Informatics

Visualizing Feature Coupling Evolution by Utilizing Source Code Co-Change and Issue Tracking Data

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Sebastian Lukas
Matrikelnummer 01126390

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Thomas Grechenig

Wien, 13. Oktober 2021

Unterschrift Verfasser

Unterschrift Betreuung

Informatics

Visualizing Feature Coupling Evolution by Utilizing Source Code Co-Change and Issue Tracking Data

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Sebastian Lukas
Registration Number 01126390

to the Faculty of Informatics

at the TU Wien

Advisor: Thomas Grechenig

Vienna, 13th October, 2021

Signature Author

Signature Advisor

Visualizing Feature Coupling Evolution by Utilizing Source Code Co-Change and Issue Tracking Data

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Sebastian Lukas
Matrikelnummer 01126390

ausgeführt am
Institut für Information Systems Engineering
Forschungsbereich Business Informatics
Forschungsgruppe Industrielle Software
der Fakultät für Informatik der Technischen Universität Wien

Betreuung: Thomas Grechenig

Wien, 13. Oktober 2021

Erklärung zur Verfassung der Arbeit

Sebastian Lukas

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 13. Oktober 2021

Sebastian Lukas

Kurzfassung

Während der Softwarewartung haben Entwickler/-innen verschiedene Informationsbedürfnisse bezüglich Softwarefeatures. Zum Beispiel müssen sie wissen, wo ein Feature im Quellcode implementiert wurde, welchen Einfluss Featureänderungen auf den Rest der Software haben, oder wie und wann sich ein Feature weiterentwickelt hat. Aufgrund der Größe und Komplexität von Softwareprojekten ist es für Entwickler/-innen eine mühsame und zeitintensive Aufgabe sich dieses Wissen zu erarbeiten.

Diese Arbeit präsentiert einen Visualisierungsansatz, der die Daten von Versionskontrollsystemen und Issue-Tracking-Systemen nutzt, um Entwickler/-innen dabei zu unterstützen, Features im Quellcode zu finden, die Auswirkungen von Featureänderungen zu analysieren und die Featureentwicklung zu verstehen. In dieser Arbeit wurde ein Prototyp einer Visualisierung inkrementell entwickelt und in einer szenariobasierten Expertenevaluierung verwendet, mit der erfahrene Entwickler/-innen Aufgaben im Bereich der Softwarewartung durchgeführt haben.

Mithilfe der Visualisierung konnten die Teilnehmenden Quellcodeteile, wie Dateien oder Methoden, eines Features schnell finden, ohne, dass sie mit der Codebasis vertraut waren. Sie konnten auch jene Codeteile identifizieren, auf die sich eine Featureänderung auswirken könnte. Außerdem konnten die Teilnehmenden die Versionshistorie nutzen um festzustellen, seit wann bestimmte Features miteinander gekoppelt sind. Diese Ergebnisse zeigen, dass die in dieser Arbeit entwickelte Visualisierung Entwickler/-innen bei typischen Wartungsaufgaben, wie der Featurelokalisierung, unterstützt und deren Wissen über die implementierten Features erweitert.

Keywords: *Softwarewartung, Visualisierung, Softwarevolution, Featurelokalisierung, Change Impact Analysis, Prototypentwicklung, Szenariobasierte Expertenevaluierung*

Abstract

During software maintenance, developers have different information needs regarding software features. For example, they need to know where a feature is implemented in the source code, which impact a feature change has on the rest of the software, or how and when the feature has evolved. Due to the size and complexity of software projects, feature comprehension is a cumbersome and time-consuming task during maintenance.

This thesis proposes a visualization approach leveraging the data from Version Control Systems and Issue Tracking Systems to support developers locating features in the source code, analyzing the impact of feature changes and understanding feature evolution. In this work, a prototype of the proposed visualization has been elaborated and was used in scenario-based expert evaluation sessions where experienced developers performed tasks related to software maintenance.

With the help of the visualization, participants could quickly locate feature-related source code entities, like files or methods, without being familiar with the codebase. They could also identify source code entities that a feature change might affect or investigate the history of features to determine, since when certain features have been coupled to each other. These results indicate that this thesis contributes a visualization idea to support developers during typical maintenance tasks like feature location or feature comprehension.

