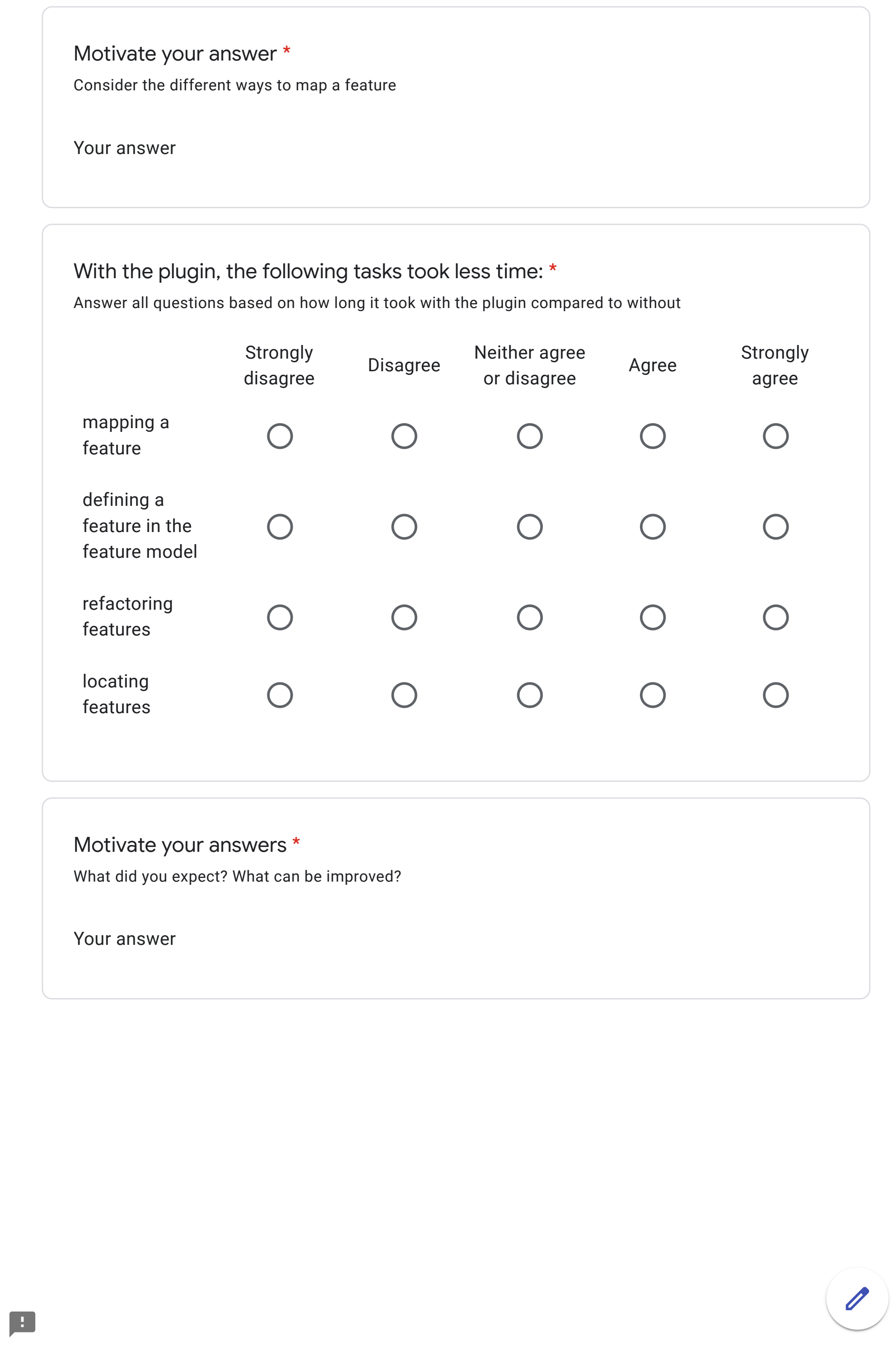
	Strongly disagree	Disagree	Neither agree or disagree	Agree	Strongly agree
It was easier to locate the feature 'Food' compared to the feature 'Head'	○	○	○	○	○

