CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Visualization of feature-traceability in variant-rich systems

Master's thesis in Software Engineering and Technology

ANTON SOLBACK

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

Visualization of feature-traceability in variant-rich systems

ANTON SOLBACK

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Division of Software Engineering and Technology
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Visualization of feature-traceability in variant-rich systems
ANTON SOLBACK

Supervisor: Thorsten Berger, Department of Computer Science and Engineering
Examiner: Regina Hebig, Department of Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Division of Software Engineering and Technology
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019

Visualization of feature-traceability in variant-rich systems
ANTON SOLBACK
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Purpose. Feature-traceability is defined as the ability to trace features and the different aspects of it in code. Variant-rich systems are systems that have many variants, such as software product lines. Feature-traceability in variant-rich systems is hard, as a result, the purpose of this thesis is to suggest views that will improve feature-traceability and other feature related problems in variant-rich systems.
Methods. The purpose was achieved by performing a literature survey, then, by following a design science methodology, views for improved feature-traceability was conceived. These conceived views were successively evaluated and improved upon further.
Result. The result of this thesis is an analysis of the current state of feature-traceability in research, concept views that present ideas of what could be implemented in a feature-traceability tool. Additionally, an open-source plugin for the integrated development environment Eclipse called Feature Dashboard. Feature Dashboard enables the user to, not only, trace features to their implementation, but to get information such as which features are implemented in the same file, observe where in the folder structure a feature is located, metrics about features, feature metrics for folders/files, and inspect which features are shared between projects.
Conclusion. This thesis shows that Feature Dashboard can be used to improve feature-traceability, maintainability, understandability, and other feature management related tasks for a project. As Feature Dashboard is open source, it offers a platform that enables future researchers to, either implement new views or to compare the result of Feature Dashboard to other views. Furthermore, anyone can suggest improvements and extend it with additional functionality, thus benefiting everyone.

Keywords: feature, feature-traceability, visualization, eclipse, plugin, variants, SPL.

Acknowledgements

Special thanks to Thorsten Berger, Jan-Philip Steghöfer, Sina Entekhabi, and to everyone that participated in the various evaluations throughout the thesis.

Anton Solback, Gothenburg, June 2019

Contents

List of Figures

List of Tables

Listings

1
Introduction

In this chapter, an introduction to the thesis is given. Firstly, in Section 1.1 the context of this thesis is presented. Then, in Section 1.2, the problems that motivate this thesis are given. Section 1.3 covers the goal for this thesis. Following the goal, section 1.4 explains the purpose. Then, in Section 1.5 methods used in this thesis are briefly explained. Lastly, Section 1.6 gives an outline of the report.

1.1 Context

The context for this thesis is software engineering with variant-rich systems, where many different developers work on such systems over long timespans. They also need to maintain and evolve features, which can exist in many variants. To have multiple variants of a software product is especially common in embedded/cyber-physical systems, automotive systems, telecommunications, and industrial automation.

1.2 Problem

There have been a number of problems identified regarding the context for this thesis and in this section they are presented.

1.2.1 Tracing features

In order to know where features in a codebase are located, their location needs to be stored. To achieve this, there are examples of tools that can be used to externally document locations of features in a codebase. However, as a codebase grows, such external documentation goes out of date, which often results in developers utilizing 'code browsing' in order to understand it [1]. Reading through source files, especially when the codebase is comprehensive, is time-consuming. Therefore, it is vital to use a method that constantly keeps traces up-to-date.

To keep traces up-to-date, Ji et al. [2] suggests placing annotations directly in the source code. Annotations are placed in the comments of a source code file, allowing annotations to be used regardless of programming language. The idea is that annotations should naturally evolve alongside the code. Thus, a user would not have to access a third-party program to first find the trace and then update it. As is demonstrated in the paper, the cost of maintaining annotations were very low [2].

1.2.2 Understandability

Imagine the following scenario: A developer has just started working at a company and is given a task to fix a bug related to a specific feature. Not being familiar with the codebase in question, significant time has to be spent to learn it. In the best case, there already exists documentation which explains where different parts are located and how the codebase is structured. In the worst case, there does not exist any comprehensive documentation and the developer has to resort to code browsing and reading source code comments (if there are any). Regardless, there is a substantial initial learning curve involved. Even for senior developers, starting to work at a new product which is large and complex, at least a couple of months is given to learning. Therefore, having a way to decrease the time for a new developer to be efficient is beneficial for everyone involved. Gaining a better understanding of the code, in general, is not just beneficial for new developers, but to anyone who is working with the code.

An additional scenario is if there are multiple variants of the same software, at some point, the developer might want to merge them into a software product line (SPL). During this task, it is important to understand which features are shared between variants, which are not, where they are located, which features depend on each other, for example. To have this knowledge, it requires an excellent understanding of all variants in question.

1.2.3 Maintainability

As a codebase gets larger and more complex, it will become increasingly harder to maintain. Knowing the locations of features, additional information such as metrics could also be used to make it easier to maintain the features. For instance, if it was possible to see that a feature has grown a lot in size recently, maybe it is time to split that feature into smaller parts. Perhaps it is noticed that a specific feature is scattered all over the codebase. Or that many features are tangled together in a specific file. Depending on this information, it could be a good idea to refactor the code. It would also be possible to correlate this information with other code quality metrics such as bug reports. For instance, it is indicated that a feature has become increasingly scattered and tangled with more and more features over the recent month. Simultaneously, more and more bug reports regarding this feature have been reported. As such, having more detail information about features located in a codebase could increase its maintainability.

1.2.4 Visualizing feature-to-code traces

As already suggested, with the introduction of embedded annotations it is possible to indicate in the code where specific features are located. However, this only solves a part of the problem, namely; a developer still has to go through the source code and look for these annotations to get an overview. It is possible to do a full-text search, however, that is not effective. As such, how can these annotations be extracted and then visualized?

1.3 Goal

The goal of this thesis is to conceive, realize, and evolve a feature visualization technique. An important part of this is, firstly, to investigate the state-of-the-art and what other relevant information is needed.

As a result of what is discussed in Section 1.2, how are features best visualized? What views should be present to convey the necessary information? How should features that belong to a certain file be visualized? What information/metrics about features are necessary? Are these types of visualizations, metrics, and information helpful in understanding features and how they are realized in code? These are questions that the thesis will try to answer.

1.4 Purpose

As already discussed, tracing features is a significant problem and there is not a generally accepted best solution to this problem. However, if an effective solution is discovered, the implications can be immense. As Ji et al. [2] discussed, simply having annotations in the code indicated a great benefit in feature-related maintenance. Combining this methodology with a more powerful visualization tool that also can provide meaningful metrics, the process of understanding a codebase and tracking the features could increase significantly.

As Wang et al. [3] states in their paper, locating features manually has a substantial cost attached to it. These features must then be recorded in some external documentation and as already discussed, this process is error-prone, and it takes considerable time from developers. Having a tool, which just parses source code files, provides visualizations and metrics about features would not only remove the time investment but also the errors made. The only thing that should be maintained is the annotations that describe where features are located. However, this should be straightforward as they evolve naturally as developers code and as previously mentioned, the maintenance cost is low [2].

1.5 Method

The goal of this thesis is to be achieved by firstly reviewing already written literature within the visualization of feature-traceability. After this, a design science methodology is utilized to develop views. These views are to be implemented in a tool that allows a developer to investigate which features are present in a codebase. After the mentioned views have been implemented in this tool they should be evaluated.

1.6 Outline

This thesis is organized as follows: Chapter 2 provides the reader with information regarding background concepts. Chapter 3 introduces the methodology used in this thesis to obtain the result. Following the methodology in Chapter 4, the result of this thesis is presented. Chapter 5 goes over the different evaluations that were performed as a part of this thesis. Chapter 6 provides a discussion about the different parts of the thesis. Chapter 7 presents the reached conclusion.

2
Background

In this chapter, concepts are explained that can be beneficial to be aware of before continuing reading this thesis.

2.1 What is a feature?

When developing a software artifact, there are certain features associated with it. A feature often describes the functional or non-functional requirements of that artifact [4]. In terms of a software system, a feature is often thought of and explained to stakeholders in terms of some specific characteristic [5]. Therefore, knowing what features a specific system offers is very important, not only to categorize it but also for stakeholders to understand what it can do.

2.2 Software product line

If a company sells a commercial product then it might not always be the case that every customer wants all the functionality that the product offers. For instance, they might not need a specific feature and is not willing to pay for that feature when buying the product. Or the customer has specific environmental demands that the product has to adapt to. Therefore, in order to satisfy customers, a *variant* is created which has a different set of features. This process could continue and after a while, there are many variants that suit different customers and purposes. One of the negative effects of this is that all these different variants have to be maintained separately. As a result, it can be beneficial to merge these variants into a *software product line*. To do this, all variants are analyzed and common features are extracted to form a base on which other features are subsequently added to. Similarly, when buying a car, the manufacturer offers the base configuration of a specific model. From there, it is possible to add different features that would be preferable to have. As Benavides et al. states, software product line engineering promotes creating a family of software products from features that already exist, instead of creating separate variants from scratch [6].

2.3 Feature model

A software product's behavior can often be described by simply listing the features which are offered. However, in order to describe a software product line (SPL), a

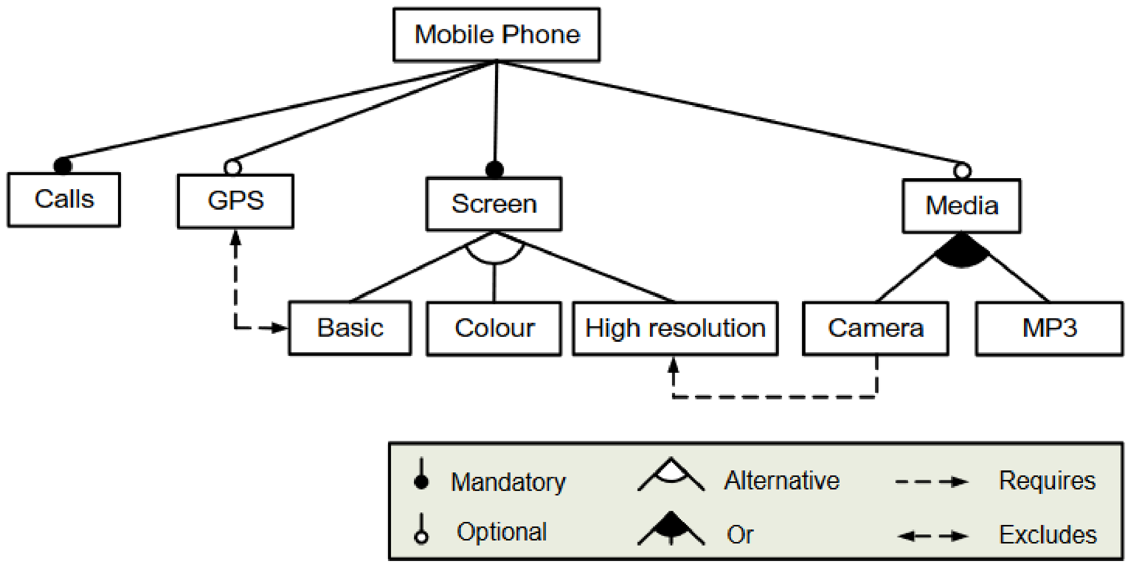


Figure 2.1: Example of a simple feature model [6].

feature model is used [6]. A feature model is used to describe all possible configuration options that can be made for an SPL [6]. In Figure 2.1, a simple feature model can be found for a phone. As can be seen in the figure, there are multiple different relationships that can be specified in such a model. One of these indicates all mandatory features that will always be included with the product and another indicates that one feature might depend on another being present, and so on.

2.4 Clone & own

Instead of creating an SPL from the beginning, another option is to create variants that later could be merged into an SPL. One popular way that variants are created is by performing, what is called, *Clone & Own*. Creating new variants by Clone & Own is achieved by first locating pieces of code in a variant that should be reused. Then, the targeted variant(s) are copied or merged into a new variant. At that point, excess functionality is removed and new features are introduced into the code [7].

2.5 Trace recovery

In order to visualize the location of a feature in a project, that location first has to be identified. In this section, different methods used and challenges regarding this task is explained.

2.5.1 Manual

In software artifacts, what a specific feature belongs to, for instance, a method or class, is seldom recorded and maintained. When there is only the original developer of an artifact, source code comments will most likely be enough due to the developers' preexisting knowledge and understanding. However, as the artifact increases in size and as more developers join the development, the locations of features are usually not known anymore. When this is the case, the feature locations need to be recovered which is a daunting and error-prone task [3]. However, a study performed by Krüger et al. [8] notes, among others, that there is still little known about the actual effort of performing these feature locations and what factors influence them. Regardless, the easier it is to understand the connection between source code and a feature, the easier it is to modify it [9]. Thus, a way of clearly describing the features to developers in order to get a better understanding of how the different parts are connected to each other is needed.

2.5.2 Automated

Instead of manually going through each resource and documenting traces, a natural improvement is to try and automate this process. There is no shortage of contributions to this field as many have tried to come up with the next tool that will revolutionize the industry. Dit et al. [10] performed a mapping study of this field where 89 papers were analyzed. These papers were then categorized by their suggested approach, the three categories were: dynamic, textual, and static [10]. Then there are also papers that have suggested a combination of two or more approaches.

Dynamic feature location works by analyzing running code and inspecting what is executed for a specific type of input. This input could be a test case that tests a specific feature and by analyzing which code is executed it is known what code belongs to the feature. Static feature location does not rely on a running program to determine feature location but instead analyzes the source code files. Textual feature location is similar to static feature location, however, it relies on specific markers in the source code such as annotations or comments. It is also possible to combine these different approaches with each other. For instance, Andam et al. [11] suggests a static feature location approach with the help of machine learning. When a feature is located, annotations are placed in the code which can later be found using textual feature location. Combining dynamic and static feature location techniques could imply that first, dynamic location is used to narrow the search space and then a static analysis is applied [10]. An approach that combines these two methods is suggested by Eisenbarth et al. [12]. Firstly, execution traces are gathered from the targeted system and based on these traces, analysis is performed [12]. The parts of the code which have been indicated to belong to a feature can then be analyzed.

Although there are many contributions to this field, Andam et al. [11] states there is still no solution that works in the real world. This is because they have low accuracy and to analyze a project in the first place, they require a lot of work to be properly configured [11]. Thus, one time-consuming task is simply replaced by another.

3
Methodology

In this chapter, the research questions are presented and the methodology used in this thesis is explained.

3.1 Research Questions

The research questions that the thesis will answer are the following.

1. **What feature-traceability views have been proposed in literature?**
   As the question implies, a literature survey is performed to establish how feature-traceability views are visualized in research. This allows establishing best practices in terms of what has been observed to work and what does not.
2. **What information regarding features is required to realize views that provide added value?**
   What information regarding a features in a project is required in order to realize views that provide added value regarding them?
3. **Do the implemented views support feature-traceability and other feature related problems?**
   As highlighted in Section 1.2 there are a number of problems that are related to not having an effective feature-traceability tool. For instance, do developers think that having a tool such as this could increase the maintainability of a project?

3.2 Thesis workflow

In this section, the workflow and the different phases of the thesis are presented.

Firstly, a literature survey was conducted, this was necessary in order to answer *RQ1*. During the literature survey, tools/views that have been previously presented in research regarding feature-traceability were investigated. After performing the literature survey, the result was analyzed further to identify issues and/or limitations with the suggested visualizations. Then, there were a number of concepts created to explore different techniques and approaches that could be used when realizing views. After different ideas had been explored, the development began of Feature Dashboard. Feature Dashboard is a plugin in the integrated development environment (IDE) Eclipse which enables feature-traceability alongside development. When the development of the plugin had reached a certain point, the focus shifted towards implementing views. During this time, an iterative process of creating, improving, and evaluating views was started which was the last stage of the thesis. At this point, it was possible to answer *RQ2* and *RQ3*, namely, what information regarding features is required for the specific visualizations and if they support feature-traceability and other feature related management tasks.

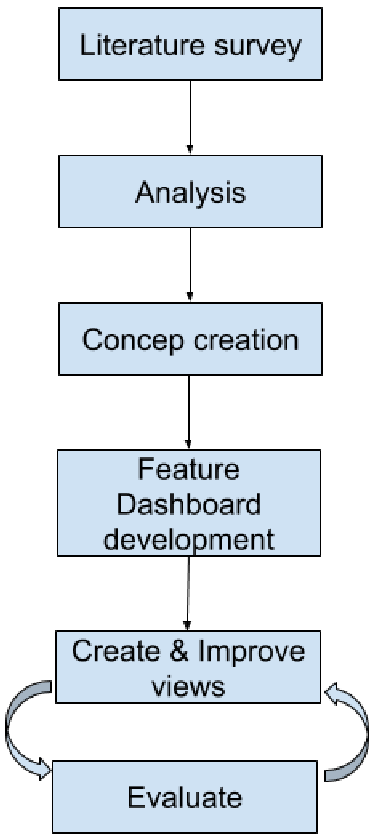


Figure 3.1: Thesis workflow.

3.3 Literature survey

As detailed, the first task that was performed in the thesis was a literature survey. This is to gather information about the state-of-the-art within the field and to understand what has been demonstrated not to work (if any). Another important aspect is that by analyzing the result and sections such as future work, additional inspiration for new views can be gathered. In this section, it is first described which methods were used in finding papers. Then, it is explained how a paper was deemed relevant to inspect. Finally, points of interest are outlined which were used when analyzing a relevant paper further.

3.3.1 Search process

In order to find relevant papers for this literature survey, there were various methods utilized. The main method was to start from an already know paper and from there, find papers which are suggested as related work. Then, this task was to be performed recursively for each paper that is suggested. Additionally, by using services such as Google Scholar, it is possible to see which papers have cited a specific paper. For instance, if paper A has deemed to be relevant, then it is possible to see which papers have cited paper A.

Additionally, different search engines were also used to find relevant papers such as Google Scholar and IEEE Xplore. Google scholar is very useful since it collects results from multiple different organizations. This saves time as the websites for these organizations do not need to be visited. However, the exception is IEEE Xplore, as their search engine allows for the creation of predicates when searching for specific keywords. The following is an example of a predicate that was used: *(visualization **AND** feature) **OR** feature-traceability.* It is also possible to specify where these keywords should appear. For instance, it is possible to specify if a keyword should appear in the meta-data only or in the text as well. Keywords that were used when searching are the following: *feature, feature-traceability, visualization, feature-location, feature-to-code, traceability, software product line, software variants, variant management.*

3.3.2 Paper relevance

For the literature survey, there was a criterion established to determine if a paper was relevant or not. The criterion was that the paper has to suggest or present a visualization technique for feature-to-code traceability. For instance, there are tools that suggest views in other areas of SPL development such as tracing features to feature-models, however, in this literature survey, the main focus is on views that trace features to their implementation in code.

3.3.3 Paper contents

When a paper has been determined to be relevant, specific parts of the content were examined further:

- **What type of views are suggested**
- **What visualization techniques are used?**
- **Is there any future work suggested?**
- **Have the views been formally evaluated? If so, what conclusion can be made?**

3.4 Evaluation

This section describes methodologies used to evaluate different parts of the result. Chapter 5 presents the result of these evaluations.

