Language Server for Feature Annotations: Integration with commonly used IDEs

Tim Schulz

Bachelor‘s Thesis – December 30, 2024

Chair of Software Engineering

Supervisor:	Prof. Dr. Thorsten Berger
Advisor:	Johan Martinson

Abstract

As a developer, you spend a lot of time searching for the location of features in code. This task needs to be done in feature-oriented programming when maintaining code bases or when customizing a software project. To reduce the time spent searching, feature location techniques have been developed. The automatic approaches lack the robustness needed for correct feature mapping. To address the challenge, domain-specific languages were proposed to annotate the use of features in the code itself. To properly use the annotations tool, support is required. Existing tool support is depending on single IDEs. To broaden the available tool support, Taymour Kiwan and I designed a language server-based implementation of HAnS to provide support for writing and navigating feature annotations. While decoupling the functionality of HAnS from its IDE, we were limited by the LSP, which makes it impossible to implement the same amount of functionality the HAnS plug-in possesses while being IDE independent.

Contents

1 Introduction

The goal of software development is to create software that meets the expectations of customers and users. To achieve this, software is carefully planned and designed. A common practice is to design code in a way that it can be reused, often in the form of libraries or frameworks [22]. When large companies create software, they often use a process called "clone-and-own", where a significant portion of the code is cloned from a previous project and modified to fit the current one. This strategy is cost-effective and requires less specialized knowledge compared to creating everything from scratch, however it does not scale with the number of variants [9, 0].

In order to effectively manage and adapt these large and complex codebases, a project is split into smaller independent parts, which share specific characteristics known as features. They are used by software developers to describe specific functionalities of a software project [18]. However, these features, spread across different locations of the software project, often remain unrecognizable when looking solely at the codebase. This creates a problem during maintenance, as updating or fixing bugs requires a software engineer to be very familiar with the codebase. If the engineer does not know the code well, they must spend time searching through documentation or the code itself, which can be very time-consuming, especially in larger projects or when much of the code is copied from another project [18]. Actually, feature location is one of the most time consuming often occurring problem [4, 26, 19, 21]. Automated techniques often fall short in capturing nuances of software development, leading to imprecise annotations [20]. Therefore, Ji et al. (2015) suggest that developers should annotate code manually during the development process, because the time spent writing the annotations reduces the time to necessary find assets that belongs to a corresponding feature, which is also known as feature mapping [10].

The significant problem with manual annotation is that developers often neglect to document their code adequately [24], which leads to difficulties in maintenance, understanding and collaboration, especially in large projects. Therefore, developers require tool support, that assists in annotating their code. One such tool is HAnS (Helping Annotate Software), an IntelliJ IDE plugin, which aids developers in feature mapping. It provides functionalities for recording feature locations, i.e. mapping features to code, files and folders, browsing features, visualizing them, code completion and refactoring annotations [13]. The grammar that HAnS uses for these annotations is based on the work by Schwarz et al. , who formulated a syntax for embedded feature annotations [20], which is either *&begin[]* and *&end[]* tags to surround code that corresponds to a specific feature, or *&line[]* tags for single lines. It's important to remember that tags must be as comments in the code, not to interfere with the actual code [13, 8].

While HAnS provides developers with tool support for recording embedded feature annotation in the IntelliJ IDE, we want to further encourage developers to record feature locations during development by making HAnS IDE independent. This usage limitation is significant for several reasons. First, it hinders the adoption of the plugin in diverse development environments, thereby limiting its user base. Second, it stifles collaboration in teams where members might prefer different development tools. Third, it restricts the plugin's potential for integration with emerging IDEs and text editors. Addressing this problem is crucial as it enables broader dissemination and utilization of the plugin's features, promoting a more inclusive and efficient development ecosystem [1].

In this thesis, we engineer an implementation of the HAnS plugin as an language server. Transitioning from a single-IDE dependency to a versatile language server framework not only expands the usability of the plugin but also enhances the developer experience across various IDEs and text editors. Therefore, it is necessary to decouple the plugin from its native IDE. This will be done by using the existing plugin's grammar as an inspiration to develop a robust LSP [13]. Additionally, one challenge that has already been identified is that implementing the LSP might not be sufficient to solve all our problems, as different IDEs handle LSPs differently. For example, using an LSP in Visual Studio Code requires an extension to handle the communication. Another challenge is the need to run our LSP concurrently with an existing language server, as our LSP won't offer language support on its own [11].

1.1 Contribution

The purpose of this research is to reduce the time that is spend searching for specific code, while maintaining, searching for code that belongs to a feature when using feature oriented programming or merging code. The time spent only increases with the size of the code base and its complexity. A lot of large and complex code bases can be found in industry. To reduce the time spent searching, domain specific languages have been proposed to mark code using manual annotations. Those annotations only help when used correctly while writing or maintaining code. To help with this task the correct tooling is necessary. Due to ever growing codebases and the lack of IDE independent tooling we propose a solution to the problem.
Furthermore we bring tooling for feature oriented programming to more IDEs, thereby improving the ability to manage features, while ensuring scale ability and adaptability of code bases. With the development of the HAnS-LSP version we gain insight in how the LSP work, how to implement an already existing extension as a language server and what limitations does a language server have, that hinders implementation of extensions.

1.2 Organization of this Thesis

In this thesis I want to answer whether a language server is a good way to implement functionalities of an extension to make them IDE-independent and furthermore I want to ask if our implementation of an extension as a language server is well integrated and if the usability of the server is not significantly worse than the extension, when using the Language Server Protocol for the communication between the language server and an IDE. The insights we derive can then be used to further improve our implementation of the HAnS extension as an LSP or for developing other LSP-based extensions. First we take a look at the background of feature location and tool support. Then I present the language server we built and we are going over the result of a user study we did to determine the usability of our implementation. Since in this thesis we take a look at two iterations of the development of the language server the implementation and the test are split into two parts each. in the end we conclude by taking a look at the results of our user studies and our experience developing the server to determine if the sever is well implemented and what can be improved. I will also try to answer the question: “Does it make sense to use an LSP to implement features that an extension would normally provide?”.

2 Background, Motivation, and Related Work

To better understand the challenges and solutions in software development, it's important to explore the concepts of features, feature location existing tool support and Language Server Protocol as they are the foundation of our development practice.

2.1 Feature

In software development a feature refers to a distinct functionality that delivers specific value or capability to the user [25]. Features can be simple functions, like a "search" function, or complex systems, like "user authentication." They represent high-level requirements that the software is expected to support, and are often used as a foundation for planning, organizing, developing, and managing the codebase.

2.2 Feature location

As for feature location, it's the process of identifying the specific parts of a software system's codebase that implement a particular feature. It involves tracing the functionality described at a high level down to the actual code responsible for that behavior [6]. Locating features is one of the most time consuming steps in maintaining or reusing a software project [13].

2.3 Tool support

Tool support is the use of software tools to assist developers in various software development tasks. These tools help with improving productivity, accuracy, and efficiency by automating repetitive tasks[15]. There has been significant work in feature location and annotation in software development. For example, Robillard and Murphy [18] discussed the importance of tools that help developers understand and navigate large codebases. Their work highlighted the challenges faced by developers when dealing with unfamiliar code and the need for tools that provide better code comprehension support.

2.4 JSON-RPC

JSON-RPC is a remote procedure call (RPC) protocol, which uses the JavaScript object Notation (JSON) format do deliver the calls [16]. A typical message sent using the JSON-RPC protocol consists of:

1. version of the protocol
2. the method that is called
3. the parameters the method is called with
4. id of the message

2.5 Language Server Protocol

Language Server Protocol (LSP) makes HAnS IDE independent, the same way LSP has been implemented for many language features such as auto completion and code navigation. These features had to be individually implemented for each IDE. A more effective solution is to decouple these features from the IDE entirely. Therefore, Microsoft developed the Language Server Protocol (LSP), which defines a standard for communication between a server that implements the features and the IDE using JSON-RPC [11]. The integration of LSPs has revolutionized how development tools interact with coding languages, enabling a decoupled yet highly integrated development experience.
Additionally, more recent studies have focused on the use of LSP to decouple language-specific features from IDEs. For instance, Amann et al. [1] demonstrated the benefits of using LSP for providing consistent development experiences across different environments. Their research showed that LSP could effectively standardize feature implementation, reducing the need for redundant development efforts across multiple platforms. However the LSP standard does not define how LSPs should be integrated in tools, this is the reason why, for example some IDEs require an extension.

2.6 LSP4J

In our implementation of the language server, we are using the LSP4J framework. It is an implementation of the LSP standard and the Debug Adapter Protocol for Java by the Eclipse Foundation. The framework implements the types and the JSON-RPC communication which are defined in the LSP[12]. It provides us with the basic structure of a language server and interfaces for our implementation of language features.

2.7 ANTLR

Most language server need a way to parse text documents or other files in order to work with them. ANTLR generates Parser, Lexer and tree walkers from a single context-free grammar [17].

2.8 Motivation

Despite these advances of having tool support, there has been limited research specifically addressing the adaptation of feature mapping plugins, such as HAnS, as a language server. This would make HAnS available to a broader range of developers and allow for more research in feature mapping.

2.9 Related work

To track features in code, an assortment of possible solutions was proposed. The automatic location techniques are not usable in practice, since they are prone to errors. Hence, research is looking for manual ways to trace features [10]. One way this may be done is through code annotation using embedded feature annotations.
One tool to aid developers was presented by Martinson and Jansson et al. [13]. This tool, named "HAnS", maps features to code, visualizes the feature model and helps annotate features by providing editing support. In their study, they concluded that the tool reduces the amount of annotation mistakes [13]. The used annotations is based on the notation Schwarz et al. introduced, which consists of either *&begin[]*, *&end[]* or *&line[]* annotations for in code references and specific *.feature-to-file* and *.feature-to-folder* files which map features to files or folders respectively [20]. The notation uses a *.feature-model* where the names of the features and there structure is defined in a notation, which is based on the Clafer modeling language [3]. However, the grammar, which defines feature annotations, is based on FAXE, a lightweight feature annotation extraction tool. FAXE proposed a set of ANTLR grammars to parse files to extract those annotations [20].
The HAnS tool, however, is exclusively available for the JetBrains IDE suite, which reduces the number of developers that would benefit from such a tool. There have been other tool implementations that help with feature location and related problems, namely, FLORiDA, Feature Dashboard and FeatRacer, which are particularly valuable in feature-oriented development [2, 7, 14]. Similarly to HAnS, these tools also suffer from dependence on a single IDE.

3 Methodology

In this section we discuss the methodology used to implement the language server version of the HAnS plugin and how we implemented features from HAnS using the Language Server Protocol. in this chapter we also discuss the strategy used to determine the usability of the language server. During the implantation it was my intention to achieve two goal:

- RO 1: implement an LSP version of the HAnS IntelliJ plugin
- RO 2: verify the usability of the designed tool

The complete source code and everything regarding the user study can be found in our GitHub[1].

3.1 Research design

To reach and verify our research objectives we used an iterative engineering study approach similar to design science. In this thesis we look at the first two iterations of the design process of the HAnS language server. Each iteration consists of 4 steps.

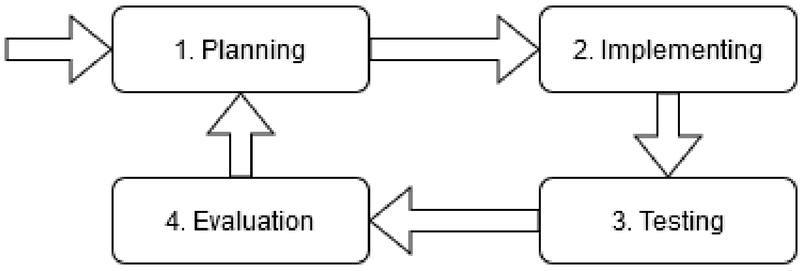


Figure 3.1: Design process

[1]`https://github.com/isselab/HAnS-LSP`

1. Planning: Planning the next part of the implementation, while also considering the evaluation of the last iteration.
2. Implementation: Implementing the next part of the project.
3. Testing: Testing the implementation through unit-tests and an integration-test.
4. Evaluating: Evaluating the implementation, using the results of the previous testing phase.

3.2 Tools and technologies used

JAVA: we used Java as the language to implement the language server, because the other used tools support it as a programming language and is high abstraction makes it easy to implement complex language features.

Git(Github): since the language server is developed in a a team, we are using a version control tool to enable easy collaboration.

LSP4J: LSP4J provides us with the necessary interfaces and types to implement a language server, and it handles the JSON-RPC based connection between the server and an IDE, which makes it easier to focus on the language features we want to implement.

ANTLR: to reduce the errors, we could introduce when building our own parser, we used ANTLR to build parser, lexer and tree walker.

Yo Code: Yo Code is an extension and customization generator provided by Microsoft. It generates extension templates to help development of extension for VS Code and Visual Studio.

Maven: to use the language server we developed within an VS Code extension it is necessary to build a Fat JAR to encompass all the required code to run and ship the language server

System Usability Scale: The System Usability Scale (SUS) was used to measure how well the tooling integrates into programming workflows. It is a way to measure the Usability of a system. Usability is not some concrete value, since it is only perceived by the user and depends on the context the system is used in [5]. The System Usability Scale works by asking participants to rate how much they agree with 10 statements on a scale from one to five, with one being strongly disagree and five being strongly agree.

Using the answers to the questions, a score can be calculated. The score is in the range of 0 to 100.

3.3 Development of the language server

The main part of the project consisted of implementing the most important features of HAnS into an LSP server. The features of HAnS we implemented are:

- Parsing *.feature-model*, *.feature-to-file* and *.feature-to-folder* files
- Parsing files containing code and annotations
- Code completion for writing annotations
- Finding annotations in code

Since an LSP server has limited ways of interacting with an IDE compared with a plugin, we had to think about how we could integrate the features. Our goal was to only use the methods that are part of the LSP protocol to achieve the most IDE independence possible during the course of this thesis. The LSP can be extended to have custom methods. The extension of the LSP requires implementing methods on the server and the IDE side, which would mean we would have to write an extension to almost every IDE. Writing an extension specifically for every IDE is what we want to avoid by implementing HAnS as a language server.
The implementation was done by using the frameworks and classes provided by LSP4J. To manage dependencies and libraries required for implementing the LSP, we utilized Maven as the build tool. Using Maven also allowed us to build the Fat JAR we require for the extension and helps with reducing the complexity of manually handling resources. The parsing of documents was handled through the use of parser and lexer generated by ANTLR. The grammars used to generate the parser are based on the ones suggested by FAXE.

3.4 Development of the Visual Studio extension

In order to test and use the language server we need a way to connect it with an IDE. We chose to test the language server in VS Code since it is according to the Stack Overflow Developer survey 2024 the most used IDE by developers [23]. To use an LSP, an extension is needed, that's why we implemented a simple VS Code extension, which launches a jar of our Server using the Visual studio code API

3.5 Testing and validation

To ensure that the entire system functions as expected, we chose a testing approach consisting of autonomous unit tests and integration tests. This approach is based on the the V-model. The V-Model is a software development process that extends the traditional waterfall model by showing the parallel execution of development and testing phases, where each development stage corresponds to a specific testing phase, ensuring systematic validation and verification throughout the life cycle [27]. We focused only on system integration, integration-tests and unit-tests, since the acceptance test has been already verified by the HAnS development team [13].

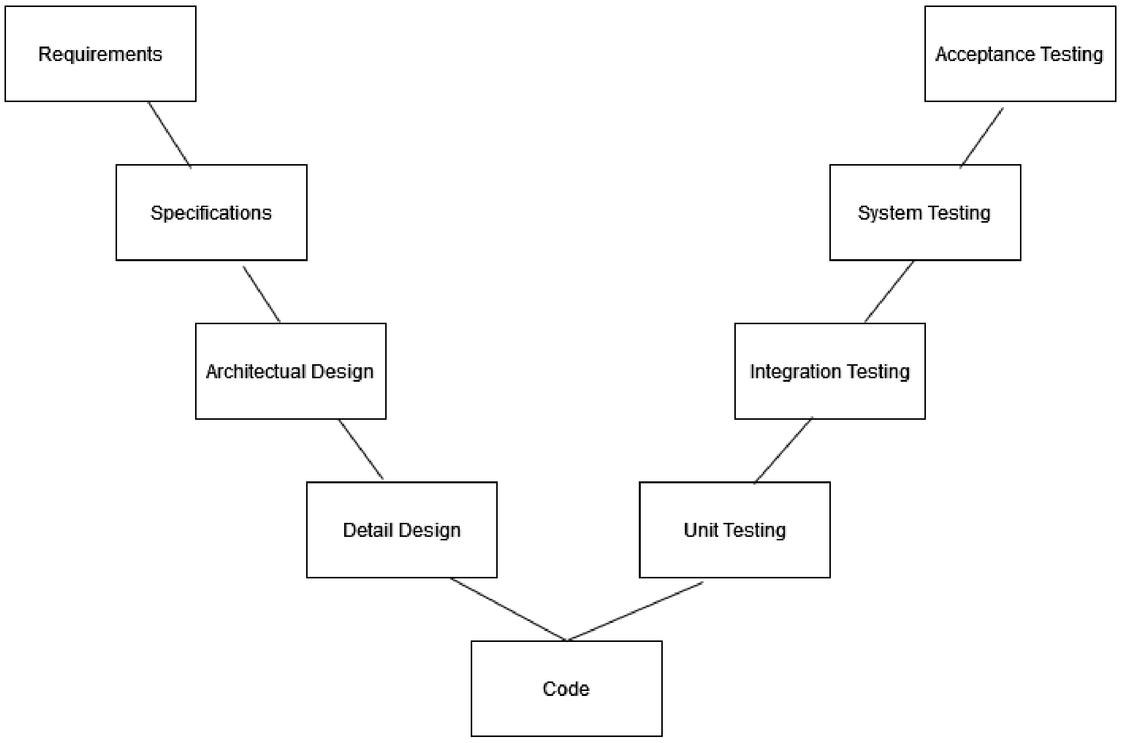


Figure 3.2: V-Model.

