CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Help Annotating Software "HAnS" - Visualisation

IntelliJ IDE plugin for feature-oriented software development

Bachelor's thesis in Computer science and engineering

Kenny Bang
Johan Berg
Seif Bourogaa
Lucas Frövik
Alexander Grönberg
Sara Persson

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

Bachelor's thesis 2021

Help Annotating Software "HAnS" - Visualisation

IntelliJ IDE plugin for feature-oriented software development

Kenny Bang
Johan Berg
Seif Bourogaa
Lucas Frövik
Alexander Grönberg
Sara Persson

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

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

Help Annotating Software "HAnS" - Visualisation
IntelliJ IDE plugin for feature-oriented software development
Kenny Bang Johan Berg Seif Bourogaa Lucas Frövik Alexander Grönberg Sara Persson

Supervisors: Thorsten Berger, Ho Quang Truong
Examiner: Jan-Philipp Steghöfer

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

Typeset in LaTeX
Gothenburg, Sweden 2021

Help Annotating Software "HAnS" - Visualisation
IntelliJ IDE plugin for feature-oriented software development
Kenny Bang, Johan Berg, Seif Bourogaa, Lucas Frövik, Alexander Grönberg, Sara Persson

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Developing large software systems that are feature-oriented is a complex and time-consuming task that is further hindered by the recurring and repetitive undertaking of feature lookup. However, feature lookups' inefficiency can be solved using embedded annotations that quickly and easily allow for recovery of features and their locations. Feature lookup is one of the problems this paper aims to solve by creating a plugin for IntelliJ that enables the use and exploration of feature annotations through multiple visualisations. With this plugin, the user will be able to: visualise where in the code its features are implemented, visualise where the code intersects of two or more features and see useful metrics related to the implemented features.

The project results were determined by utilising a user review of the plugin and comparing the completed plugin with predetermined product specifications. While some functionality were not able to be implemented due to limitations in the project, the results achieved suggest that our plugin provides an intuitive and easy way to perform feature lookup. The completed product thus serves as a useful tool to feature based programming for the IDE IntelliJ.

Keywords: Annotations, Features, Feature location, Visualisation, IntelliJ, Plugin, Software development

Sammandrag

Att utveckla stora "feature-oriented" mjukvarusystem är en komplex och tidskrävande uppgift som hindras ytterligare av återkommande och den upprepade uppgiften av feature lookup. Däremot, ineffektivitet av feature lookup kan lösas genom att använda inbäddade annoteringar som med lättnad tillåter återvinning av features och deras plats i programmet. Feature lookup är ett av problemen som denna rapport syftar till att lösa genom att skapa ett plugin för IntelliJ som möjliggör användningen och utforskningen av feature annotations genom flertalet visualiseringar. Med detta plugin kommer användaren kunna: visualisera var i koden dess features är implementerade, visualisera var features delar på samma kod och kunna kolla på användbar metrik relaterat till de implementerade features.

Projektets resultat fastslogs genom en användarundersökning av pluginet och genom att jämföra pluginet med förutbestämda produktspecifikationer. Då viss funktionalitet kunde ej implementeras p.g.a begränsningar i projektet, antyder de uppnådda resultaten att vårt plugin tillhandahåller ett intuitivt och enkelt sätt att utföra feature lookup. Den färdigställda produkten tjänar således som ett användbart verktyg för feature based programming för IDEn IntelliJ.

Contents

Glossary

Annotations are code markers that allow the developer to track what and where they have written a feature. Like a bookmark, the annotation allows the developer not to lose time and energy recalling and scavenging for the page or location they are searching for by keeping track of necessary location information.

Features represents an observable functionality in the Software.

Feature tangling/"Tangling" means that features are connected to each other in a way where a change in one feature could result in a change in another feature.

Feature locating/"feature lookup" is the task to find where a feature is located in a project.

Metrics are "a set of numbers that give information about a particular process or activity" [11].

Feature Metrics are numbers calculated from annotated features.

HAnS Is an acronym for Help Annotating Software.

HAnS-Vis is the plugin developed in this project for visualising annotated features. "Vis" refers to visualisations.

HAnS-Edit is a plugin developed by another group, consisting of masters students, for annotating features, which will be referred to as HAnS-Edit. "Edit" refers to editing support of annotations.

IDE Integrated Development Environment.

HTML Hypertext Markup Language, the standard language for web browsers.

CSS Cascading Style Sheet, the language to style an HTML document.

API Application Programmable Interface.

JVM Java Virtual Machine.

1
Introduction

A feature is a term commonly used when describing software systems. In the context of software development, a feature represents an observable functionality in the software, an abstraction to describe the software's functionality. Developers and project managers use these abstractions to describe and communicate software systems to each other and clients. A significant portion of software development uses features and requires the mechanic of adding new features, maintaining features, or extracting features in old software projects. Furthermore, managing features in software systems requires knowledge about the features, their locations and their implementation. However, this is not always adequately or correctly documented, requiring developers to manually find and extract all the features and locations in a project. Moreover, this aforementioned "feature-lookup" task is both time-consuming and error-prone [1], [2], [3].

To give an example of a system where feature-lookup creates issues when not handled well, imagine a large and old video game with multiple developers and features. In this hypothetical example, the features are poorly documented, the original developers are no longer part of the team, and the features are too many and complex to remember on an individual basis. Much of the development time will go to finding, documenting, fixing or extracting features in this system. If a new developer wants to look up the feature that handles the player character's movement in the game, they might have to sift through much of the code or code structure to find what they are seeking.

The example above is an example of a flawed Feature-based system. However, feature-oriented programming is common, both good and bad examples, and for developers creating these systems, so is the IDE Intellij. According to a survey conducted in 2020, IntelliJ is the most widely used IDE among JVM developers as 62% of them use the IDE [4, p. 33].

1.1 Aim

This project aims to save time for developers of feature-based systems by creating an IntelliJ plugin that can visualise features and feature metrics. Visualising features using interactive graphs can help developers efficiently handle time-consuming tasks such as feature lookup. Furthermore, to achieve this efficiency, the following four goals were created to steer the created product towards the above aim.

Goals of the plugin:

- Being able to extract features from code to use as data for visualisations.
- Displaying where the extracted features can be found in the code, so developers do not need to put time into looking features up manually.
- Displaying how the code of the extracted features are connected to prevent developers from making code changes that will unintentionally affect more than one feature.
- Displaying an overview of the features a developer is working on in the form of metrics.

The project's objective is also to have this plugin available for as large a user base as possible, making IntelliJ a prime candidate to develop this plugin in, as IntelliJ currently does not have any plugins visualising features.

Further expansion of the plugins visualisation goals are explored later in the text (See chapter 3 for a product specification that expands on the goals mentioned here by presenting the visualisations, and the graphical views specifications').

1.2 Scope

The plugin HAnS-Vis is just one of two parts of the complete plugin HAnS. The second part of the plugin, HAnS-Edit, is developed in tandem with a team of master students and is centred around providing editing support for feature-based programming in IntelliJ. The overarching goal of both of these plugins is to merge them to create a more comprehensive plugin that handles both editing and visualisation of features.