3.4.1 Semi-structured interviews

The first methodology used to evaluate the conceived views was the usage of semi-structured interviews. The difference between semi-structured and structured interviews is that it further promotes a conversation between the interviewer and the interviewee [13]. This allows subjects to explore things regarding the subject that they think is important. Even though it is not entirely structured, there should still be a protocol with predetermined questions which ensures that interviews consistently covers the same topics [13]. As such, the protocol used for these interviews can be found in Appendix A.2. Feedback received during the interviews was recorded and later transcribed.

3.4.2 Questionnaire

At the end of the thesis, a questionnaire was used in order to answer *RQ3*. To achieve this, there was a combination of closed and open questions. The closed questions were to evaluate whether or not subjects thought the views were helpful in different aspects such as tracing features and allow better understandability of a project. However, to gather additional feedback on what was missing or other problems that were not known beforehand, open questions were also in the questionnaire for each view. The questionnaire was distributed by sending requests to subjects through email and other messaging applications, such as Skype.

3.4.3 Controlled experiments

In addition to distributing a questionnaire at the end of the thesis, controlled experiments were performed to test the initialization time of the views as the input increases. Controlled experiments are experiments where all variables are controlled except for one, namely, the independent variable [14]. It is the hope, by changing the independent variable and controlling all other variables, to identify the effect of the independent variable. It is important for the internal validity of the experiment that only the independent variable is changed, otherwise, there could be additional factors that can affect the outcome of the experiment [14].

3.5 Design Science

In this thesis, as previously mentioned, the methodology used to obtain a result was *design science*. Hevner and Chatterjee [15], present a number of guidelines regarding design science some of which are presented here and how they were adhered to in the thesis.

3.5.1 Problem relevance

This guideline specifies that the artifact developed should provide solutions to specific and relevant problems. In this case, the problems revolve around feature-traceability in variant-rich systems, see Section 1.2 which provides a description of the problems identified

3.5.2 Design as a search process

This is the search for the optimal views for feature-traceability and the steps taken to arrive there. In this section, numerous sources that were used as inspiration when creating views are presented.

Firstly, the literature survey executed gave valuable inspiration, not only from the research that had previously suggested feature-traceability tools/views but other aspects of feature management views. After previous relevant research had been identified, the tools and views were further analyzed from a user point of view. The goal was to identify issues/problems that would hinder a user from effective feature-traceability. With these issues in mind, a number of concepts were created which explored how to visualize different aspects of feature-traceability. Since the issues identified should be avoided, it promoted additional creative thinking. Research regarding visualizations in software engineering was also consulted such as work by Moody [16] and El Ahmar et al. [17, 18]. Inspiration was also drawn from views that are not related to the area of feature-traceability. For instance, when trying to create concepts of how to visualize how features propagate through Git branches, visualizations offered by Github and other open source websites were investigated.

When eventually implementing views in the plugin, additional factors gave inspirations on how to visualize different aspects of feature-traceability such as functionality offered by the visualization library used. In addition to taking advantage of this functionality, the limitations of the library also have to be considered. Since the views were going to be implemented in Eclipse, it was also possible to utilize the fact that multiple views can be opened at the same time.

Another major source of inspiration was the result of evaluations held during the thesis, which gave ideas for how to improve existing views and additional views that could be implemented.

3.5.3 Design as an artifact

Another part of design science is that the result, namely the views, should be represented in an appropriate manner. In this case, it is about providing a means for developers to inspect and analyze features that are located in the source code of a specific project. Thus, the views will be implemented in a plugin for the integrated development environment (IDE) Eclipse. Hence, allowing developers to view information about features in the same program as they are using to develop code, which removes the need for a third party program.

3.5.4 Design evaluation

An important part of design science is that the result has to be evaluated and therefore there were different methods used to achieve this. Firstly, semi-structured interviews were held with a small selection of subjects. The main purpose of these interviews was to gather feedback on how to further improve the views, not to answer *RQ3*. As such, semi-structured interviews were used as they promote discussion between interviewer and interviewee to a higher degree than structured-interviews.

Subjects included were students performing master theses with the goal of determining the cost of merging software variants into SPL's. Since the work that they are performing is exactly what is meant to be made easier with Feature Dashboard, their feedback was important. Since they were using other tools to help them in their work, advantages and disadvantages regarding these tools could be gathered as well. From this evaluation, useful feedback was received that was taken into account during the next design iterations.

For the second evaluation, both Feature Dashboard and the views were evaluated. Unlike the first evaluation, a questionnaire was used at this point instead. The reasoning behind using a questionnaire was that this project was executed as a part of ReVaMP2. ReVaMP2 is a collaboration between universities and companies in Europe and the thesis supervisor has had previous contact with developers at some of these companies and the idea was to contact them and gather feedback [19]. However, due to that these companies are located in other countries and the developers having limited time, it was decided that a questionnaire was going to be utilized. However, this posed additional challenges since the tool would have to be distributed to subjects in order for them to use it and then provide feedback. As Feature Dashboard is a plugin, if an intended subject previously did not have Eclipse installed, then they would have to install Eclipse, the plugin, and additional dependencies. To make it as easy as possible to take part in the questionnaire, a version of Eclipse with the plugin and its dependencies pre-installed was provided. A description outlining the usage of the tool and views was also provided to subjects. Furthermore, the description provided examples of all views that were to be evaluated. There were also two projects, to simulate two variants, provided which would allow subjects to use the tool as intended.

Lastly, there were a series of controlled experiments in order to evaluate the initialization time of the views as the input increases. This was to offer performance metrics of how the views in the tool perform for future researchers and developers looking to use the tool.

3.5.5 Research contributions

The research contributions include an open source tool that has views that helps a user to trace features to their implementation in the code. This makes it possible for other interested parties to easily use the tool and/or contribute to the tool instead of developing something new from the ground up. Additional contributions include a literature survey along with an analysis of the result. Concept views and visualization ideas for different aspects of a project are also provided.

4
Result

In this chapter, the result of the thesis is presented. Firstly, an overview of the area of feature-traceability is given in the form of a literature survey. During the literature survey, there were a number of observations made which are presented. After the literature survey, a number of concepts views that could be present in a feature-traceability tool were created with the observations from the literature survey in mind. Afterward, the tool that the views were realized in, is presented. This tool is called Feature Dashboard and is an Eclipse plugin that will allow users to trace features to their location.

4.1 Literature survey

In this section, the literature survey that was conducted at the beginning of the thesis is presented. The goal of this literature survey was to find already suggested tools and views in the area of feature-traceability.

4.1.1 Result

Florida: Feature location dashboard for extracting and visualizing feature traces [11]. In this paper, a tool for tracing features to their location in code is suggested. There are two main views offered: Feature-file (tracing features to the source files they are implemented in) and Feature-folder (tracing features to their location in the folder structure). As can be seen in Figure 4.1, the traces are visualized using a graph where features and files are represented by nodes and if a file implements that feature then there is an edge between them. The feature-folder view is visualized using a different technique. As can be seen in Figure 4.2, features are still being represented as nodes but the folders are now boxes which are nested in each other to represent the folder structure. There is an additional view which is a table and provides different metrics for each feature. In terms of future work, they explain that they plan to use the underlying version-control system to extract metrics and have a view for this [11]. During the development of this tool, feedback on the views was provided by two industry contacts who were involved in the development. Other than that, there was no evaluation conducted.

Visualizing Software Product Line Variabilities in Source Code [20]. In this paper, a tool that is based on the IDE Eclipse is suggested which is called CIDE [20]. Features are visualized by first assigning a color for each feature that interests the user and then different parts of the IDE are color coded. For instance, different

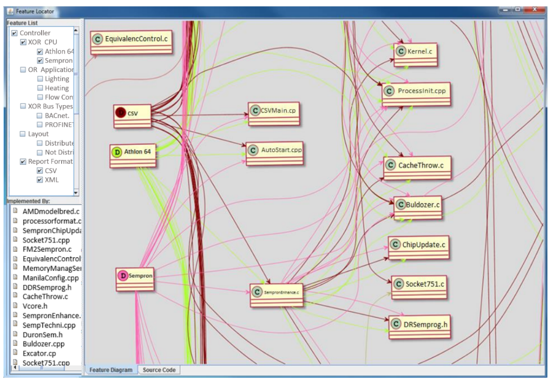


Figure 4.1: Feature-file view in FLOrIDA [11].

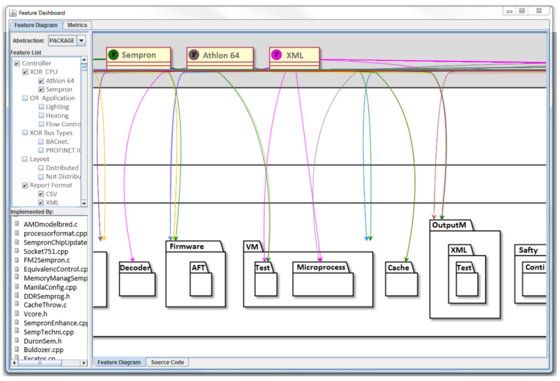


Figure 4.2: Feature-folder view in FLOrIDA [11].

folders and packages in the project explorer are color coded as well as specific parts of a source file. Since the tool is not evaluated in this paper one of the points for future work is to evaluate it and to evaluate different ways of creating views based on file contents [20].

FeatureIDE: A tool framework for feature-oriented software development [21]. In this paper, a framework for Feature-Oriented Software Development (FOSD) is being presented called FeatureIDE [21]. FeatureIDE is a collection of different tools that focuses on feature-oriented software development and one of these tools is the already mentioned CIDE. As such, visualizing feature-to-code is achieved in the same way as in CIDE. However, as mentioned by Meinicke et al. [22], just using the project explorer does not offer an effective overview of the project. As such, an additional view that is introduced in FeatureIDE is the *Collaborative diagram*. As can be seen in Figure 4.3 a file is selected and it is possible to see here that the features *Hello*, *Beautiful* and *World* are implemented in this file. No future work is suggested and no evaluation is presented.

View infinity: A zoomable interface for feature-oriented software development [23]. In this tool, it is possible to trace features to their implementation in the source code. The techniques used for visualization is similar to the already discussed tools such as CIDE and FeatureIDE, namely, associating colors with features and highlighting code in the editor. However, what is unique about this tool is the *zoomable* view. On the left, the current view is displayed and on the right, the different zoom levels are displayed (see Figure 4.4). The top level shows the feature model, next is a file view that shows different files that implement a specific feature and the final zoom level is looking at the source code for a specific file. They have provided a short summary of a usability evaluation that was performed and reached the conclusion that subjects were positive about the tool. Future work is to provide a more extensive evaluation and integrate the tool in a modern IDE.

Feature cohesion in software product lines: An exploratory study [24]. In this paper, the views that are suggested are not specifically about showing the location of features in the code. However, the views show the cohesion and coupling of features in the code, thus giving the user an idea of how features interact with one another in the code. The views suggested are based on clustering layouts. As

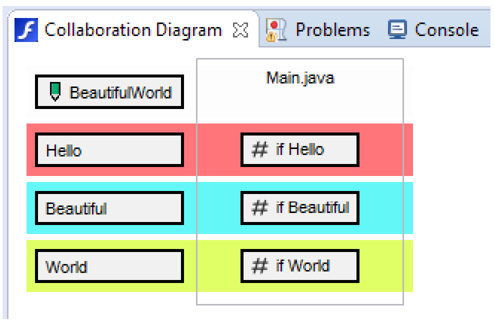


Figure 4.3: Collaborative diagram in FeatureIDE [22].

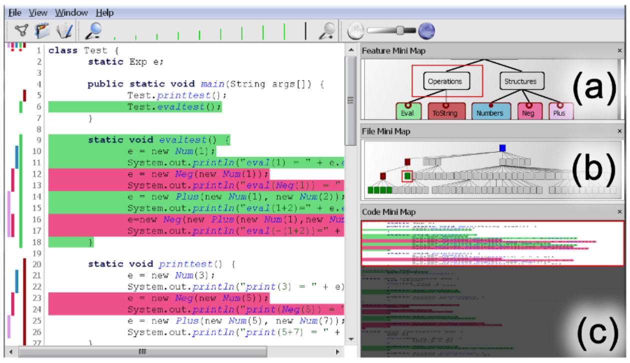


Figure 4.4: Overview of View infinity [23].

with other tools, different colors are assigned to selected features. No evaluation has been made on these views and there is no future work presented.

FeatureCommander: colorful #ifdef world [25]. In this paper, a prototype for tracing feature-to-code is presented. The idea behind the different visualizations is similar to CIDE, however, there are some differences. Looking at Figure 4.5, it is possible to see how much in percentage a feature or features use of a file. Nothing is suggested in terms of future work but they have performed an evaluation mainly testing the scalability and that subjects liked the idea of having different colors differentiating the features.

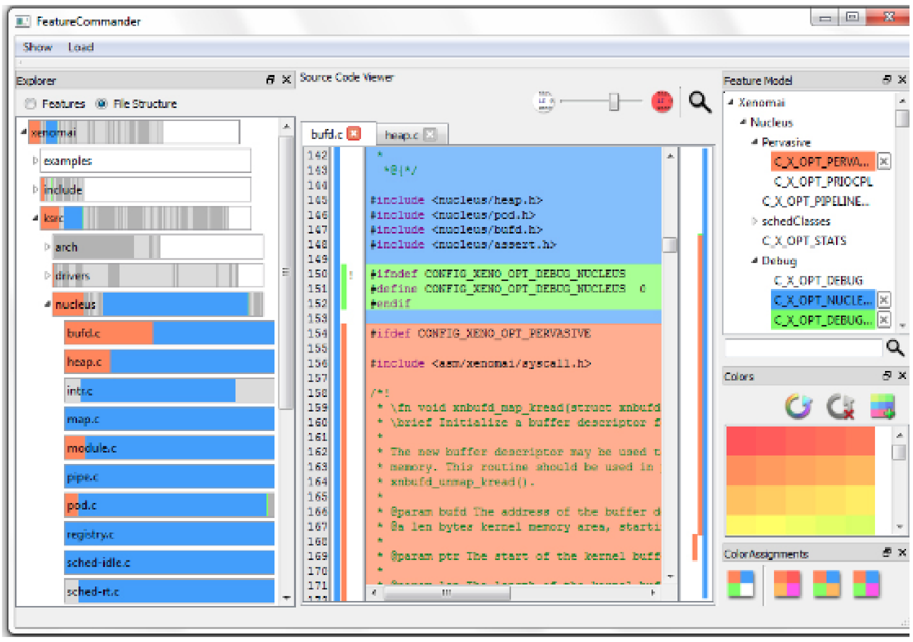


Figure 4.5: Overview of FeatureCommander [25].

On feature traceability in object oriented programs [1]. There are three different views suggested: visualization of feature metrics and roles, if there is any overlap amongst features, and evolution of features. The type of visualization used is the use of 3D boxes and depending on height, width, depth, and color different characteristics can be implied. These views have not been evaluated and the future work presented is to analyze more projects with these views.

4.1.2 Identified issues

During the literature survey and while examining the suggested views, a number of issues were identified. These issues can be problems that the authors themselves point out or by analyzing the views from a user point of view.

4.1.2.1 Colors

As can be seen in the different views already suggested, there is a common theme amongst them. And that is the usage of colors when displaying features and their location in code. However, using colors to distinguish features amongst each other has some problems associated with it. One of these problems is brought up by Kästner et al. [20], namely that if many features are to be displayed at the same time, at some point the user is going to have to use hues that are very similar to each other. Another problem is that there are colors that will not be possible to use due to contrast issues with the text and keywords in the editor. In Figure 4.4, it is possible to see the editor and the colors used by the text. If, for instance, a dark blue color would be used, then it would not be possible to see different parts of the code. Even though Feigenspan et al. [26] concludes that the usage of colors increases some aspects of program comprehension it might not beneficial in this specific use case. Additionally, some of these tools also require the user to manually assign colors to different features, adding an extra mandatory step by the user. Later, the user also needs to remember which feature is assigned to which color or navigate to whichever view where this information can be found since it is not immediately obvious. Another issue is regarding people that are color blind and their ability to use these views. As such, can views be created that does not have an over-reliance on colors to distinguish features?

4.1.2.2 Visual bloat

Since a lot of the previous work focuses on separating features by highlighting code in the editor, this is mainly something that was identified by looking at the views suggested by Andam et al. [11] and Apel and Beyer [24]. Looking at Figure 4.1, the traces between features and files are visualized by the use of a graph. In this graph, features and files are represented by nodes and if a feature is implemented in a specific file, then there is an edge between these nodes. In this particular example, there are a lot of features and files which implies a lot of connections. This means that it is difficult to distinguish which file node is connected to which feature node, or vice versa. One might be able to fix this by using smart layout algorithms for the nodes/connections so that there are fewer overlapping connections. However, having increasingly complex layout algorithms imply that views will take longer time to initialize. As a result, another important factor when creating views was how one could minimize the number of visual elements in the view but still provide effective feature-traceability.

4.1.2.3 Overview

Many tools presented in the literature survey, take advantage of similar visualization techniques. Particularly, the user assigns colors to features and correspondingly, files, folders, project explorer, and additional elements are color coded. However, as is mentioned by Meinicke et al. [22], using the project explorer for this intention makes it hard to establish an overview of the project. This was their reasoning as to why they introduced the Collaborative diagram to FeatureIDE (see Figure 4.3). As such, another priority when designing views was to conceive views that would require as little interaction as possible from the user but still offer an effective overview of the specific system.

4.2 Initial views

After having performed the literature survey and identified the previous work, a number of concepts were created to come up with different visualization ideas. There were no outside parties that had established requirements that were adhered to when creating these concepts. Concept views were created that would allow the user to do the following: a user should be able to trace a feature to files which implement this feature, the user should get an overview over the project and see where the feature is located in the folder structure, give the user the possibility to see how a feature has evolved in the project.

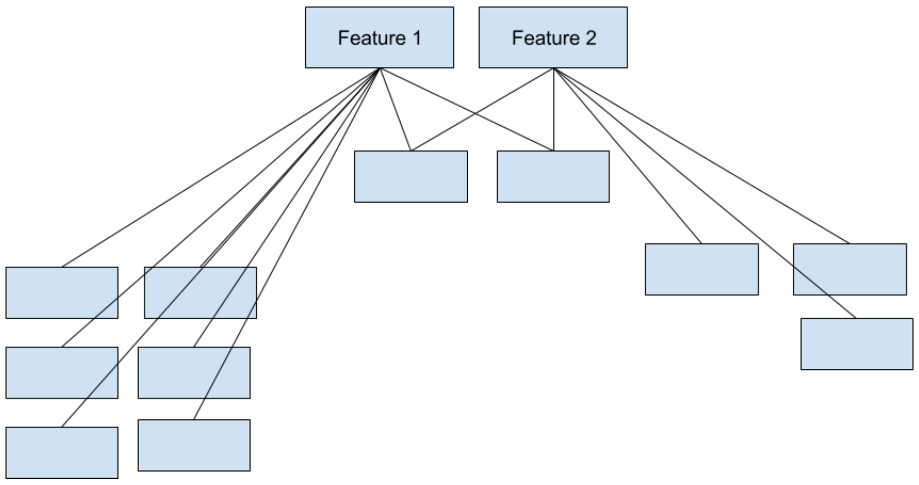


Figure 4.6: First feature-file concept.

Since views were going to be implemented in Eclipse, it was important to determine which visualization library that was going to be used in order to be aware of both the limitations and advantages of the library. There were a number of different libraries that were under consideration, however, it was decided that GEF Zest Version 5 [27] would be used. The reasoning behind this choice was that the library was the most up-to-date and still receives active support which is important for future development.