Keywords: *Software Maintenance, Visualization, Software Evolution, Feature Location, Change Impact Analysis, Prototyping, Scenario-based expert evaluation*

Contents

CHAPTER 1

Introduction

During software maintenance, developers often perform modification tasks, like bug fixes or improvements. In order to perform such maintenance tasks efficiently and correctly, developers need a deep understanding of the software and its functionalities. Increasing the comprehensibility of the software, functionalities are represented as logical units called features [1]. Features describe the functionalities of the software from the perspective of application users, developers and maintainers [2]. As maintenance tasks are often related to particular software features, developers have to comprehend how these features are implemented in the source code. However, due to the size and complexity of modern software projects, feature comprehension is a cumbersome and time-consuming task. Therefore, different approaches have evolved to support developers with the challenges they face during feature comprehension. This chapter introduces the reader to the fundamental problem of feature comprehension, and explains the motivation of why existing approaches still can be improved. Further, the aim of the work is explained in Chapter 1.3 and Chapter 1.4 previews the structure of this thesis.

1.1 Problem Description

Feature comprehension is primarily based on locating the source code parts which realize the feature. This is referred to as feature location [1, 3]. Most maintenance tasks require the developer to find all source code parts they have to change, making feature location one of the most essential and common activities. In practice, developers use different strategies to perform feature location, like browsing the source code or searching for specific keywords. However, these strategies are mostly time-consuming and error-prone and depend on the developer's intuition and luck [4]. Therefore, researchers developed other approaches to help maintainers with feature location. These approaches are described in Chapter 2.2.

Another important aspect when performing modification tasks is feature coupling. Usually, software features are not isolated but are coupled to others. This means that the source code of one feature shares some functionality with other features [5]. Developers need to be aware of those couplings because if they change the functionality of one feature, they might otherwise unintentionally change the behavior of other features. However, discovering feature couplings is hard for different reasons. First, developers need to map a feature to concrete source code parts and investigate the coupling of those parts to the rest of the software. Investigating the coupling is closely related to the activity of Change impact analysis (IA), where developers identify all source code parts which have to be modified in order to implement a change request [6]. Secondly, as the maintained software gets further developed, new features are introduced and existing ones are changed or removed. Therefore, also feature couplings change over time.

One source of information developers use to understand how features and their couplings have changed over time is the version control system, which stores all changes made to the source code. For example, developers know that a certain feature worked last week, but now it does not. They then start searching for recent changes to the source code which are related to the task. Next, they investigate these changes in order to find the cause of the bug. However, the version control system stores a large amount of data, and developers often struggle to find all relevant changes for their tasks. This is because the changes stored in the version control system can not always be traced to the features they address due to missing feature references and adverse change descriptions. Another reason is that comprehending software code changes often requires a large amount of mental work because the representation of a change in a source code diff can become complicated depending on the code style and the applied changes. For example, the movement of code lines in a non-formatted code base with no code style guidelines might be harder to understand than introducing new code lines in a well-maintained software project.

1.2 Motivation

To help developers with program comprehension, researchers developed tools that provide a visualization of software evolution. They provide visual resources because they help developers to effectively analyze and understand the large amount of data developers are dealing with during software evolution [7]. Most of today's tools are capable of visualizing the evolution of source code entities like modules, classes or packages, but provide only minimal support for feature comprehension [1]. Although there exist some tools that visualize feature couplings like [8, 2], they are only visualizing the relationship of those features on an abstract level, but not on a source-code level. According to the literature survey for this thesis, there is no tool available that visualizes features and their relationships on a source code level.

This thesis hypothesizes that such a tool could help developers with feature comprehension and decrease maintenance costs. For example, when given a maintenance task, developers could visualize the source code parts of the features related to their task. Further, they see what parts are dependencies on or from other features and propagate the changes there accordingly. This helps developers with feature location and makes them less prone to introducing bugs. In addition, the visualization of feature evolution based on source code can also support developers during bug fixing. This is because the visualization filters a large amount of historical data and only presents relevant changes to the users. Then they can analyze those changes to find the source code parts which contain the bug.

One important aspect of feature coupling evolution is the detection of source code couplings. Researchers have developed different approaches to detect source code couplings. For example, they could leverage the source code history to examine co-changes of source code entities. Co-changed source code entities are entities, like files or modules, which have changed together simultaneously [9]. The idea behind this approach is that entities that have changed together multiple times are probably related to each other. However, this approach and others were not investigated in the context of feature coupling evolution. An open question is how such approaches can also be used for investigating feature coupling evolution.