Motivate your answer *

Your answer

With the plugin it was easier to: *

	Strongly disagree	Disagree	Neither agree or disagree	Agree	Strongly Agree
map a feature	○	○	○	○	○
define a new feature	○	○	○	○	○



Motivate your answer *

Consider the different ways to map a feature

Your answer

With the plugin, the following tasks took less time: *

Answer all questions based on how long it took with the plugin compared to without

	Strongly disagree	Disagree	Neither agree or disagree	Agree	Strongly agree
mapping a feature	○	○	○	○	○
defining a feature in the feature model	○	○	○	○	○
refactoring features	○	○	○	○	○
locating features	○	○	○	○	○

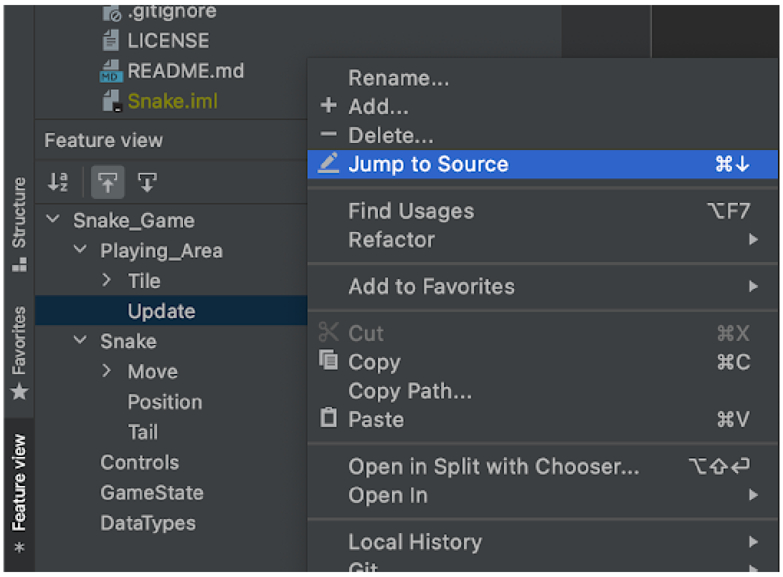
Motivate your answers *

What did you expect? What can be improved?

Your answer

L

To what extent do you agree with the following statements regarding the Feature Model View? *

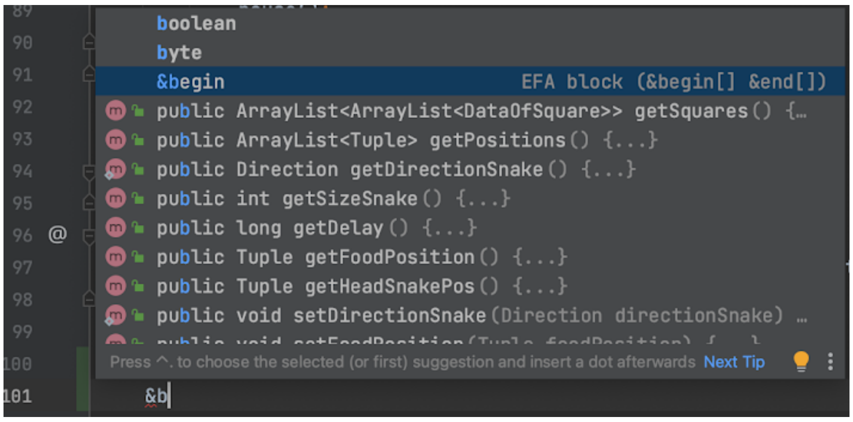


	Strongly disagree	Disagree	Neither agree or disagree	Agree	Strongly Agree
I would like to use the 'Feature Model View' frequently.	○	○	○	○	○
The 'Feature Model View' unnecessarily complex.	○	○	○	○	○
The 'Feature Model View' is easy to use.	○	○	○	○	○
The various functions in the 'Feature Model View' are well integrated	○	○	○	○	○
I imagine that most developers would learn to	○	○	○	○	○

use the 'Feature Model View' very quickly					
The 'Feature Model View' is cumbersome to use.	○	○	○	○	○
I feel very confident using the 'Feature Model View'	○	○	○	○	○