The verification of the code through unit test and the test strategy used is discussed in the thesis written by Taymour Kiwan, who collaborated with me on the development of the server. This thesis only focuses on the integration of the tool.
To test the integration and usability of the developed language server and the extension used to launch it, a user study was designed by me. The user study conducted on computer science students at our university. In the study the participants were split into two groups. Both groups are given the same six tasks to complete and the same

questionnaire to answer. The only difference between the groups is that the first group uses the extension and the LSP for the first tree tasks while the second group uses the extension for the second half. This design is called a cross-over study, which is commonly used in research. The tasks consist of some simple annotation and code browsing tasks in an existing project. The task and the project used are similar to the one used in the HAnS study[13] and are already annotated to work with HAnS. The questions we gave the participants consist of three parts. First, some general information about the participant; second, some questions about whether they think that the tool we provide helps with writing annotations and on whether they think it helps with browsing code. The last part contains general questions about the integration of the tool and the System Usability Scale, which we use to determine the usability of the tool.

4 Implementation

In this chapter I will talk about our implementation of a Language Server and the features we consider important for the efficient use of the feature annotation domain-specific language used by HAnS.

4.1 Structure of the language server

To ensuring modularity and ease of extension, the HAnS-LSP server is following the structure provided by the LSP4J framework. The Structure of the Language Server can be seen in Figure 4.1.

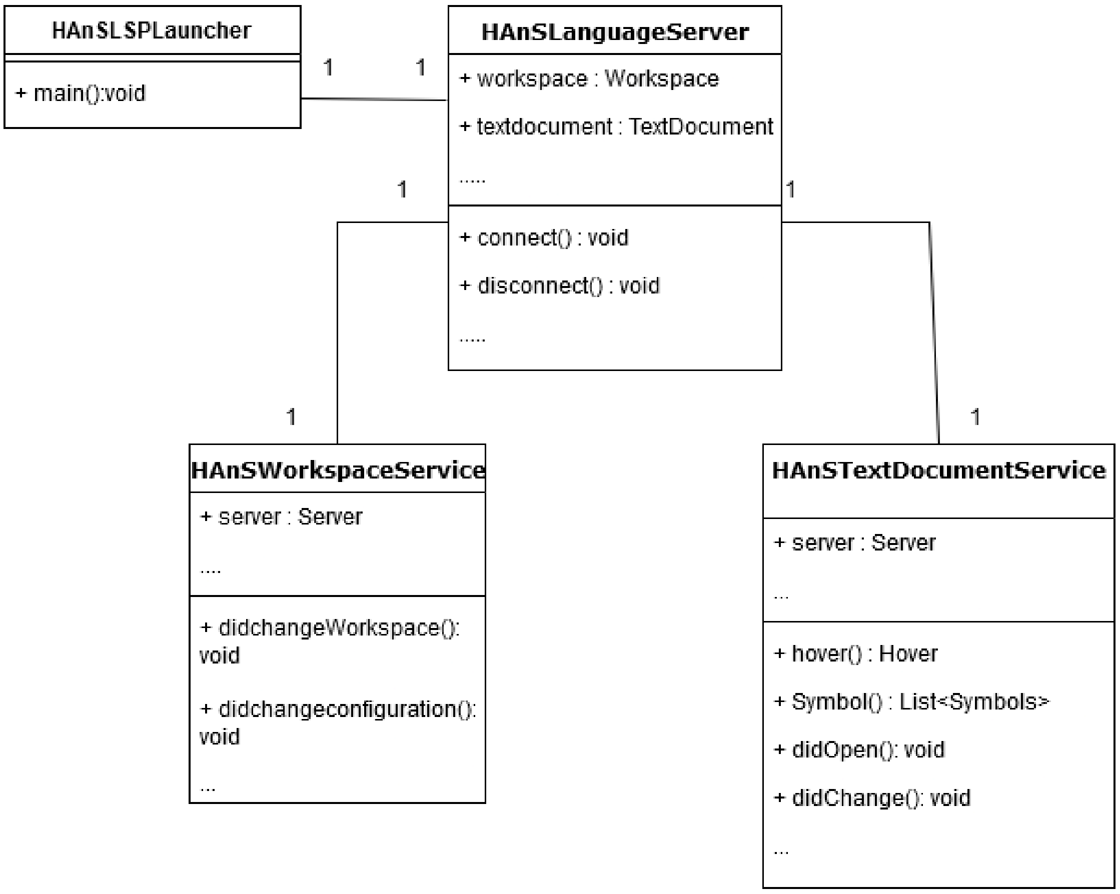


Figure 4.1: HAnS-LSP structure example

The entry point to launch the server is the HAnSLSPLauncher, which creates a new instance of the server and ensures a proper startup. At the core of the server is the HAnSLanguageServer, which implements the language server interface, defines the capabilities the Server provides and coordinates the communication between the other components. Other services we implemented:

- HAnSTextDocumentService: Manages file related operations, such as hover information, processing changes, symbol information and handles "got to" commands.
- HAnSWorkspaceService: Manages the workspace, including configuration changes, workspace folders and workspace-related actions.

The server also includes four ANTLR grammars and the Lexer, Parser and Tree-walker generated by ANTLR to parse feature models, feature-to-file, feature-to-folder and code files. This is critical for providing the functionality of HAnS. Additionally it contains a pom.xml file as part of the Maven configuration to manage dependencies and the build processes.

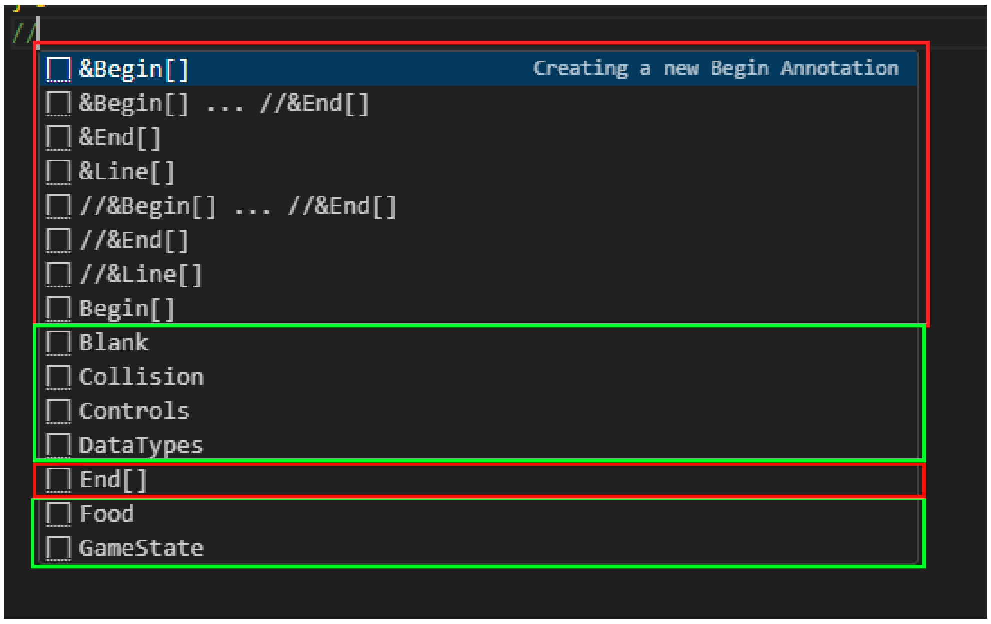
To support debugging we implemented a logging system. The logging system uses files, since in the implementation of the language server already needs the Java stdout/stdin to communicate with a client. All information about features are stored in a tree structure which is constructed while parsing the feature model.

4.2 First iteration

During the first iteration, we focused on building a robust base implementation of an LSP-base version of the HAnS plugin to simplify further expanding and development. The first Feature we implemented after building a launchable prototype is the ability to parse feature trees. The grammar used to generate the parser is similar to the one proposed by FAXE, but since we wanted the ability to parse feature models, which define or and xor relationships between features, we implemented correct parsing of those. To find the corresponding feature model, we implemented a function to find the correct *.feature-model* file by checking if the current folder has an *.feature-model* file. if no *.feature-model* file can be found, it walks up the folder structure to its parent folder and continues searching . It starts at the folder the currently opened file is in and runs until it finds a *.feature-model* file or reaches the outmost folder opened in the workspace.

4.2.1 Code completion

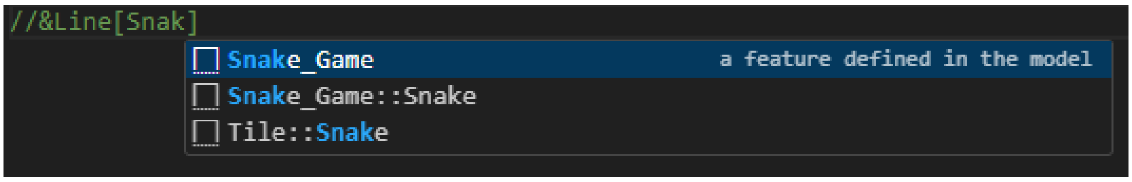
After implementing the ability to parse feature trees, implementing the code completion feature was possible. The user of the tool gets the names of the features and the annotations recommended while writing code, which leads to faster annotation of code.



```
//
&Begin[]                    Creating a new Begin Annotation
&Begin[] ... //&End[]
&End[]
&Line[]
//&Begin[] ... //&End[]
//&End[]
//&Line[]
Begin[]
Blank
Collision
Controls
DataTypes
End[]
Food
GameState
```

Figure 4.2: Code completion example in VS Code

The tool can recommend different versions of the annotations to the user to make it possible to quickly write code annotations. The user can choose to generate a *&Begin[]* annotation with an accompanying *&End[]* annotation or without one. The user can also choose between different versions with the comment delimiter or without.
Furthermore the recommendations include the names of features which are present in the feature model that belongs to the corresponding file. Names for features that are used multiple times in the model only get recommended with the corresponding parent feature name in the front.
For example, if the feature “Snake” is defined twice in a feature model and the feature “Snake_game” is the parent of the first occurrence of the feature “Snake” and the feature “Tile” is the parents of the second occurrence of the features “Snake”, the tool only recommends “Snake_game::Snake” and “Tile::Snake” as seen in Figure 4.3.



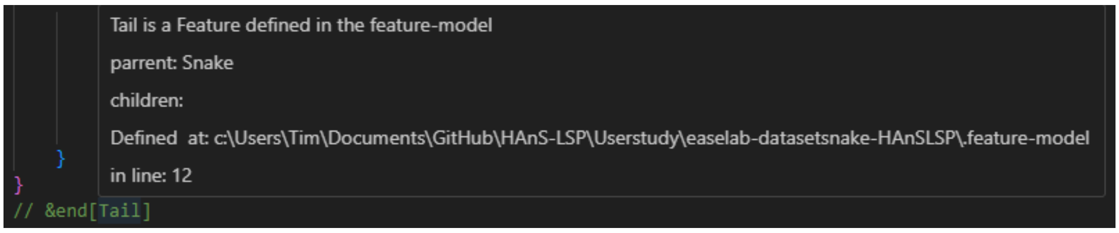
```
//&Line[Snak]
Snake_Game                    a feature defined in the model
Snake_Game::Snake
Tile::Snake
```

Figure 4.3: Code completion example duplicated feature names

As shown in Figure 4.2, where you can see the code completion window of VS Code with the the recommendations for annotations marked in red and recommendations for feature names marked in green. the recommendation process in handles by the IDE. The server provides the IDE only with all completion items that can be recommended.

4.2.2 Hover

To provide the user with more information, the tool provides hover windows when the user hovers over features and annotations with his mouse cursor. When hovering over annotations, the hover window describes what the used annotation means, and when hovering over features, it shows information about the feature. It tells the user where a feature is defined and what the parent-feature and child-features are. An example Hover is shown in Figure 4.4.



```
Tail is a Feature defined in the feature-model
parrent: Snake
children:
Defined  at: c:\Users\Tim\Documents\GitHub\HAnS-LSP\Userstudy\easelab-datasetsnake-HAnSLSP\.feature-model
in line: 12
    }
}
// &end[Tail]
```

Figure 4.4: Hover over feature example

4.3 Second iteration

In the second iteration we focused on implementing the remaining features we mentioned in section 3.3. First we implemented parsing of *.feature-to-file* files, *.feature-to-folder* files and code files. The grammars used are once again based on the ones FAXE provides but changed a bit to make working with the Parse Trees a bit easier. The server now also provides the IDE with information about Symbols in the current document, which is necessary for the implementation we chose to implement finding locations in code. Without providing the correct symbol information it wouldn't be possible to call the "go to definition" and "go to reference" methods we want to use for the implementation of a fast way to browse the used feature annotations.

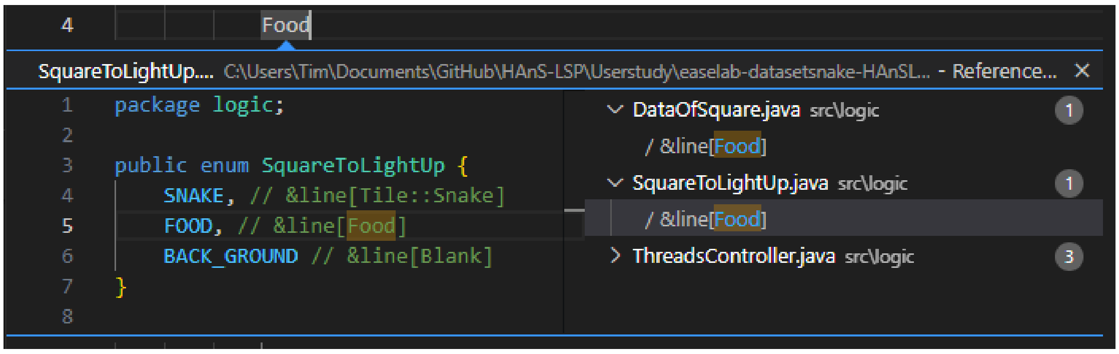
4.3.1 Go to definition

When the user clicks the "go to definition" option after right clicking a feature, the IDE opens the *.feature-model* file the feature is defined in, at the position where the feature is defined.

4.3.2 Go to reference

To simplify the location of feature annotations in code we decided to implement the "go to reference" method. When the user calls the "Go to reference" method, the IDE receives all locations of:

1. code annotations that reference the feature
2. the *.feature-to-folder* and *.feature-to-file* files, where the feature is used in
3. all files which are mapped through a feature to file mapping



```
Food
```

SquareToLightUp.... C:\Users\Tim\Documents\GitHub\HAnS-LSP\Userstudy\easelab-datasetsnake-HAnSL... - Reference... ✕

```
package logic;

public enum SquareToLightUp {
    SNAKE, // &line[Tile::Snake]
    FOOD, // &line[Food]
    BACK_GROUND // &line[Blank]
}

```

```
⌄ DataOfSquare.java src\logic        1
    / &line[Food]
⌄ SquareToLightUp.java src\logic     1
    / &line[Food]
> ThreadsController.java src\logic   3
```

Figure 4.5: VS Code reference example

VS Code handles multiple locations by displaying an extra window to the user with locations listed on the right side and a small version of the file containing the location.

5 Results

In this chapter we assess if we implemented all features of HAnS we deemed necessary for an efficient use of feature annotations. Furthermore we evaluate if the extension provides good tool support and if the LSP and the extension integrate well in to the workflow of a programmer using Visual Studio Code. Since the integration tests where only conducted with VS Code, it remains unclear how other IDEs handle the language server.

5.1 Development outcome

When starting development of the language server we set out to implement the most important features of the HAnS IntelliJ extension.

5.1.1 Correct parsing

Correct parsing is one of the most important parts of the system since all other functions rely on the correctness of the parsing. While implementing the parsing, we used ANTLR to help build the correct parser and lexer. The grammars used are largely identical to the one suggested by FAXE, which where also the bases for the ones used in HAnS. The parsing done by our server is correct for all tests run by us. The only issue with our implementation is that, since it is an LSP, we can not directly access the changes on a document that are stored in the memory of the IDE. It would be necessary to track all changes in a separate version of the document that is stores on the server side. For this reason the tool currently can only guarantee correct function of the server when the user saves the document.