This project is limited to only the visualisation of features, feature extractions, and feature metrics. This limitation is due to the fact that the visualisation part is expected to require a substantial amount of time and because there is already a master thesis project working on the editing support of an accompanying plugin, HAnS-Edit.

By narrowing the project down to only the visualisations, the focus can be put on one important aspect, while the master students' project implement the second aspect, and in the end have two products that can cooperate as an all-encompassing plugin for feature development.

Creating an IntelliJ plugin with graphical visualisations necessitated a visualisation library. Additionally, this library required the delimitation of being regularly maintained, well documented, and use a BSD-compatible license [10] for purposes of product longevity and open-source plugin publication. This limits the available libraries for the project.

1.3 Report Structure

The following chapter, background, aims to prepare the reader for the following chapters. Although the multiple informative topics in chapter two may seem varied, they are all necessary for understanding the subsequent text.

Chapter 3, product specifications, act as an extended goal chapter, only exclusively on the visualisations. The product specification outlines the ideal product in detail. All features and functionality that would be ideal for adding to the plugin, regardless of the limitations on the project, are described here. The minimum requirements of the chosen visualisations precede the specifications.

Chapter 4, methodology, details how the project's libraries fetch data from feature annotations and visualise them into easy-to-use graphs. It also describes the process and methodology of a user review, where test users tried the project in an interview format.

Chapter 5, results, state the level of completion the project reached by comparing it with the product specification and details the implementation.

The final chapter, Discussion, discusses the result, the user review, and what could be improved in the future. Moreover, it examines what conclusions can be drawn from creating a visualisation plugin for feature-based programming.

2
Background

This chapter presents previous projects that are similar to the project being created and related research. The chapter's primary focus is to clarify feature annotations, describe visualisations, and give helpful information for following subject comprehension.

2.1 Related works

Former related research and projects have been conducted such as FeatureDashBoard and FLOrIDA [5],[1]. Both are tools for extracting and visualising annotated features. They both support the importance and necessity of annotation-based plugins and systems. A commonly noted benefit in papers on this subject is the time saved by minimising the feature localisation, and the structural and efficiency gain by implementing these systems [1], [5].

Researchers have created an open-source library called FAXE - Feature Annotation eXtraction Engine - that automatically parses and receives annotations from software projects and creates an object model with this information [6]. FAXE has then been used to create a dashboard, and this dashboard visualises how files and folders map to features and which features a specific file contains [1], [5]. This previously mentioned dashboard, called FeatureDashBoard, is a plugin created for Eclipse [5], but currently, there are no similar plugins for IntelliJ.

2.2 Annotation's purpose and use

Annotations are code markers that allow the developer to track where a feature is implemented. If two or more features share the same annotation then those two features depend on the same code, this means the features are tangled. There are three different types of annotations, namely:

- In-file annotations
- File annotation
- Folder annotations

In-file annotations are annotations declared directly in the code and allow the developer to declare that a particular line or block of code belongs to certain features. There are two types of in-file annotations:

- Line annotations
- Block annotations

Line annotations are used when a single line of code belongs to a feature. This is declared by writing a *&line[feature-name]* in a comment on the line the code is written.

```
Tuple position = new Tuple( x: 10, y: 10);
ThreadsController c = new ThreadsController(position); // &line[Move]
c.start();
```

Figure 2.1: The feature **Move** is annotated to the line *ThreadsController...*

Block annotations are used when multiple lines of code belongs to a feature. These lines are encapsulated with *&begin[feature-name]* and *&end[feature-name]* statements in comments above and under the block to declare which lines belong to a feature.

```
// &begin[Update]
public void ChangeColor(Color d){
    this.setBackground(d);
    this.repaint();
}
// &end[Update]
```

Figure 2.2: The feature **Update** is annotated to the method *ChangeColor()*

File annotations are declared in a file called *.feature-to-file*, which is located in the same folder as the mapped files and specifies that entire files belong to certain features.

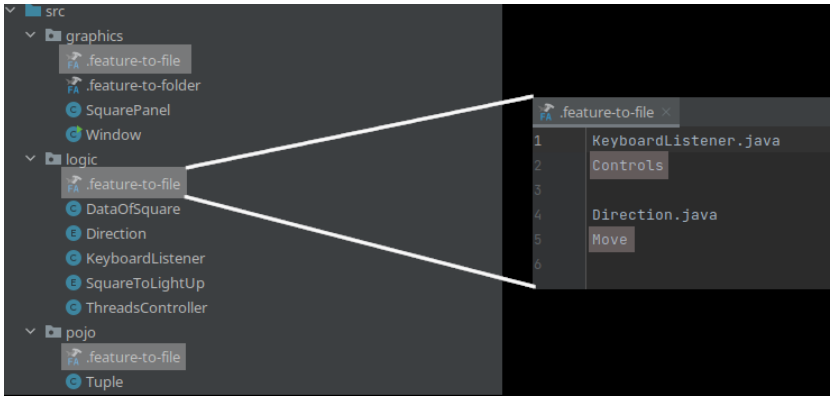

Figure 2.3: The feature **Controls** is annotated to the file *KeyboardListener.java* and the feature **Move** is annotated to the file *Direction.java*

Finally, **folder annotations** are declared in a file called .feature-to-folder. This file is located in the folder that is annotated and specifies that that folder and its content belong to certain features.

Figure 2.4: The feature **Graphics** is annotated to the folder *graphics* and all of its content

As annotations can be used to keep track of features and where said features are implemented, developers can minimise the time spent locating features, which developers spend a major part of their time doing [1], [7]. Annotations are also good for software traceability and software maintainability as it allows developers to document differences between software directly in the project. Annotations could be read by software to create an object model, which could then be used to give the user a graphical representation of which features exist in a project, where said features are located, the structure of the software and the degree of scattering for a feature in a project, i.e. how many different files and folders a feature is implemented in.

2.3 Visualisation's purpose and use

Data visualisation is the graphical representation of information wherein the data is presented more naturally for the human mind to comprehend than rows and rows of

texts that could be exceedingly difficult to read, providing an accessible way to see and understand the information. By visualising the feature annotations and where said features are located, the user could easily locate where features are present in the project, rather than going through a substantial amount of folders and files. Likewise, by visualising which features are connected and dependent on each other in the project, the user can understand which features would be impacted if one feature was changed at a specific location. FeatureDashBoard offers multiple visualisations that aid the user with the points mentioned above. The first two visualisations are called Feature-to-File view and Feature-to-Folder view. Features, files and folders are visualised in these views with a connection between them. By using these connections, a user would be able to see in what files and folders a feature is located. Another visualisation is the Feature Tangling view. There, when two features are tangled with each other, it says that they are connected and dependent. Any modifications to one of the features could result in a change in the other. To prevent this from happening, the Feature Tangling view is used to see what features are tangled with each other. Lastly, the Metrics view shows calculated feature-related numbers using a table; a few of them are:

- Number of file annotations - How many file annotations a feature has in the project.
- Number of folder annotations - How many folder annotations a feature has in the project.
- Scattering degree - How many folder, file, and in-file annotations a feature has in the project.
- Tangling Degree - How tangled a feature is with other features in the project.
- Lines of Feature Code - How many lines of code that belongs to the feature.