Did you find Live templates to map features to code useful? [Second image below] *

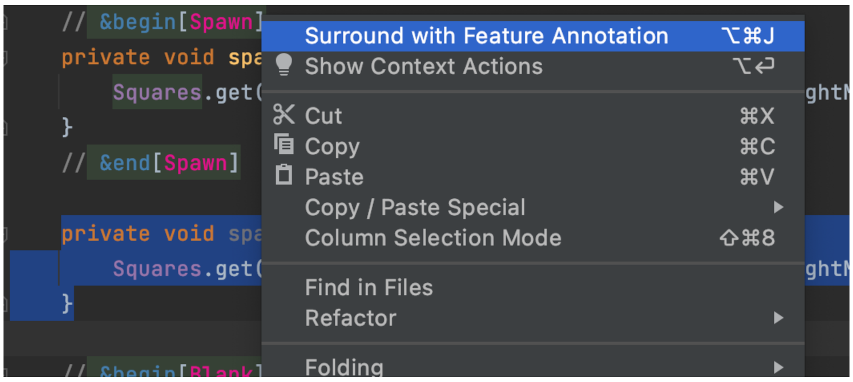
Typing &b, &l or &e to get a suggestion or marking a piece of code and surrounding it with a feature annotation (ctrl-alt-J).



```
boolean
byte
&begin                                               EFA block (&begin[] &end[])
public ArrayList<ArrayList<DataOfSquare>> getSquares() {…
public ArrayList<Tuple> getPositions() {...}
public Direction getDirectionSnake() {...}
public int getSizeSnake() {...}
public long getDelay() {...}
public Tuple getFoodPosition() {...}
public Tuple getHeadSnakePos() {...}
public void setDirectionSnake(Direction directionSnake) …
Press ^. to choose the selected (or first) suggestion and insert a dot afterwards Next Tip
&b
```

Your answer

Another type of live template



```
// &begin[Spawn]
private void spa
    Squares.get(
}
// &end[Spawn]

private void spa
    Squares.get(
}

// &begin[Blank]
```

Page 3 of 4

Back Next

Never submit passwords through Google Forms.