4.2.1 Feature-to-File

In this section, concepts of a view that would allow a user to see which files implement a feature are presented.

The first concept is presented in Figure 4.6. Here, the selected feature and files which implement said feature are represented by nodes in a graph. If there is a connection between nodes, then it means that the feature is implemented in that file.

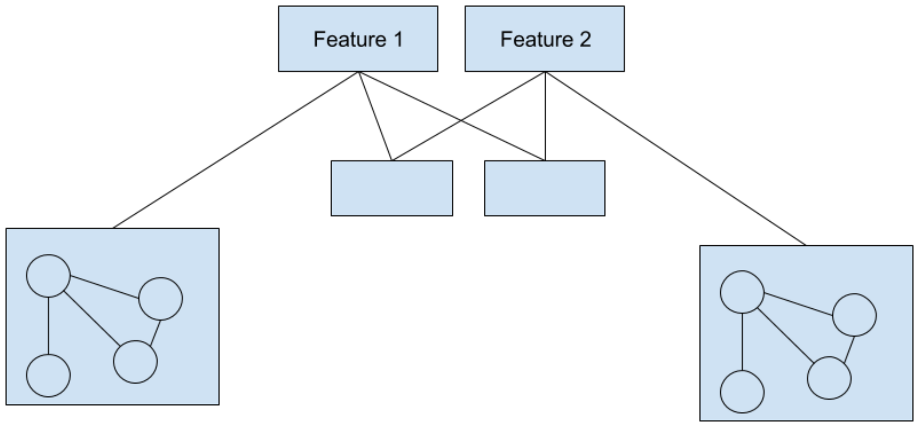


Figure 4.7: A variant of Figure 4.6.

Figure 4.7 is similar to the previous concept, however as written in Section 4.1.2, one of the issues identified is that there are too many visual elements in the view. One way to deal with this, as presented by Moody, is by utilizing *modularization* or *hierarchically structuring* [16]. Modularization implies dividing large systems into smaller subsystems as a means to help with information overloading. As such, by utilizing a feature from the visualization library used, information can be modularized into nested graphs. In this case, files belonging to just one particular feature will be placed in the respective nested graph. Files that implement multiple selected features should be positioned outside. As such, the user would quickly identify which files belong to a specific feature.

One of the disadvantages of using the nested graph functionality in Figure 4.7, is that it requires one additional interaction to show files that belong to that specific feature. Therefore, in Figure 4.8 the idea was still to utilize modularization but still have as little connections as possible and to remove the extra step needed to view the files which belong to just one of the selected feature. As such, files belonging to the same feature are grouped in a container and then a connection is made from this container to the feature.

A common property amongst the previous views is the dependency of connections from one node to another. This takes more time to render and introduces more visual elements to the screen. Thus, a view that does not rely on connections was created. In Figure 4.9, the features selected are represented by geographical shapes and if that feature is implemented in a file, a smaller version of that shape is placed in the node which represents the file. As Moody [16] writes, geographical shapes play a special role as it is the main property on which objects are identified in the real world. Even though there is no need for connections, and it is easy to see if a file implements

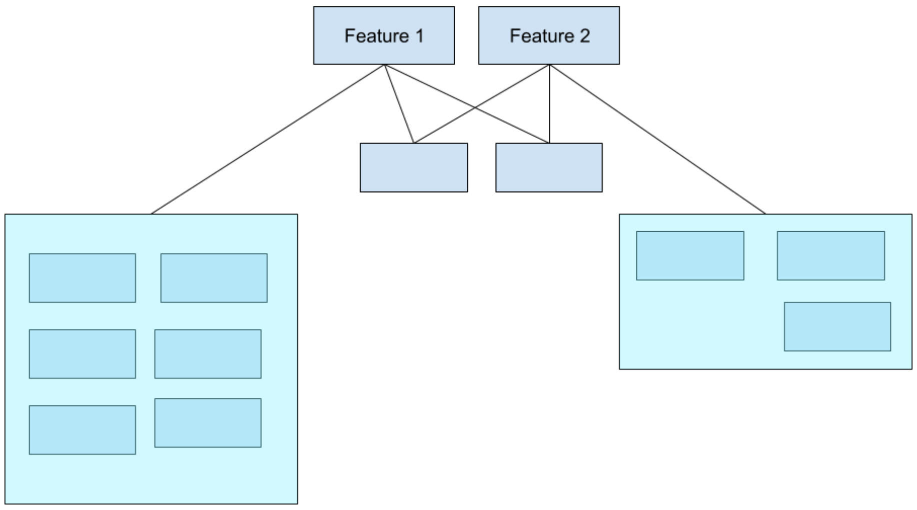


Figure 4.8: Concept to give better overview than Figure 4.7.

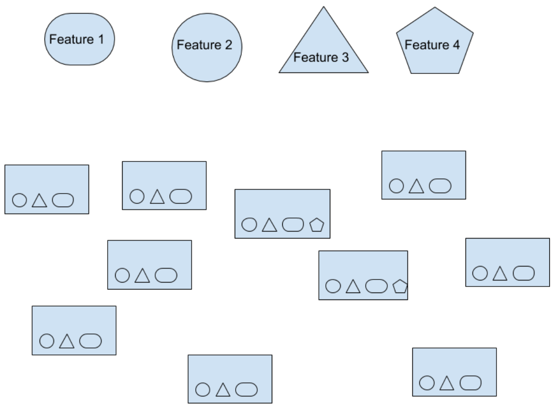


Figure 4.9: Concept that does not rely on connections between nodes.

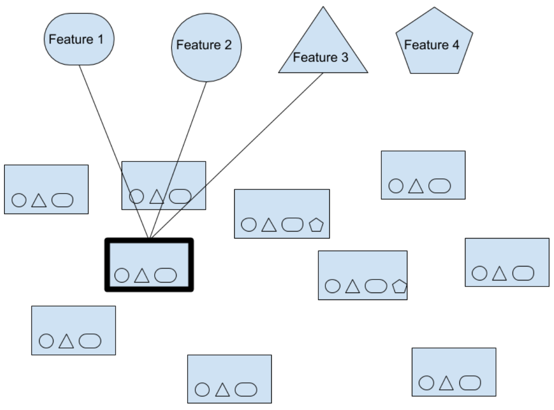


Figure 4.10: When a node in Figure 4.9 is highlighted.

more than one feature. One major limitation, similar to that of the usage of colors, is that there are only so many shapes that will be clearly distinguishable from each other. Ideas to alleviate this problem can be seen in Figure 4.10. Here, a specific node is focused and only then are connections shown.

The majority of the aforementioned views use a type of graph, in that there are nodes attached to each other by a connection. Inspiration taken from FeatureIDE and the collaborative diagram (see Figure 4.3), resulted in Figure 4.11. If two or more features are selected, all files belonging to the different features are displayed and if a feature is implemented in a file then this is indicated.

4.2.2 Feature-to-Folder

In this section, another type of view is described. This view should allow a user to trace features to their location in the folder structure of a project.

In Figure 4.12, the idea is to have a tree view over the folder structure. Additionally, the goal behind this concept was to reduce the number of columns created for the tree by combining folders that do not contain any features, to one node in-

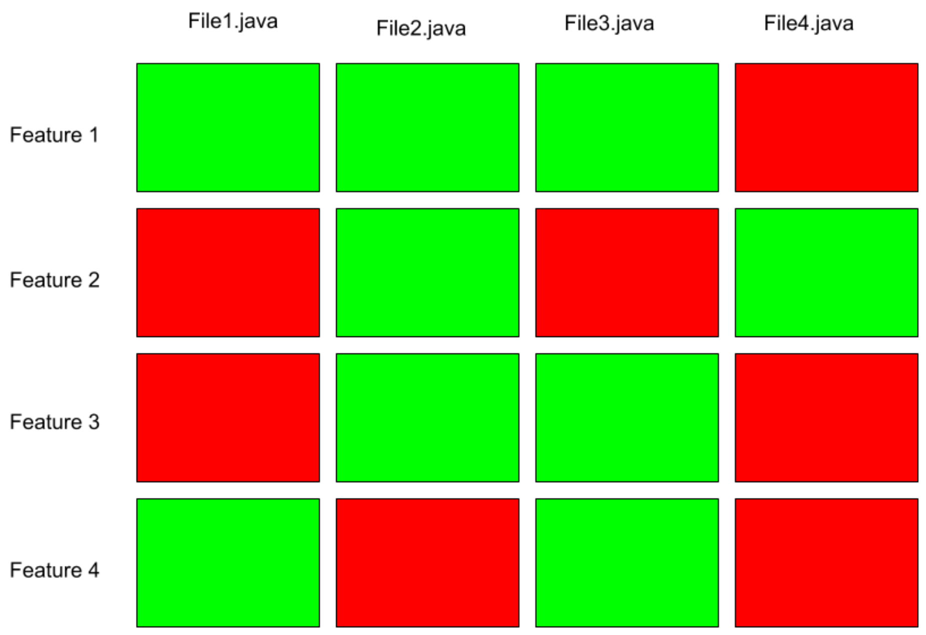


Figure 4.11: Concept inspired by the collaborative diagram in FeatureIDE.

stead of two or more. In certain programming languages, there are conventions that dictate a certain folder structure which creates empty folders which would probably not be interesting for the user.

Figure 4.13 demonstrates a slight alteration on the previous concept. Here, all folders are given a node to better visualize at which depth the feature is located at. Furthermore, instead of having one node for each feature that connects to each folder node, there is one feature node for each location. However, having one feature node for each location would quickly lead to a lot of nodes in the view, especially if a feature is extensively scattered across the project. As such, Figure 4.14 was created that has one feature node for each feature. However, the problem here is that now the nodes have to be placed intelligently so that connections are not overlapping.

4.2.3 History

As can be seen in the literature survey (Section 4.1), of the already suggested tools that have some sort of feature-location views, only Antoniol et al. [1], provides an example how to view the history or evolution for a feature. As such, concept views are presented in this section that visualizes a feature's evolution in a project.

4.2.3.1 Determining delta

When visualizing something that changes over time, the delta at which data points are retrieved needs to be determined. The delta used by Antoniol et al. [1] is one week. However, considering the state of the software industry today, where many development teams are working with agile workflows where many changes are happening on a daily basis, a week was determined to be too large of a delta. Since most (if not all) large-scale projects use an underlying version control system, it was decided to utilize this when gathering data about the state of the project in the

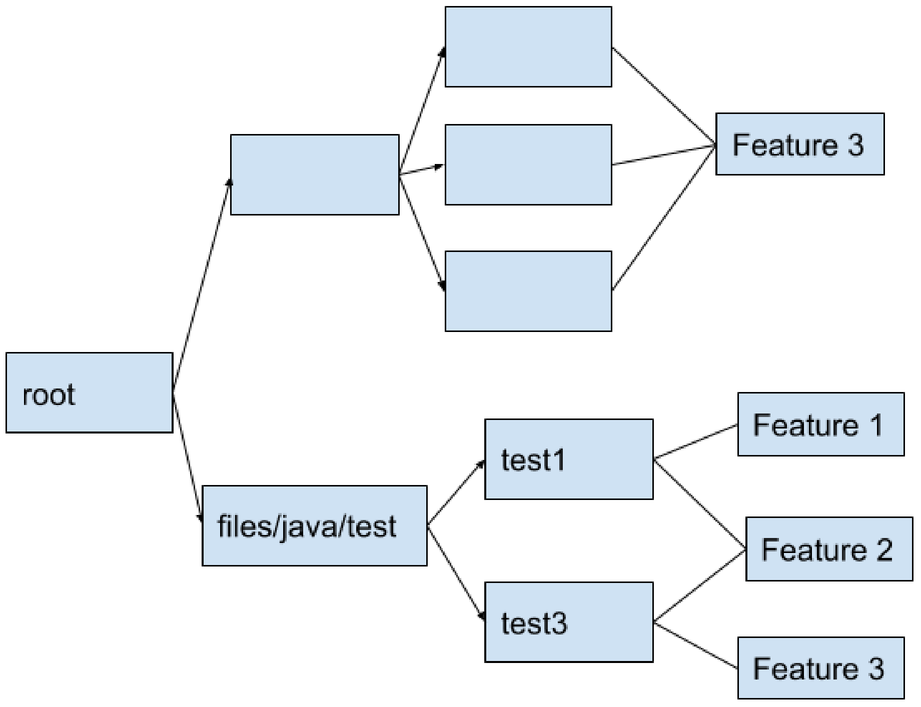


Figure 4.12: First feature-folder concept.

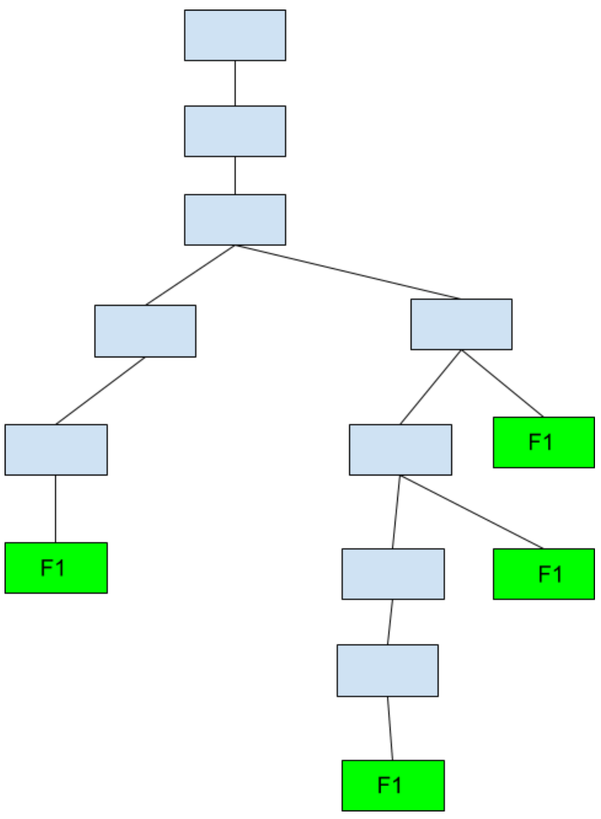


Figure 4.13: One feature node for each occurrence.

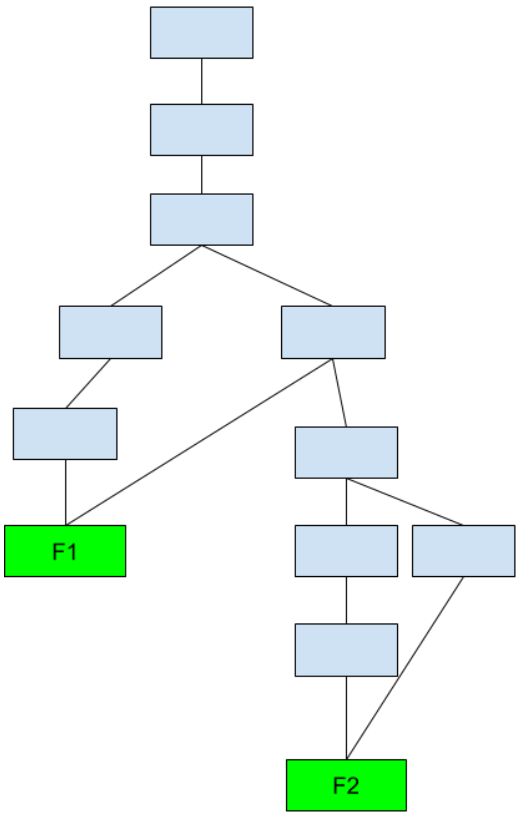


Figure 4.14: One node for each feature.

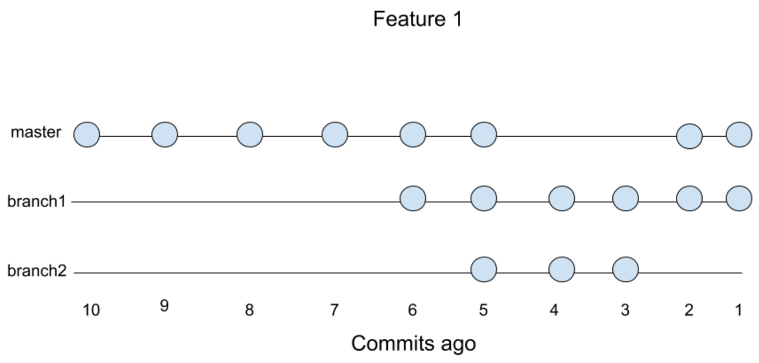


Figure 4.15: Showing if a feature is present in a commit for each branch/variant.

past. The versioning system chosen to support was Git since it is one of, if not the most, popular system to use according to a report in 2016 [28]. In Git, whenever a change is introduced by a developer, that change is saved in a *commit*. As a result, a commit is used as the delta in the concept views.

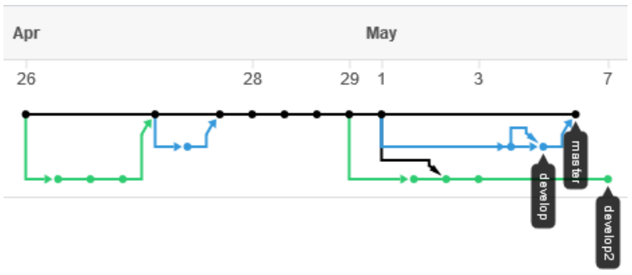


Figure 4.16: Github's visualization of branches.

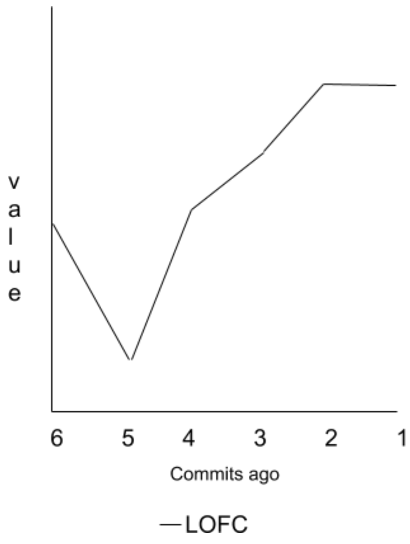


Figure 4.17: Metrics graph.

4.2.3.2 Concepts

The first view that came to mind when trying to see the history for a specific feature was a simple graph. This graph would plot values for a feature's metrics and how they change over a number of commits. Figure 4.17 shows a graph which displays the lines of feature code for a feature over a number of commits. Here, it would be possible to have any metric that can be associated with a feature.

Having multiple variants of a software product, at some point, a developer might want to compare these variants to each other. Figure 4.15 shows a concept which assumes that different variants are on different branches instead of separate projects. There are three variants: master, branch1, branch2. In this example, *Feature 1* is

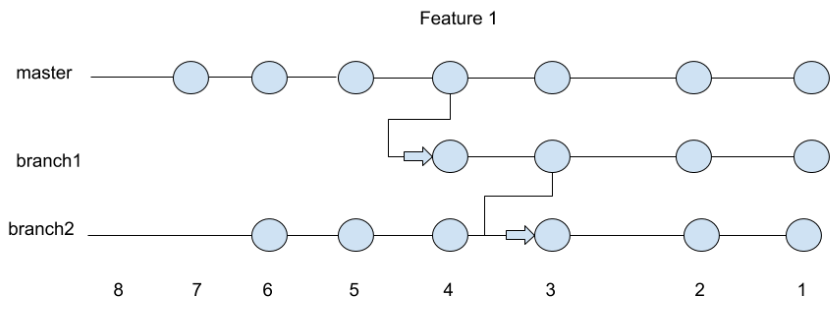


Figure 4.18: Example of combining Figure 4.15 and Figure 4.16.

selected and a circle represents that the feature was present in that specific commit. In this view, it is possible to see when a commit introduces a feature and if the other variants adopt that feature and at which point.