The additional information that is given with the Metric view provide an overview of the features that could be useful for a developer. Two features could be compared on a specific metric to see which one is most tangled or has the most lines of code and possibly identify which feature impacts the system the most.

2.4 HAnS-Edit

HAnS-Edit which has been developed in tandem with this project provides syntax highlighting and code completion for feature annotations. Furthermore it is also used by this project to calculate feature metrics.

3
Product specifications

This chapter presents the ideal visualisations for the project, with no regards to any limitations that are put up during development. This chapter also presents the minimum requirements for each visualisations in the created plugin. Furthermore, as the project aims to visualise feature data there are detailed specifications for each visualisation. The product specification functions as a more specific goal chapter for the visualisation, to later compare with the implemented product in the result and discussion chapters.

There are five presented visualisations in this chapter. To ensure a minimal viable product the specifications for the visualisations are divided into two sets:

- Essential visualisations: The visualisations deemed necessary for a finished minimum viable product.
- Non-essential visualisations: The visualisations deemed optional or unnecessary, and features that will not be in the final implementation but that could be implemented beyond this project's deadline.

To determine if a visualisation is essential for the minimum viable product, a visualisation must fulfill all four of the following requirements:

- Has the visualisation been used previously in similar projects or research and is therefore expected by the user?
- Will the visualisation serve the purpose of efficiency for developers more than other visualisations?
- Will it be possible to create this visualisation in IntelliJ, using the provided libraries?
- Does it serve an essential purpose to the plugin?

Visualisations failing to fulfil all four requirements are classified as non-essential and, as such, are not required to be included in the implemented edition of the plugin. Determining the first item, of the above four requirements, was done by looking at the earlier created similar projects mentioned in 2.1, such as FeatureDashBoard. Determining the second item, of judging the visualisations level of service to the purpose of

efficiency, was done by internal discussions within the project's group and supervisors. This requirement prioritises the more useful visualisations according to the project's aim. Determining the third item was done by researching IntelliJ's capabilities with specific visualisations in mind and rejecting the ones that appeared incompatible or impossible in the allocated time or with the available resources. Determining the last item was done by internal discussions within the project's group and supervisors.

3.1 Essential visualisations

In this section the visualisations deemed essential and their minimum requirements are listed, and the visualisations are briefly summarised.

- Feature Location Visualisation
  The visualisations main purpose is to locate the files and folders of features with the help of a graph (see Figure 3.1). The minimum requirements of the Feature Location Visualisation is:
  - A visualisation with folder annotations of an identifiable node shape.
  - A visualisation with file annotations of an identifiable node shape.
  - Clearly show in what specific files and folders any specific feature can be found implemented.
  - To help locating features in a project.
  - A tool window that shows the visualisation.
  - A feature (or more) has to be selected to visualise the graph.

  This visualisation is essential to the project since knowing where a feature is located will greatly benefit developers who could easily find and work on a feature in a project without reading through code to locate it.

- Feature Tangling Visualisation
  The visualisations main purpose is to show feature tangling using a graph (see Figure 3.2). The minimum requirements of Feature Tangling Visualisation is:

  - Show one central feature, and its tangled features.
  - Show the degree of tangling through the size of the edges.
  - Allow for clarification of tangling degree, by hovering over a node and then clearly displaying the tangling degree number.

  This visualisation is designated essential because it has been used by previous similar projects such as Feature Dashboard [5]. More importantly, the visualisation allows the user to view features that tangle with a specifically chosen feature. This is helpful to prevent developers from making changes in the code that accidentally affects unintended features.

- Metrics view
  The visualisation's main purpose is to present the metrics of features, the specific metrics are brought up in section 3.3.3. The minimum requirements of Metrics view are:
  - Show metrics for any selected features.
  - Have a clear and functional table to display the view.
  - Have a radio button, or similar element, to toggle on all or toggle off all but one feature.
  - Have a radio button, or similar element, to toggle on all or toggle off all but one metric.

  This view is essential in the project since it is a vital tool for developing a project or maintaining and reusing an old project. The metrics view was also used in previous similar works such as FLOrIDA [1] and Feature Dashboard [5], making the metric view an expected visualisation to have in a feature annotation visualisation related plugin.

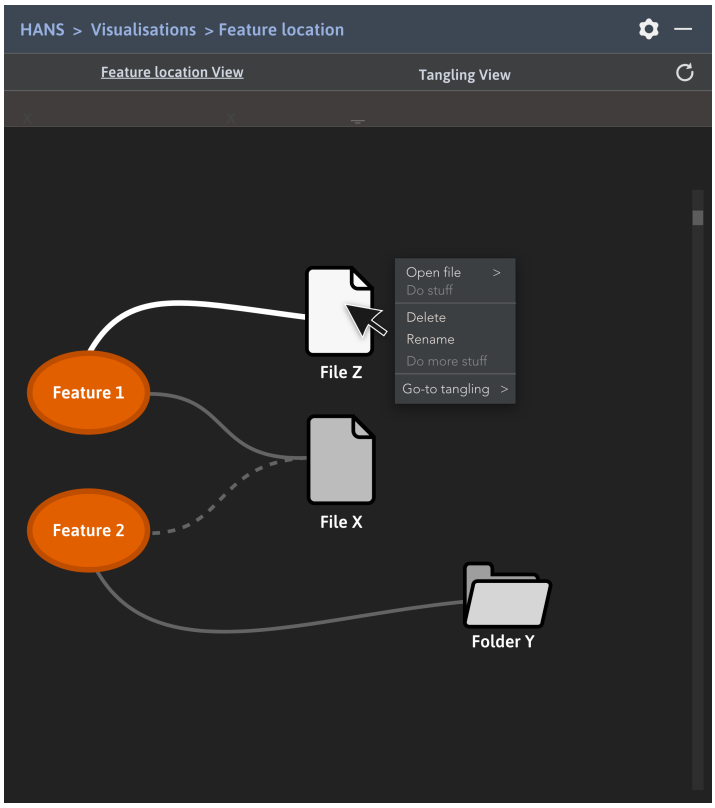

Figure 3.1: Mock-up of the Feature Location Visualisation

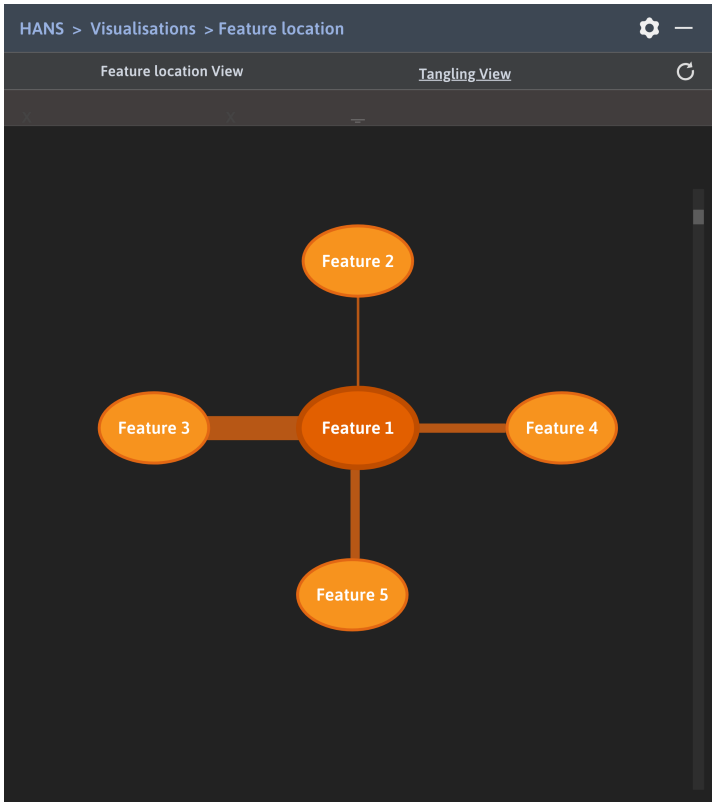

Figure 3.2: Mock-up of the Feature Tangling Visualisation

3.2 Non-essential visualisations

In this section the visualisations deemed non-essential are listed and briefly summarised.