5.1.2 Code completion

Code completion is a major functionality of HAnS since we can only get programmers to write annotations while coding when it doesn't require a lot of effort and time. Unfortunately, The way HAnS helps annotate code is not possible to create using an LSP, because of the limited refactoring options of an LSP, since the refactor method can only refactor on one position at a time. The best thing we could do is provide code completion for annotations and feature names.

5.1.3 Locating Features in Code

To locate the use of a feature in code we use the “go to definition” and “go to reference” methods of the LSP, which works to navigate to uses of the feature. It allows the user to quickly jump to uses of the feature and it allows for easy location of features respectively. The only downside of working with this approach is that when using the “go to reference” method other LSPs which are working on the file at the same time provide references for other things at the same time. For example: when JAVA code contains a method that is annotated with a block annotation and the user calls the “go to reference” method on the feature that is written in the *&Begin[]* brackets the JAVA LSP will provide the IDE with locations where the method is used, which will be mixed with the ones our language server provides, because this is how multiple parallel running LSP are handled in VS Code.

5.2 Evaluation of the user study

Two rounds of the user study were run to determine how well the developed LSP integrates into VS Code. The first run was conducted at the end of the first iteration. In that iteration we focused on building a good base for the next one and wanted to know if we can get qualitative feedback with the user study design we are using.

5.2.1 First iteration of the user study

The first round was conducted with three bachelor students, which had never used the annotations before and all of them had at leased one year of experience with JAVA.
All of them agreed that it was easier and faster to write feature annotations, when using the Tool. Regardless of if it is a mapping, in a *.feature-to-folder* or a *.feature-to-file* file, or a direct annotation.
The result of the System Usability Scale result of the iteration is 82,5 out of 100, witch means the integration of our language server into the VS Code workflow is quite good.

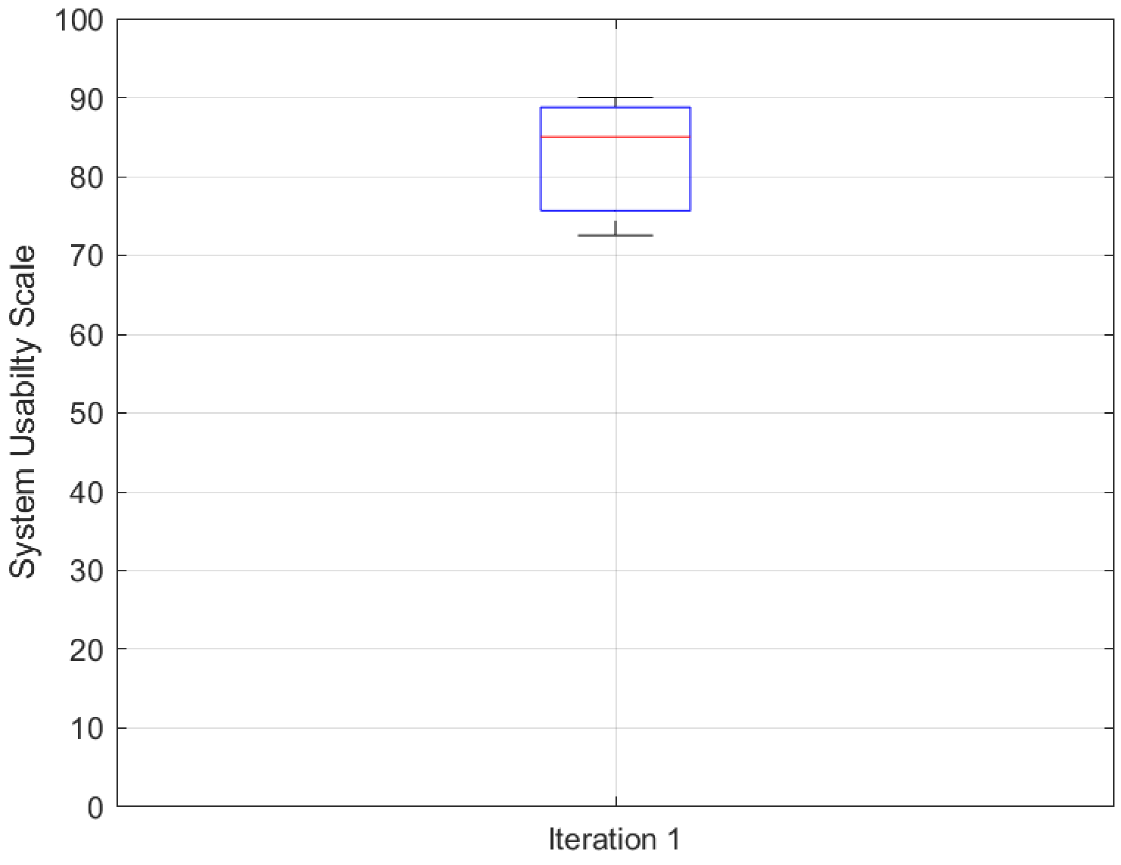


Figure 5.1: System Usability Scale of the first iteration

Our server got the worst score in the the following questions: “I think that I would need the support of a technical person to be able to use this system”, “I needed to learn a lot of things before I could get going with this system” and

I felt very confident using the system. That could mean that our explanation is not detailed enough or the information we provide to introduce the annotations and the tool is not precise enough. A different problem seems to be that the set-up process is not straight forward as it requires a bit of time and effort. Two participants explicitly stated that setting up the tool is the greatest challenge they faced.

5.2.2 Second iteration of the user study

In the second round of the study we wanted as many participants as possible. In the end seven students participated in the study. With the exception of one person, everyone agreed that it was faster and easier to write feature annotations, when using the tool, compared to without. The person who did not agree chose the neither agree nor disagree option. All participants had at least one year of experience using JAVA, but no one used annotations before participating in the user study. The average System Usability Score of the second round was 75,36.

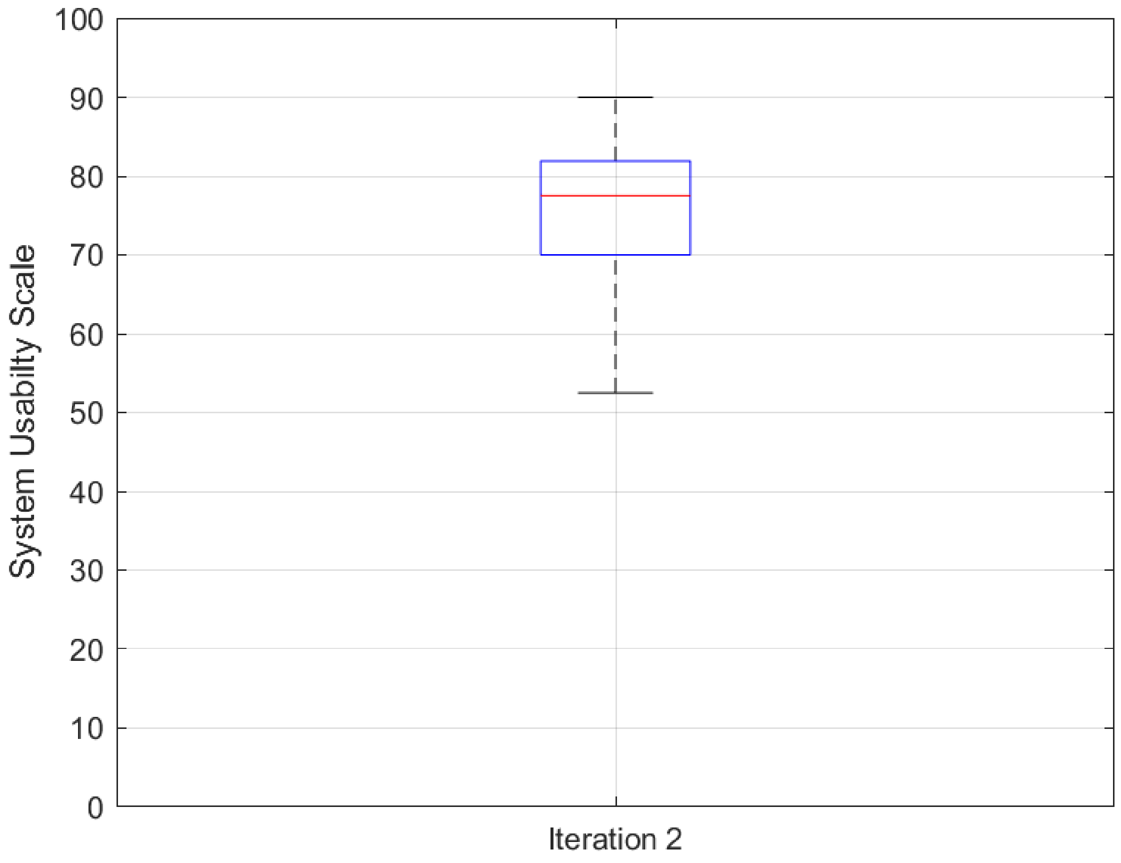


Figure 5.2: System Usability Scale of the second iteration

The lowest score calculated during the second run is 52,5 and the highest is 90. A score of more than 70 means it is still well integrated. The questions with the worst scores are: “I think that I would need the support of a technical person to be able to use this system”, “I found the system unnecessarily complex” and “I needed to learn a lot of things before I could get going with this system”. The biggest problem with the implementation seems to be with the knowledge the user needs to use the system, the complexity of the setup process. The question about the biggest problems faced by the participants confirm that.

5.2.3 Conclusion of the user study

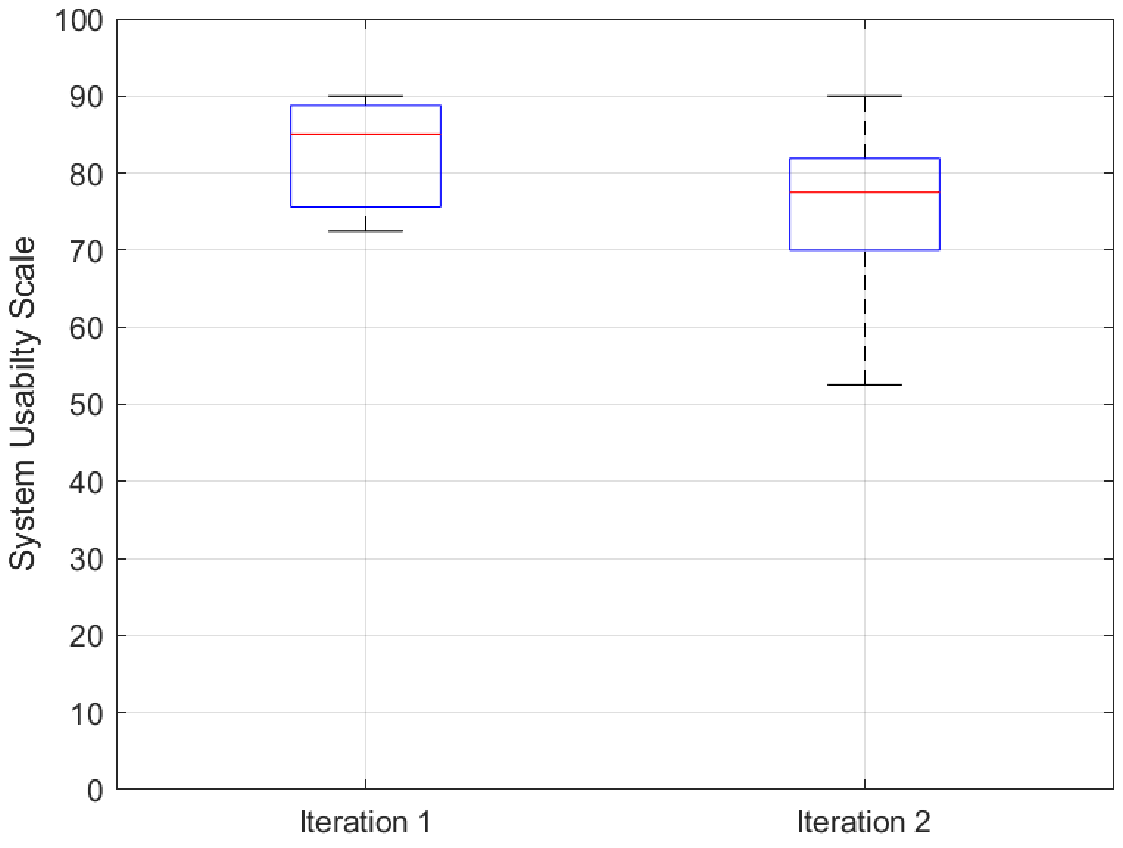


Figure 5.3: System Usability Scale comparison

When looking the results of both runs of the study, the results are pretty similar. The difference of 7,2 in the average System Usability Score between both iteration can be explained by the low amount of participants in both runs of the user study.

	Iteration 1	Iteration 2
#Participants	3	7
#Participants Group 1	2	5
#Participants Group 2	1	2
Average SUS	82,5	75,36
Highest SUS	90	90
Lowest SUS	72,5	52,5

Table 5.1: Comparison of both iterations

Furthermore, the System Usability Score is just a way to measure the usability of a system, which is not a real value that can be measured. Usability is different for every person and as such the System Usability Score con only give us an estimate. Since the difference in the measured values does not significantly change and is above 75 in both cases, we can conclude that the tool is well integrated in the workflow of a programmer that is using VS Code. No user reported any major issues with the way we implemented the features or the way they interact with the tool. Which furthermore point to the tool being implemented well enough. The biggest problems users faced and the suggestions to further improve the implementation are consistent between both runs.

5.2.4 Discussion

The implementation of the base functionality of HAnS into a language server was a success. Because of how the functionalities of HAnS line up with commands the LSP natively supports. But not all steps of the conversion were, without problems. The LSP limits the interaction with the IDE and the information the IDE receives from other sources or stores inside its memory, for example the changes stored in a temp file. The use of the LSP also comes with limitations of what the server natively can support, for example the LSP does not support all functionalities an extension provides. Furthermore, it is not possible to provide syntax highlighting in the LSP and integrating it manually into the LSP would require to implement it in both an extension and on server side, which is against what we are trying to achieve with the LSP, since someone would still have to implement a plugin for every IDE. Another limitation of the implementation of a language server is with the amount of refactoring that is possible with a language server, since the Language Server Protocol does not allow for refactoring at different places in the same document, let alone in different documents.

The results of the study only suggests that our implementation was a success, since the number of participants is not high enough for a definitive answer. The System Usability Score of more than 75 in both runs point to the conclusion, that the tool is well implemented. The difference in the average System Usability Score between both iteration is not significantly different and can be explained by the low number of participants in both runs.
The tasks were chosen to test the features the toll provides and is therefor not an ideal representation of the a work a developer would do and the example project given is relatively small compared to real projects. Therefore, it needs to be tested if the study we conducted translates to the real world. This thesis goal was to develop a IDE independent version of HAnS using the LSP. But how a language server is implemented is not defined by the LSP. during testing and development we only focused on VS Code. How well the toll integrates into other IDEs need to be tested separately. The extension and

the language server provide tool support for writing annotations and for browsing existing code basses. The biggest problem the tool seems to have is the amount of knowledge a new user has to learn to use the tool and the annotations effectively and the long and for such a tool complex set up process.

Future work could be optimise the existing server by using concurrency in the longer calculations and function calls like the parsing of the code base or the searching in the tree. The implementation of other features of HAnS could also be a possible follow up task. Another task could be to try to optimise the integration of the extension and the plugin by directly taking the feed back into account we gathered during the user study, which suggest to implement syntax highlighting and highlighting features which are annotated but not defined in the feature model. A different topic could be to check how much effort it would take to get the language server working in different IDEs to determine how IDE independent a Language Server really is or testing how much effort it takes to support languages with other comment styles, since we are currently only supporting C type comments and only tested with JAVA code.

6 Conclusion

To increase maintainability of code bases or the modularity of software systems, we want developers to properly document their code. To help with the documentation, feature-oriented Domain specific Languages were proposed, but for developers to take advantage of the languages propose we need tool support. Until now, there was no IDE independent tool support and developers, who use not supported IDEs had to change their IDE or don't use tooling designed for those languages.
HAnS-LSP aims to address the issue of IDE dependent tooling by trying to become IDE independent. It is designed to only use the LSP to deliver the basic functionalities of HAnS. The tool provides helpful functionalities like code completion, locating uses of annotations in code and navigating to annotations. We provide an extension to Visual Studio Code, which allows the use of the language server.
With this project we wanted to answer whether it is possible to implement the base version of the HAnS plugin as a language server. We managed to implement all the features deemed important, but not without problems, since the Language Server Protocol is more restrictive than APIs used to develop Extensions. Only minimal communication between the server and the client is part of the base features of the LSP. Since the features of HAnS line up with methods the LSP provides by default, we were able to implement a working server. More complex or specific functionalities an extension could implement are not possible with the basic methods the LSP natively provides. To verify the usability of the design, a user study containing the System Usability Scale questions was conducted. Even though the number of participants were quite low, the data points toward it being well integrated into the development workflow.
To fully evaluate the usability and integration, a study with more participants should be conducted.
The development of the tool assists with creating more tooling to help with feature-oriented programming. Further, it creates a foundation for improving the tool support to increase the accessibility of tools and increases the efficiency of programmers. Future research could revolve around on increasing the usability of the tool, addressing limitations of the LSP and broadening the support of IDEs and programming languages.