As previously described, these views utilize Git to get information about the code in the past, however, there is additional information that can be gathered from Git. Figure 4.16 demonstrates how Github visualizes how branches diverge and merge into one another for a specific project. As such, Figure 4.18 shows a view when combining Figure 4.15 and 4.16. Unlike Figure 4.15, this view is solely intended to compare different branches and how features move across them.

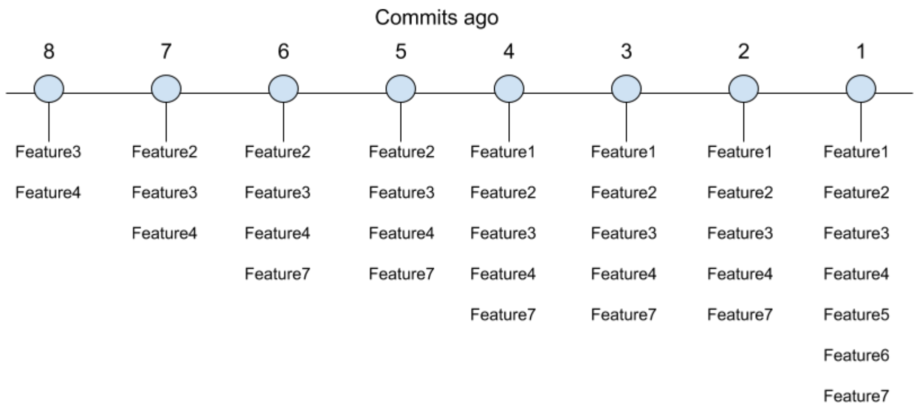
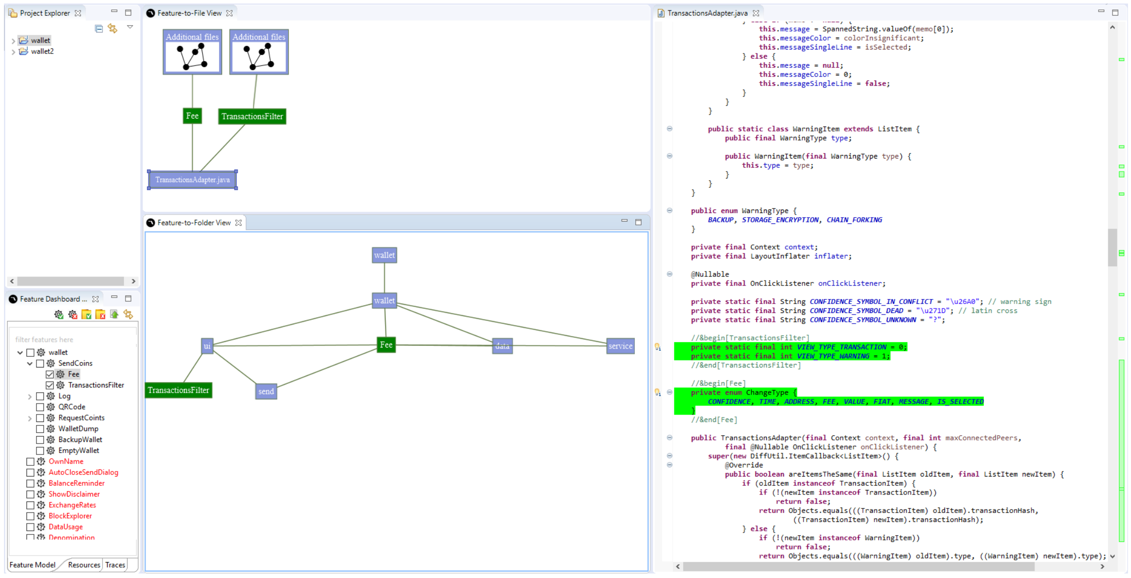


Figure 4.19: History of common features between variants.

When comparing variants, viewing which features that are shared between them could be interesting when, for instance, performing variability management. Figure 4.19 shows for each commit, a list of common features between two variants. Inspecting the view, as time moves on, two variants are becoming increasingly similar in terms of which features they implement. If this trend continuous, then it would be beneficial to merge them together.

4.3 Feature dashboard

An important part of the design science methodology, as detailed in Section 3.5, is that the result should be realized in an appropriate manner. As such, in this section, the tool *Feature Dashboard* is presented. Feature Dashboard is a plugin in the IDE Eclipse and allows users to trace and get information regarding features at the same time as developing software. In Figure 4.20, an example is presented of having some views offered by Feature Dashboard opened at the same time as writing code.



```java
                this.message = SpannedString.valueOf(memo[0]);
                this.messageColor = colorInsignificant;
                this.messageSingleLine = isSelected;
            } else {
                this.message = null;
                this.messageColor = 0;
                this.messageSingleLine = false;
            }
        }
    }

    public static class WarningItem extends ListItem {
        public final WarningType type;

        public WarningItem(final WarningType type) {
            this.type = type;
        }
    }
}

public enum WarningType {
    BACKUP, STORAGE_ENCRYPTION, CHAIN_FORKING
}

private final Context context;
private final LayoutInflater inflater;

@Nullable
private final OnClickListener onClickListener;

private static final String CONFIDENCE_SYMBOL_IN_CONFLICT = "\u26A0"; // warning sign
private static final String CONFIDENCE_SYMBOL_DEAD = "\u271D"; // latin cross
private static final String CONFIDENCE_SYMBOL_UNKNOWN = "?";

//&begin[TransactionsFilter]
private static final int VIEW_TYPE_TRANSACTION = 0;
private static final int VIEW_TYPE_WARNING = 1;
//&end[TransactionsFilter]

//&begin[Fee]
private enum ChangeType {
    CONFIDENCE, TIME, ADDRESS, FEE, VALUE, FIAT, MESSAGE, IS_SELECTED
}
//&end[Fee]

public TransactionsAdapter(final Context context, final int maxConnectedPeers,
        final @Nullable OnClickListener onClickListener) {
    super(new DiffUtil.ItemCallback<ListItem>() {
        @Override
        public boolean areItemsTheSame(final ListItem oldItem, final ListItem newItem) {
            if (oldItem instanceof TransactionItem) {
                if (!(newItem instanceof TransactionItem))
                    return false;
                return Objects.equals(((TransactionItem) oldItem).transactionHash,
                        ((TransactionItem) newItem).transactionHash);
            } else {
                if (!(newItem instanceof WarningItem))
                    return false;
                return Objects.equals(((WarningItem) oldItem).type, ((WarningItem) newItem).type);
```

Figure 4.20: Opening Feature Dashboard views at the same time as developing.

4.3.1 Annotations

What is perhaps the most important part of this tool is how features are found in the code. As mentioned in Section 2.5.2, there exists a number of different ways of extracting these locations. Feature Dashboard uses the textual approach by parsing source files and looking for *annotations*. The syntax used for annotations is the same as suggested by Ji et al. [2]. There are three types of annotations: *line*, *start-block*, and *end-block*. Listing 4.1 shows an example of these annotations and how they are used to indicate the location of features in the code. In the aforementioned example, there are two features: *Main* and *ConsolePrint*. Lines 3-6 belongs to Main and line 5 belongs to ConsolePrint.

```
public class HelloWorld {
    // &begin[Main]
    public static void main(String[] args){
        // &line[ConsolePrint]
        Arrays.stream(args).forEach(System.out::println);
    }
    // &end[Main]
}
```

Listing 4.1: Examples of feature annotations in a Java source file.

4.3.2 Mapping files

In addition to being able to add annotations in the source code, there is another way of declaring that a specific resource belongs to a feature. In some cases, a user knows that an entire file or folder belongs to a feature. In those cases, going through all resources and adding annotations can become tiresome. Similarly to Andam et al. [11], there are also specific *mapping files* that either map a file or folder to a feature. These files have the file extension of `.feature-file` and `.feature-folder`. For `.feature-file`'s there are two different syntaxes that are supported, these can be found in listings 4.2 and 4.3.

```
Feature1: File1.java, File2.java
Feature2: File3.java, File4.java
```

Listing 4.2: One syntax for .feature-file.

```
File1.java File2.java
Feature1
File3.java File4.java
Feature2
```

Listing 4.3: Another syntax for .feature-file.

The reasoning behind having two different syntaxes for the `.feature-file` is because there already exists datasets that use these two types of syntaxes. Therefore, being able to support both of them was important.

As previously mentioned, the `.feature-file`'s are used when an entire file is associated with a feature. However, if a user knows that a folder and all of its contents (includes all sub-folders) belong to a feature, it is possible to use the `.feature-folder` file instead. In this file, a user simply lists a feature on each line that should be associated with the contents in the folder which the file is located in. Listing 4.4 shows an example of how this file could be placed in a folder structure. All resources in *Root* will successively belong to any feature(s) specified in the mapping file.

```
Root
    .feature-folder
    File1.java
    File2.java
    Sub1.1
        File3.java
        File4.java
    Sub1.2
        File5.py
```

Listing 4.4: Example of a .feature-folder file in a folder strucutre.

As can be noted from Listing 4.4, the mapping files will be scattered across the codebase that is annotated. This is intentional as there are several advantages of having these files scattered across the codebase. One of these advantages is that there will be no single large file that contains all information. As this file increases in size, it will become increasingly difficult to maintain and to confirm that every mapping is correct. Another advantage is if the folder structure is refactored at some point. In the case of a single file, every line that is affected would have to be manually changed which would undoubtedly be an error-prone task. With the scattered approach, as a folder is moved, so will the mapping file.

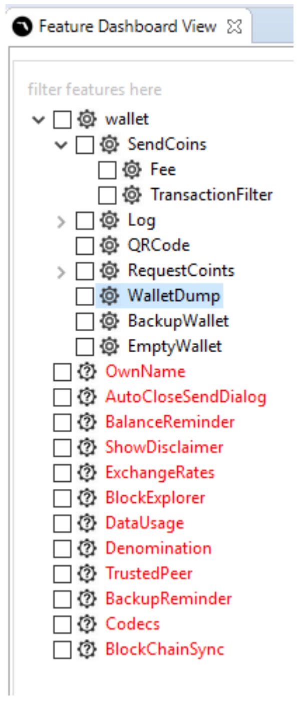
4.3.3 Views

In this section, the different views that are included in Feature Dashboard are presented. At the end of the thesis, these views were evaluated and the result can be inspected in Section 5.2.

4.3.3.1 Feature Dashboard View

This is the main view of the plugin, from here, the features that are located in the parsed project are selected. Figure 4.21 shows an example of this view and each feature found can be selected by pressing in the respective checkbox. Depending on what is selected in this view, it will change what is shown in (almost) all other views.

It can be noted that some features in the view have different layout properties. This is because of the possibility to add a feature model (see Section 2.3) in the root of the project. If the plugin detects a feature model, the features will be placed in accordance with this feature model. The feature model is specified in a language called *Clafer*, which allows a user, (among other things) to specify a feature model textually [29, 30]. This file is added in the root of the project and is given a name with the file extension *.cfr*.



Feature Dashboard View

filter features here

```
wallet
    SendCoins
        Fee
        TransactionFilter
    Log
    QRCode
    RequestCoints
    WalletDump
    BackupWallet
    EmptyWallet
OwnName
AutoCloseSendDialog
BalanceReminder
ShowDisclaimer
ExchangeRates
BlockExplorer
DataUsage
Denomination
TrustedPeer
BackupReminder
Codecs
BlockChainSync
```

Figure 4.21: Feature Dashboard View.

As can be noted in Figure 4.21, some features are colored red. This means that the feature was found when parsing annotations, but it was not found in the feature model. This enables the user to either update the feature model or correct any mistakes that have been made.

4.3.3.2 Feature-to-File view

As the name suggests, this view allows the user to trace features to files that implement the said feature. To view this information about a feature, the user selects the desired feature in Feature Dashboard View (see the previous section).

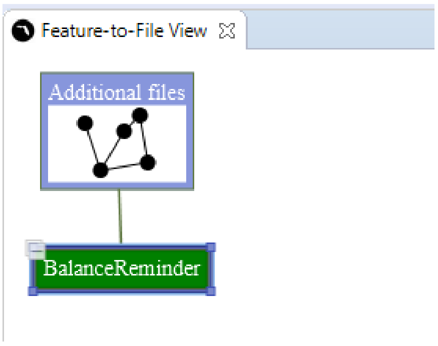


Figure 4.22: Feature-to-File view with one feature selected.

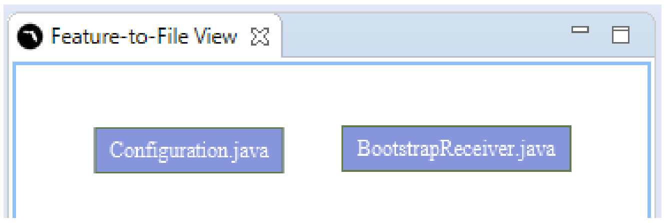


Figure 4.23: Files belonging to feature *BalanceReminder*.

Figure 4.22 shows what is displayed when the feature *BalanceReminder* has been selected. The node which represents the feature is connected to another node that contains a nested graph that has the label 'Additional files'. Double-clicking on it will show the view presented in Figure 4.23, in which files are also represented by nodes. Firstly, if the user is interested to see which lines inside a specific file belong to the feature, it is possible to double-click the node. This action will open the file in the editor and the appropriate lines of code will be highlighted. It is also possible to see the project relative path for the file in question by hovering with the mouse over the node in the view. To get back from Figure 4.23 to 4.22 the user has to double-click anywhere inside the view (except on the actual nodes) or zoom out.

The view will have slightly different behavior depending on if the selected features are *tangled* with one another. Two features are said to be tangled if they are both implemented in the same file. In Figure 4.24, multiple features have been selected, namely *BalanceReminder*, *Log*, *ExchangeRates*, *BIP70*, and *BlockChainSync*. Unlike Figure 4.22, there are file nodes in the initial view that connects to a feature node. This implies that the features *Log*, *BalanceReminder*, and *ExchangeRates* are all implemented in the file `Configurations.java`. It can also be noted that the feature *Log*, does not have a connection to a nested graph node which the other feature nodes have. This is because that feature is only implemented in a single file, namely `Configurations.java`. All other features are implemented in additional files that the others are not implemented in. As with the first view, if a user double-clicks a file node, for instance `SendCoinFragments.java`, that file will be opened in the editor and code belonging to *BIP70* and *ExchangeRates* will be highlighted.

The same design philosophy, as outlined in Section 4.2.1, regarding taking advantage of modularization was used here as well. Moreover, as detailed in Section 4.1.2,

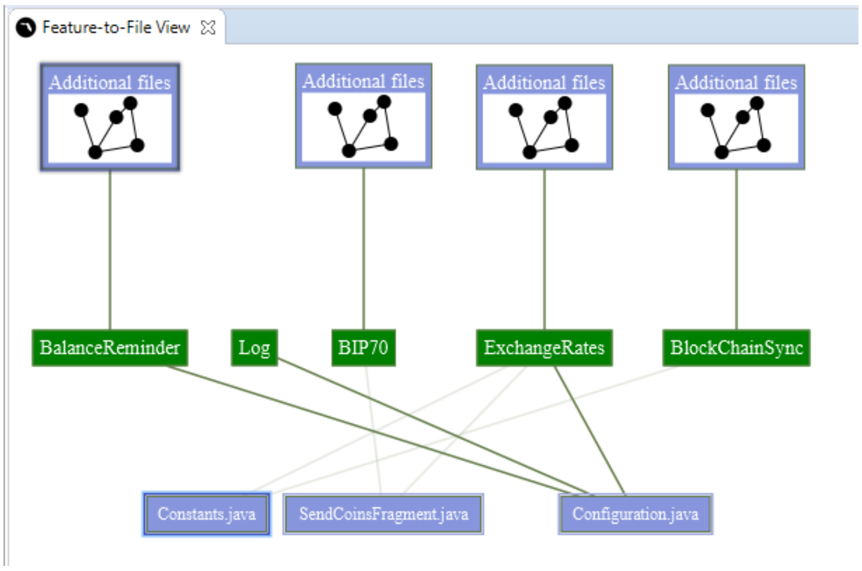
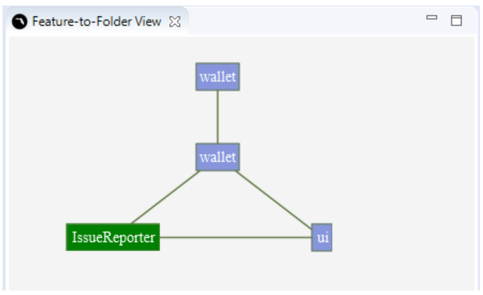


Figure 4.24: Feature-to-File view with multiple features selected.

one of the issues found with other views was that the number of connections made the view difficult to interpret as there were, in some cases, too many connections. Here, the choice was made to have connections from files to features so the user can quickly gather an overview of the features that are tangled together and in which files. As El Ahmar et al. [18] lists, there are a number of visual variables that can be used in visual representations. Of these, brightness has been taken advantage of when visualizing the connections in this view. As can be seen in Figure 4.24, some of the connections are less bright than the others. By default, all connections are made less bright and when a file node is selected, those connections will be made brighter to clearly indicate which files belong to those features.

It is also important to note that in the case of `Constants.java`, for instance, there could be more features located in this file but among the selected features, only *ExchangeRates* and *BlockChainSync* are implemented in it.

4.3.3.3 Feature-to-Folder view



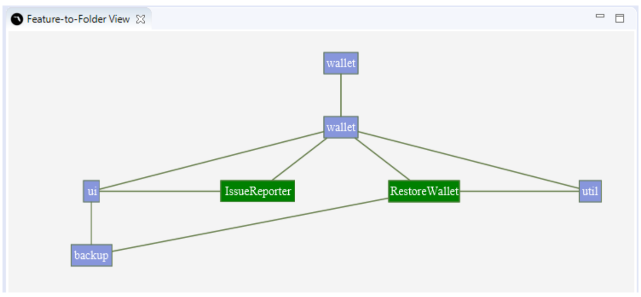


Figure 4.25: Feature-to-Folder view with a single feature selected.