- History view
  The visualisations main purpose is to show project changes over time
  The history view's visualisation features a line graph showing the metric value on the y-axis and the number of commits ago on the x-axis. This would allow the user to view changes in metrics over a time unit, commits, meaning uploaded or saved changes to the code.
  This visualisation is designated non-essential as the ability to view how feature metrics have changed from commit to commit is not essential to feature annotations systems but an extra feature that could enhance or lightly help the development experience.
- Common features view
  The visualisations main purpose is to show common features between projects
  The common features' visualisation is based on a similar visualisation from Feature Dashboard [5]. The visualisation is supposed to visualise the features from different projects or variants that the projects have in common.
  The common features visualisation is designated non-essential because features

are extracted from one instance of IntelliJ. Since multiple projects are not allowed in one instance of IntelliJ there is not a way to extract features from multiple projects.

3.3 Visualisation specifications

In this section, the visualisations' specifications are presented. The Following Visualisation specifications are ideal blueprints for the functionality and design. The project does not require the implementation of all of the specifications to achieve the project's goals.

3.3.1 Feature Location Visualisation

Purpose

The Feature Location Visualisation aims to have a way to visualise all features in a code base and display where the features are implemented. Showing where features map to files and folders within the project aids users in quickly finding the correct code to edit when working with feature-based programming.

Nodes and Edges

The Feature Location Visualisation should offer the user an interactive graph view where features, files, and folders are represented as nodes with edges mapping the relationship between them. The nodes and edges should change if the user clicks or hovers over them to give feedback to the user when interacting with the graph. The edges should appear as thin curved grey lines connected to a non-selected node, and thicker more colourful lines when connected to a selected node. Through the increased thickness, the edges visualise which feature exists in which file/folder. Moreover, when a feature is selected, the visual changes make the relevant connections clearer, allowing the users to perform actions such as finding the feature's relevant files and thus finding the relevant code in easy and efficient ways.

In the Feature Location Visualisation, there should be three types of nodes and two types of edges. The three nodes should be: features, whose icon is an orange coloured circular shape; files, whose icon is a standard file icon; and lastly, folders, whose icon is a standard folder icon. The two edge types should be: solid and dashed. The solid line should represent file and folder annotations, and the dashed line should represent code annotations, meaning code-line and code-block annotations.

Hovering

Further interactivity should appear with hovering. Hovering over any element in the graph, whether it is an edge or node, supports the users' engagement through visual feedback. By hovering, the user confirms an interest in the element and the element, in return, shows interactability or further relevant information when hovered over. Hiding information behind a hovering action helps hide less prioritised information until the user specifically requests it. When hovering over a node, the previously hidden information should appear in a static box to the right of the visualisation window. The information displayed in the box should vary depending on what type of node it is. If the node represents a feature, no information should be displayed. However, there should be an option to display some metrics from the metrics view here if the user chooses. Neither should any information be displayed if hovering over an edge, as these two elements have no additional information that needs to be displayed relevant to this visualisation. If the node is a file, the information box should show the user the file's folder. If the file only contains file annotations, this is the only information that is displayed. However, if the file contains code annotations, the box should also display how many lines of code each annotated feature contains; if the node is a folder, the box should display what files are found within that folder.

Filtering

The feature to file visualisation should toggle between four filter views, "Visualise all annotations", "Visualise line annotations", "Visualise file annotations", and "Visualise folder annotations". These options should be available in the top left-hand corner of the visualisation window in a dropdown menu called "Select annotations to visualise". Applying filtration to this visualisation is another functionality that makes it easier for users to find the data they are looking for.

Context menu

Right-clicking on a node should show a drop-down menu, where the user can either rename or delete the node, both in the graph and in the code where they are implemented. File, feature and folder nodes should have specific options in this menu. Files should have the option of "open", opening to the code in IntelliJ. Folders should have the option of "expand", which should expand the folder in the project structure if it was not already. If the folder is expanded it instead should say "collapse", to collapse the opened folder. Features should have the option of "go-to tangling", which should send the user to a view called Feature Tangling Visualisation, with the feature selected and in focus. There should also be the options of renaming or deleting the features, files, and folders.

Direct navigation interaction

By clicking on a file, the user should go to where the feature is implemented in that file.

Meaning the plugin should open the code in IntelliJ and to the correct position. This code navigation requires the plugin to receive the exact path of the files that a feature maps to, which the HAnS-Edit plugin should handle. Being able to quickly navigate from visualised annotations to where they are written in the code could save time for developers.

Feature selection

One of the features that HAnS-Edit aims to provide is the ability to select which features that a user wants to focus on. By using the HAnS-Edit API to extend this functionality to the Feature Location Visualisation, the user should be able to select which features should be visualised at one time to give users an even easier time to localise feature implementation. The feature selection functionality should also be extended to the other views of this project to hide undesired information from the user.

3.3.2 Feature Tangling Visualisation

Purpose

The Feature Tangling Visualisation aims to give developers an idea if changes to the code will affect certain features. Specifically, what features will be affected if a developer were to change the code of another feature. Tangling, in this case, means that two or more features are mentioned in the same annotations. Thus, this means two features are dependent on the same code.

Edges

The visualisation should show which features tangle with a different selected feature, which the edges in the visualisation should visualise. Furthermore, to which degree the feature is tangled with the other features through the varying thickness of the connecting edges.

Presentation

The visualisation should have a mind-map-like structure with only feature nodes. The graph should work similarly to the graph in Feature Location Visualisation. The design and elements should have similar design to Feature Location Visualisation.

Hovering

Similarly to the Feature Location Visualisation, there should be a static box to the visualisation's right, showing additional information when hovering over the nodes. This

hovering info should also expand to the edges to show the degree of tangling numerically. In this visualisation the static box should also show more data on the hovered feature. However, if the user hovers over the selected feature, no information should be displayed as it connects to all other nodes. In the box, the user should see in what different ways the two features are tangled. The different categories of tangling are the following:

- Tangled line annotations
- Tangled block annotations
- Tangled file annotations
- Tangled folder annotations

Interaction

The nodes and edges should perform the same by interaction. The drop-down menu should have the same options of "rename" and "delete" as in the Feature Location Visualisation when right-clicking on a node. Additionally, the drop-down menu should have the option of "go-to Feature Location Visualisation". This option opens up a Feature Location Visualisation of the right-clicked node with only that node and the files and folders connected. This option should also be available if clicking on the edge between two nodes. In that case, the Feature Location Visualisation should display both of the features of the connecting edge.