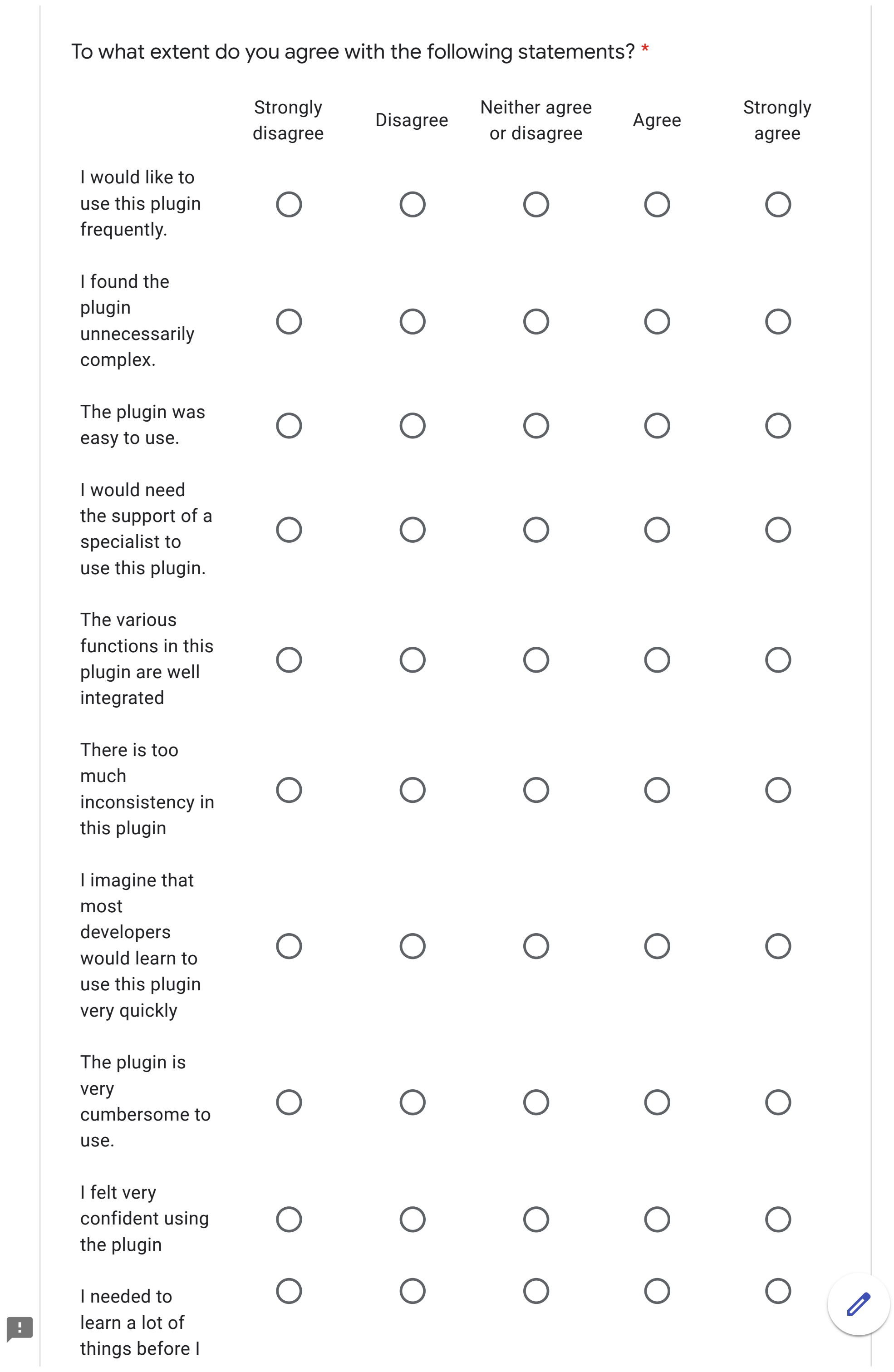
This content is neither created nor endorsed by Google. Report Abuse - Terms of Service - Privacy Policy