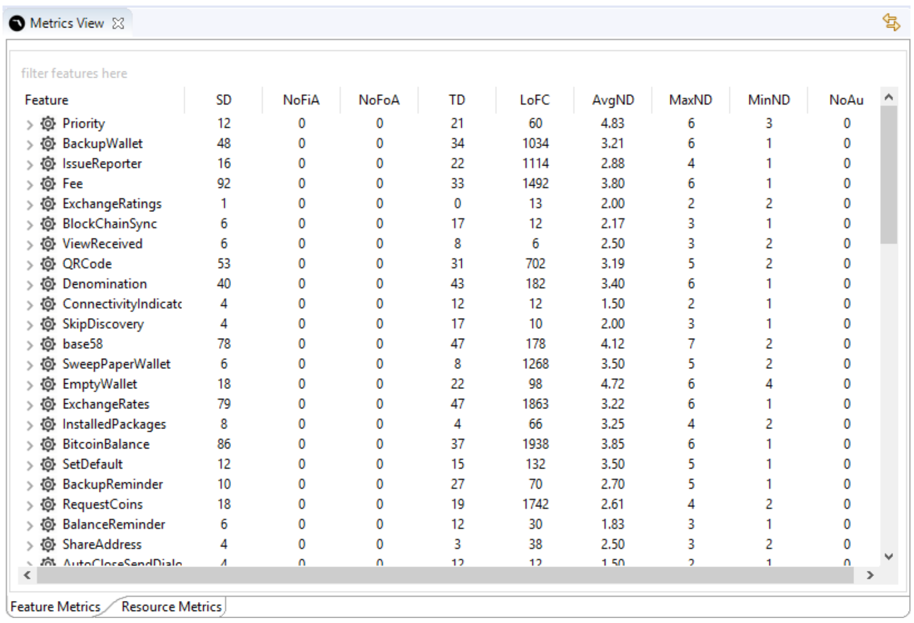
Figure 4.26: Feature-to-Folder view with multiple features selected.

In this view, it is possible to see where in the folder structure one or more features are located. When selecting a feature, the folder structure will be created and each folder will have a node with a connection to its parent folder. As with all other views, the feature is represented by a node with a green background color. A connection will be made between a feature and a folder node if the files that the feature is implemented in are located in that folder. In Figure 4.25 the feature *IssueReporter* has been selected and from the connections it is possible to deduce that files implementing this feature are located in the folders `wallet/wallet` and `wallet/wallet/ui`. An additional example where two features are selected can be seen in Figure 4.26.

4.3.3.4 Metrics view

Similarly to Andam et al. [11], Feature Dashboard also offers a view that allows the user to retrieve different types of metrics about the features located in a project. Two abstraction levels were made for the metrics: feature and resource. The views shown in figures 4.27 and 4.28 show metrics for features and resources respectively.

The metrics for each feature are the following:



Feature	SD	NoFiA	NoFoA	TD	LoFC	AvgND	MaxND	MinND	NoAu
Priority	12	0	0	21	60	4.83	6	3	0
BackupWallet	48	0	0	34	1034	3.21	6	1	0
IssueReporter	16	0	0	22	1114	2.88	4	1	0
Fee	92	0	0	33	1492	3.80	6	1	0
ExchangeRatings	1	0	0	0	13	2.00	2	2	0
BlockChainSync	6	0	0	17	12	2.17	3	1	0
ViewReceived	6	0	0	8	6	2.50	3	2	0
QRCode	53	0	0	31	702	3.19	5	2	0
Denomination	40	0	0	43	182	3.40	6	1	0
ConnectivityIndicatc	4	0	0	12	12	1.50	2	1	0
SkipDiscovery	4	0	0	17	10	2.00	3	1	0
base58	78	0	0	47	178	4.12	7	2	0
SweepPaperWallet	6	0	0	8	1268	3.50	5	2	0
EmptyWallet	18	0	0	22	98	4.72	6	4	0
ExchangeRates	79	0	0	47	1863	3.22	6	1	0
InstalledPackages	8	0	0	4	66	3.25	4	2	0
BitcoinBalance	86	0	0	37	1938	3.85	6	1	0
SetDefault	12	0	0	15	132	3.50	5	1	0
BackupReminder	10	0	0	27	70	2.70	5	1	0
RequestCoins	18	0	0	19	1742	2.61	4	2	0
BalanceReminder	6	0	0	12	30	1.83	3	1	0
ShareAddress	4	0	0	3	38	2.50	3	2	0
AutoCloseSendDialo	4	0	0	12	12	1.50	2	1	0

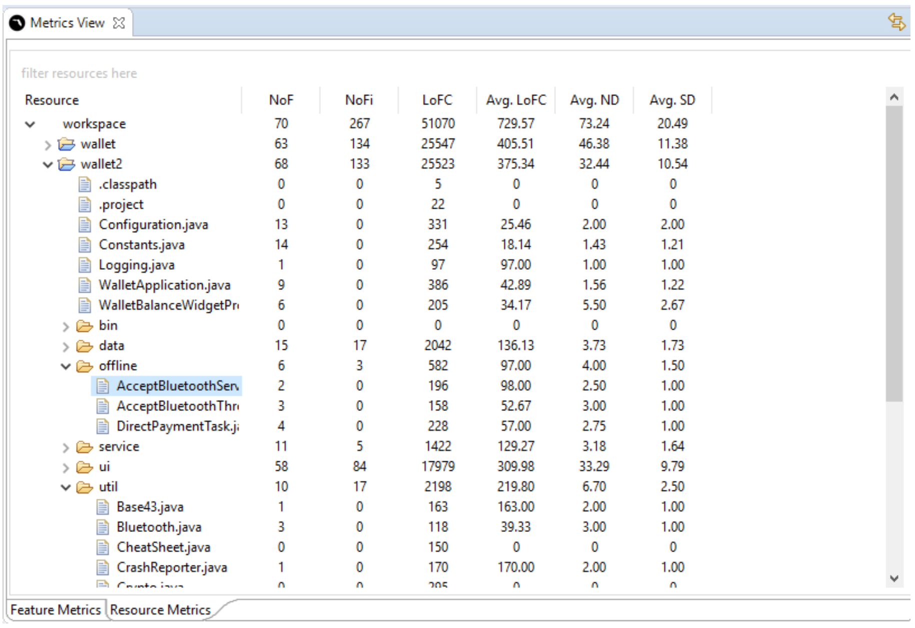
Figure 4.27: Metrics for features.

- **Lines of feature code (LOFC)**
  Counds the number of lines referenced by annotations in the code. When a feature is mentioned in a .feature-file or .feature-folder the entire file is taken into account.
- **Number of File Annotations (NoFiA)**
  The number of entire files belonging to a feature as a result of .feature-file and .feature-folder files.
- **Number of Folder Annotations (NoFoA)**
  Number of entire folders that has been indicated to belong to a feature.
- **Scattering degree (SD)**

Amount of in-file annotations referencing a feature (see Section 4.3.1), plus NoFiA and NoFoA.

- **Tangling degree (TD)**
  A feature is tangled if, in a file, there are multiple annotations referencing different features. For instance, if a file has two annotations and they reference different features, then the tangling degree for those features is increased by one.
- **Nesting depth (ND)**
  The project root has a depth of zero and for each folder, the depth increases by one. The depth is also increased for every annotation block that the current annotation is located inside. Maximum, average, minimum nesting depth is provided for each feature as well.
- **Number of authors (NoAu)**
  Amount of authors that contributed to a source file that the feature is located in.

The metrics for resources are mostly the same as for features, however, there are some differences. Firstly, the average is shown for LOFC, ND, TD, and SD. For folders, the average is calculated over the number of files in that folder and for files, the average is calculated over the number of features in the file. A metric introduced for resources is the number of features (NoF) which shows the number of unique features located in a folder/file. Lastly, it is also possible to see the number of files (NoFi) for folders.



Resource	NoF	NoFi	LoFC	Avg. LoFC	Avg. ND	Avg. SD
workspace	70	267	51070	729.57	73.24	20.49
wallet	63	134	25547	405.51	46.38	11.38
wallet2	68	133	25523	375.34	32.44	10.54
.classpath	0	0	5	0	0	0
.project	0	0	22	0	0	0
Configuration.java	13	0	331	25.46	2.00	2.00
Constants.java	14	0	254	18.14	1.43	1.21
Logging.java	1	0	97	97.00	1.00	1.00
WalletApplication.java	9	0	386	42.89	1.56	1.22
WalletBalanceWidgetPr	6	0	205	34.17	5.50	2.67
bin	0	0	0	0	0	0
data	15	17	2042	136.13	3.73	1.73
offline	6	3	582	97.00	4.00	1.50
AcceptBluetoothSer	2	0	196	98.00	2.50	1.00
AcceptBluetoothThr	3	0	158	52.67	3.00	1.00
DirectPaymentTask.j	4	0	228	57.00	2.75	1.00
service	11	5	1422	129.27	3.18	1.64
ui	58	84	17979	309.98	33.29	9.79
util	10	17	2198	219.80	6.70	2.50
Base43.java	1	0	163	163.00	2.00	1.00
Bluetooth.java	3	0	118	39.33	3.00	1.00
CheatSheet.java	0	0	150	0	0	0
CrashReporter.java	1	0	170	170.00	2.00	1.00

Figure 4.28: Metrics for folders/files.

4.3.3.5 Tangling view

One prominent point of feedback from the first evaluation (see Section 5.1), was the lack of a view that showed which features a specific feature is tangled with. In the metrics view (see the previous section), it is possible to see the tangling degree for a feature but it is not shown which features a feature is tangled with. As a result of this feedback, the view shown in Figure 4.29 was conceived. By selecting a feature from the Feature Dashboard View (see Section 4.3.3.1), it will show which features the selected feature is tangled with. It is important to note that it only shows from the perspective of the selected feature. As can be seen in Figure 4.29, there are only connections to/from the selected feature which in this case is *BackupReminder*, but not between any other features. This does not mean that they are not tangled, it is simply because the decision was made only to show how the selected feature is tangled with other features.

In previous views, the same visualization is used for feature nodes to quickly identify them. In this view, however, there are only feature nodes that are visualized. As such, a concept described by Moody [16] is used, namely: *perceptual popout*. This means that specific elements should 'pop-out' or be distinguishable without conscious effort [16]. This is achieved by, firstly, using a layout algorithm called *Spring layout*, which pushes connected nodes away from each other, it is possible to convey which node is selected by connecting all tangled nodes with the selected node. Furthermore, selected features are displayed in a different color, clearly distinguishing them further.

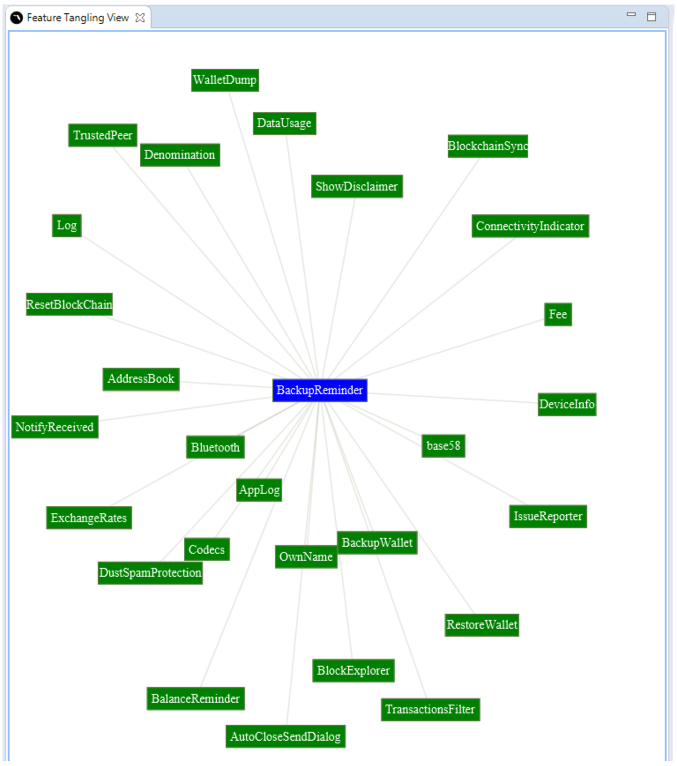
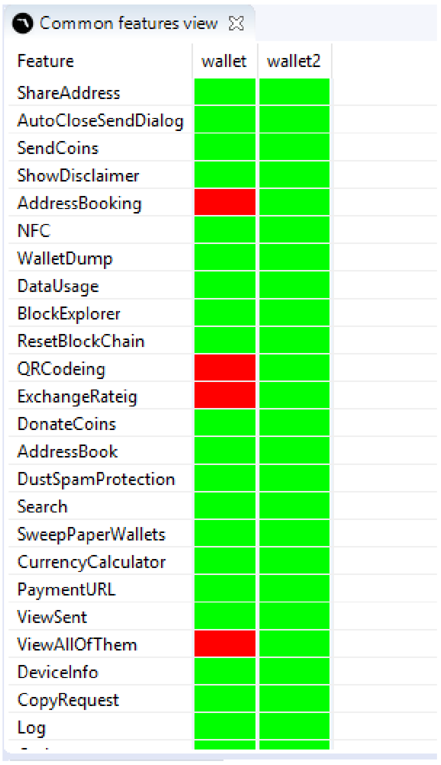


Figure 4.29: Features tangled with *BackupReminder*.


Common features view

Feature	wallet	wallet2
ShareAddress	🟩	🟩
AutoCloseSendDialog	🟩	🟩
SendCoins	🟩	🟩
ShowDisclaimer	🟩	🟩
AddressBooking	🟥	🟩
NFC	🟩	🟩
WalletDump	🟩	🟩
DataUsage	🟩	🟩
BlockExplorer	🟩	🟩
ResetBlockChain	🟩	🟩
QRCodeing	🟥	🟩
ExchangeRateig	🟥	🟩
DonateCoins	🟩	🟩
AddressBook	🟩	🟩
DustSpamProtection	🟩	🟩
Search	🟩	🟩
SweepPaperWallets	🟩	🟩
CurrencyCalculator	🟩	🟩
PaymentURL	🟩	🟩
ViewSent	🟩	🟩
ViewAllOfThem	🟥	🟩
DeviceInfo	🟩	🟩
CopyRequest	🟩	🟩
Log	🟩	🟩

Figure 4.30: Common features between two variants.

In this view, it is shown which features a feature is tangled with, however, there is still information that might be interesting for a user that is missing. Particularly, in which files/folders are the features tangled? As such, it is possible to double-click a connection between two features. This will open the *Feature-to-File* and *Feature-to-Folder* view (see Section 4.3.3.2 and 4.3.3.3 respectively) with those two features as a selection. Subsequently, it will allow the user to see in which files two features are tangled in and see their location in the folder structure.

4.3.3.6 Common features view

An additional point of feedback from the first interview was being able to compare two projects with one another and see which features are and are not shared between them. Correspondingly, the view, *Common features*, was conceived (see Figure 4.30). As can be seen, in the first column all unique features found between the two projects are listed. Projects have a column each and in each cell, it is indicated if that feature is present in that variant or not.

5 Evaluation

This chapter presents the result of the evaluations performed during the thesis.

5.1 Semi-structured interviews

In this section, the result of the interviews is presented. For each view, a short description of the feedback will be presented and quotes taken from the interviews will be presented where appropriate.

Feature-To-File. As can be seen from the following quote, this view was not received positively and the usefulness of it was questioned. This was the case for all subjects interviewed.

> *[...] with this functionality it does not seem to be necessary, but if it could show which features depend on other features then it could be very useful I think [...]*

As can read from the quote, some useful feedback was received in how it could be improved in the future.

Feature-To-Folder This view was received positively as subjects thought it gave a good overview of the project. There was no specific point of feedback regarding the view that the subjects thought was bad or needed improvement.

> *[...] It could be helpful as a complementary view to see where they exist in the file structure [...]*

Feature/Project metrics view Of the views presented, these views were the views that received the most positive feedback. The ability to easily see each feature in a project and sort on LOFC, TD, for instance, was highly appreciated.

> *[...] especially tangling. Is it possible to see which other features it is tangled with?*

When asked if any metric was missing none of the subjects had any suggestion, however, as can be read from the quote, subjects wanted a view where it was possible to see which features are tangled. Furthermore, similarly to the TD, when subjects saw the number of features found across for projects, they pointed out that firstly, when summing features across projects, it should show the number of unique features. Furthermore, they pointed out that offering a view that allows them to see common features across the project was very important.

Feature Selection view. Regarding this view, there was no specific feedback other than it was appreciated that it was very easy to select different features and see information regarding them. Another positive feedback was that the features in the view adapted to the way they are located in the feature model and if any features were missing from it.

Concept Views. As can be seen in the interview protocol, some questions were asked about the viewpoint of the subject regarding the ability to view the history of a feature. As such, some of the concepts that were created (see Section 4.2.3) were shown to gather their opinion and feedback. Subjects stated that this was indeed something that they would want to have and that it would provide added value. For instance, regarding Figure 4.17, it was suggested to show one specific metric for all features instead of just showing one feature and comparing different metrics. Moreover, another comment regarding Figure 4.15 was that the possibility to click on a specific commit and directly see information about the state of the project at this point in time should be available.

Overall. The feedback overall was positive and they all expressed that views such as these would be helpful in their work merging variants into an SPL. Even though it was not specifically evaluated, additional feedback regarding interactions in the tool was received. Such as, that it should be possible to go from one view to another by interacting with elements in the view.

5.2 Questionnaire

In this section, the result of the questionnaire is presented. Since the subjects were going to use the tool themselves, there could be problems preventing them to use it. As such, subjects had to indicate whether or not they had used the tool or simply inspected the examples provided in the documentation. Having used the tool or not is defined as having at least opened each view once and seen one example.

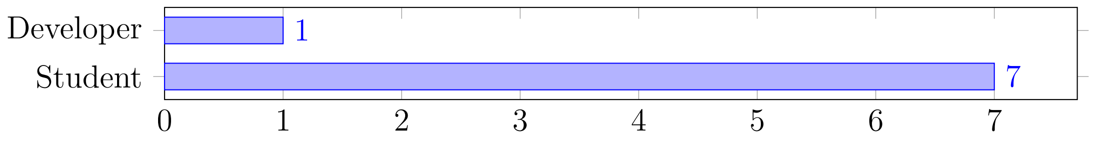


Figure 5.1: Subjects' background

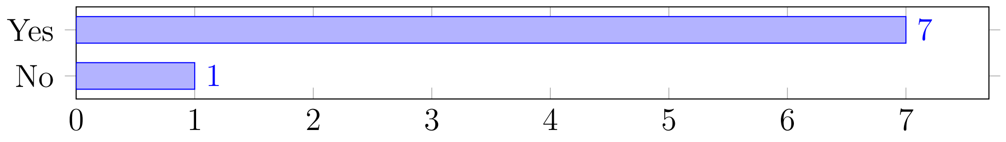


Figure 5.2: If subjects used the tool before taking the questionnaire

5.2.1 Feature-to-File

There were two comments regarding this view, they were both pointing out problems with the interaction of viewing additional files for a feature. Going back to the overview after having seen additional files was the problem in both cases.