3.3.3 Metric view

Purpose

The purpose of the metric view is to give the user a way to view more detailed information about the features annotated in a project. With the metric view, to give an example, a user can view how many lines of code there is in a feature and in how many files the feature code is scattered across.

Metrics

The view should be able to display all of the following metrics to the user:

- Scattering degree: How scattered a feature is around the project.
- Number of file annotations: How many file annotations a feature has in the project.
- Tangling Degree: How tangled a feature is with other features in the project.
- Lines of Feature Code: How many lines of code that belongs to the feature.

- Number of file annotations: How many file annotations a feature has in the project.
- Number of folder annotations: How many folder annotations a feature has in the project.

Metric settings

The user should be able to change which of the metrics listed above they want to be displayed in the view through a menu in File -> Settings -> Tools -> HAnS-Vis. The settings menu should present all available metrics for features in a table. In the first column is the metrics. In the second column, the user can toggle each metric on or off.

Using HAnS-Edit, it should be possible for the user to choose if they want to view metrics for just one feature or select multiple features. The selected metrics should be presented as a table in the metrics view, where each selected metric will have one column in the table. The selected features should make up the rows of the table, making it possible to view metrics for several features at once. The table should allow a developer to overview relevant information about a feature to make more informed decisions.

4
Methodology

The first step of working on the project was to specify how the plugin should function and look and get clear goals to work towards. While this preliminary step was performed, there was also research into multiple visualisation libraries used to fulfil the project's specifications. Furthermore, this research brought insight into what specifications were impossible and allowed adjustments to them.

After the research and design stage was done, work started on the actual implementation of visualisations. In order to implement these visualisations, they were divided into main features that could be worked upon independently. This was to make the workflow efficient and make sure that as much as possible could be implemented simultaneously. During the implementation stage, there were weekly checkups on what has been done during the past week to know what has been done, what needs to be changed, and what is left to implement.

Once a visualisation was functional but not necessarily finished, the visualisation was internally reviewed. The feedback was then analysed to figure out what should be done and how. Using the feedback, changes were made to the visualisation. The visualisations went back to the implementation stage to implement new changes and continue to implement unfinished functionality. Once largely complete, the visualisations were tested with a user review (see 4.2). The user review was used to get feedback from users on potential changes or concerns. Then these concerns could be fixed in later implementation iterations of the visualisation.

4.1 Libraries & Tools

The relevant libraries and dependencies used in the created project are explained here for clarification purposes. The information soon presented might be necessary to understand the following chapters or give background on essential concepts. This subchapter will describe three such libraries and dependencies, which were all used in the project. As such, libraries merely considered or later removed are not described here, neither are libraries or dependencies that are less relevant for understanding.

Vis.js is a JavaScript web-based visualisation library that render the visualisations on a HTML canvas. The library enables manipulation and interaction with dynamic data [8]. The library consists of several components, the ones relevant for this project are:

- **Network** is the component that is used for the visualisations. Network displays networks consisting of nodes and edges that are customisable in both appearance and layout. It also enables interactions that can handle user actions.
- **DataSet** is one of the other components of vis.js which is used to handle the node/edge data that "Network" requires for its visuals. The data required for Network is retrieved and modified using both JavaScript and DataSet methods

The decision was made to use this library even though it is a JavaScript library, which comes with the disadvantage of requiring data transfer between Java and JavaScript. Finding a library suited for the project's requirements came with difficulties, the set delimitation on finding a BSD-licensed library vastly narrowed the range of selection. The reason why this was demanded is because it provides minimal restrictions and copyright violations on distribution of the product[10]. The Vis.js library was determined as the best choice because it is very well documented and user-friendly.

JCEF (Java Chromium Embedded Framework), is a framework for embedding browser components, such as HTML, CSS and JavaScript, in applications written in Java[9]. This became a tool required for displaying the visualisations with vis.js in this Java-based project.

JSON (JavaScript Object Notation), is a lightweight format for storing and transporting data. It uses key and value pairs to map keys to data. Due to the way JSON data is structured, with keys being mapped to values, it is easy to parse and generate.

4.2 Review

To test the product created in this project there was a user review performed close to the end of the development stage. This user review was created to be less comprehensive than a user study after feedback from the project's supervisors. The motivation being that studying the effects of the product created was not the focus of the project. The purpose of then performing a review, rather than merely having an analysis by the project's members, or doing nothing at all, was to increase the project's scientific foundation.

The review took the form of an interview, of a qualitative and flexible shape where the test user, the people who partook in the the review, interacted with a demo of the plugin. The motivation for choosing this format was:

1. The plugin demo needed a supervised format because of the incompleteness of the demo at the time of the review. One additional reason for the format was the test users relative unfamiliarity of the plugin and annotated features. Therefore, a supervised interview in combination with the demo was deemed appropriate.
2. One purpose of the review was to understand if the plugin improved the human experience and how? Which is a question that is difficult to quantify, and therefore, the review took a qualitative rather than a quantitative shape.
3. A lack of time and available test users made a smaller review, with quality research ideal.
4. A flexible interview, where questions, interactions, and tasks may appear or disappear during the interview to fully explore the experience of the plugin, was decided on for two reasons. One, the human experience differs and seeing how the users interact with and without our input and then asking for the thought process and experience could be valuable and varied. And two, because the demo was incomplete and required flexibility during the review.

The review aimed to examine the helpfulness of visualisations of feature annotations and the success of the project's plugin. In particular, it looked at users' reaction to the project's proposed solution of feature lookup, specifically, the feature location visualisation. The review's aim is tested by seeing if the test user can use the plugin as intended and how they feel about it. The target audience of the plugin was the same demographic as the review, i.e. programmers who use, or are interested in, feature annotations. Multiple tasks and questions were written in advance for the review (See planned questions and tasks in appendix A.1), but because of the flexible nature of the format, they were not strictly followed.

The two test subjects were chosen for: their availability; belonging to the reviews intended demographic; and for belonging to a group of similar knowledge on annotations, and visualisations. This all lead to the test users being people who are connected

to the members of the project, but not a part of the visualisation plugin project. The test users were then contacted, and the test was performed over two different days.

Limitations of the review included: things that will not be included or studied in the plugin because they are not completed in the plugin at the time of review, or irrelevant to the aim: layout, unless hindering to user experience; some of the possible interactions, such as highlight and drop-down menu options; design, unless misleading or hindering the test; certain features such as selecting features—they are instead given a few selected features. The review was also only done on one iteration of the plugin, because of time restraints, which could invalidate the results of the review when the visualisations and plugin change. Therefore the changes made after the review was limited to backend changes, minor details, or requested changes. With the assumption that the results of the reviews would remain largely the same.

The test users were informed before the review about the project, visualisations, and feature annotations. They had similar low levels of familiarity with the subjects. Furthermore, the goal was to have individuals with similar understanding tested and not mix with individuals who do not fit this demographic. Therefore this limited the individuals that could be used for the review.