Google Forms

HAnS evaluation

*Required

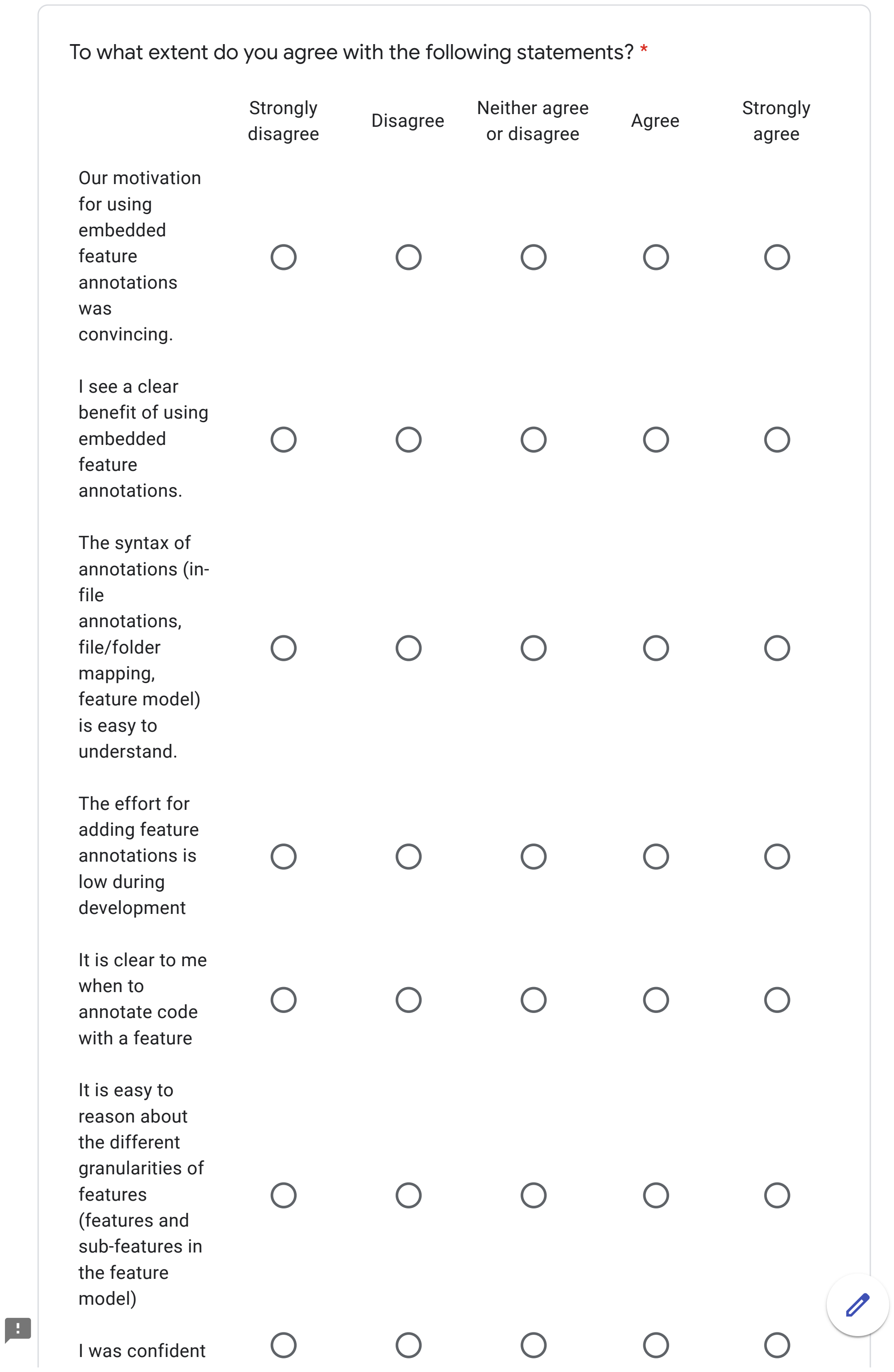
Overall evaluation of the plugin



To what extent do you agree with the following statements? *

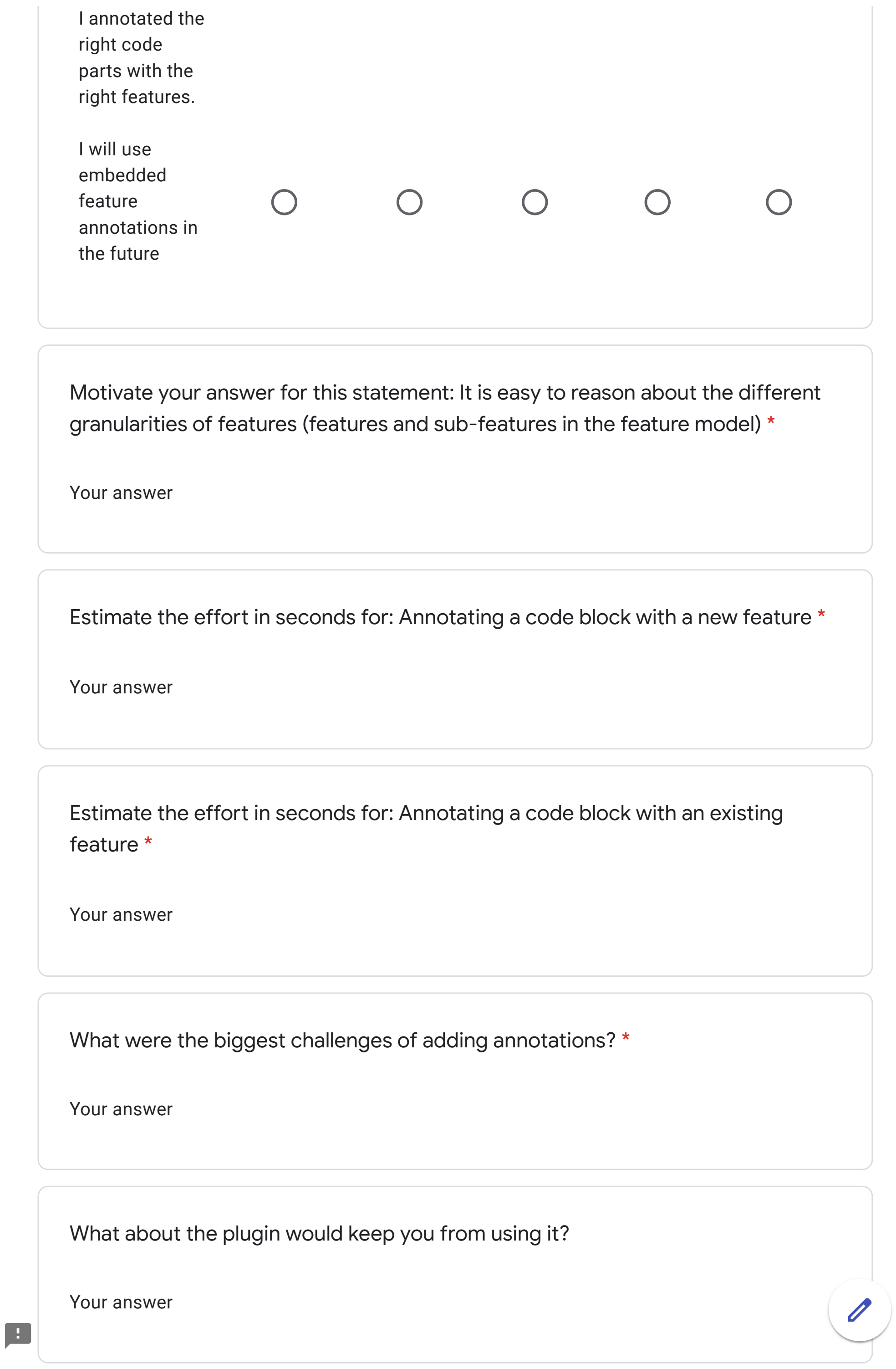
	Strongly disagree	Disagree	Neither agree or disagree	Agree	Strongly agree
I would like to use this plugin frequently.	○	○	○	○	○
I found the plugin unnecessarily complex.	○	○	○	○	○
The plugin was easy to use.	○	○	○	○	○
I would need the support of a specialist to use this plugin.	○	○	○	○	○
The various functions in this plugin are well integrated	○	○	○	○	○
There is too much inconsistency in this plugin	○	○	○	○	○
I imagine that most developers would learn to use this plugin very quickly	○	○	○	○	○
The plugin is very cumbersome to use.	○	○	○	○	○
I felt very confident using the plugin	○	○	○	○	○
I needed to learn a lot of things before I	○	○	○	○	○

could get going
with this plugin



To what extent do you agree with the following statements? *

	Strongly disagree	Disagree	Neither agree or disagree	Agree	Strongly agree
Our motivation for using embedded feature annotations was convincing.	○	○	○	○	○
I see a clear benefit of using embedded feature annotations.	○	○	○	○	○
The syntax of annotations (in-file annotations, file/folder mapping, feature model) is easy to understand.	○	○	○	○	○
The effort for adding feature annotations is low during development	○	○	○	○	○
It is clear to me when to annotate code with a feature	○	○	○	○	○
It is easy to reason about the different granularities of features (features and sub-features in the feature model)	○	○	○	○	○
I was confident	○	○	○	○	○



I annotated the right code parts with the right features.					
I will use embedded feature annotations in the future	○	○	○	○	○

Motivate your answer for this statement: It is easy to reason about the different granularities of features (features and sub-features in the feature model) *

Your answer

Estimate the effort in seconds for: Annotating a code block with a new feature *

Your answer

Estimate the effort in seconds for: Annotating a code block with an existing feature *