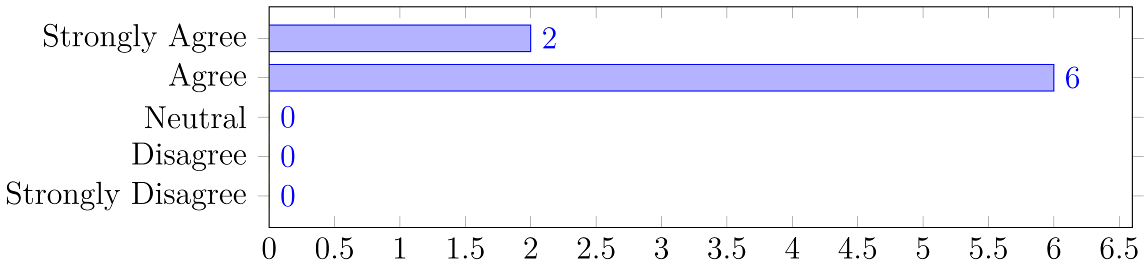


Figure 5.3: I think the feature-to-file view was useful in tracing features to files which implement them

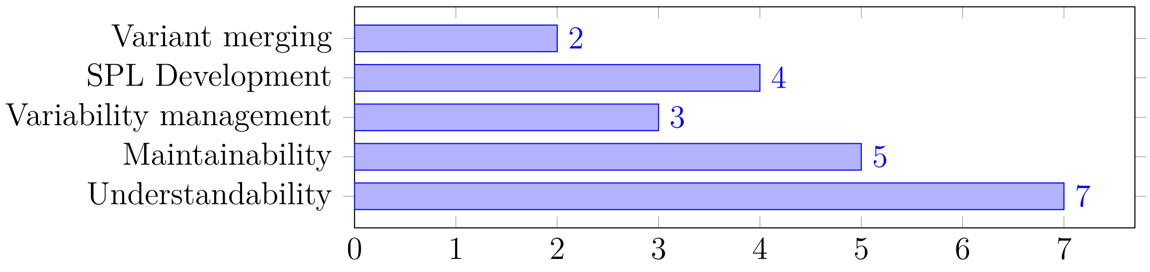


Figure 5.4: I think the feature-to-file view will be helpful in the following aspects

5.2.2 Feature-to-Folder

Comments for this view was regarding the fact that when trying to view multiple features, the view became too cluttered. It became too cluttered since there were many connections, some of which overlapped other connections and nodes. One subject suggested having connections between files and features in this view as well, effectively merging Feature-to-File and Feature-to-Folder.

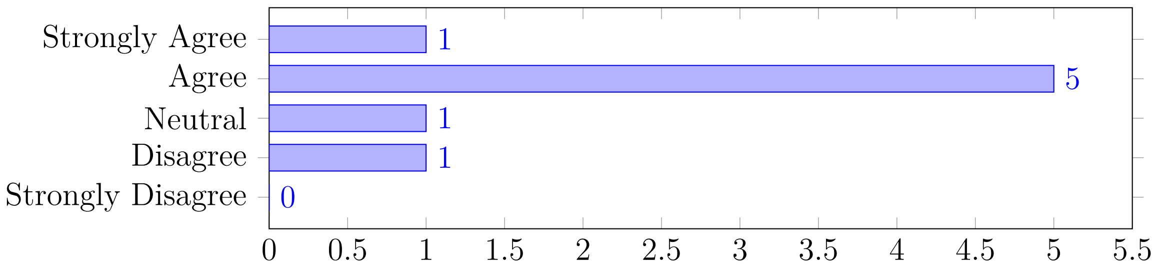


Figure 5.5: I think the feature-to-folder view was useful in tracing features to folders

5.2.3 Feature tangling view

One subject had a very positive comment about the view, which can be found below while another stated that it provided a good overview but was not useful for specifics.

> *I really like the idea of visualizing feature entanglement this way! From the example I find it fairly clear and legible. [...]*

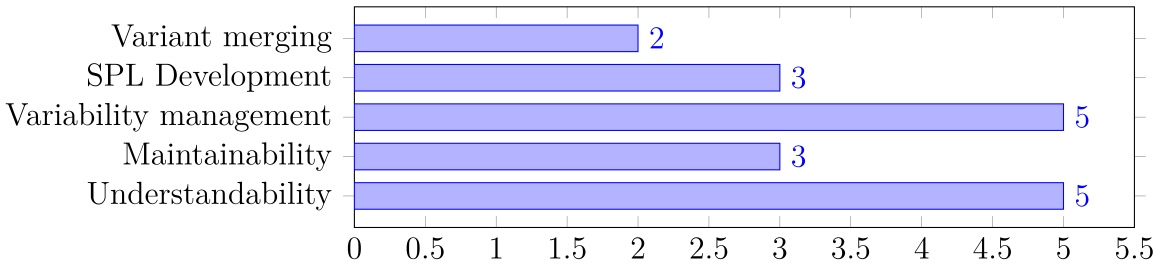


Figure 5.6: I think the feature-to-folder view will be helpful in the following aspects

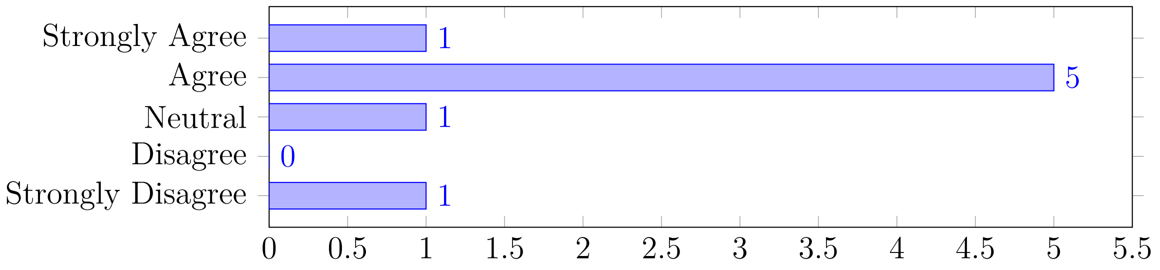


Figure 5.7: The Feature tangling view was effective in showing which features are tangled together

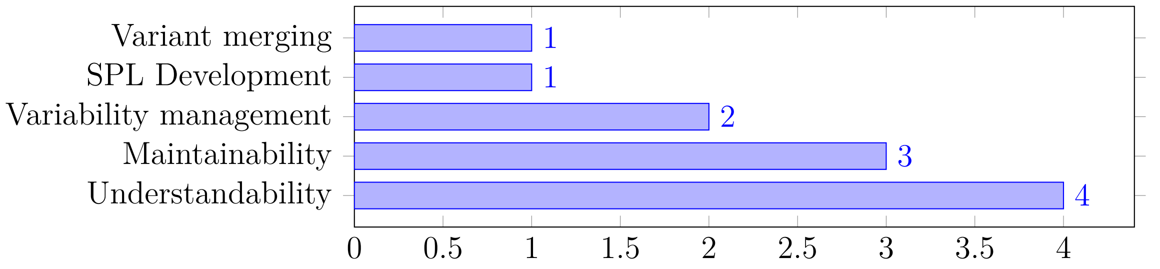


Figure 5.8: I think the feature tangling view will be helpful in the following aspects

5.2.4 Common features view

One of the comments pointed out that this view would not be useful if a subject does not perform variability management.

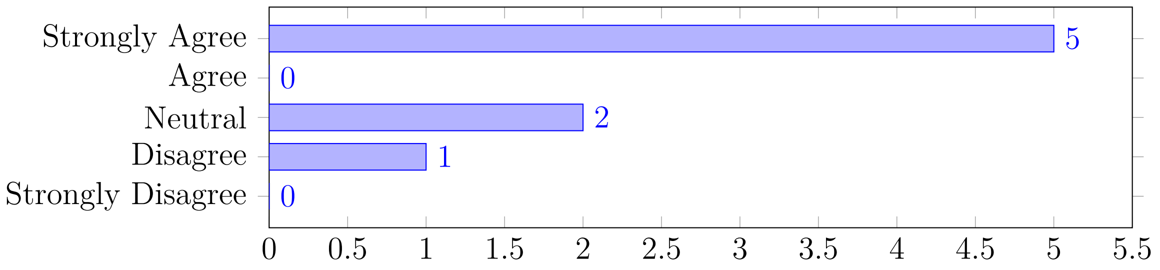


Figure 5.9: The Common Features view showing common features between projects/variants was useful

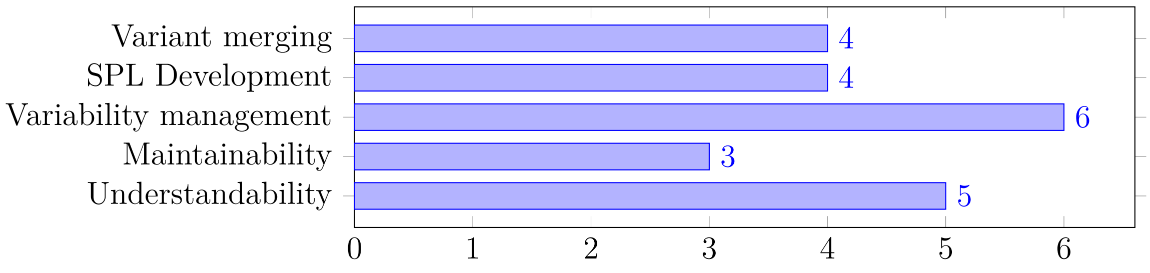


Figure 5.10: I think Common Features view will be helpful in the following aspects

5.2.5 Metrics view

Comments pointed out that there should have been further explanation of the metrics and that it was not appreciated that there were only numbers in the view and no lines indicating the different rows.

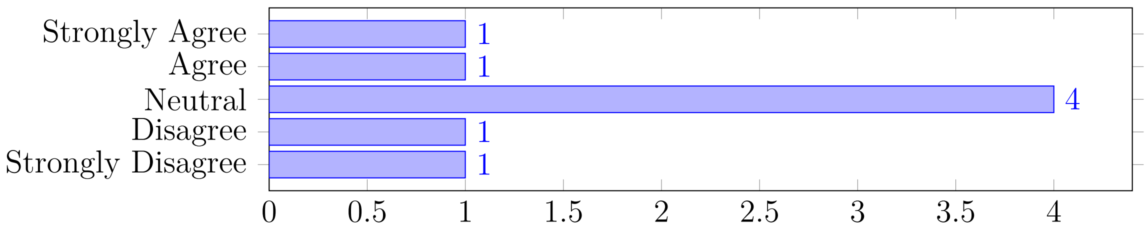


Figure 5.11: I thought that the metrics offered on a Feature-level provided useful information

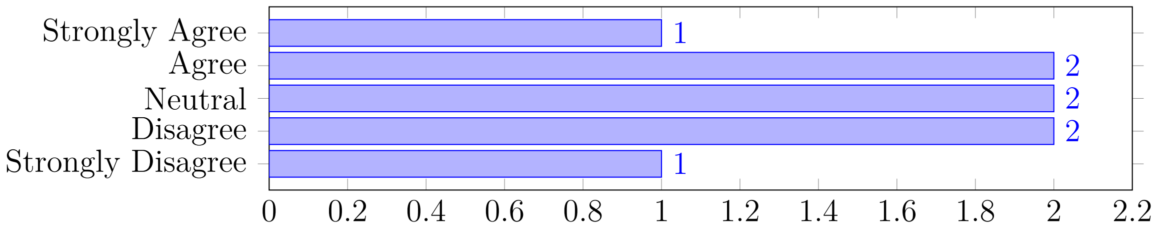


Figure 5.12: I thought that the metrics offered on a Project-level provided useful information

5.2.6 General

At the end of the questionnaire, subjects were asked if any views or functionality was missing, but there was no feedback in this regard. Subjects were also asked if they had previous experience with any other feature-location tool and the advantages/disadvantages regarding it. This was to understand if anything can be learned or avoided from already established tools. Four subjects indicated that they had previous experience with FeatureIDE and had the following to say.

Easy to track features in the code

best tool in its domain that i found, but buggy and resource-intensive

FeatureIDE is quite mature and fully formed

Subjects were also asked which views are essential for feature-traceability. Each subject answered that they thought a view where features are traced to files is needed. A view which helps to trace features to folders received six votes, five thinks a view is needed of how features are tangled with each other, a view which lets a user see common features between projects is required by four subjects, and lastly, two subjects think a metrics view is needed.

Additionally, in questions about the usability of the tool were included in the questionnaire. These questions were in line with the SUS usability assessment which is a 'quick and dirty' way of assessing the usability of the tool [31]. For each question there are five different possible answers, ranging from 'Strongly agree' (SA) which is a score of five, to 'Strongly disagree' (SD) which is a score of one [31]. The following questions are a part of the usability assessment.

1. I think that I would like to use the tool frequently
2. I found the tool unnecessarily complex
3. I thought the tool was easy to use
4. I think I would need the support of a technical person to be able to use this tool
5. I found the various functions in this tool well integrated
6. I thought there was too much inconsistency in this system
7. I would imagine most people would learn to use this tool very quickly
8. I found the tool very cumbersome to use
9. I felt very confident using the tool
10. I needed to learn a lot before I could get going with the tool

[31].

Question \ Answer	SA	A	N	D	SD
1		2	2	2	1
2		1		4	2
3		4	2	1	
4				2	5
5		4	2	1	
6			4	1	2
7	1	3	2	1	
8		1	1	4	1
9		1	6		
10	1		1	3	2

Table 5.1: Subjects' opinion regarding the usability of Feature Dashboard

The following is done to calculate the score: for each odd question the score is subtracted by one, for each even-numbered question the score is subtracted from five, then the total is multiplied by 2,5 and when this is done for each subject, an average is calculated [31]. Using the values in Table 5.1, the calculated score for Feature Dashboard is 65,71.

5.3 Scalability

At this point, the usability of both views and tool have been evaluated. In this section, a controlled experiment is presented where the scalability of the views is tested from a performance perspective. The views that were tested are the same as were being evaluated in the second usability evaluation.

As the plugin is written in Java, to get precise measurements of the initialization time, the standard Java library method `System.currentTimeMillis()` was used. When this method is called, a timestamp in milliseconds is returned which allows measuring the time elapsed between two points. As such, all the numbers displayed in tables is the time elapsed in milliseconds, except for one exception.

The value for the independent variable was in all cases (except for Feature Dashboard view and metrics view), decided by increasing it until the tool either was unresponsive, crashed or ran out of memory. Then, the value was decreased until a stable execution could be performed and two additional decrements were made to the independent variable in order to provide additional coverage.

These tests were performed on a virtual machine running Microsoft Windows 10. The virtual machine had access to two processor cores which are clocked at a 2GHz base frequency, with a boost frequency of 3.10GHz and a total of 4GB of RAM.

5.3.1 Feature Dashboard View

As can be seen in Table 5.2, the independent variable is the total lines that were searched (each line was 50 characters long unless an annotation was specified on it). Note that during these tests, there were 12,000 annotations found across the lines searched to simulate a real-world project. The reasoning behind this number originates from the paper by Andam et al. [11], in which, an automated feature-location tool is suggested that places the same feature annotations in the code as is read by Feature Dashboard. Executing the tool on a project with 3,200,000 lines of code, just over 6,000 annotations were placed in source files [11]. Assuming that feature-location tools have low accuracy (see Section 2.5.2), the double amount of annotations were considered in this test.

Lines	Run 1	Run 2	Run 3	Run 4	Run 5	Average
250,000	6678	6420	6218	6673	8924	6983
500,000	19522	19473	19076	18576	19100	19149
1,000,000	66412	67927	66879	69239	67497	67591

Table 5.2: Measurements for feature dashboard view in milliseconds

5.3.2 Feature-file

For the Feature-to-File view (see Section 4.3.3.2), there were two tests performed. One test to measure the time it takes to view all files that belong to a feature by clicking on the nested node connected to a feature. For this test, the independent variable was the number of files belonging to a feature. The time measured, was when a user clicks to see the files belonging to a feature and when all files are displayed. For this test, it was not possible to measure the time programmatically, since the implementation for this functionality is a part of the visualization library and thus not accessible. As such, the time was measured in seconds using a stopwatch on a mobile phone. The results can be found in Table 5.3. The second test measured the time it takes to view two features if they are tangled together in many files. As such, the independent variable was the number of files two features are tangled in. The result for the second test can be found in Table 5.4.

Files	Run 1	Run 2	Run 3	Run 4	Run 5	Average
500	∼6	∼12	∼6	∼4	∼7	7
1000	∼9	∼11	∼7	∼9	∼9	9
1500	∼12	∼14	∼12	∼12	∼10	12

Table 5.3: Measurements for viewing files implementing a feature in seconds

Tangled files	Run 1	Run 2	Run 3	Run 4	Run 5	Average
250	5622	6835	5785	5529	6339	6022
500	16089	13339	12316	12318	12382	13288
750	18407	20922	22329	18488	21001	20229

Table 5.4: Measurements for viewing features if tangled in milliseconds

5.3.3 Feature-to-Folder

The Feature-to-Folder view (see Section 4.3.3.3) visualizes the folder structure and the folders in which a feature is located in, as such, the independent variable was the number of folders that have to be visualized when viewing a feature. The results can be inspected in Table 5.5.

Folders	Run 1	Run 2	Run 3	Run 4	Run 5	Average
250	4541	2718	2132	3141	1665	2839
500	8569	5084	4099	4373	4187	5262
1000	12251	12361	11348	10610	12035	11721

Table 5.5: Measurements for feature-to-folder in milliseconds

5.3.4 Feature tangling view

This view (see Section 4.3.3.5) visualizes features that are tangled with another feature. As such, the independent variable is the number of features tangled together in a file. The measurements can be inspected in Table 5.6.

Tangled features	Run 1	Run 2	Run 3	Run 4	Run 5	Average
250	5410	3539	3581	3525	3532	3917
500	16089	13339	12316	12318	12382	13288
750	21898	21775	25972	23834	22665	23229

Table 5.6: Measurements for feature tangling view in milliseconds

5.3.5 Common features

By analyzing this view (see 4.3.3.6), a user can find which features are shared between two or more projects. For this test, three different projects were created that each shared a number of features (which is the independent variable). The results in Table 5.7 shows the initialization time to view the common features when the third project is selected.

Common features	Run 1	Run 2	Run 3	Run 4	Run 5	Average
1000	1845	1810	1836	1846	1886	1844
2000	5510	6193	5125	5146	5153	5425
3000	8588	7293	7341	7319	7422	7593

Table 5.7: Measurements for common features view in milliseconds

5.3.6 Metrics view

This view allows a user to view metrics for features and feature metrics for resources. In order to simulate a project, there were 100 folders and 250 files created for which metrics would have to be calculated. The independent variable was the number of features per file. To view the results, see Table 5.8.

Features per File	Run 1	Run 2	Run 3	Run 4	Run 5	Average
25	2525	1428	1117	1121	1273	1504
50	7809	5211	5713	4392	5645	5754
100	29788	34290	28652	35132	29625	31497

Table 5.8: Measurements for viewing metrics in milliseconds

6
Discussion

In this chapter, different parts of the thesis are discussed.

6.1 Result

The main point of discussion regarding the result is the absence of a view that allows the user to inspect the evolution of a feature. Still, there was a view created (see Figure 6.1), which shows how metrics have changed over time for a feature. However, due to time constraints, the process in which information is gathered for

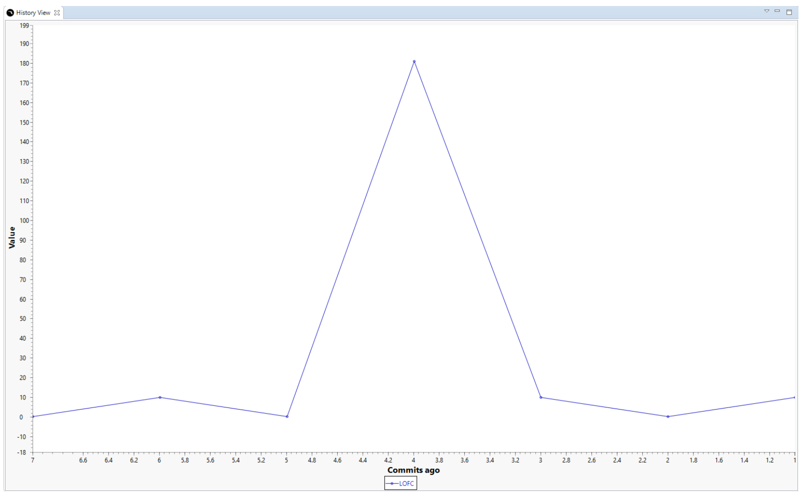


Figure 6.1: How a feature's LOFC has changed over commits.

a feature could not be made to work consistently. Therefore, the view is still under development and not available to users. One factor that is thought to contribute to the inconsistency is that Git is integrated into Eclipse. As such, Eclipse will automatically perform Git commands as soon as a change is detected. Which commands executed are not know and therefore it is hard to determine if it affects the process negatively or not. Regardless of what might affect the result, more work is required.