List of Figures

List of Tables

Bibliography

[1] S. Amann et al. “A Study on the Usage of the Language Server Protocol in Open Source IDEs/Editors”. In: *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering.* 2011.

[2] Berima Andam, Andreas Burger, Thorsten Berger, and Michel R. V. Chaudron. “FLOrIDA: Feature LOcatIon DAshboard for extracting and visualizing feature traces”. In: VaMoS ’17. Eindhoven, Netherlands: Association for Computing Machinery, 2017, pp. 100–107. ISBN: 9781450348119. DOI: `10.1145/3023956.3023967`. URL: `https://doi.org/10.1145/3023956.3023967`.

[0] Sven Apel, Don Batory, Christian Kastner, and Gunter Saake. “Software Product Lines”. In: *Feature-oriented software product lines.* 2013th ed. Berlin, Germany: Springer, 2013, pp. 1–44.

[3] Kacper Bąk, Zinovy Diskin, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. “Clafer: Unifying class and feature modeling”. In: *Software & Systems Modeling* 15 (Dec. 2014). DOI: `10.1007/s10270-014-0441-1`.

[4] T.J. Biggerstaff, B.G. Mitbander, and D. Webster. “The concept assignment problem in program understanding”. In: *[1993] Proceedings Working Conference on Reverse Engineering.* 1993, pp. 27–43.

[5] John Brooke. “SUS: A quick and dirty usability scale”. In: *Usability Eval. Ind.* 189 (Nov. 1995).

[6] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. “Feature location in source code: a taxonomy and survey”. In: *Journal of Software: Evolution and Process* 25 (2013). URL: `https://api.semanticscholar.org/CorpusID:7630279`.

[7] Sina Entekhabi, Anton Solback, Jan-Philipp Steghöfer, and Thorsten Berger. “Visualization of Feature Locations with the Tool FeatureDashboard”. In: *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B.* SPLC ’19. Paris, France: Association for Computing Machinery, 2019, pp. 1–4. ISBN: 9781450366687. DOI: `10.1145/3307630.3342392`. URL: `https://doi.org/10.1145/3307630.3342392`.

[8] *FAXE Online Appendix.* `https://bitbucket.org/easelab/faxe`. Accessed: 2024-12-18. 2021.

[9] J. Feichtinger, T. Kehrer, and D. Strüber. “SiLift: Lifting Clone-and-Own to Software Product Lines”. In: *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering.* 2019.

[10] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. “Maintaining feature traceability with embedded annotations”. In: *Proceedings of the 19th International Conference on Software Product Line*. SPLC ’15. Nashville, Tennessee: Association for Computing Machinery, 2015, pp. 61–70. ISBN: 9781450336130. DOI: 10.1145/2791060.2791107. URL: https://doi.org/10.1145/2791060.2791107.

[11] *Language Server Protocol*. https://microsoft.github.io/language-server-protocol/. Accessed: 2024-12-18.

[12] *LSP4J*. https://github.com/eclipse-lsp4j/lsp4j. Accessed: 2024-12-19.

[13] J. Martinson and H. Janson. “HAnS: IDE-Based Editing Support for Embedded Feature Annotations”. In: *SPLC ’21: Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume B*. 2011.

[14] Mukelabai Mukelabai, Kevin Hermann, Thorsten Berger, and Jan-Philipp Steghöfer. “FeatRacer: Locating Features Through Assisted Traceability”. In: *IEEE Transactions on Software Engineering* 49.12 (2023), pp. 5060–5083.

[15] G.C. Murphy, M. Kersten, and L. Findlater. “How are Java software developers using the Eclipse IDE?” In: *IEEE Software* 23.4 (2006), pp. 76–83.

[16] *official JSON-RPC website*. https://www.jsonrpc.org/. Accessed: 2024-12-19.

[17] Terence Parr and Kathleen Fisher. “LL(*): the foundation of the ANTLR parser generator”. In: PLDI ’11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 425–436. ISBN: 9781450306638. URL: https://doi.org/10.1145/1993498.1993548.

[18] M. P. Robillard and G. C. Murphy. “Concern graphs: Finding and describing concerns using structural program dependencies”. In: *Proceedings of the 24th International Conference on Software Engineering*. 2002.

[19] Julia Rubin and Marsha Chechik. “A Survey of Feature Location Techniques”. In: *Domain Engineering: Product Lines, Languages, and Conceptual Models*. Ed. by Iris Reinhartz-Berger, Arnon Sturm, Tony Clark, Sholom Cohen, and Jorn Bettin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 29–58. ISBN: 978-3-642-36654-3.

[20] Tobias Schwarz, Wardah Mahmood, and Thorsten Berger. “A Common Notation and Tool Support for Embedded Feature Annotations”. In: SPLC ’20. Montreal, QC, Canada: Association for Computing Machinery, 2020, pp. 5–8. ISBN: 9781450375702. DOI: 10.1145/3382026.3431253. URL: https://doi.org/10.1145/3382026.3431253.

[21] Yulia Shmerlin, Irit Hadar, Doron Kliger, and Hayim Makabee. “To Document or Not to Document? An Exploratory Study on Developers’ Motivation to Document Code”. In: vol. 215. June 2015, pp. 100–106. ISBN: 978-3-319-19242-0. DOI: 10.1007/978-3-319-19243-7_10.

[22] Ian Sommerville. *Software engineering*. 9th. Pearson, 2011.

[23] *Stack Overflow Survey 2024, integrated development environment.* https://survey.stackoverflow.co/2024/technology#1-integrated-development-environment. Accessed: 2024-12-19.

[24] Christoph Johann Stettina and Werner Heijstek. “Necessary and neglected? an empirical study of internal documentation in agile software development teams”. In: *Proceedings of the 29th ACM International Conference on Design of Communication.* SIGDOC ’11. Pisa, Italy: Association for Computing Machinery, 2011, pp. 159–166. ISBN: 9781450309363. DOI: 10.1145/2038476.2038509. URL: https://doi.org/10.1145/2038476.2038509.

[25] C.R. Turner, A. Fuggetta, L. Lavazza, and A.L. Wolf. “Feature engineering [software development]”. In: *Proceedings Ninth International Workshop on Software Specification and Design.* 1998, pp. 162–164.

[26] J. Wang, X. Peng, Z. Xing, and W. Zhao. “How Developers Perform Feature Location Tasks: A Human-Centric and Process-Oriented Exploratory Study”. In: *Journal of Software: Evolution and Process.* Vol. 25. 11. 2013.

[27] “WATEERFALLVs V-MODEL Vs AGILE: A COMPARATIVE STUDY ON SDLC”. In: 2012. URL: https://api.semanticscholar.org/CorpusID:263064309.

A Resources for user study

A.1 Introduction

Purpose:

The purpose of this repo is to test and evaluate the embedded feature annotations plugin HAnS-LSP for Visual Studio Code. *Do not download the repo content, you will be given a different path to download everything you need*

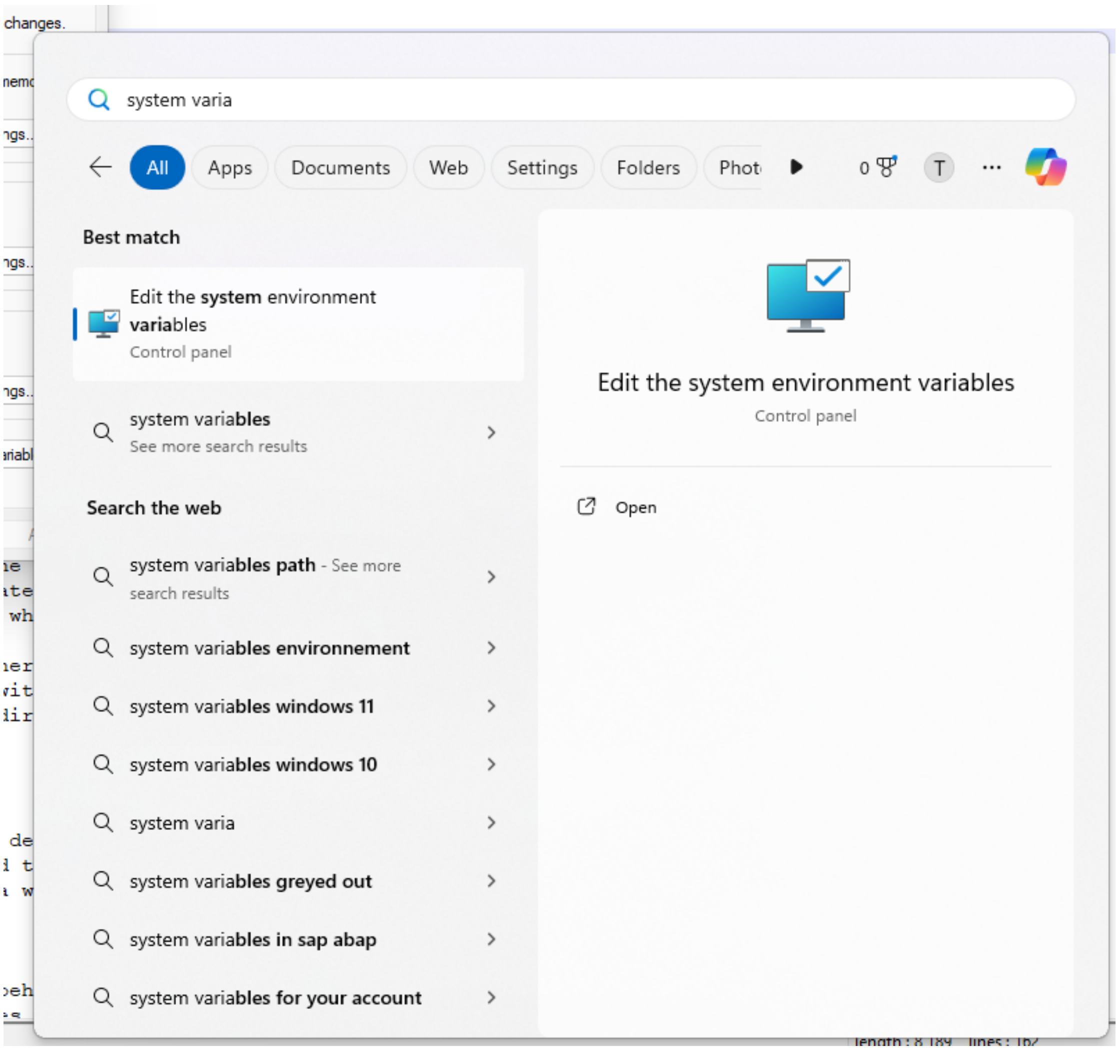
Requirements:

- Visual Studio code installed
- Required JDK 22+
- Installed HAnS-LSP-vscode plugin
- JAVA_Home Set to JDK 22+
- Make sure you turn on auto-save in Visual studio code or save your changes after every step for the extension to work correctly.

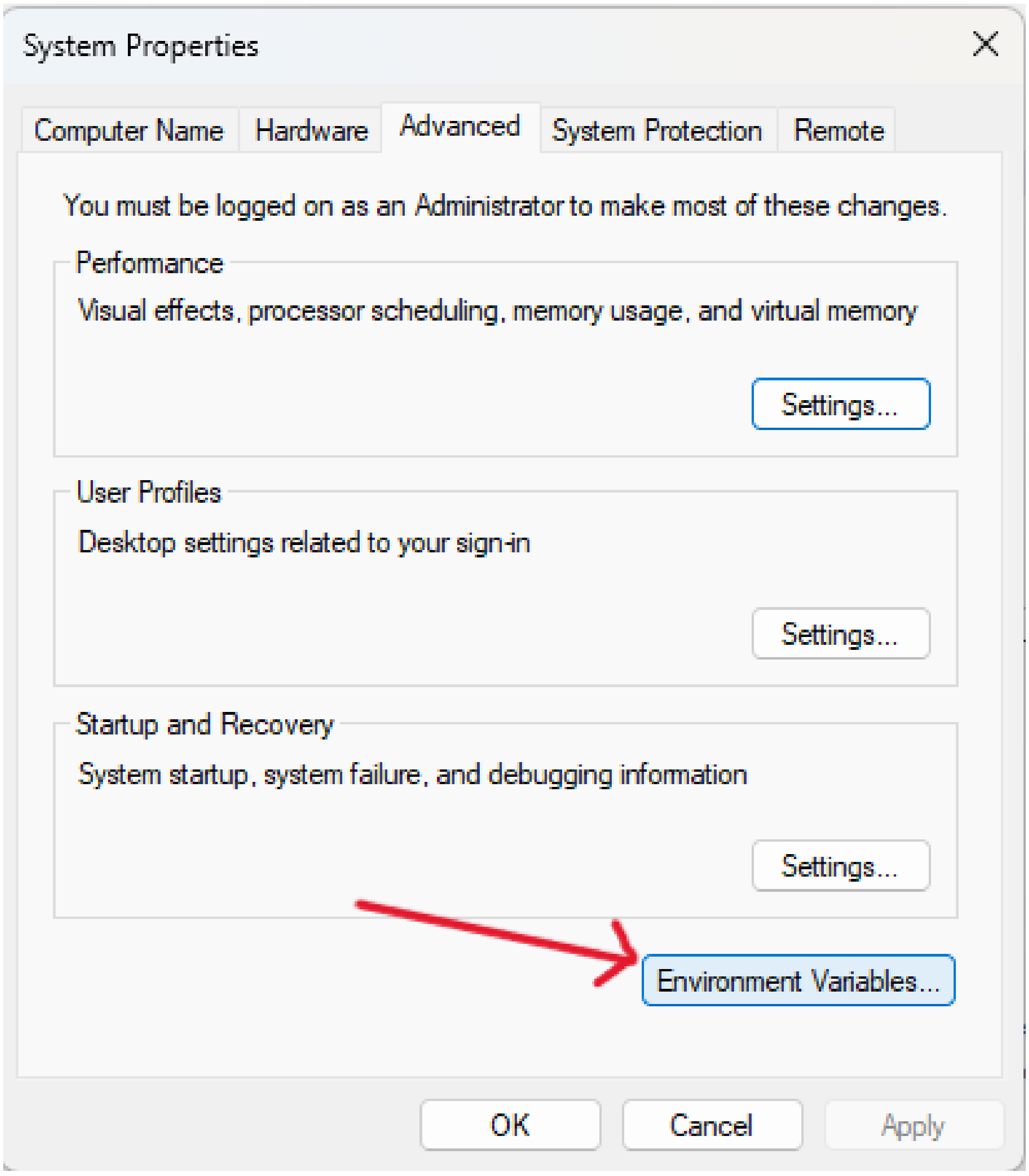
JAVA HOME SETTINGS

How to set JAVA_HOME on windows:

Search for 'Edit the system environment variables'


system varia
All
Apps
Documents
Web
Settings
Folders
Best match
Edit the system environment variables
Control panel
system variables
See more search results
Search the web
system variables path - See more search results
system variables environnement
system variables windows 11
system variables windows 10
system varia
system variables greyed out
system variables in sap abap
system variables for your account
Edit the system environment variables
Control panel
Open

click on ‘Environment Variables’


System Properties

Computer Name | Hardware | Advanced | System Protection | Remote

You must be logged on as an Administrator to make most of these changes.

Performance
Visual effects, processor scheduling, memory usage, and virtual memory
Settings...