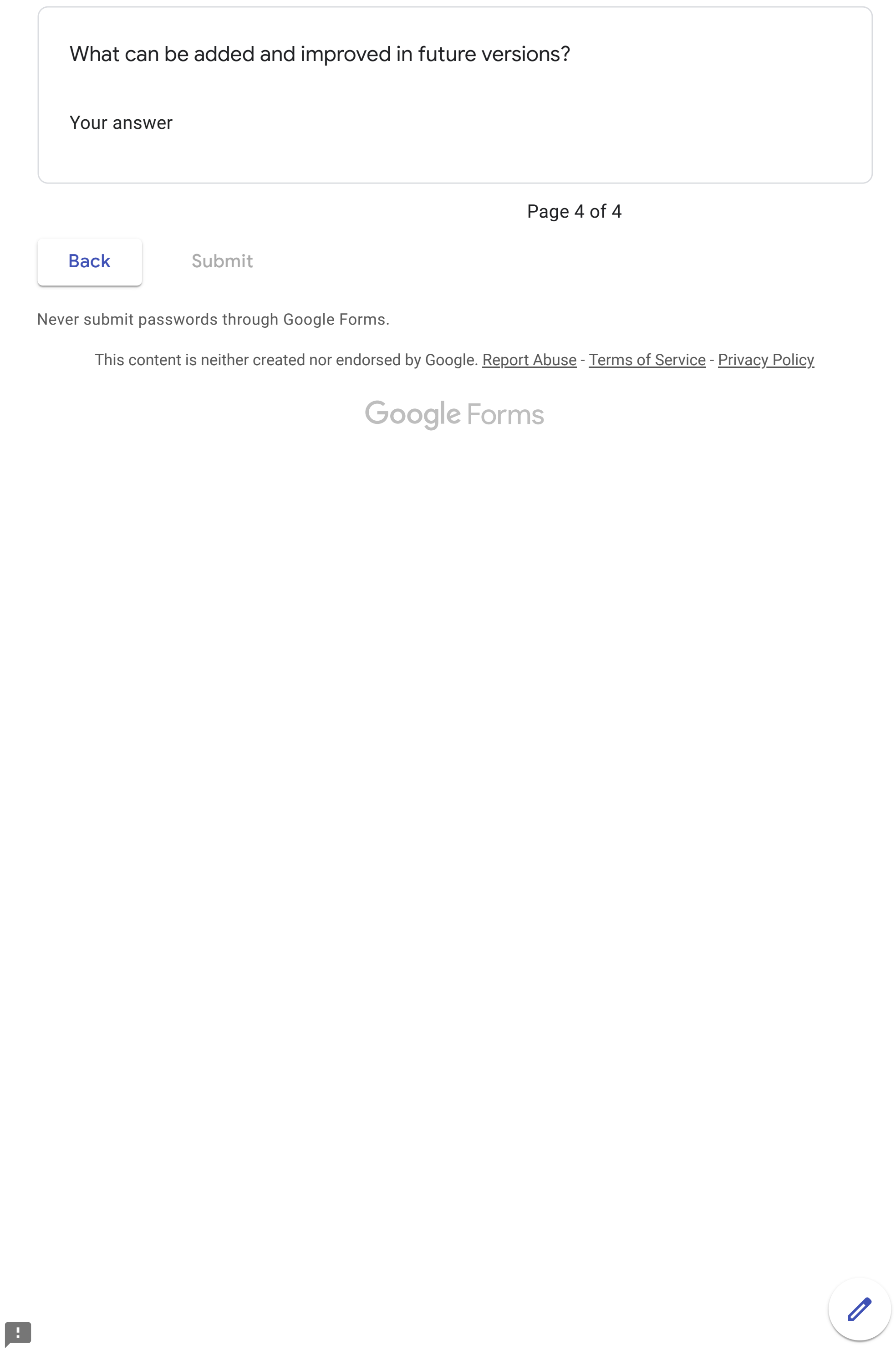
Your answer

What were the biggest challenges of adding annotations? *

Your answer

What about the plugin would keep you from using it?

Your answer



What can be added and improved in future versions?

Your answer

Page 4 of 4