As detailed in Section 4.2.3, agile is one of the most popular workflows used by software development teams today. Agile workflows encourage small commits and therefore, a lot of commits have to be inspected. An observation made during development was that it takes significant time to *checkout* each commit. For large-scale projects, it would be a time-consuming task. To mitigate this issue, users could specify how many commits back (to an upper limit) they are interested in.

However, looking at just a few commits has proven to be a time-consuming process. Another problem with performing checkout on each commit is that an `index.lock` file will be created. This indicates that a process is currently modifying the `index` file and therefore other processes are blocked from doing similar tasks, this includes the developer as well.

6.2 Evaluation

In this section, the evaluations that were performed during the thesis are discussed.

6.2.1 Interviews

These interviews were a success, as they produced useful information which resulted in new views and improvements to existing ones. An interesting observation is that the first concepts created for Feature-to-File (see Section 4.2.1), offers what subjects thought were missing from the evaluated view. Namely, the possibility to see if features are tangled with one another and in which file.

Since participating subjects already used a tool based on Eclipse, useful feedback on what to avoid was also received in this regard. One of these points of feedback was when interacting with elements in a view, another view with relevant information should be opened automatically. This inspired the interaction between the Feature Tangling and Feature-to-File views, however, there could be more of these interactions added.

6.2.2 Questionnaire

An interesting observation is that in the first evaluation, one of the views that received the most negative feedback was Feature-to-File. For this evaluation, however, it was one of the most appreciated. Though, there were some problems with going back and forth between the two different 'views' which should be improved further by, for instance, adding a button with can be pressed to go back and forth.

One of the views that were least appreciated was Feature-to-Folder. Mainly it seems to be a problem with how nodes are placed in the view. At the moment, a tree layout algorithm is utilized to position the nodes which do not allow for the separation of nodes. The feature nodes will, as a result, be considered as folder nodes. A straightforward improvement would be to write an algorithm that can make this distinction to position the nodes correctly. There was one comment that would like to, effectively, merge Feature-to-File and Feature-to-Folder. However, if Feature-to-Folder in its current state, was considered too cluttered, then combining these two views would most likely lead to the same feedback. One possible solution is to allow users to double-click a connection between a feature and a folder node. Subsequently, all files located in that folder belonging to the feature would be presented.

Regarding the metrics view, there seem to be two main problems. Firstly, the metrics shown in the view were not explained enough. Which could be fixed by contributing to the built-in help page in Eclipse, offering a more comprehensive explanation of the metrics. Secondly, not including visual aids to track metrics to either features or resources was not appreciated. This can be fixed by, for instance, coloring rows in two alternating colors.

There was one developer that participated in the questionnaire. As such, this feedback was inspected individually to see if there were any differences in the feedback received compared to students. However, there were no significant differences observed. The same was done for the subject that did not use the tool before taking the questionnaire, however, there was no significant difference here either.

There were four subjects that indicated that they had previously used a feature-traceability tool and everyone indicated that this tool was FeatureIDE. Inspecting the comments regarding the perceived advantages and disadvantages of FeatureIDE, it is possible to understand why. Such that it is fully formed, it is easy to trace features in the code, and that it is the best tool in the domain. However, FeatureIDE is extensive and has a collection of multiple different tools and a downside to this is as can be read, it is resource intensive and 'buggy'. Due to these facts, one way that Feature Dashboard can set itself apart from FeatureIDE and steal away users, could be to focus specifically on feature-traceability and keeping the tool as resource inexpensive as possible.

When inspecting the score obtained from the SUS test, it reveals that the usability is considered 'poor' but on the border of being 'good' [32, 33, 34]. Since it was not the scope of this thesis to achieve excellent usability, the result is considered good enough. However, it does show that there is ample room for improvement.

6.2.3 Scalability tests

Since it is expected that this tool will be used a majority of the time when a developer is writing code, initializing views and provide a visualization regarding a specific aspect of a feature cannot take to long. The results for Feature Dashboard view and the metrics view are the views that require the most time to initialize. However, these views are intended to be initialized only once or twice per session. As an example, once the project has been parsed, there is no need to do it again for some time. When it comes to Feature-to-File view, for instance, this is a view that the user will interact with constantly. As such, if the initialization is too long, then it will not be a pleasant experience using it. Inspecting the result for these views, it can be identified that the views in some cases require a moment to initialize. In Section 5.3.1, an example was given of how many annotations were present in a large-scale project. An additional number to keep in mind was that for this project, there were only 43 unique features identified [11]. As such, the results are considered acceptable.

6.3 Threats to validity

In this section, different threats against the different aspects of the thesis are discussed.

6.3.1 Result

Not finding already suggested work when executing the literature survey is a significant threat. These papers could potentially have suggested extremely important lessons or information that could have been useful to know in advance.

Another threat against validity is the number of evaluated design iterations. As can be seen in Chapter 5, two iterations were evaluated. Performing more evaluations allows for more feedback to be considered when designing views. The reasoning as to why only two iterations were evaluated, was due to the back-end development of the plugin required more time than expected.

6.3.2 Interviews

In this section, threats against the internal validity regarding the semi-structured interviews were held are discussed.

6.3.2.1 Internal

One of the variable that was thought of that was going to affect the result of the evaluation was the unfinished tool. As such, subjects did not directly use the tool. The interviewer demonstrated different examples when the subjects were told to focus on the contents in the views instead of how interaction is the tool worked. Therefore, any impact unfinished interactions in the tool would have had on the result was removed. Furthermore, the interviews were recorded and later transcribed, removing the possibility of feedback being misinterpreted or changed when compiled.

Another potential threat regarding the validity, as can be reading the protocol in Appendix A.2, is the leading nature of the questions that were asked which could have affected the feedback that was received. However, this evaluation was used solely as a means to obtain feedback on the views and to start a discussion between the interviewer and interviewee. As such, the threat to validity is considered to be low in this regard.

6.3.3 Questionnaire

In this section, threats against the external and internal validity for the questionnaire tests are discussed.

6.3.3.1 Internal

When evaluating the views for a second time, as with the first evaluation, a possible threat against the internal validity was the interaction with the tool. For instance, if the views received poor feedback, it is not possible to say if the tool had affected that result negatively or positively. Therefore, the SUS test was included in the questionnaire. With the addition of these questions, it was possible to separate the feedback of the tool and views. If the tool received an excellent usability score while the views were received negatively it is possible to say that the tool did not affect the views negatively, for example. Furthermore, subjects would have to think twice if something applies to the views or the tool.

Additionally, subjects received documentation regarding the functionality of the views/tool so that unintended behavior would not be interpreted as intended behavior and vice versa. This is normally not a problem during interviews since there is a constant dialog between interviewer and interviewee, but here it has to be stated explicitly. However, if subjects do not read the documentation before using the tool then there is nothing to stop this.

Even though any potential subject was heavily urged to use the tool, in the event where it would not be possible it was still allowed to provide feedback. It was permitted since in the documentation and in the questionnaire, there were examples of all views which subjects could inspect. Additionally, subjects had to mark if they had used the tool or not in the questionnaire. This enables an analysis of the result afterward to see if a subject who had not used the tool has had a different experience as suppose to treat all answers equally. For instance, it would have to be highlighted if complete opposite feedback had been received from subjects who had or had not used the tool.

6.3.3.2 External

Considering the size of the population, which is developers that would like to get more information about a project in terms of its features, experienced developers and inexperienced developers alike. Eleven subjects that were included during the evaluations are not enough to make any significant claims regarding the generalizability of the entire population. However, increasing the generalizability slightly, is the fact that subjects were using the tool on their own in the second evaluation. With no predetermined example which was followed, the result can be generalized on more scenarios.

6.3.4 Scalability tests

In this section, threats against the external and internal validity for the scalability tests are discussed.

6.3.4.1 Internal

Since the computer on which the tests were executed on had limited performance, the greatest threat against the internal validity is in the event when another background process that consumes a lot of processing power is running. Instead of the tool being able to take advantage of, for instance, 70 percent of the processing power it might now only have access to 30 percent. Since it can not be completely determined if there was something that interfered with the test, five runs were executed for each test and then the average was calculated.

6.3.4.2 External

The results can be generalized onto different projects since it is shown how long the initialization time is, depending on a specific variable. However, this does not apply as much for the metrics view since this view depends on variables that shift drastically from project to project. Moreover, the results can not be generalized on different computer configurations because changing the processing and memory capabilities will affect the results.

6.4 Research questions

The first research question asked was the following.

> *RQ1: What feature-traceability views have been proposed in literature?*

As can be inspected in Section 4.1.1, views that have been previously been proposed in literature are presented. A common visualization technique is to assign colors to features and then different parts of the view are color coded which allows a user to associate the resource with features and, as a result, trace features to their location. Andam et al. [11] suggested a slightly different approach in tracing features to files by utilizing a graph and indicating that a feature is implemented in a file as an edge between two nodes. Antoniol et al. [1] also suggests views that takes advantage of a slightly different approach. The views presented takes advantage of 3D visualizations and allows a user to see how a feature has evolved in the code, for instance.

The second research question asked was the following.

> *RQ2: What information regarding features is required to realize views that provide added value?*

Therefore, for each view that was implemented in Feature Dashboard, an explanation regarding what type of information that is required in order to visualize them are presented.

- <u>Feature-to-File</u>
  The obvious thing that is required, is the ability to in some way associate a feature with a file. The view also offers the possibility to allow the user to view if two or more selected features are implemented in the same file. In order to visualize this, all of the locations for all the selected features have to be obtained and then it also has to be known for each location, which files it is implemented in. When double-clicking a file node, an editor will be opened and the relevant feature code will be highlighted. For this, it is necessary to save, for each location, the start and end line of a block annotation or just the line of a single line annotation. Furthermore, if a feature is mapped to an entire file, either through a `.feature-file` or `.feature-folder` file, the amount of lines for that file needs to be known to highlight the entire file. To open the file in the editor, a direct reference to that file object in Eclipse also has to be stored for each location.

- <u>Feature-to-Folder</u>
  When a feature is selected, all folders that either contain files that belong to a certain feature or if an entire folder has been mapped to a feature needs to

be visualized. This requires all the locations for a specific feature to be known and the parent folder all the way to the project root. When this is known, it is possible to display the view.

- Feature metrics
  In order to visualize this table and to calculate the different metrics shown in the view, a number of various types of information regarding each feature are required, such as: amount of features in the same file, the amount of annotations for a feature, how many `.feature-file/folder` files are used to associate a feature to either a folder/file, at which depth is the file located in the folder structure, and what depth is the feature annotation nested at.

- Feature tangling
  In order to visualize this view, all feature locations for all features in the project has to be known. Then, examine whether these features are implemented in the same files as the selected feature. If this is the case, then there should be a connection visualized between the two nodes. In this view, it is also possible to double-click a connection and then Feature-to-File and Feature-to-Folder view will be opened. As such, the information required by those two views needs to be stored as well.

- Common features
  Here, all features for two or more projects have to be known, which allows determining all unique features. Then, check if a feature is present in a project or not.

The last and third research question was the following.

> *RQ3: Do the implemented views support feature-traceability and other feature related problems?*

In order to answer this question, subjects either used the tool or inspected examples of the views provided in Feature Dashboard and then answered a questionnaire. The questionnaire allowed subjects to specify to which extent they agreed with a specific statement. Furthermore, it was also possible to indicate whether or not it was perceived that the views helped with other feature related problems such as understandability, maintainability and variant merging. Before discussing the result for each view, remember that feature-traceability is defined as the ability to trace features and the different aspects of it in code, such as location, size, and dependencies. For Feature-to-file, all subjects either agreed and strongly agreed that the view helped tracing features to files and it was indicated that it helped with other aspects, with understandability being the most common answer which all except for one agreed with. Feature-to-Folder was also received positively, except for two subjects that did not agree that it was useful in tracing feature to folders. While subjects indicated that the view was helpful in a lot of other aspects, it was not as positively received as Feature-to-File. Similarly to Feature-to-Folder, six subjects either agreed or strongly agreed that feature tangling view was effective in showing which features are tangled together. There were also subjects who indicated that this view would be helpful in other aspects. For common features view, five subjects strongly agree that the view was useful. Subjects also thought that this view helped them regarding other feature related problems. Lastly, the metrics view was not received as positively as the other views, however, there were still subjects that either agreed or strongly agreed that the metrics were useful. To then answer the question, yes, the views implemented support feature-traceability and other feature related problems based on the result gathered from the questionnaire.

7 Conclusion

In this thesis, a tool is presented which proposes views that aim to improve feature-traceability. *Feature-to-File* allows the users to trace features to files in which they are implemented in, *Feature-to-Folder* allows the user to see where a feature is located in the folder structure of a project, *Feature Tangling* allows the user to identify features that are implemented in the same file(s) as another feature, *Common Features* allows the user to compare projects with each other and spot which features are shared between them, and *Metrics View* allows the user to view metrics about each feature in a project and also inspect feature metrics for each folder/file.

The result was achieved by first performing a literature survey, investigating the state-of-the-art in feature-traceability research. The findings were analyzed and there were identified issues with the suggested approaches. These issues included: an over-dependence of colors, providing poor overview of the project, and visual bloat. With the result of the literature survey in mind, concept views were created which served as a creative process but also an inspiration for future development. Afterward, by utilizing a design science methodology, the aforementioned views were conceived. However, firstly, the foundations for an open source plugin for the IDE Eclipse called *Feature Dashboard* were created. Subsequently, appropriate views were realized in this plugin which allows the user to import a project and analyze features located in the project. As Feature Dashboard is open source, it offers a platform for future researchers to implement new and interesting views or to compare the results of Feature Dashboard against other implementations, for instance. Furthermore, it can easily be adopted by a developer who could either use the tool or contribute to the development.

At the end of the thesis, Feature Dashboard was evaluated, and it shows, while the interactions in the tool need improvement, that the views proposed do help users with feature-traceability, understandability, maintainability, and other feature management related tasks. Additionally, scalability tests were performed which provides benchmarks for the initialization time of the views.

7.1 Future Work

Feature Dashboard is, at the moment, only provided as a plugin in Eclipse. It is intended to offer Feature Dashboard as a lightweight Eclipse standalone, which only contains Feature Dashboard and other essential features such as an editor. The idea is that users who are not using Eclipse as their main IDE will be incentivized to use the tool without having to download Eclipse in its entirety. Furthermore, if someone is interested in the metrics that Feature Dashboard can calculate for a project, but does not want to use the views, it is also intended to extract this part of the plugin as a command line tool.

In Section 5.3, a scalability evaluation was performed where the initialization time of the views in the plugin was tested. However, as the input scales, are the views still *usable*? This is an entirely different question that also needs to be answered.

Continuing to improve views and adding functionality to Feature Dashboard is also a priority. Improvements to the views include taking more advantage of more visual variables like size, brightness, texture, orientation, and shape [17], among others. An example where this can be applied to further enhance the views is in Feature-to-Folder. At the moment, if a feature is located in two separate folders, it is not clear as to how many files belong to this feature in those two folders. In one folder, there can be one file while in the other, there are a dozen, for instance. Using size, it would be possible to change the connections depending on how many files there are in those folders. Additionally, continuous improvement for the history view is needed so data can be collected consistently. Moreover, as detailed in the previous chapter, one potential issue with retrieving history is that it would take a significant amount of time for large projects. Therefore, it would be important to implement this view and evaluate how long users would be willing to wait for the history of a feature to be established.

Bibliography

[1] G. Antoniol, E. Merlo, Y.-G. Guéhéneuc, and H. Sahraoui, "On feature traceability in object oriented programs," in *Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering.* ACM, 2005, pp. 73–78.

[2] W. Ji, T. Berger, M. Antkiewicz, and K. Czarnecki, "Maintaining feature traceability with embedded annotations," in *Proceedings of the 19th International Conference on Software Product Line.* ACM, 2015, pp. 61–70.

[3] 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, no. 11, pp. 1193–1224, 2013.

[4] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki, "What is a feature?: a qualitative study of features in industrial software product lines," in *Proceedings of the 19th International Conference on Software Product Line.* ACM, 2015, pp. 16–25.

[5] P. Shaker, "Feature-oriented requirements modelling," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 365–368. [Online]. Available: http://doi.acm.org/10.1145/1810295.1810394

[6] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Information systems*, vol. 35, no. 6, pp. 615–636, 2010.

[7] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Enhancing clone-and-own with systematic reuse for developing software variants," in *2014 IEEE International Conference on Software Maintenance and Evolution.* IEEE, 2014, pp. 391–400.

[8] 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.* LLC/CRC Press, 2018.

[9] T. M. Shaft and I. Vessey, "The role of cognitive fit in the relationship between software comprehension and modification," *Mis Quarterly*, pp. 29–55, 2006.

[10] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[11] B. Andam, A. Burger, T. Berger, and M. R. Chaudron, "Florida: Feature location dashboard for extracting and visualizing feature traces," in *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems.* ACM, 2017, pp. 100–107.

[12] T. Eisenbarth, R. Koschke, and D. Simon, "Aiding program comprehension by static and dynamic feature analysis," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01).* IEEE Computer Society, 2001, p. 602.

[13] R. Longhurst, "Semi-structured interviews and focus groups," *Key methods in geography*, vol. 3, pp. 143–156, 2003.

[14] D. I. Sjøberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanovic, N.-K. Liborg, and A. C. Rekdal, "A survey of controlled experiments in software engineering," *IEEE transactions on software engineering*, vol. 31, no. 9, pp. 733–753, 2005.

[15] A. Hevner and S. Chatterjee, "Design science research in information systems," in *Design research in information systems.* Springer, 2010, pp. 9–22.

[16] D. Moody, "The "physics" of notations: toward a scientific basis for constructing visual notations in software engineering," *IEEE Transactions on software engineering*, vol. 35, no. 6, pp. 756–779, 2009.

[17] Y. El Ahmar, X. Le Pallec, S. Gérard, and T. Ho-Quang, "Visual variables in uml: a first empirical assessment," in *Human Factors in Modeling*, 2017.

[18] Y. El Ahmar, S. Gérard, C. Dumoulin, and X. Le Pallec, "Enhancing the communication value of uml models with graphical layers," in *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS).* IEEE, 2015, pp. 64–69.

[19] "Revamp2 - round-trip engineering and variability management platform and process," http://www.revamp2-project.eu/, accessed: 2019-03-20.

[20] C. Kästner, S. Trujillo, and S. Apel, "Visualizing software product line variabilities in source code." in *SPLC (2)*, 2008, pp. 303–312.

[21] C. Kastner, T. Thum, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel, "Featureide: A tool framework for feature-oriented software development," in *Proceedings of the 31st International Conference on Software Engineering.* IEEE Computer Society, 2009, pp. 611–614.

[22] J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, T. Leich, and G. Saake, *Mastering Software Variability with FeatureIDE.* Springer, 2017.