Recorded tests was performed over the digital video chat platform zoom, which lasted about 30 to 45 minutes. Three group members conducted the review, with roles of discussion leader, moderator, and note-taker. There were planned questions (see planned questions and tasks in appendix A.1) and unplanned questions. The purpose of the formulating of the questions and tasks was to simulate regular use of the plugin and to further the aim of the review and thesis. The depth of the questioning depended on whether the answer truly answered the question that was asked and whether it was relevant to the aim of the review. The depth of the questions did not depend on how many unsatisfactory (as in aim, not in judgement) answers came before. The number of people to test was determined to be a minimum of 2 and a maximum of 5, primarily for time constraints and available test users. In the end the review was done with the minimum number of participants, which was believed to adequately give a sense of the reception and usefulness of the visualisations.

The user reviews gave results in an informal interview format, and therefore the data needed to be selected before presented. This process of judging what data is relevant was done by weighing the reliability of the answers/actions and judging on the applicability to thesis and review scope and aim. The reliability of the answers/actions were based on a predetermined system given below:

> The review process is assumed to provide a correct and honest response, by prioritising feedback that is observed rather than verbal. Some weight will be given to direct answers, but more weight is given to feedback delivered

spontaneously as a reaction to the usage of the demo, or if the test user expands on a question as to why they have that opinion with a reasonable argument. This is not a perfect system, but was deemed acceptable by the creators and supervisors, as a review process.

Then finally the answers from the interviews were analysed by the above criteria, and the relevant results were written in this results chapter. The relevant data gathered was then used to support the project's plugin and purpose in the discussion chapter.

5
Results

This section contains the results of the project. The plugin is presented following the visualisations achieved and finally the results of the user review.

5.1 HAnS-Vis

The current version of HAnS-Vis contains the three essential visualisations described in the specification chapter (see chapter 3.). Due to time constraints, no work on the non-essential visualisations was done. The three essential visualisations, the feature location view, the feature tangling view, and the metric view, can be found in IntelliJ once the plugin is installed. The feature location and tangling view can be found in IntelliJ's right toolbar implemented in a JCEF browser. The metric view can be found at the bottom toolbar by default but can be moved to other toolbars if preferred. The metric view is displayed in a window using IntelliJ native libraries.

5.2 Implementation

As stated before, two of the three views are implemented in a JCEF browser that uses the Javascript library Vis.js. In order to get all the visualisation to work, there has to be a way to transfer the data needed for the visualisations from Java to JavaScript. Essentially, this is done in two steps:

- Generate JSON-formatted string of said data
- Send said string to the visualisations

First, a JSON-formatted string is generated from a Map<String, Object> object containing all the relevant data needed for the visualisations through a Utility-class called *JSONHandler.java*. This JSON-formatted string is then sent to the visualisations, which parses the relevant data and renders it for the user.

Feature Location Visualisation is implemented partly in Java and partly in HTML, CSS, and JavaScript. The backend is entirely implemented in Java as, due to JCEF

constraints, there is no way to extract all the relevant data from the project through JavaScript and JCEF. Instead, the data is extracted from the project through Java by a Utility-class called *FeatureAnnotationUtil.java*. This data is converted to a JSON-formatted string then sent to the visualisation, upon being requested from the visualisation, through JCEF's onQuery method, which enables strings to be sent through Java and JavaScript.

The frontend is entirely implemented in HTML, CSS and JavaScript. It displays an interactive graph of all the features present in the opened project. The graph displays where features are annotated in the project and what type of annotation it is. Each feature is represented by a node in the visualisation. Likewise, each file and folder that has been annotated is represented by its own nodes. For the user to easily differentiate between nodes, a file node is displayed as a file icon, and a folder node is displayed as a folder icon. When a feature node is connected to another node with an edge, it implies that the said node contains code that implements that feature. Depending on what kind of annotation is used, the line of the edge will be different. When folder and file annotations are used, the line is solid; however, if the annotation used is an in-file annotation, the line used as an edge is a dashed line. This line distinction makes it easier for users to quickly overview what type of annotations are used in the project and where they are used.

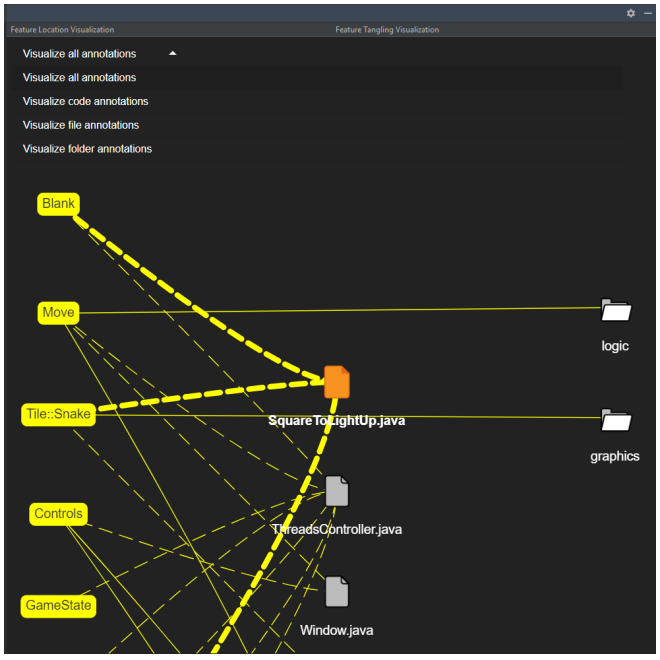

Figure 5.1: The Feature Location Visualisation with the options for the drop-down menu. The file SquareToLightUp.java has been clicked on and is highlighted, as well as the edges to its' related features

The frontend is implemented so that the user can choose which level of granularity to display, with each type of annotation being their own level of granularity. In order to choose which level of granularity to display, the user can pick the granularity in a context menu in the visualisation window. Upon selecting a new level of granularity, the visualisation is automatically re-rendered to display the correct data. This will allow users to quickly find the type of annotation they are looking for and give the user a way to navigate the annotations present in the project easier. Essentially, this is done by reading different indices in the JSON-formatted string since each index for a specific feature contains a different granularity of data.

Furthermore, visual feedback is used to make the graph more user-friendly, easier to understand, and clearly emphasise that the graph is interactive. Hovering over any node or edge will highlight it to make it easier for users to navigate the graph, especially for larger projects where many edges will be present. If the user hovers over a node, additional highlighting of all connected edges and their respective nodes will be displayed. Likewise, if the user hovers over a file or folder node, the user receives additional information about the said node. If the node being hovered over is a file node, the user gets information where said file exists and how many features and how many annotated lines of code those features have in said file. However, if the node being hovered over is a folder node, the users get information telling them how which files are present in said folder.

Figure 5.2: A hover box is shown when hovering over a file showing what folder it is located in. The lines of code a feature takes up in a file are also shown

The visualisation contains a number of interactive elements. Features have the options

to view the feature in Feature Tangling Visualisation. Furthermore, there is additional interactivity with the nodes. Clicking on a node will change the colour or icon of the node as well as highlighting all connected nodes, making it easier to navigate for the user.