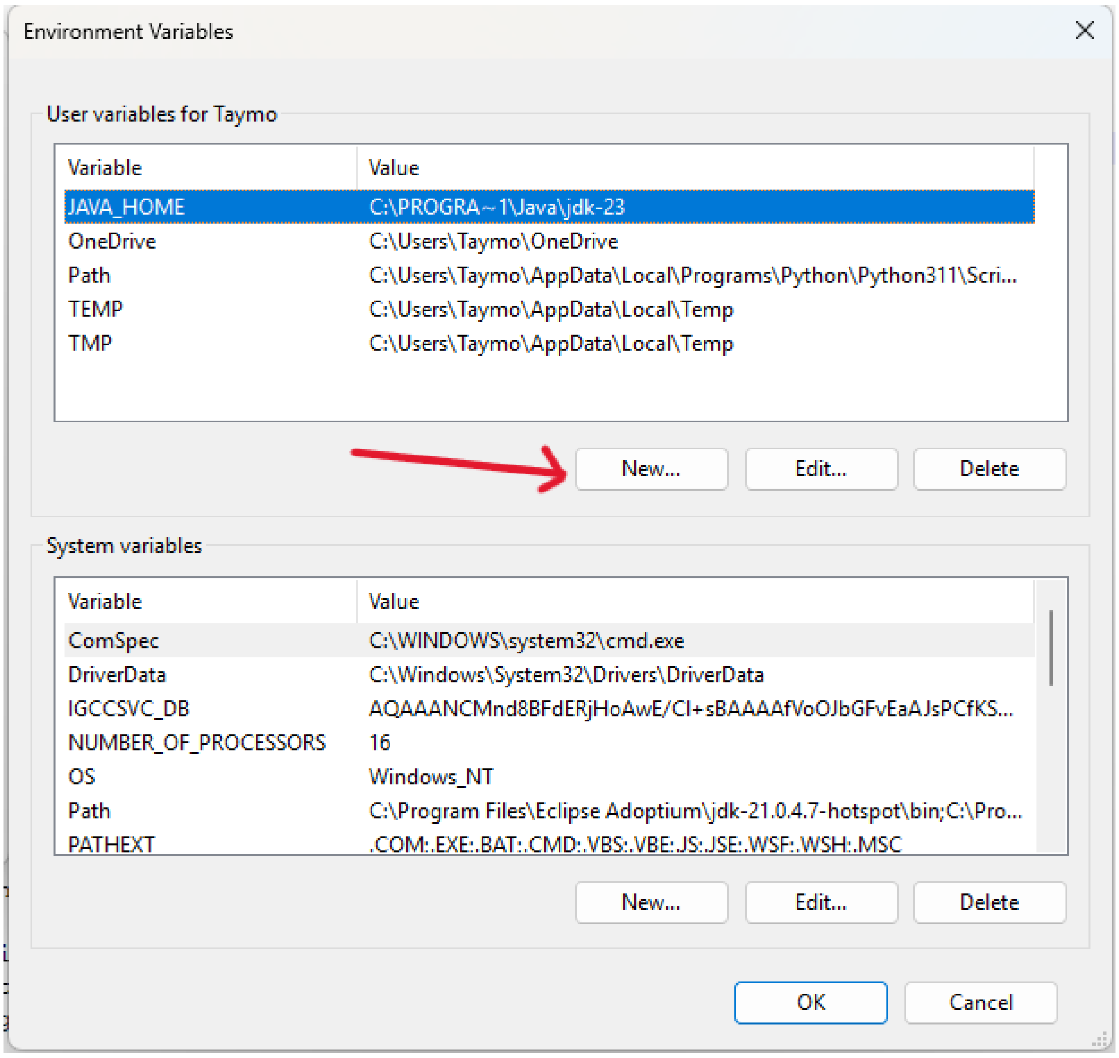
User Profiles
Desktop settings related to your sign-in
Settings...

Startup and Recovery
System startup, system failure, and debugging information
Settings...

Environment Variables...

OK | Cancel | Apply

click 'new…' to create a new variable.


Environment Variables

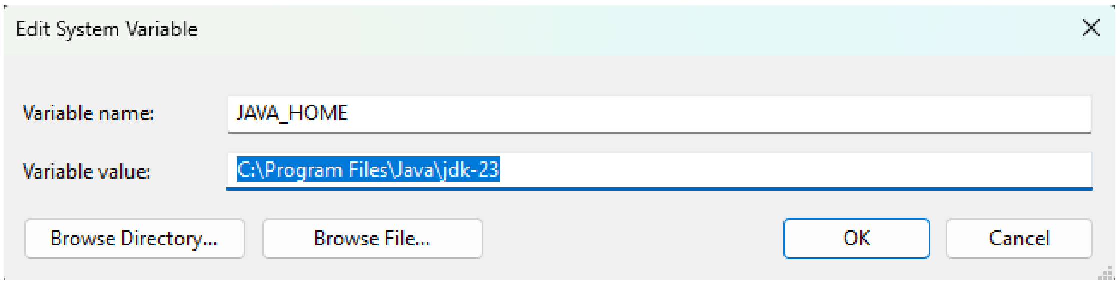
User variables for Taymo

Variable	Value
JAVA_HOME	C:\PROGRA~1\Java\jdk-23
OneDrive	C:\Users\Taymo\OneDrive
Path	C:\Users\Taymo\AppData\Local\Programs\Python\Python311\Scri...
TEMP	C:\Users\Taymo\AppData\Local\Temp
TMP	C:\Users\Taymo\AppData\Local\Temp

System variables

Variable	Value
ComSpec	C:\WINDOWS\system32\cmd.exe
DriverData	C:\Windows\System32\Drivers\DriverData
IGCCSVC_DB	AQAAANCMnd8BFdERjHoAwE/Cl+sBAAAAfVoOJbGFvEaAJsPCfKS...
NUMBER_OF_PROCESSORS	16
OS	Windows_NT
Path	C:\Program Files\Eclipse Adoptium\jdk-21.0.4.7-hotspot\bin;C:\Pro...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC

Name the Variable ‘JAVA_HOME’ and add the path to your JDK 22+ as the value

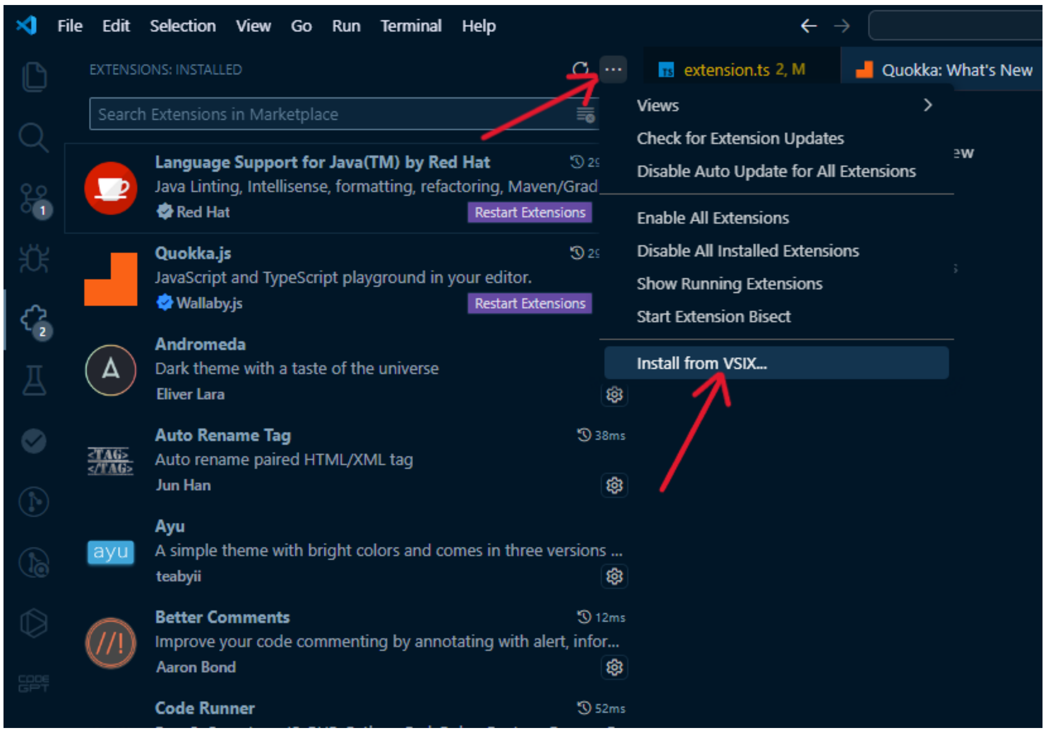

Edit System Variable

Variable name: JAVA_HOME

Variable value: C:\Program Files\Java\jdk-23

Installation of the HAnS-LSP plugin:

- Open Viusual studio Code -> Extensions (ctrl shift x) -> three points (Views and more actions)”



Language Support for Java(TM) by Red Hat
Java Linting, Intellisense, formatting, refactoring, Maven/Grad
Red Hat

Quokka.js
JavaScript and TypeScript playground in your editor.
Wallaby.js

Andromeda
Dark theme with a taste of the universe
Eliver Lara

Auto Rename Tag 38ms
Auto rename paired HTML/XML tag
Jun Han

Ayu
A simple theme with bright colors and comes in three versions ...
teabyii

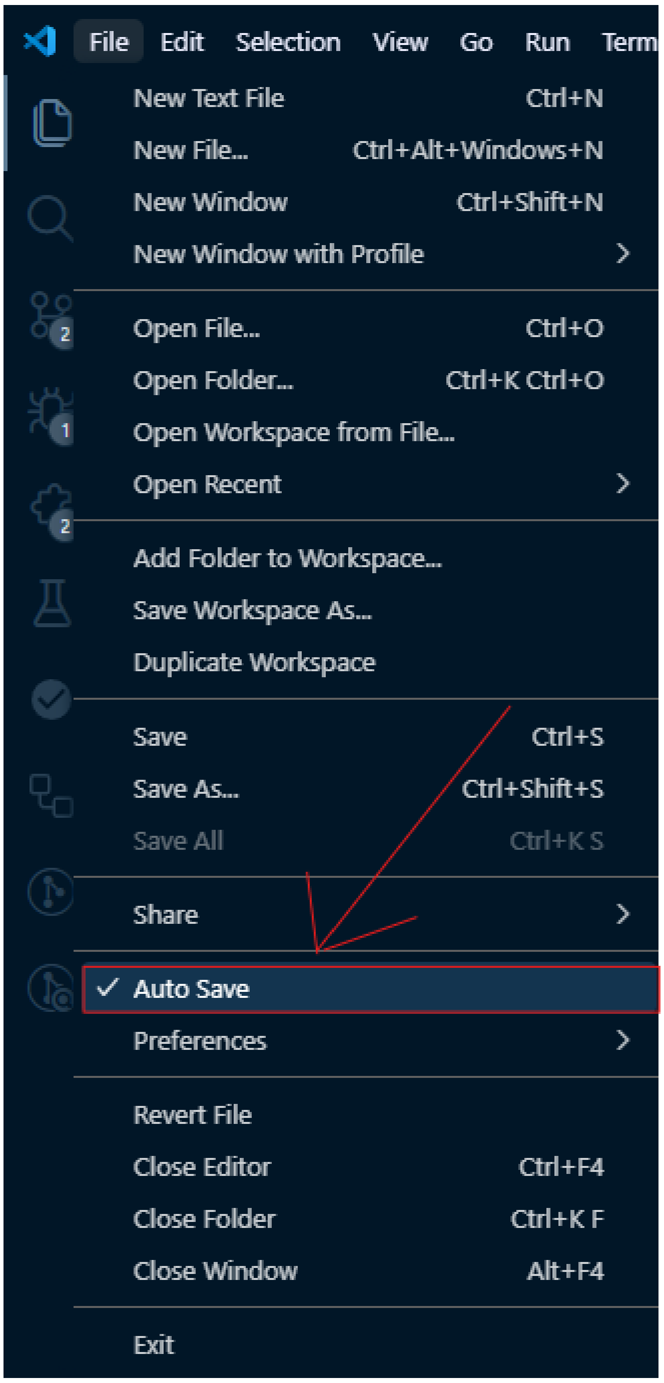
Better Comments 12ms
Improve your code commenting by annotating with alert, infor...
Aaron Bond

Code Runner 52ms

- Choose the path to the VSIX file of the plugin after unzipping the downloaded Userstudy.zip.

Visual Studio auto-save

-auto save would save tthe effort of pressing save after every change :)

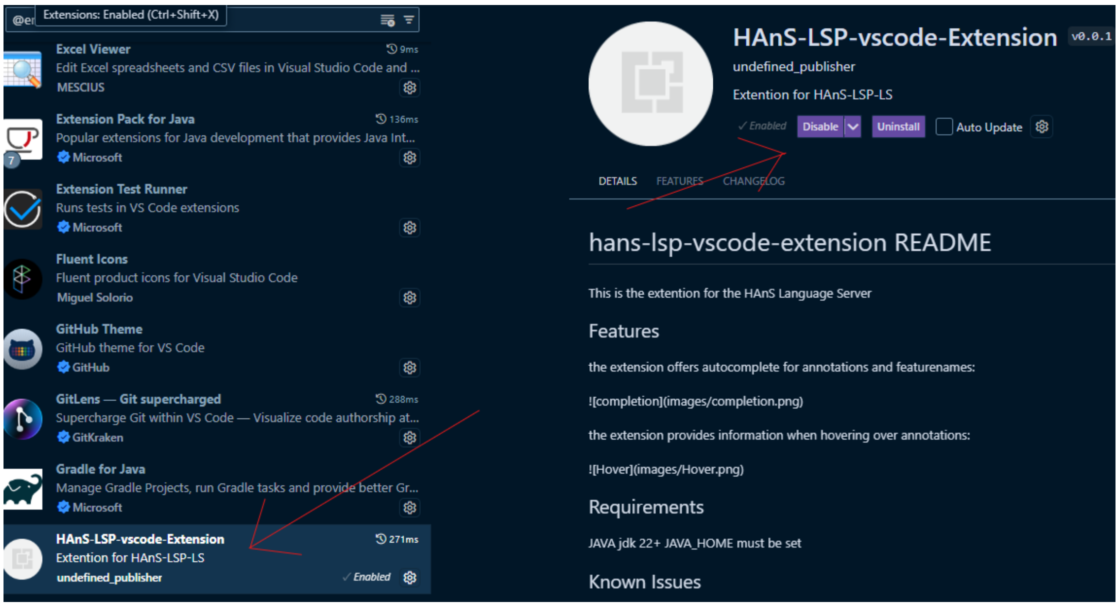


alt tag

How to disable the extension

1. To temporarily disable the HAnS-LSP extension -> go to extensions -> search for HAnS-LSP -> click the gear button at the right of an extension entry.

2. Click “Disable”.



Extensions: Enabled (Ctrl+Shift+X)

Excel Viewer 9ms
Edit Excel spreadsheets and CSV files in Visual Studio Code and ...
MESCIUS

Extension Pack for Java 136ms
Popular extensions for Java development that provides Java Int...
Microsoft

Extension Test Runner
Runs tests in VS Code extensions
Microsoft

Fluent Icons
Fluent product icons for Visual Studio Code
Miguel Solorio

GitHub Theme
GitHub theme for VS Code
GitHub

GitLens — Git supercharged 288ms
Supercharge Git within VS Code — Visualize code authorship at...
GitKraken

Gradle for Java
Manage Gradle Projects, run Gradle tasks and provide better Gr...
Microsoft

HAnS-LSP-vscode-Extension 271ms
Extention for HAnS-LSP-LS
undefined_publisher Enabled

HAnS-LSP-vscode-Extension v0.0.1
undefined_publisher
Extention for HAnS-LSP-LS
✓ Enabled Disable Uninstall Auto Update

DETAILS FEATURES CHANGELOG

hans-lsp-vscode-extension README

This is the extension for the HAnS Language Server

Features

the extension offers autocomplete for annotations and featurenames:

![completion](images/completion.png)

the extension provides information when hovering over annotations:

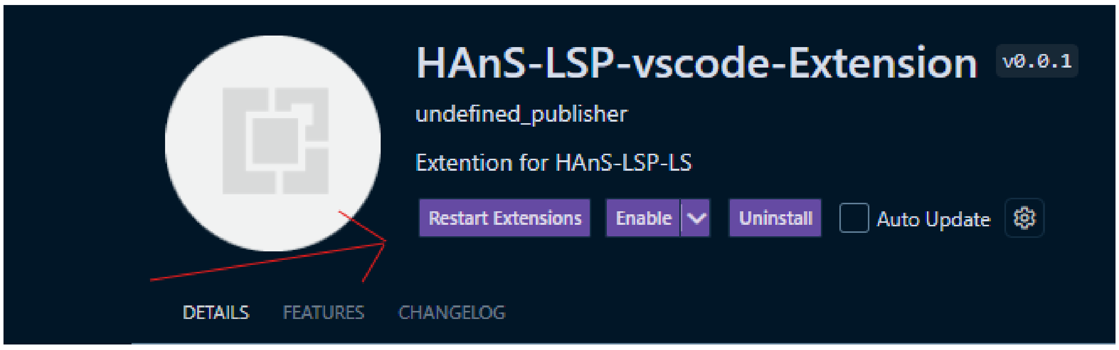
![Hover](images/Hover.png)