1.3 Aim of Work

This work proposes a visualization idea that supports developers during software maintenance tasks, like feature location, change impact analysis, and feature evolution comprehension. The work aims to answer whether developers find such a visualization purposeful and what benefits it provides them. Further, this work elaborates the visualization requirements and how they should be implemented to satisfy developers' information needs.

Another goal of this work is to develop an implementation of the proposed visualization. This implementation should utilize the data from VCS and ITS to represent features in the visualization. The visualization should also use different coupling types, like co-changes, to represent the relationships between features and source code entities accordingly.

The prototype is used in scenario-based expert evaluation sessions to assess the visualization and how it supports developers with feature location, change impact analysis and feature comprehension. The visualization should help software maintainers determine where features are implemented in the source code, which features are coupled to each other, and how features and their couplings have evolved.

1.3.1 Research Questions

The thesis should answer the following research questions:

RQ-1: How purposeful do experts rate a visualization concept based on the combination of co-changes from VCS and ITS data regarding

- feature location
- impact analysis
- feature evolution

RQ-2: How to visualize feature coupling evolution by utilizing source code co-change and issue tracking data?

RQ-3: How does the proposed visualization idea support developers with

- locating features
- analyzing feature change impact
- comprehending feature evolution

In order to answer *RQ-3*, the following sub-research questions should be answered.

RQ-3.1: How do developers benefit from visualizing feature-related source code entities for feature location and impact analysis?

RQ-3.2: How do developers benefit from visualizing logical-coupled source code entities for feature location and impact analysis?

RQ-3.3: What are the advantages and drawbacks of different abstraction levels of source code entities for feature location and impact analysis?

RQ-3.4: How does visualizing feature evolution support developers with feature comprehension?

1.3.2 Methodology

The following methods have been used to answer the research questions.

Argumentative research Based on a broad literature review and state-of-art-analysis, argumentative research as described in Galliers [10] has been conducted. This process aimed to elaborate an approach to visualize features in the source code by leveraging the data from Version Control Systems (VCS) and Issue Tracking Systems (ITS). This research process is naturally more unstructured and might be biased by the author's opinion. The developed approach has been evaluated in expert interviews.

Expert interviews To evaluate if the proposed approach is purposeful to developers, expert interviews have been conducted. Participants filled out a questionnaire during these interviews, which was evaluated quantitatively, but participants were also encouraged to explain their answers qualitatively. The gained insights were also used for the requirement engineering of the prototype implementation.

Incremental prototyping The prototype was implemented by using incremental prototyping as described by Graham [11]. During the implementation phase, an entity-relationship diagram and a software architecture diagram were created.

Scenario-based expert evaluation To evaluate the proposed visualization, a scenario-based expert evaluation has been conducted. In these sessions, experienced developers used the prototype to execute some tasks related to feature location, change impact analysis, and feature evolution comprehension. During these sessions, participants filled out a questionnaire, which was evaluated quantitatively. Further, participants were encouraged to also explain their answers in more detail verbally.

1.4 Structure

This chapter will briefly describe the structure of this thesis.

Chapter 1 explains the general problem and describes the motivation and aim of the thesis.

Chapter 2 introduces the fundamentals of feature location, change impact analysis, and visualizations. Additionally, it will explain aspects of Version Control Systems and Issue Tracking Systems, which are essential for this thesis.

Chapter 3 presents similar state-of-the-art systems in the scientific area. It also explains the differences between those tools and the developed approach. Also, some commercial tools are briefly described in the end.

Chapter 4 describes the information needs that developers have during software maintenance and what challenges they usually face when dealing with evolutionary data. The chapter ends with a description of the main requirements the tool should implement.

Chapter 5 explains the technical considerations, architecture and some implementation details of the developed prototype.

Chapter 6 describes how the requirements and prototype have been evaluated. Especially how the evaluation sessions are performed is explained in detail.

Chapter 7 will present the results of the evaluation and Chapter 8 will discuss the gained insights. It will also elaborate on the result's threats to validity.

Finally, Chapter 9 will conclude the thesis and the gained insights. Further, it will describe where future work can build upon.

CHAPTER 2

Fundamentals

This chapter describes the main concepts which will be used throughout this thesis.

2.1 Features and Feature Coupling

According to Fischer et al., a feature is „an observable and relatively closed behavior or characteristic of a (software) part“ [2]. They are a natural unit to describe the software's functionalities from the perspective of users and developers.