Feature Tangling Visualisation, like the *Feature Location Visualisation*, consists of a backend and a frontend. The backend is implemented the same. The frontend, however, although implemented in HTML, CSS and JavaScript, differs.

The frontend features an interactive graph of a chosen feature and the features it is tangled with. The feature is selected from the Feature Location Visualisation using a feature nodes' context menu. The feature is centered with edges going to its tangled features. The edge thickness represents the degree of tangling between two features.This makes it easier for users to quickly get an overview of what features are tangled with another feature and gives an easy way to visualise the degree of tangling between features. This visualisation contains a number of interactive elements as well. In order to save time for the users, features can be right clicked to display a context-menu allowing the the user to open the feature in the Feature Location Visualisation.

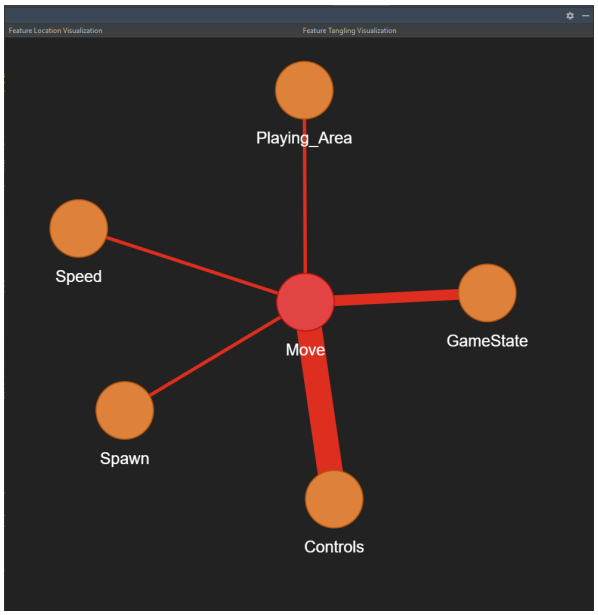

Figure 5.3: Visualisation on what feature **Move** is tangled with. The edge thickness between **Move** and **Controls** is the thickest, indicating a higher level of tangling between them

Feature Metric Visualisation is implemented entirely in Java using IntelliJ's own API, the Java Swing Library and the Java AWT library. The metric view displays a table with different useful metrics regarding all the features present in the project. The fol-

lowing metrics are supported in the metric view:

- NoFiA - Number of file annotations
- NoFoA - Number of folder annotations
- SD - Scattering degree
- TD - Tangling Degree
- LoFC Lines of Feature Code

These metrics give the user insight into the features, how big these features are, how independent they are and how much of the project is made up of individual features. Additional functions such as searching for a specific feature and sorting by a specific metrics is also available.

Filter features here

Feature	SD	NoCA	NoFiA	NoFoA	TD	LOFC
Playing_Area	3	2	1	0	0	11
Tile	0	0	0	0	0	0
Food	5	3	0	2	0	13
Spawn	1	1	0	0	0	5
Blank	2	2	0	0	0	17
Tile::Snake	3	2	0	1	0	2
Update	3	3	0	0	0	8
Snake	0	0	0	0	0	0
Move	4	2	1	1	0	49
Collision	0	0	0	0	0	0
Head	0	0	0	0	0	0
Tail	0	0	0	0	0	0
Controls	3	1	2	0	0	38
GameState	1	1	0	0	0	7

Figure 5.4: Metrics view showing the calculated numbers from the features.

5.3 User review results

The user reviews gave results in an informal interview format, and therefore the data needed to be selected before presented. This process of judging what data is relevant is described in the previous chapter (see review, chapter 4.3). Therefore the data presented here is pre-selected and judged on applicability.

The conclusions of two completed user reviews are:

- Feature lookup and efficiency was deemed to be greatly improved by feature view, and marginally by metric view. One test user particularly mentioned that these improvements would be amplified if the feature based project was large or if the user arrived at the project after its conception.

- A tutorial might be useful for understanding some of the plugins mechanics. Clarifying such options as the ability to right click nodes to make a context menu appear.
- One test user was able to act comfortably, confidently, and sometimes independently while navigating and using the feature view. The other test user was behaving similarly, but had to ask for guidance more.
- The test users were able to perform most tasks with minimal guidance and clarification, though some tasks were at first solved incorrectly. The reason for wrong answers was often over misunderstandings of terms such as features and annotations.
- Most answers were favorable.
- Both test users spontaneously brought up a wish for a horizontal distance increase between the feature nodes and file/folder nodes.

6
Discussion

This chapter discusses the result and if the requirements have been achieved, the user review results, what could be improved in the future if others wish to improve upon the project, and what conclusions can be drawn from the project.

6.1 Result

The three views created in the project, although a bit limited in some areas, still work well and are useful when handling the feature related problems the project set out to improve. Especially the Feature Location Visualisation provides a very quick and intuitive way to navigate an annotated project and find the features you need. The Feature Location Visualisation does however encounter some issues when handling larger projects. This is because the layout only stacks features vertically, thus making feature rich projects difficult to overview in full. The Metrics view is able to show feature-related metrics with search and sort functions. However, the metrics are only calculated once when opening up a project and are not updated if more annotations were to be added unless the project is reopened.

To obtain a truly unbiased opinion of the usefulness of the plugin, more reviews from developers experienced in the field of feature based programming would be required. Our plugin does however work as intended and provides the functionality designed to provide.

As stated earlier in the chapter, the plugin is limited in some regards. All the requirements were not implemented in the final result of the product. This absence was due to communication issues between the Bachelor's group and the Master group that caused the API not to be implemented in time, and thus all data could not be provided. The Master group also planned to create functionality to select features to visualise, however this functionality was not completed in time to have this project implement it. Because the API could not provide what was expected from it in time, we had to implement some of the functionality independently under a tight timeline. Thus, certain functionality that we could not implement ourselves in time had to be left out. In the end, the product needed to be more limited than we wanted it to be, and the last requirements have to be fulfilled in the future when the API has been fully implemented.

6.2 Review

The results of the reviews showed that the test users were able to complete most tasks independently or with not too much guidance and answered most questions favourably. There were some deviations from positive feedback, such as asking for a tutorial of some aspects to help new users such as them, unfavourable opinions on design and layout despite not being a focus of the test, and some requests for features. Nevertheless, the users demonstrated that the product could be used and that they understood the tasks that were being carried out. Furthermore, as the tasks were designed to simulate regular use of the product, the product is hopefully valuable for actual use in software development.

According to the user reviews, the project's aim to save developers time by helping in tasks such as feature lookup was successful. With the use of the visualisations in the demo, the test users reported a clear preference for having the plugin, rather than not, for the tasks presented. However, further testing would be required to show exactly to what degree it is helpful or what percentage of time is saved by the product's availability.

After the review, there was one significant esthetical change to the product, as mentioned in the results chapter. Both users brought up the small distance between the feature nodes and the file/folder nodes. Furthermore, as both test reviewers brought up this design decision spontaneously, their words held much weight. Therefore the product was changed to accommodate the perceived flaw. The changes made after the review are untested compared to the old version, but as most are backend changes, minor details, or requested changes, the assumption is that the results of the reviews remain the same or improved.