Requirements

JAVA jdk 22+ JAVA_HOME must be set

Known Issues

3. Click “restart extensions”



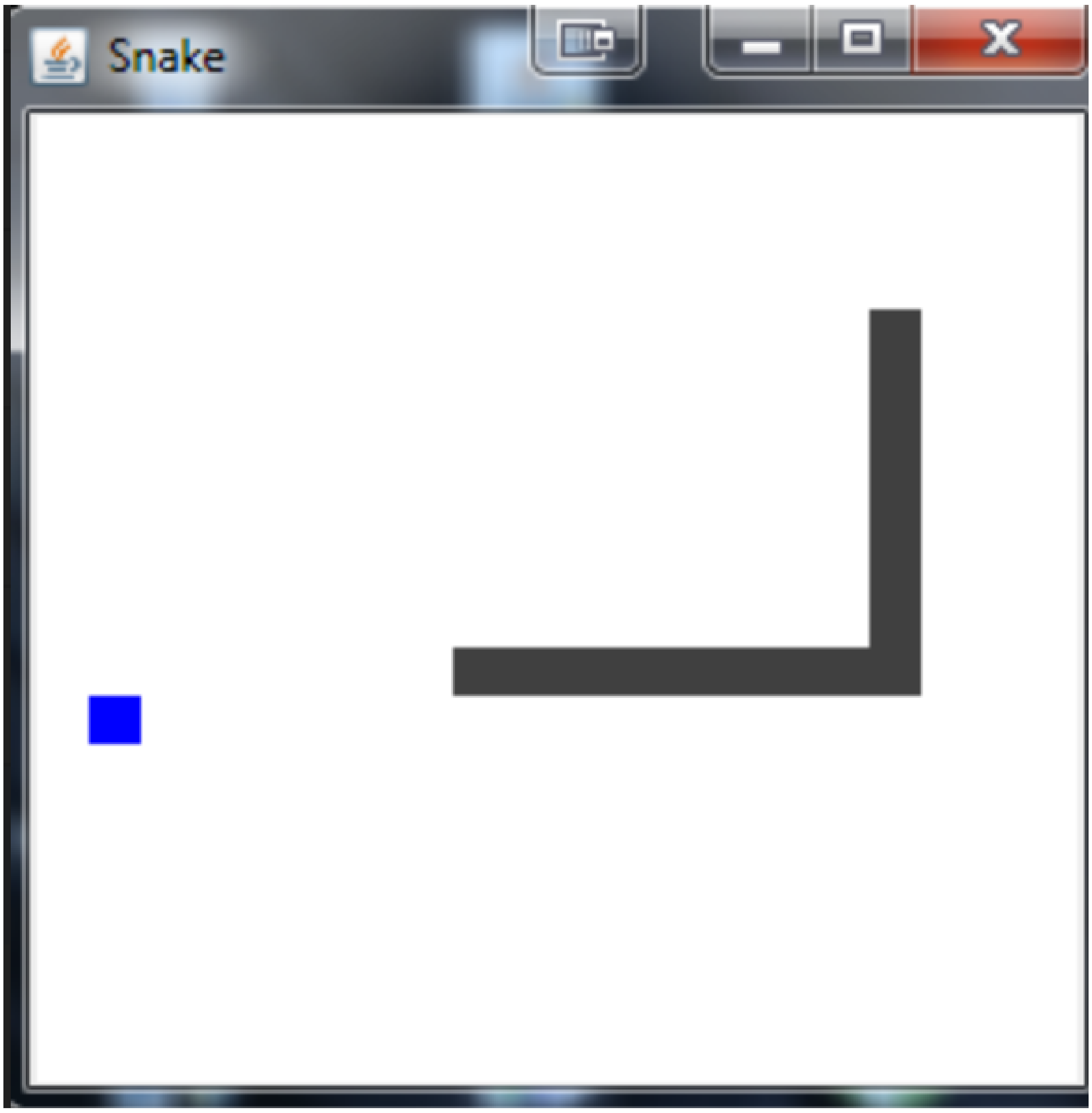
HAnS-LSP-vscode-Extension v0.0.1
undefined_publisher
Extention for HAnS-LSP-LS
Restart Extensions Enable Uninstall Auto Update

DETAILS FEATURES CHANGELOG

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. That's HAnS has been created for Intellij users. Our mission is to create a Language server protocol to make HAnS IDE independent.

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.

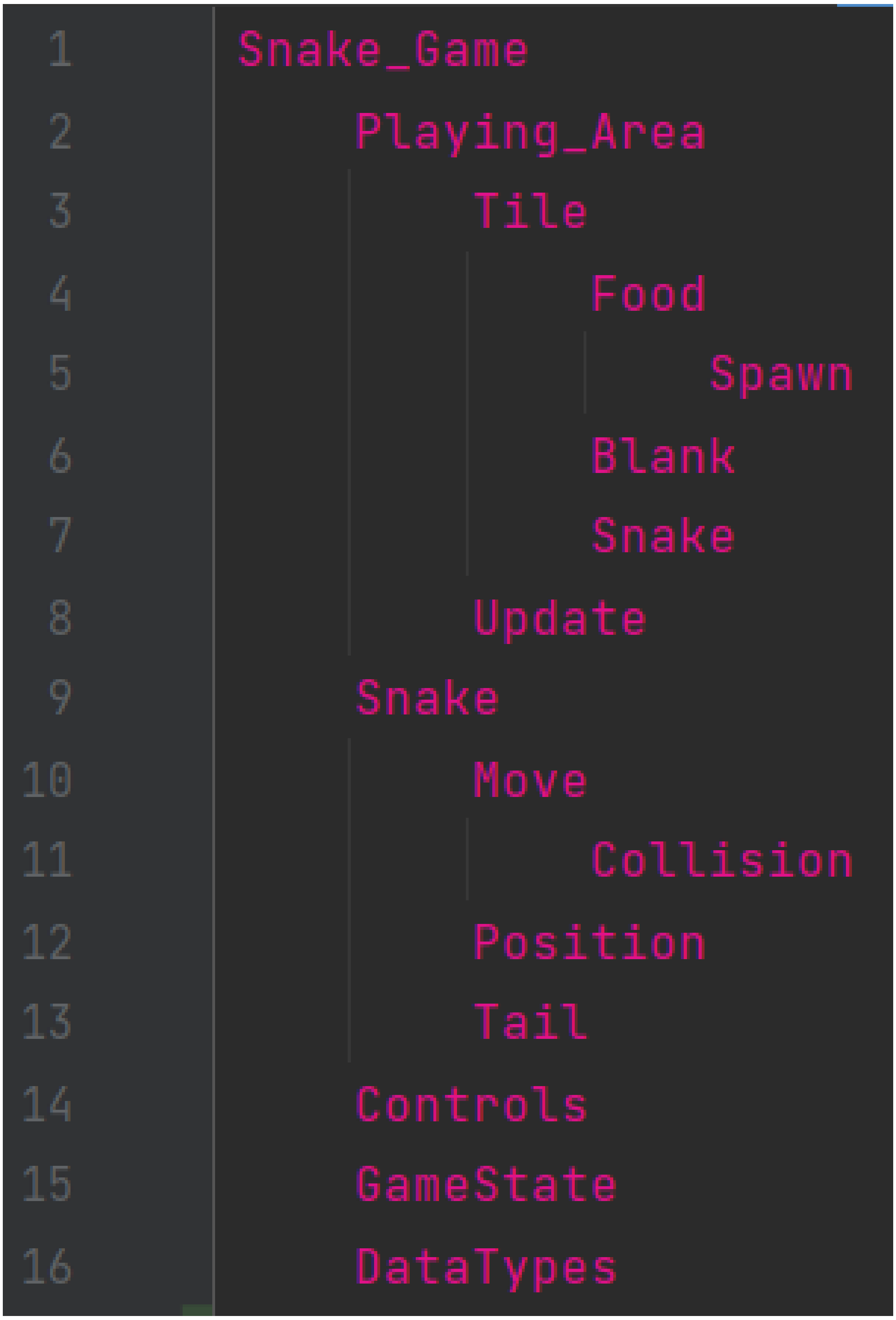
Feature location

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

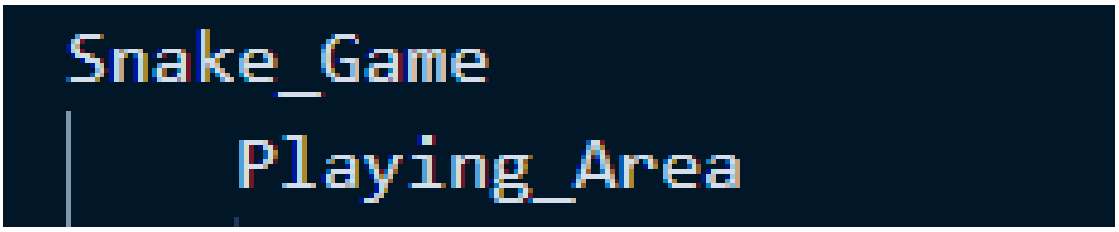
The usage of feature annotations is to map sections of code to functionality of the software. The intent is that this can help developers with feature location. The system of annotations that the HAnS plugin uses is able to map features to any file type and programming language (except languages that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

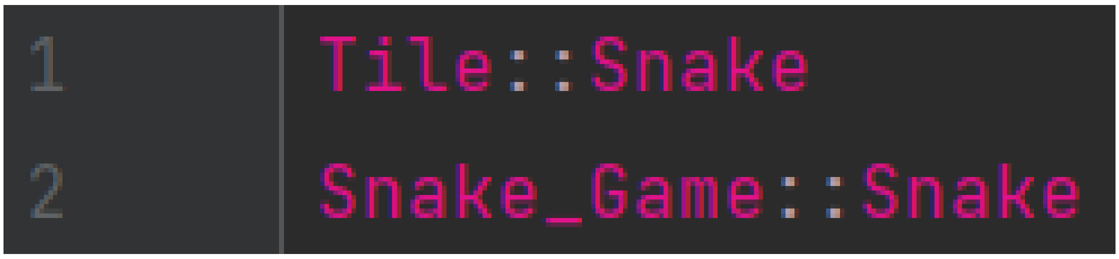
in this example Playing_Area is a child of Snake_Game



```
Snake_Game
    Playing_Area
```

Feature Reference Names

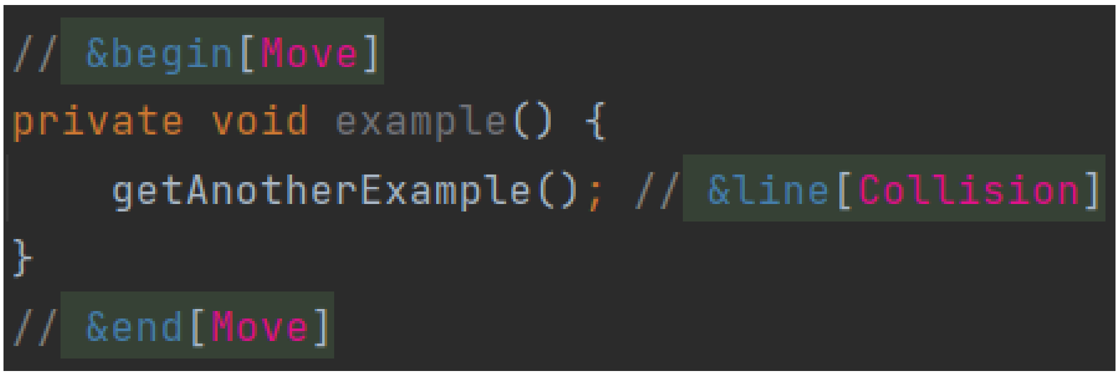
Inside the feature hierarchy model, features with the same name may appear twice or more often. To reference features uniquely the individual feature is pre-extended with its ancestor until the combined feature reference is unique (separated by "::"). This technique is called Least-Partially-Qualified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

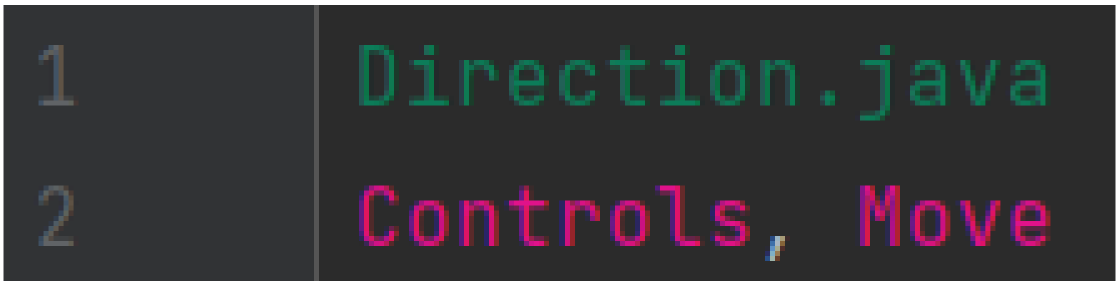
The feature-to-code mapping serves to link specific blocks and lines of code to one or more features. The parts of the source code which are mapped to a certain feature are called annotation scopes. An annotation scope is surrounded by annotation markers and contains at least one feature reference. In 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

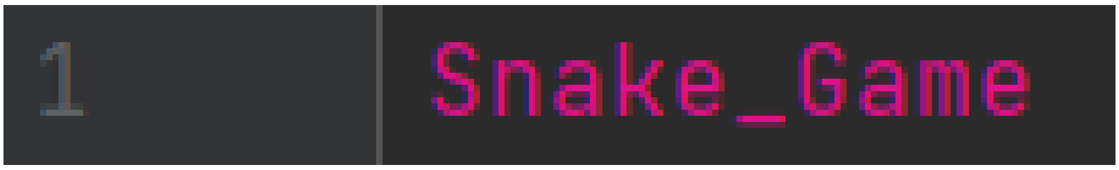
The feature-to-file mapping is a specialized file with the extension `.feature-to-file` and is used to map one or more file(s) and its/their content to one or more features. All content of the linked file is considered fully to be part of the given feature references. The mapping file must be stored 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
```

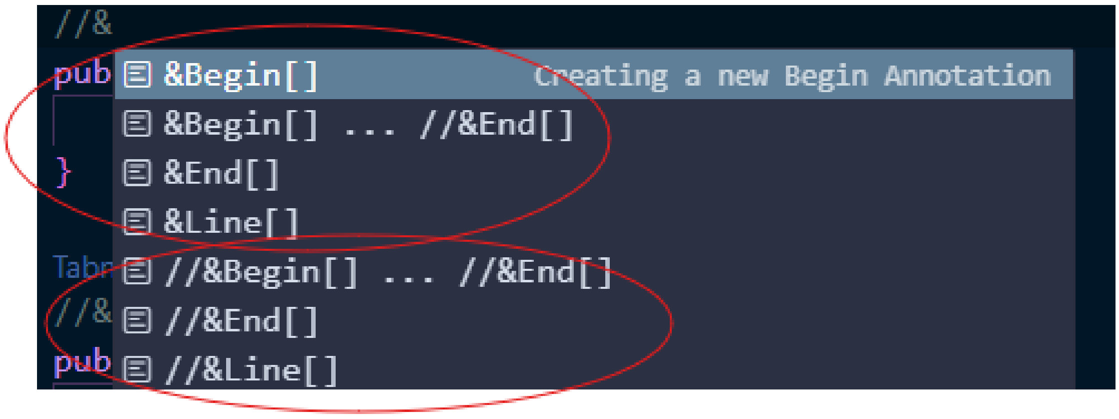
HAnS-LSP: Helping Annotate Software

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

HAnS-LSP supports: For the parsing to work properly, make sure to save your files after changes.

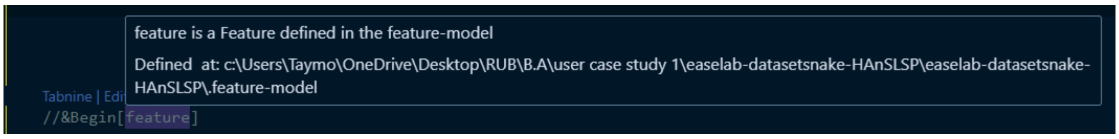
- Embedded Feature Annotations (check .feature-model to browse all defined features)

- Annotation Completion (ctrl space to get suggested completion)



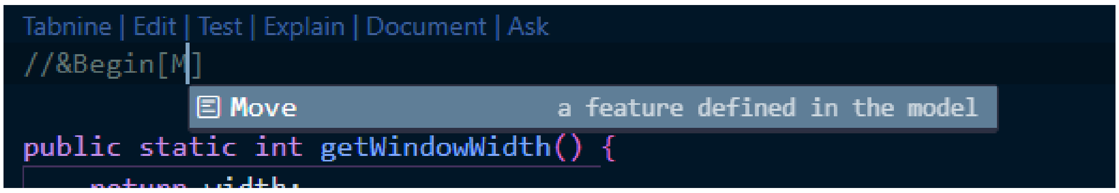
```
//&
pub  &Begin[]                        Creating a new Begin Annotation
     &Begin[] ... //&End[]
}    &End[]
     &Line[]
Tabn //&Begin[] ... //&End[]
//&  //&End[]
pub  //&Line[]
```

- Hover over annotation (either hover over the feature name, or over the annotation marker, e.g. begin, end or line). hovering will give information about that feature. **please note: hovering is keeping the mouse on the word, no clicks needed**



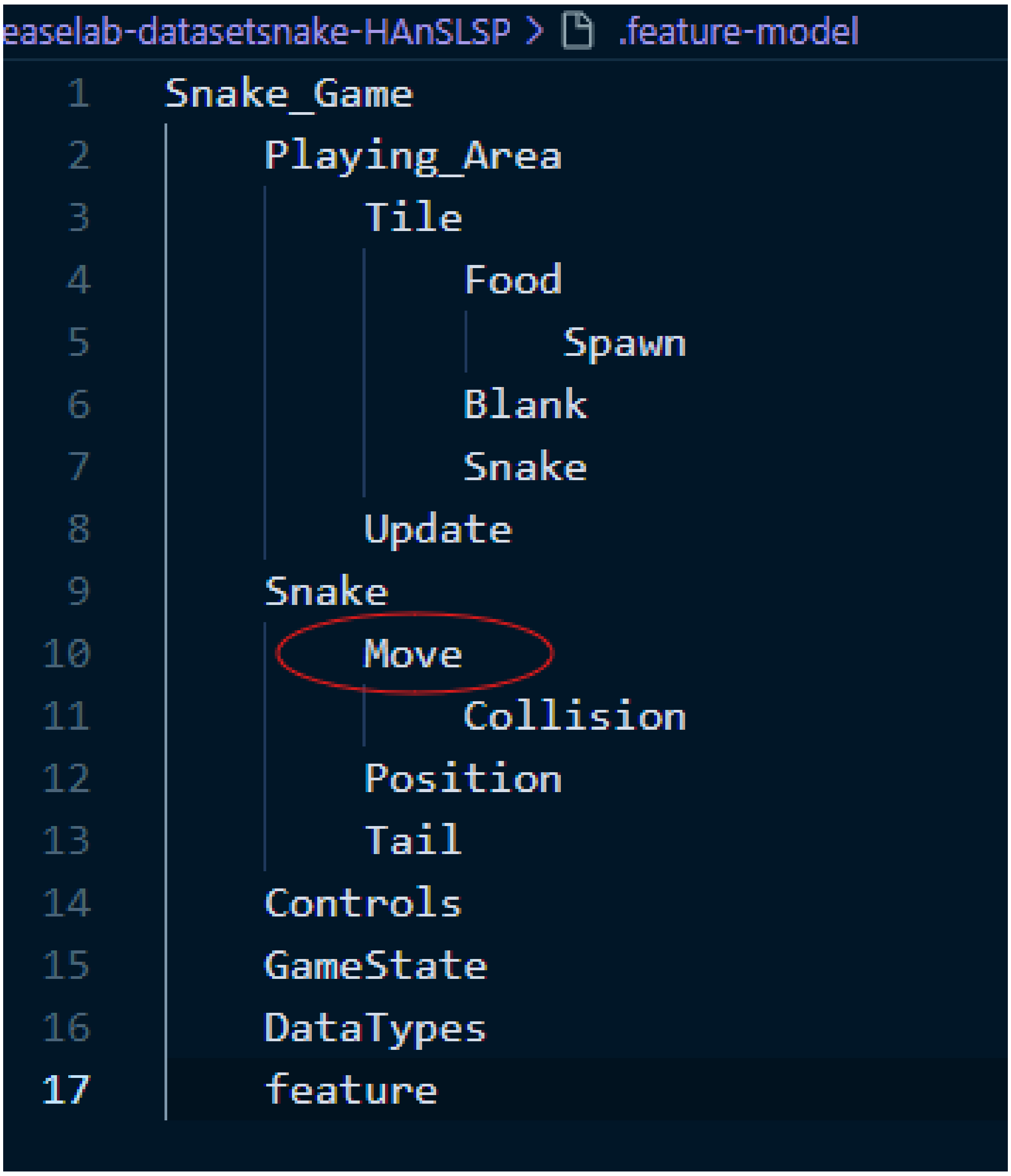
```
feature is a Feature defined in the feature-model
Defined  at: c:\Users\Taymo\OneDrive\Desktop\RUB\B.A\user case study 1\easelab-datasetsnake-HAnSLSP\easelab-datasetsnake-HAnSLSP\.feature-model
Tabnine | Edi
//&Begin[feature]
```

- feature name suggestion (ctrl space to get suggested name)