When dealing with features, they can be viewed on different levels of abstraction. At the highest level of abstraction, a feature is a description of software functionality. However, in the source code, features are usually implemented throughout different software artifacts like modules, files and methods. These software artifacts describe the other levels of abstraction. Depending on the task at hand, features are viewed on different levels. For example, a software architect might be interested in the modules and packages which implement the feature. However, developers are interested in the concrete methods that implement the misbehaving feature when fixing a bug. Thus every abstraction level serves a different need.

In large and complex software systems, the features are not isolated but are coupled to other features [12]. A so-called feature coupling occurs if multiple features are affected by the same source code. Feature couplings, which are sometimes also referred to as feature dependencies, are essential to be aware of because a change in those affects multiple features.

2.2 Feature Location

Feature location is the task of identifying the source code parts that implement a feature [3]. In practice, feature location can become a difficult task because the maintainer might be new to a project and is not familiar with the source code. The following circumstances often aggravate the situation. The author of the source code is not available anymore, the source code is hard to read or the project's documentation is sparse. Developers then start to use different strategies to locate the feature in the source code, like searching for specific keywords. Because these strategies are time-intensive and error-prone, researchers have developed so-called feature location techniques (FLT). There are many different FLT, which can be mainly differed by their approaches [3]. The most common FLT approaches are:

Static approaches These approaches investigate the control or data flow dependencies of the source code. They use structural coupling, like inheritance and method calls. These approaches are very similar to the way how developers manually examine feature location, as they tend to follow the control flow of source code. Usually, these approaches require the users to provide a source code artifact as a starting point.

Dynamic approaches These approaches examine a software system's execution at runtime. They require the feature to be executable. One of the first dynamic approaches is software reconnaissance [3]. In this approach, the are two sets of execution scenarios. The first set contains scenarios that activate the investigated feature; in the second set, the scenarios do not activate the feature. By analyzing the execution traces of all scenarios, relevant source parts can be identified, while called but feature-unrelated source parts can be eliminated. However, the success of those approaches is mostly dependent on the quality of the execution scenarios. For example, all possible program flow paths must be executed to identify all feature-related source code parts.

Textual approaches These approaches look for words used in the source code and also include comments written by the developers. The idea behind these approaches is that a domain language is used in the source code that makes it possible to search the required feature textually. These approaches use semantic coupling, which "is a measure of how loosely or closely related two software artifacts are, by considering the semantic information embedded in the comments and identifiers" [13]. These techniques can be relatively simple, like pattern matching, or be more advanced like Information Retrieval (IR) or Natural Language Processing (NLP). IR techniques use statistical methods to find relevant code by analyzing identifiers and comments that are similar to the query input provided by the user. NLP techniques are similar to IR and use a query as input, but they analyze the parts of speech of the words used in the source code. The success of those techniques is heavily based on the quality of the user's input query.

Many feature location tools combine multiple approaches to improve their results. Others use the information stored in development tools like the VCS. For example, Chochlov et al. developed ACIR, a tool that utilizes changeset descriptions from Git[1] repositories [14] to partition source code artifacts together. Users can submit a search query related to their feature and receive a ranked list of relevant artifacts.

2.3 Change Impact Analysis

IA (Change Impact Analysis) is the task of analyzing all source code parts affected by a change [3]. This means that when developers change one part of the source code, they also have to change other parts in order to keep the software working correctly. This is referred to as change propagation. In the change process, IA is performed after feature location and uses the result of feature location as a starting point to estimate all source code parts impacted by the change. Methodologically, feature location and IA are different and treated separately in the literature. However, the approaches used to perform IA are very similar to the FLTs [3].

2.4 Types of Coupling

An essential role in many approaches for feature location and IA is the usage of different types of couplings. Coupling is the strength of relationships between two modules or routines [15]. As such, the coupling is also measured to evaluate the quality of software architecture. In order to keep a software architecture maintainable and extensible, the coupling should be kept low. The following gives an overview of the different types of couplings used in scientific research.

2.4.1 Structural Coupling

In the literature, the structural coupling is determined by different metrics that are based on the source code [12]. They can differ in their complexity and the abstraction level on which they work. For example, some of these metrics cover class-to-class interaction, while others use class-method or method-method interactions.