6.3 Future work

While the plugin at the time of writing this report has managed to implement most of the functionality deemed essential in our product specification, there is additional functionality that, if implemented, could further enhance the experience of the plugin's users. What would benefit the plugin the most would be to have total integration with the intended HAnS-Edit plugin that was developed in tandem with this project. While this project provides the means to visualise annotations in a project quickly and find where features are located in the project, together with HAnS-Edit, the user would have syntax highlighting of features in the code. Furthermore, the user would be able to choose which features to visualise in, for example, the feature location view rather than visualise all features in the project.

There is also the matter of the views not deemed essential in the specification. Due to the project's time constraints, the non-essential views ended up in future work.

As stated in 3.2, the History view is not essential for feature lookup. Instead, it would serve as an additional quality of life feature to aid in keeping track of the project development. It would, of course, be beneficial for feature programming developers to have a way to see when a feature was last updated and with what. As such, it still fits our stated purpose in 1.1. However, it would require additional libraries to utilise git data, which would be libraries that could not be reused in other visualisations. Therefore this view was not prioritised, mainly because it would take additional time from the prioritised elements.

The common Feature Location Visualisation aims to compare features between different projects. The view requires comparison between several projects in IntelliJ. However, seeing as IntelliJ only allows one project to be open per window, the workload of having information from several different instances of IntelliJ communicated between each other was determined to be too great to be achievable during the project's timeframe.

In the visualisation specification section 3.3, both the Feature Location Visualisation and Feature Tangling Visualisation are planned to have a a selection menu of features to visualise. Though, as previously mentioned in 6.1, feature selection which was to be implemented through HAnS-edit, which as not yet been implemented. This impediment causes any functionality that requires filtering or selection, such as creating a view with only one selected feature, to be put into future work. This addition would greatly enhance the user experience by tidying up the visualisations and allow users to even quicker find the features they are looking for and should be a priority in future iterations. Giving the users the functionality to rename features directly from the visualisations would be a great addition to the plugin. As of right now, users would have to manually rename every feature in the annotations, a time consuming task. Furthermore, functionality to add to rename, open and delete files or folders in the editor directly from the visualisations should be implemented. By allowing users to have this functionality in the visualisations, users would not need to manually traverse the project structure to open, rename or delete files or folders - which becomes more time consuming as projects scale in size. The hover box is also in a different placement than mentioned in the product specification. Like other design/layout mentioned in the specification chapter, like colour choices and radio buttons, the final implementation was preferred once properly implemented and seen.

6.4 Conclusion

If features are the foundation in Feature-oriented software development, then embedded annotations are the missing keystones. As mentioned previously, prior research shows that identifying a features location, or the assignment of "feature-lookup", is a time-demanding task for developers involved in feature-oriented software development.

Therefore, the mitigation or solving of feature-lookup is a problem to resolve if one seeks to improve developers efficiency. This problem is what led to the project's aim of solving that problem. The project's scope then limited the project to only visualisations helpful with this developer difficulty. With the goal of achieving efficiency, a plugin was created that could visualise and simplify the aforementioned problem. The resulting plugin was a product able to save time for developers, though limited in its ability to filter features fully. The plugin helps developers save time by visualising where the code that implements a feature can be found. It helps developers find where code of different features intersect and give helpful information about features in the form of features. While similar plugins exist with more functionality, our project serves as a helpful addition to the feature-based programming community on IntelliJ where no other similar plugin exists.

Bibliography

[1] B. Andam, A. Burger, T. Berger and M. R. V. Chaudron. "FLOrIDA: Feature LOcatIon DAshboard for Extracting and Visualizing Feature Traces" in VaMoS, Eindhoven, Netherlands, 2017, pp. 100-107. doi:10.1145/3023956.3023967

[2] J. Krüger, T. Berger, and T. Leich, "Features and How to Find Them: A Survey of Manual Feature Location," in Software Engineering for Variability Intensive Systems: Foundations and Applications, I. Mistrik, M. Galster, and B. Maxim, Eds. Taylor & Francis Group, LLC/CRC Press, 2018. [Online] Available: http://www.cse.chalmers.se/~bergert/paper/2018-sevis-manual_fl.pdf [Accessed: May 14, 2021]

[3] W. Ji, T. Berger, M. Antkiewicz and K. Czarnecki. "Maintaining Feature Traceability with Embedded Annotations" in 19th International Systems and Software Product Line Conference, Nashville, TN, USA, 2015, pp. 61-70. doi:10.1145/2791060.2791107

[4] B. Vermeer. "JVM Ecosystem Report 2020", Snyk, London, UK, 2020. [Online] Available: https://snyk.io/wp-content/uploads/jvm_2020.pdf [Accessed: May 14, 2021]

[5] S. Entekhabi, J.-P. Steghöfer, A. Solback and T. Berger. "Visualization of Feature Locations with the Tool FeatureDashboard" in 23rd International Systems and Software Product Line Conference, Paris, France, 2019, pp. 1-4. doi:10.1145/3307630.3342392

[6] T. Schwarz, W.Mahmood, and T. Berger "A Common Notation and Tool Support for Embedded Feature Annotations" in 24th ACM International Systems and Software Product Line Conference, Montreal, QC,Canada, 2020. pp. 5-8. doi:10.1145/3382026.3431253

[7] J. Wang, X. Peng, Z. Xing, and W. Zhao, "How developers perform feature location tasks: a human-centric and process-oriented exploratory study". Journal of Software: Evolution and Process, vol. 25, 2013, pp. 1193–1224. doi:10.1002/smr.1593.

[8] Vis.js [Online] Available: https://visjs.org [Accessed: May 14, 2021]

[9] JetBrains. JCEF [Online] Available: https://plugins.jetbrains.com/docs/intellij/jcef.html [Accessed: May 14, 2021]

[10] Wikipedia. BSD licences [Online] Available: https://en.wikipedia.org/wiki/BSD_licenses [Accessed: May 14, 2021]

[11] Cambridge. Metrics [Online] Available: https://dictionary.cambridge.org/dictionary/english/metrics [Accessed: June 3, 2021]

A

Appendix 1: User review

A.1 Questions and tasks

(Q is questions, T is tasks)

Overall plugin -
Q. Do they understand how to navigate? (interpret from actions)
T. Go to X visualisations. (use the menu)
Q. How do they feel about the navigation?

Feature view -
T. Find file X.
T. Find a specific folder annotated to feature Y.
T. In what folder is file x annotated to feature Y?
T. Find a file containing code annotations to a feature.
T. Find a file containing file annotations.
T. Find how many different annotations exist for file x.
Q. How do they feel about the feature view?
Q. Would feature lookup be made easier or more efficient by this visualisation?
T. Final task: go to tangling view for feature x.

Tangle view -
Q. How do you interpret this view?
T. What degree of tangling has the least tangled feature?
T. What degree of tangling has the most tangled feature?
Q. Was the line thickness indicative enough to allow you to easily find the least- and most tangled features?
Q. Does the static box provide enough information about the different kinds of tangling that you know that you will be able to change one feature without damaging the other.
Q. How do they feel about the tangle view after using it?
T. Final task: go to metric view.

Metric view -
Q. What do you think about having the settings for the metrics view under file -> settings?
Q. How do they feel about the metric view after using it?
Q. is it useful?