```
Tabnine | Edit | Test | Explain | Document | Ask
//&Begin[M]
            Move                  a feature defined in the model
public static int getWindowWidth() {
```

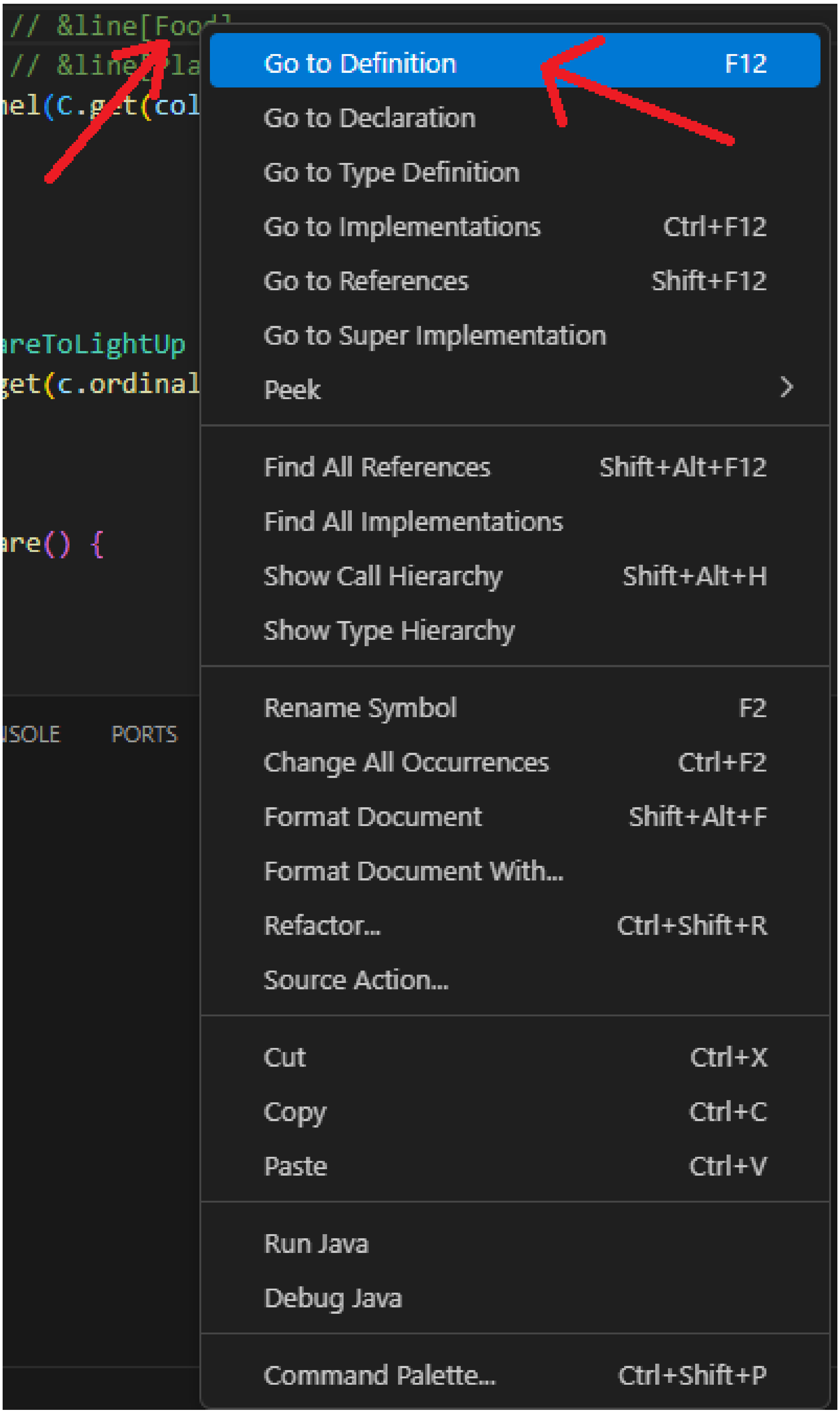
the “Move” feature is already defined in the .feature-model file



easelab-datasetsnake-HAnSLSP > .feature-model

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

- Definition: while in a code file (e.g. Window.java), right click on the feature's name written between [. . .] -> Go to -> definition. this will show you where the feature has been defined (IMPORTANT: Calling Definition when in the ".feature-model" file results in VS Code calling the reference method instead)

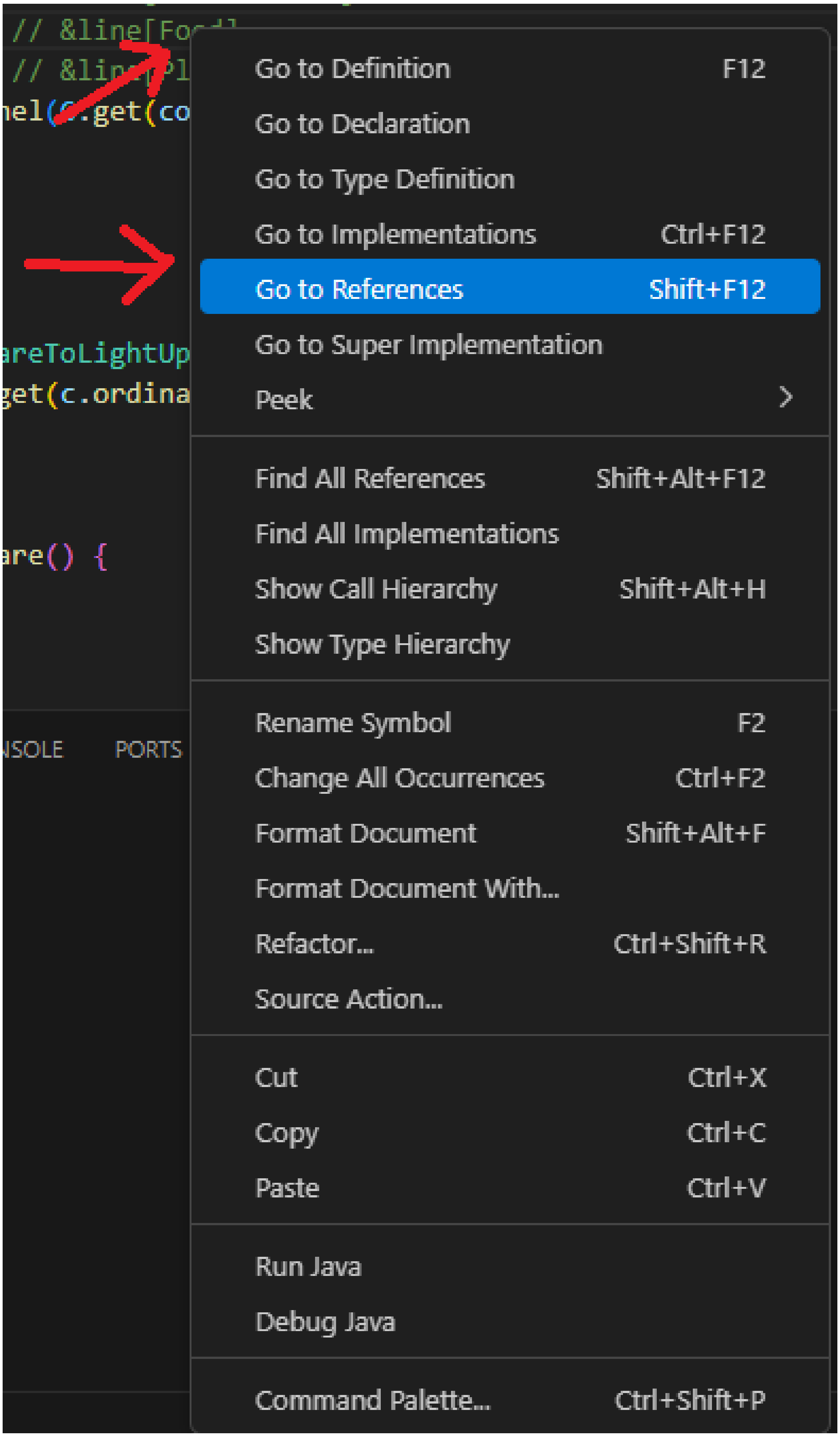