[23] M. Stengel, M. Frisch, S. Apel, J. Feigenspan, C. Kastner, and R. Dachselt, "View infinity: a zoomable interface for feature-oriented software development," in *2011 33rd International Conference on Software Engineering (ICSE).* IEEE, 2011, pp. 1031–1033.

[24] S. Apel and D. Beyer, "Feature cohesion in software product lines: an exploratory study," in *2011 33rd International Conference on Software Engineering (ICSE).* IEEE, 2011, pp. 421–430.

[25] J. Feigenspan, M. Papendieck, C. Kästner, M. Frisch, and R. Dachselt, "Featurecommander: colorful# ifdef world." in *SPLC Workshops*, 2011, p. 48.

[26] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake, "Do background colors improve program comprehension in the# ifdef hell?" *Empirical Software Engineering*, vol. 18, no. 4, pp. 699–745, 2013.

[27] "Gef - graphical editing framework," https://www.eclipse.org/gef/, accessed: 2019-05-07.

[28] "Version control systems popularity in 2016," https://rhodecode.com/insights/version-control-systems-2016, accessed: 2019-05-09.

[29] K. Bak, "Clafer: a unified language for class and feature modeling," Technical report, Generative Software Development Lab, Tech. Rep., 2010.

[30] M. Antkiewicz, K. Bak, A. Murashkin, R. Olaechea, J. Hui, and K. Czarnecki, "Clafer tools for product line engineering." in *SPLC Workshops*, 2013, pp. 130–135.

[31] J. Brooke, "Sus-a quick and dirty usability scale," *Usability evaluation in industry*, vol. 189, no. 194, pp. 4–7, 1996.

[32] "System usability scale (sus)," https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html, accessed: 2019-04-28.

[33] H. Alathas, "How to measure product usability with the system usability scale (sus) score," https://uxplanet.org/how-to-measure-product-usability-with-the-system-usability-scale-sus-score-69f3875b858f, accessed: 2019-05-28.

[34] "Measuring and interpreting system usability scale (sus)," https://uiuxtrend.com/measuring-system-usability-scale-sus/, accessed: 2019-05-28.

A

Appendix

In the appendix things that can be good for the reader to know or that provide additional value are listed.

A.1 Views

In this section, examples of the views shown to subjects during the semi-structured interviews are shown.

Figure A.1: Feature-to-File

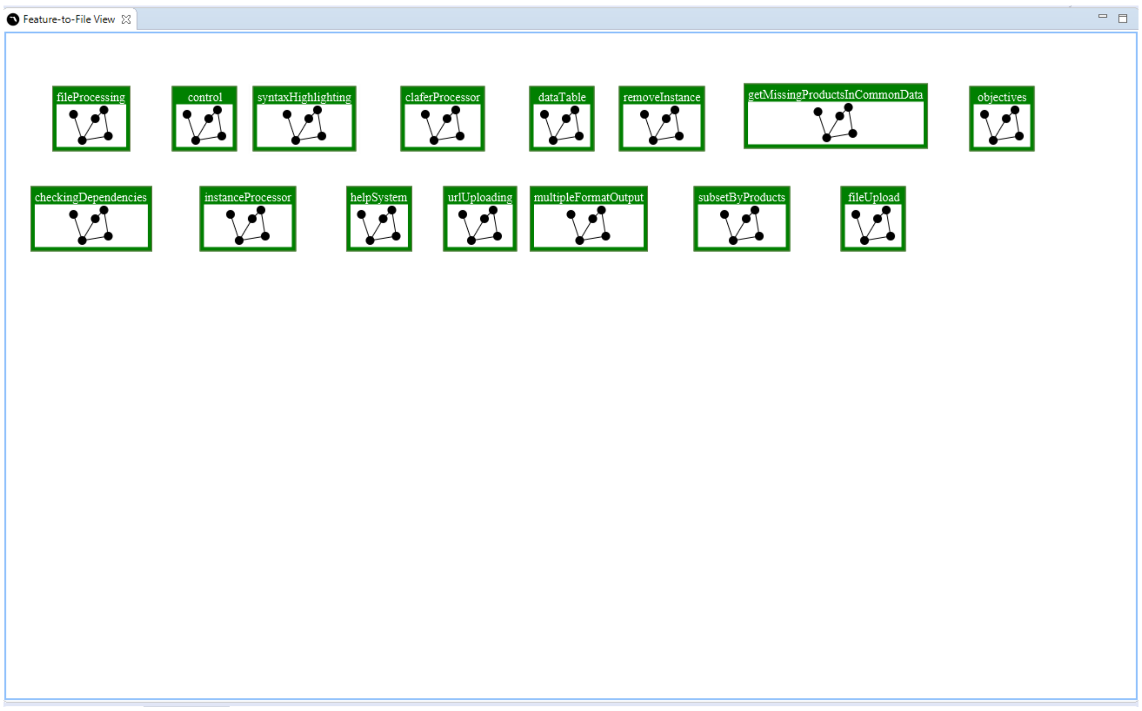


Figure A.2: Feature-to-Folder

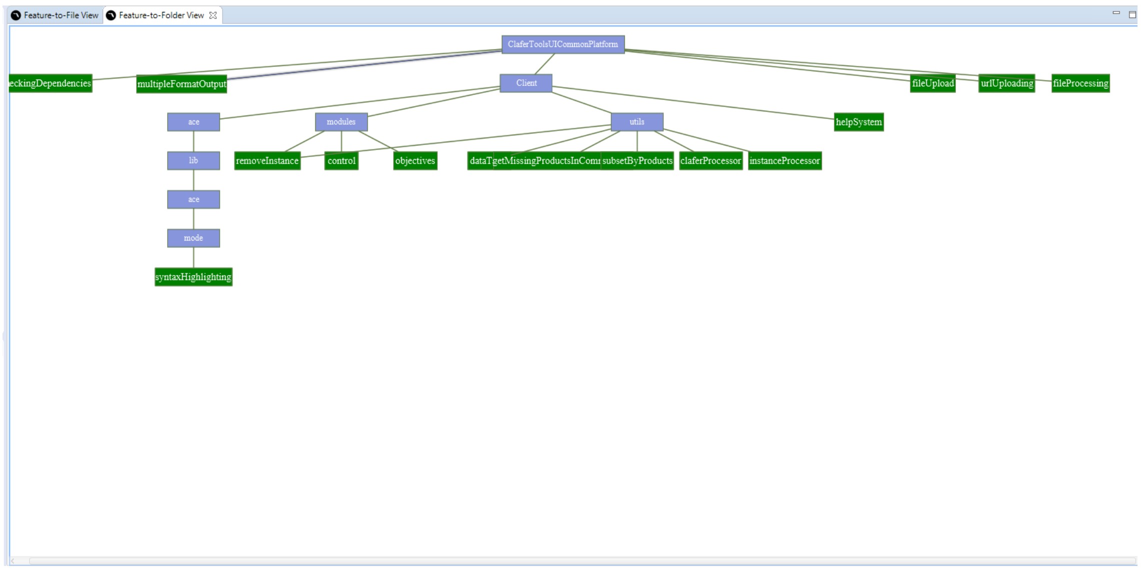
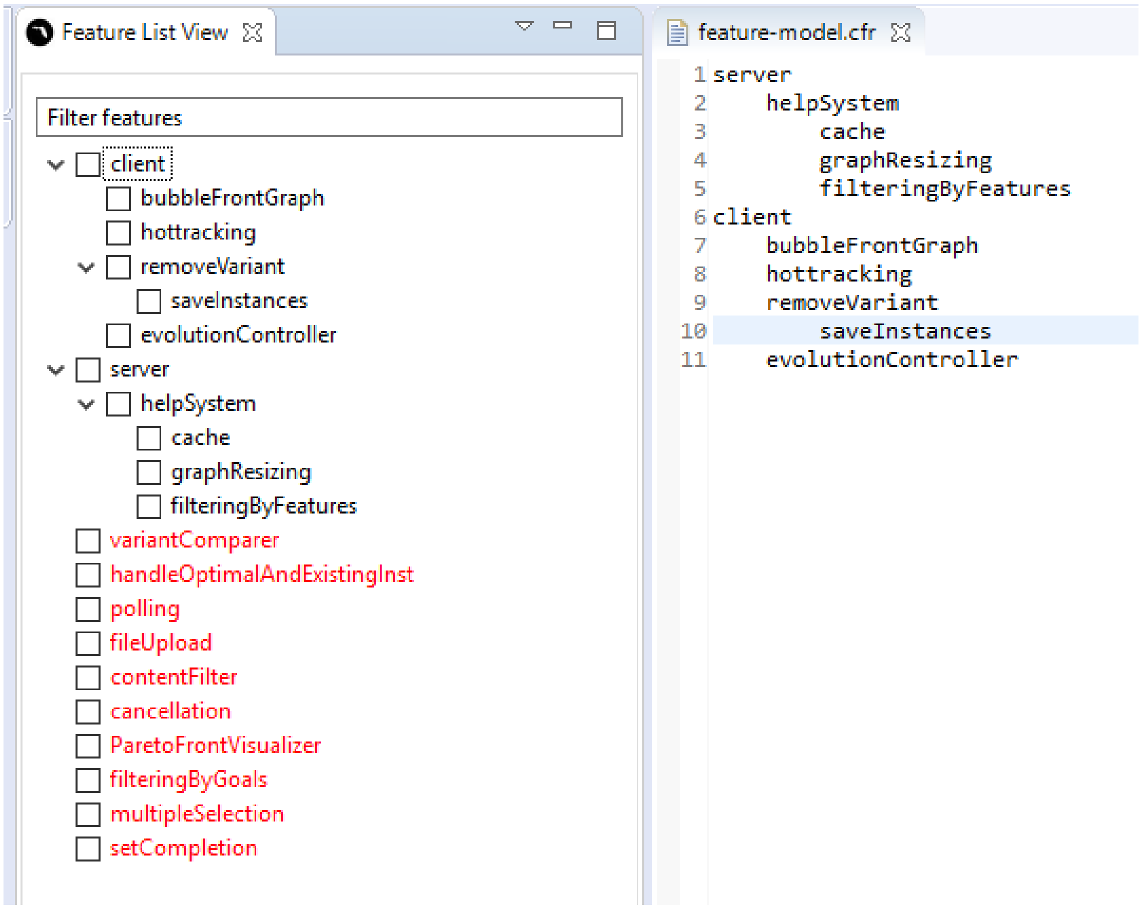


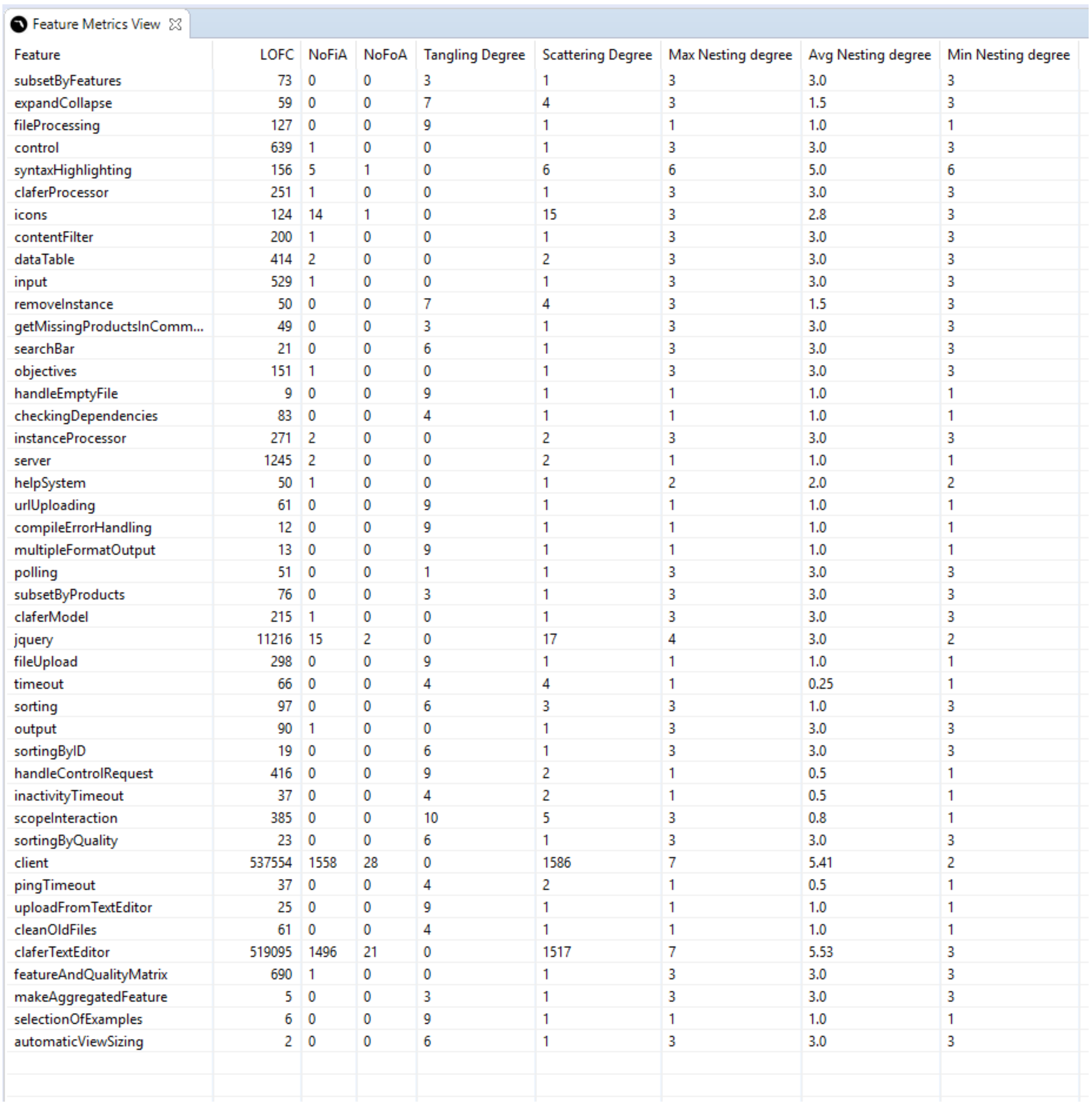
Figure A.3: Feature list



- client
  - bubbleFrontGraph
  - hottracking
  - removeVariant
    - saveInstances
  - evolutionController
- server
  - helpSystem
    - cache
    - graphResizing
    - filteringByFeatures
- variantComparer
- handleOptimalAndExistingInst
- polling
- fileUpload
- contentFilter
- cancellation
- ParetoFrontVisualizer
- filteringByGoals
- multipleSelection
- setCompletion

```
server
    helpSystem
        cache
        graphResizing
        filteringByFeatures
client
    bubbleFrontGraph
    hottracking
    removeVariant
        saveInstances
    evolutionController
```

Figure A.4: Feature metrics


Feature Metrics View

Feature	LOFC	NoFiA	NoFoA	Tangling Degree	Scattering Degree	Max Nesting degree	Avg Nesting degree	Min Nesting degree
subsetByFeatures	73	0	0	3	1	3	3.0	3
expandCollapse	59	0	0	7	4	3	1.5	3
fileProcessing	127	0	0	9	1	1	1.0	1
control	639	1	0	0	1	3	3.0	3
syntaxHighlighting	156	5	1	0	6	6	5.0	6
claferProcessor	251	1	0	0	1	3	3.0	3
icons	124	14	1	0	15	3	2.8	3
contentFilter	200	1	0	0	1	3	3.0	3
dataTable	414	2	0	0	2	3	3.0	3
input	529	1	0	0	1	3	3.0	3
removeInstance	50	0	0	7	4	3	1.5	3
getMissingProductsInComm...	49	0	0	3	1	3	3.0	3
searchBar	21	0	0	6	1	3	3.0	3
objectives	151	1	0	0	1	3	3.0	3
handleEmptyFile	9	0	0	9	1	1	1.0	1
checkingDependencies	83	0	0	4	1	1	1.0	1
instanceProcessor	271	2	0	0	2	3	3.0	3
server	1245	2	0	0	2	1	1.0	1
helpSystem	50	1	0	0	1	2	2.0	2
urlUploading	61	0	0	9	1	1	1.0	1
compileErrorHandling	12	0	0	9	1	1	1.0	1
multipleFormatOutput	13	0	0	9	1	1	1.0	1
polling	51	0	0	1	1	3	3.0	3
subsetByProducts	76	0	0	3	1	3	3.0	3
claferModel	215	1	0	0	1	3	3.0	3
jquery	11216	15	2	0	17	4	3.0	2
fileUpload	298	0	0	9	1	1	1.0	1
timeout	66	0	0	4	4	1	0.25	1
sorting	97	0	0	6	3	3	1.0	3
output	90	1	0	0	1	3	3.0	3
sortingByID	19	0	0	6	1	3	3.0	3
handleControlRequest	416	0	0	9	2	1	0.5	1
inactivityTimeout	37	0	0	4	2	1	0.5	1
scopeInteraction	385	0	0	10	5	3	0.8	1
sortingByQuality	23	0	0	6	1	3	3.0	3
client	537554	1558	28	0	1586	7	5.41	2
pingTimeout	37	0	0	4	2	1	0.5	1
uploadFromTextEditor	25	0	0	9	1	1	1.0	1
cleanOldFiles	61	0	0	4	1	1	1.0	1
claferTextEditor	519095	1496	21	0	1517	7	5.53	3
featureAndQualityMatrix	690	1	0	0	1	3	3.0	3
makeAggregatedFeature	5	0	0	3	1	3	3.0	3
selectionOfExamples	6	0	0	9	1	1	1.0	1
automaticViewSizing	2	0	0	6	1	3	3.0	3

A.2 Evaluation 1 protocol

In this section, the protocol that was used during the semi-structured interview is presented.

Listing A.1: Interview protocol

```
Feature Selection View
Q: What did you think of the Feature list view?

Q: Were there any features missing from this view?

Q: If anything, what would you have changed?
Feature-to-File View
Q: What did you think of the feature-to-file view?
```

```
Q: Was the view successful in enabling
feature-traceability?

Q: What would you like to see changed with the view and why?

Q: Would you also like to have tree-list version?
```

Feature-to-Folder View

```
Q: What did you think of the feature-to-folder view?

Q: Was the view successful in enabling
feature-traceability?

Q: What would you like to see changed with the view and why?
```

Feature Metrics View

```
Q: What did you think of the Feature metrics view?

Q: Did you find these metrics useful?

Q: Is there any metrics that you thought were missing?

Q: Any additional functionality that you were missing?
```

Project Metrics View

```
Q: What did you think of the Project metrics view?

Q: Did you find these metrics useful?

Q: Is there any metrics that you thought were missing?

Q: Any additional functionality that you were missing?
```

History Metrics view

```
Q: Is it important for you to understand how a feature
has evolved over time in terms of the metrics presented
in the Feature/Project metrics view?
```

History Branch/Project View

```
Q: What did you think of this view?

Q: Any functionality that you think is missing?
```

Overall feedback

```
Q: What were your impressions of the tool? Do you think
that it could be useful in your type of workflow?

Q: Were there any views that were missing?

Q: Is this a type of tool that you could see yourself using?

Q: This of the use case that this tool could be used for is
general understandability and maintainability of software.
If you were for instance a new developer and would have to
get to know a codebase better, would you think that this
tool would help you understand the codebase faster?
```

Q: Do you think that the maintainability of a codebase would be improved with a tools such as this?