```
// &line[Food]
// &line[Pla
el(C.get(col

areToLightUp
get(c.ordinal

are() {
```

- Reference: while in .feature-model file, right click on the feature name -> GO to -> reference. The user will then see where this feature has been annotated in the all files that belongs to the feature model.



// &line[Fo
// &lin[Pl
el(c.get(co
areToLightUp
get(c.ordina
are() {
NSOLE
PORTS
Go to Definition F12
Go to Declaration
Go to Type Definition
Go to Implementations Ctrl+F12
Go to References Shift+F12
Go to Super Implementation
Peek
Find All References Shift+Alt+F12
Find All Implementations
Show Call Hierarchy Shift+Alt+H
Show Type Hierarchy
Rename Symbol F2
Change All Occurrences Ctrl+F2
Format Document Shift+Alt+F
Format Document With...
Refactor... Ctrl+Shift+R
Source Action...
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Run Java
Debug Java
Command Palette... Ctrl+Shift+P

Study Group

- Group 1
- Group 2

A.1.1 Tasks for group 1

Tasks to complete

During the task, take notes while coding, so you can answer the questionnaire after completing all the tasks. Make sure you turn on auto-save in Visual studio code or save your changes after every step for the extension to work correctly.

First part: Annotating with HAnS-LSP

Warmup task

Add a .feature-to-folder file in the graphics folder. * Verify that the feature Playing_Area is defined in the .feature-model file. * Map the feature Playing_Area to the new .feature-to-folder file by writing it into that file. * To know how to map a feature to a folder you can check again in the readme file *

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

Task 1:

Implement a method 'changeGrowth(int x){...}' that changes the size, the snake grows by and annotate it as feature (choose a fitting name). * Hint 1: the growth depends on the growth variable * The method should be added to /src/logic/ThreadsController.java file. The feature should be defined as a child feature of Snake in the Feature Model.

Reminder: Make sure you annotate the code you write!

Task 2:

Add a .feature-to-file file in the pojo folder. * Verify that the feature "Tile" is defined in the .feature-model file. * Map the feature "Tile" to the file Tuple.java. * To know how to map a feature to a file you can check again in the readme file *

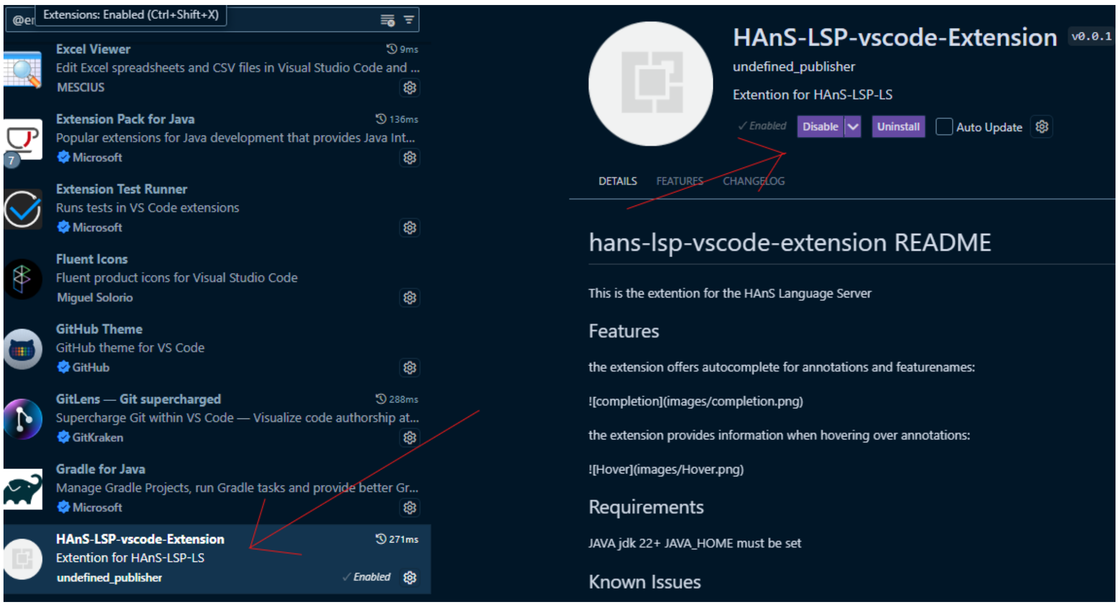
Task 3:

Add a new feature (Choose any name) to the .feature-model file, then annotate both functions, namely getWindowWidth() and getWindowHeight(), in a single block (//&Begin[you feature's name] ... both functions logic....//&End[you feature's name]). These functions can be found in the /src/graphics/Window.java file. * To know how to annotate a block of code you can check again in the readme file *

Second part: Annotating without HAnS-LSP

Disable the plugin.

1. To temporarily disable the HAnS-LSP extension -> go to extensions -> search for HAnS-LSP -> click the gear button at the right of an extension entry.
2. Click "Disable".



Extensions: Enabled (Ctrl+Shift+X)

Excel Viewer — 9ms
Edit Excel spreadsheets and CSV files in Visual Studio Code and ...
MESCIUS

Extension Pack for Java — 136ms
Popular extensions for Java development that provides Java Int...
Microsoft

Extension Test Runner
Runs tests in VS Code extensions
Microsoft

Fluent Icons
Fluent product icons for Visual Studio Code
Miguel Solorio

GitHub Theme
GitHub theme for VS Code
GitHub

GitLens — Git supercharged — 288ms
Supercharge Git within VS Code — Visualize code authorship at...
GitKraken

Gradle for Java
Manage Gradle Projects, run Gradle tasks and provide better Gr...
Microsoft

HAnS-LSP-vscode-Extension — 271ms
Extention for HAnS-LSP-LS
undefined_publisher — Enabled

HAnS-LSP-vscode-Extension v0.0.1
undefined_publisher
Extention for HAnS-LSP-LS
Enabled | Disable | Uninstall | Auto Update

DETAILS FEATURES CHANGELOG

hans-lsp-vscode-extension README

This is the extention for the HAnS Language Server

Features

the extension offers autocomplete for annotations and featurenames:

![completion](images/completion.png)

the extension provides information when hovering over annotations:

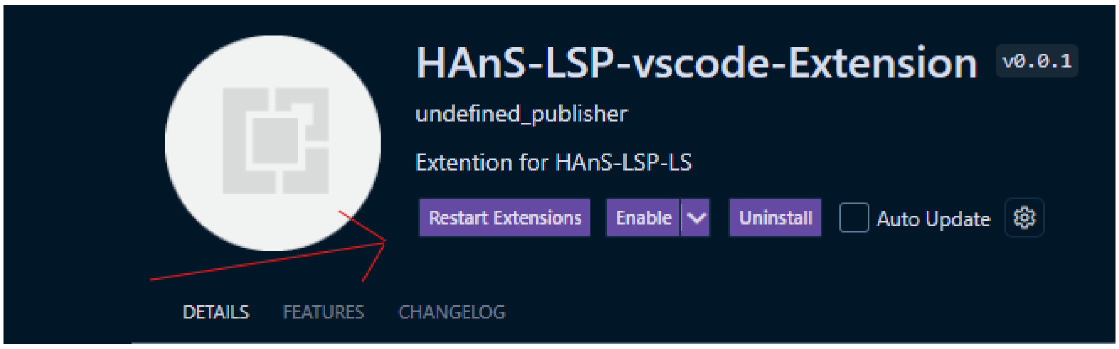
![Hover](images/Hover.png)

Requirements

JAVA jdk 22+ JAVA_HOME must be set

Known Issues

3. Click "restart extensions"



Warmup task

Add a `.feature-to-folder` file in the pojo folder. * Verify that the feature "Tile" is defined in the .feature-model file. * Map the feature "DataTypes" to the new .feature-to-folder file by writing it into that file. * To know how to map a feature to a folder you can check again in the readme file *

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

Task 4

Implement a method 'changeSpeed(long x){. . . }' that changes the speed of the snake and annotate it as feature (choose a fitting name). * Hint 1: the speed is dependent on the game speed Hint 2: gamespeed depends on the sleep time of the pause methode * * Hint 3: the pause time depends on the delay variable * The method should be added to /src/logic/ThreadsController.java file. The feature should be defined as a child feature of Snake in the Feature Model.

Reminder: Make sure you annotate the code you write!

Task 5

Add a `.feature-to-file` file in the logic folder. * Verify that the feature "Controls" is defined in the .feature-model file. * Map the feature "Controls" to the file KeyboardListener.java. * to know how to map a feature to a file you can check again in the readme file *

Task 6

In the file /src/graphics/Window.java check all feature annotations. then go to .feature-model file and check if each feature is defined there. If a feature is not defined in the .feature-model file add it.

Answer questions

After completing the tasks above, fill out the survey.

A.1.2 Tasks for group 2

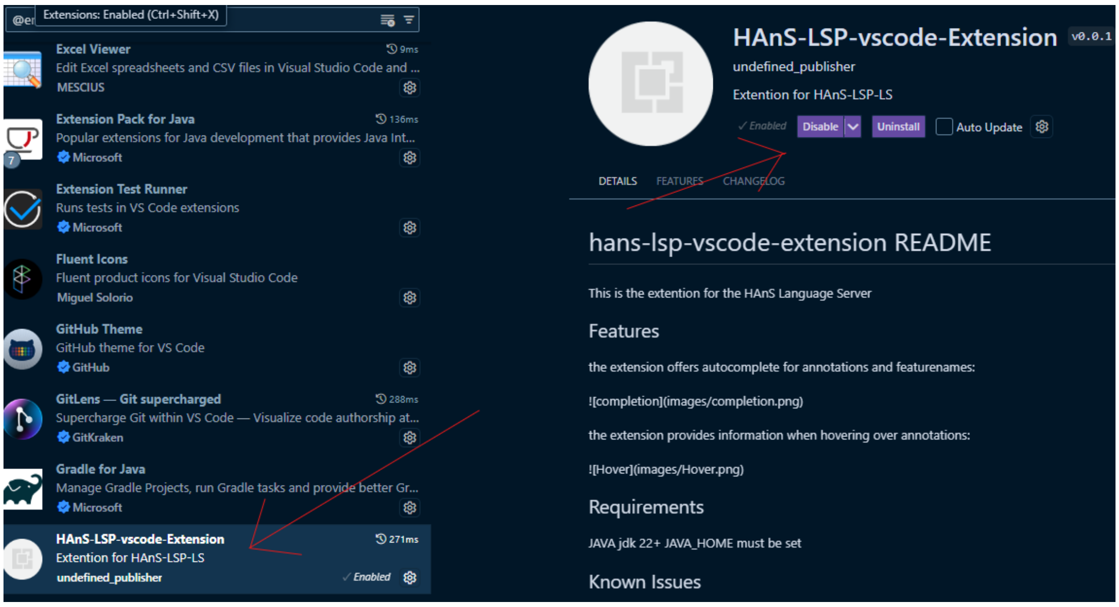
Tasks to complete

During the task, take notes while coding, so you can answer the questionnaire after completing all the tasks. Make sure you turn on auto-save in Visual Studio Code or save your changes after every step for the extension to work correctly. *See the README file for clarification*

First part: Annotating without HAnS-LSP

Disable the plugin.

1. To temporarily disable the HAnS-LSP extension -> go to extensions -> search for HAnS-LSP -> click the gear button at the right of an extension entry.
2. Click "Disable".


Extensions: Enabled (Ctrl+Shift+X)

Excel Viewer — 9ms
Edit Excel spreadsheets and CSV files in Visual Studio Code and ...
MESCIUS

Extension Pack for Java — 136ms
Popular extensions for Java development that provides Java Int...
Microsoft

Extension Test Runner
Runs tests in VS Code extensions
Microsoft

Fluent Icons
Fluent product icons for Visual Studio Code
Miguel Solorio

GitHub Theme
GitHub theme for VS Code
GitHub

GitLens — Git supercharged — 288ms
Supercharge Git within VS Code — Visualize code authorship at...
GitKraken

Gradle for Java
Manage Gradle Projects, run Gradle tasks and provide better Gr...
Microsoft

HAnS-LSP-vscode-Extension — 271ms
Extention for HAnS-LSP-LS
undefined_publisher — ✓ Enabled

HAnS-LSP-vscode-Extension v0.0.1
undefined_publisher
Extention for HAnS-LSP-LS
✓ Enabled | Disable | Uninstall | Auto Update

DETAILS FEATURES CHANGELOG

hans-lsp-vscode-extension README

This is the extention for the HAnS Language Server

Features

the extension offers autocomplete for annotations and featurenames:

![completion](images/completion.png)

the extension provides information when hovering over annotations:

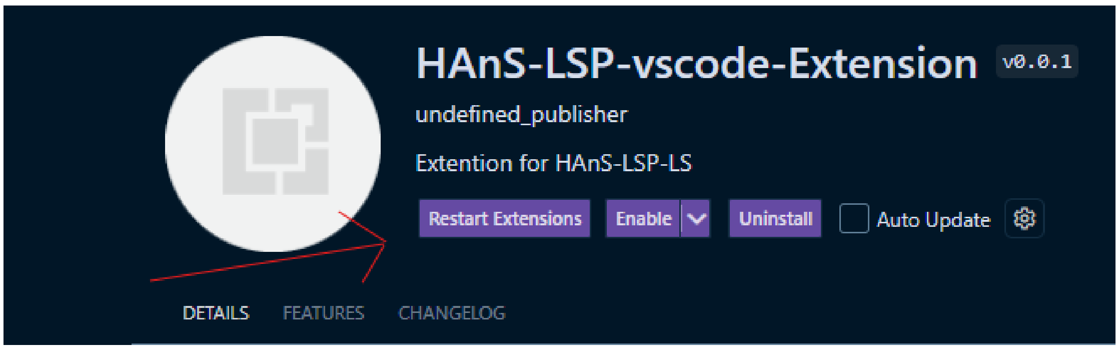
![Hover](images/Hover.png)

Requirements

JAVA jdk 22+ JAVA_HOME must be set

Known Issues

3. Click “restart extensions”



Warmup task

Add a .feature-to-folder file in the graphics folder. * Verify that the feature Playing_Area is defined in the .feature-model file. * Map the feature Playing_Area to the new .feature-to-folder file by writing it into that file. * To know how to map a feature to a folder you can check again in the readme file *

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

Task 1:

Implement a method ‘changeGrowth(int x){…}’ that changes the size, the snake grows by and annotate it as feature (choose a fitting name). * Hint 1: the growth depends on the growth variable * The method should be added to /src/logic/ThreadsController.java file. The feature should be defined as a child feature of Snake in the Feature Model.

Reminder: Make sure you annotate the code you write!

Task 2:

Add a .feature-to-file file in the pojo folder. * Verify that the feature “Tile” is defined in the .feature-model file. * Map the feature “Tile” to the file Tuple.java. * To know how to map a feature to a file you can check again in the readme file *

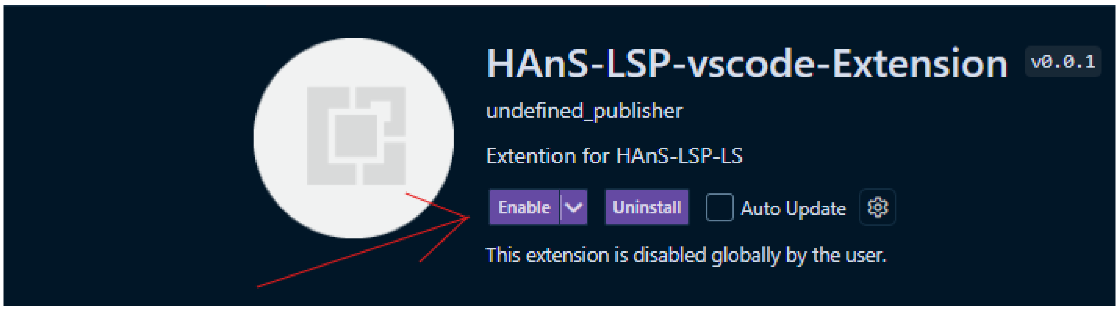
Task 3:

Add a new feature (Choose any name) to the .feature-model file, then annotate both functions, namely getWindowWidth() and getWindowHeight(), in a single block (//&Begin[you feature’s name] . . . both functions logic. . . .//&End[you feature’s name]). These functions can be found in the /src/graphics/Window.java file. * To know how to annotate a block of code you can check again in the readme file *

Second part: Annotating with HAnS-LSP

Enable the plugin.

1. to enable the HAnS-LSP extension -> go to extensions -> search for HAnS-LSP -> click the gear button at the right of an extension entry.
2. Click “Enable”.


HAnS-LSP-vscode-Extension
v0.0.1
undefined_publisher
Extention for HAnS-LSP-LS
Enable
Uninstall
Auto Update
This extension is disabled globally by the user.

Warmup task

Add a `.feature-to-folder` file in the pojo folder. * Verify that the feature “Tile” is defined in the .feature-model file. * Map the feature “DataTypes” to the new .feature-to-folder file by writing it into that file. * To know how to map a feature to a folder you can check again in the readme file *

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

Task 4

Implement a method ‘changeSpeed(long x){. . . }’ that changes the speed of the snake and annotate it as feature (choose a fitting name). * Hint 1: the speed is dependent on the game speed Hint 2: gamespeed depends on the sleep time of the pause methode * *

Hint 3: the pause time depends on the delay variable * The method should be added to /src/logic/ThreadsController.java file. The feature should be defined as a child feature of Snake in the Feature Model.

Reminder: Make sure you annotate the code you write!

Task 5

Add a `.feature-to-file` file in the logic folder. * Verify that the feature "Controls" is defined in the .feature-model file. * Map the feature "Controls" to the file KeyboardListener.java. * to know how to map a feature to a file you can check again in the readme file *

Task 6

In the file /src/graphics/Window.java check all feature annotations. then go to .feature-model file and check if each feature is defined there. If a feature is not defined in the .feature-model file add it.

Answer questions

After completing the tasks above, fill out the survey.

A.2 Questionnaire

HAnS-LSP Survey

This is a questionnaire about the HAnS-LSP services. (e.g. hover, references, definition, syntax highlighting, annotation completion); **Not about the code annotation itself.**
Link to HAnS-LSP: https://github.com/isselab/HAnS-LSP/tree/test-with-plugin
Please note that there are 2 groups, answer the following question to know to which group you belong.

* Gibt eine erforderliche Frage an

About you

Background information about your programing experience

1. To which of the following groups do you belong? *

   Markieren Sie nur ein Oval.

   ◯ 1st group: I used HAnS-LSP for tasks 1-3 and plain Visual Studio Code for the rest

   ◯ 2nd group: I used HAnS-LSP for tasks 4-6 and plain Visual Studio Code for the rest

2. How many years of experience do you have with Java? *

   Markieren Sie nur ein Oval.

   ◯ less than 1 year
   ◯ 1 - 2 years
   ◯ 2 - 3 years
   ◯ 3 - 4 years
   ◯ 4 - 5 years
   ◯ more than 5 years

3. What type of programming experience do you have? *

Wählen Sie alle zutreffenden Antworten aus.

- ☐ Educational
- ☐ Professional

4. Have you ever annotated features in any project before? *

Markieren Sie nur ein Oval.

- ◯ yes
- ◯ no

5. If yes, which tool did you use to do it?

Wählen Sie alle zutreffenden Antworten aus.

- ☐ HAnS
- ☐ Sonstiges: ______________________

Experience using HAnS-LSP compared to without it

The following questions are about your experience using our feature annotation tool and its services(hover, references, definition, syntax highlighting, annotation completion) **not the general benefits of code annotation**.

6. To which extent would you agree with the following statements? *

Markieren Sie nur ein Oval pro Zeile.

	Strongly disagree	Disagree	Neither agree or disagree	Agree	Strongly Agree
Creating annotations using the LSP was faster than without	◯	◯	◯	◯	◯
Creating code annotations using the LSP was easier than without	◯	◯	◯	◯	◯
The code completion for creating annotation is useful to me	◯	◯	◯	◯	◯
it was easy to create .feature-to-file mappings	◯	◯	◯	◯	◯
it was easy to create .feature-to-folder mappings	◯	◯	◯	◯	◯

7. To which extent do you agree with the following statements regarding the time required to complete tasks using HAnS-LSP? *

Markieren Sie nur ein Oval pro Zeile.

	Strongly disagree	Disagree	Neither agree or disagree	Agree	Strongly Agree
check if the feature is defined in the .feature-model file was faster using the LSP	◯	◯	◯	◯	◯
it was faster to locating annotation of a feature in the code	◯	◯	◯	◯	◯
using the LSP it was faster to locating feature in .feature-to-file	◯	◯	◯	◯	◯
it was easy to browse .feature-to-folder mappings	◯	◯	◯	◯	◯

Usabilty of HAnS-LSP

The purpose of this form is to gather valuable feedback to enhance HAnS-LSP, improving both its usability and performance. Our ultimate goal is to provide a seamless, user-friendly experience for all users.

8. To which extent do you agree with the following statements regarding HAnS-LSP? *

Markieren Sie nur ein Oval pro Zeile.

	Strongly disagree	Disagree	Neither agree or disagree	Agree	Strongly Agree
I think that I would like to use HAnS-LSP frequently.	◯	◯	◯	◯	◯
I found the system unnecessarily complex.	◯	◯	◯	◯	◯
I thought HAnS-LSP was easy to use.	◯	◯	◯	◯	◯
I think that I would need the support of a technical person to be able to use this system.	◯	◯	◯	◯	◯
I found the various functions in this system were well integrated.	◯	◯	◯	◯	◯
I thought there was too much inconsistency in this system.	◯	◯	◯	◯	◯

I would imagine that most people would learn to use this system very quickly.	⬭	⬭	⬭	⬭	⬭
I found the system very cumbersome to use.	⬭	⬭	⬭	⬭	⬭
I felt very confident using the system.	⬭	⬭	⬭	⬭	⬭
I needed to learn a lot of things before I could get going with this system.	⬭	⬭	⬭	⬭	⬭

9. What about the plugin would keep you from using it? *

10. What about the plugin do you like? *

11. What were the biggest challenges you have faced while using HAnS-LSP? *

12. What additional features do you think would enhance the usability of LSP-HAnS? *