Coupling between Objects (CBO) is a simple object-oriented metric in which two classes are coupled if one class uses methods or fields from the other. The little bit more advanced metric Response for a Class (RFC) considers not only directly called methods but also methods that are invoked indirectly. Information flow-based coupling (ICP) is a structural method that also considers polymorphism. ICP counts the method calls from one class invoked by another, weighted by the number of parameters. Alternative versions of that approach also consider inherited methods.

[1]https://git-scm.com/

```
class F1 {
    public void foo() {
        // method logic
    }
}

class F2 {

    public void bar() {
        new F1().foo();
    }
}
```

Listing 2.1: Example source code to show simple structural coupling

Listing 2.1 shows a simple example of structural coupling. In this example, the method `bar` of class `F2` called the method `foo` from class `F1`. Therefore, there is a structural coupling between `F1` and `F2`.

To calculate structural metrics, the source code needs to be transformed from text into a processable entity, like an Abstract Syntax Tree (AST) or a srcML XML file. An AST is a data structure that represents the syntactic constructs of source code in a tree [16, 17]. Each node in the tree represents a syntactically valid chunk of source code. However, an AST does not contain syntactic elements, which are not required for program analysis. For example, the AST removes unnecessary white spaces in the source code, making processing easier. srcML is an infrastructure that transforms source files into XML files and vica versa [18]. The srcML format wraps the original source code elements into XML tags, which reflect the language elements, like `<if>`, `<function>` or `<class>`. After transforming the source code via srcML to XML it can be queried with XPath in order to calculate the metrics. Listing 2.2 shows a small C++ code snipped, which is transformed to XML with srcML in Listing 2.3.

```
#include <iostream>

// A function
void
f (int x)
{
        std::cout << x + 10;
}
```

Listing 2.2: C++ code example [19]

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<unit xmlns="http://www.srcML.org/srcML/src" xmlns:cpp="http://www.
    srcML.org/srcML/cpp" revision="1.0.0" language="C++"
       filename="example.cpp">
     <cpp:include>#<cpp:directive>include</cpp:directive>
          <cpp:file>&lt;iostream&gt;</cpp:file>
     </cpp:include>
     <comment type="line">// A function</comment>
     <function>
          <type>
               <name>void</name>
          </type>
          <name>f</name>
          <parameter_list>(
               <parameter>
                    <decl>
                         <type>
                              <name>int</name>
                         </type>
                         <name>x</name>
                    </decl>
               </parameter>
               )
          </parameter_list>
          <block>{
               <block_content>
                    <expr_stmt>
                         <expr>
                              <name>
                                   <name>std</name>
                                   <operator>::</operator>
                                   <name>cout</name>
                              </name>
                              <operator>&lt;&lt;</operator>
                              <name>x</name>
                              <operator>+</operator>
                              <literal type="number">10</literal>
                         </expr>
                         ;
                    </expr_stmt>
               </block_content>
               }
          </block>
     </function>
</unit>
```

Listing 2.3: XML after transforming the code via srcML [19]

2.4.2 Logical Coupling

According to D'Ambros et al., „logical couplings are implicit and evolutionary dependencies between artifacts of a system that evolve together, although they are not necessarily structurally related (for example, by means of inheritance, subsystem membership, usage, etc.). They are therefore linked to each other from a development process point of view: logically coupled entities have changed together in the past and are likely to change together in the future" [20]. One advantage of logical coupling over structural coupling is that it can identify hidden dependencies [2, 21]. Hidden dependencies are dependencies between source code parts, which are structurally unrelated. That means they affect each other, but there is no indication in the source code that they belong together, such as a method call or an inheritance relation.

In a co-change analysis, the VCS is mined to determine what source code parts have changed together and how often they have changed. Based on this information, the logical coupling can be calculated. The higher the value is, the stronger the coupling between the source code parts are.

In this thesis, the logical coupling is calculated corresponding to two adapted metrics for association rules of files from Oliva et al., which were initially formalized by Zimmermann et al. [22, 23]. An association rule is an implication in the form of $f_1 \Rightarrow f_2$, where f_1, f_2 are files, and the rule means that if f_1 changes, also f_2 changes. The first metric, the support value of a rule $f_1 \Rightarrow f_2$, is the probability of finding both the antecedent (f_1) and the consequence (f_2) in the same change. Let $c(f_1)$, $c(f_2)$ be the amount of changes containing a modification of f_1, respectively f_2. Accordingly, $c(f_1, f_2)$ is the amount of changes containing a modification of f_1 and f_2. Therefore, the formula to calculate the support value of an association rule is

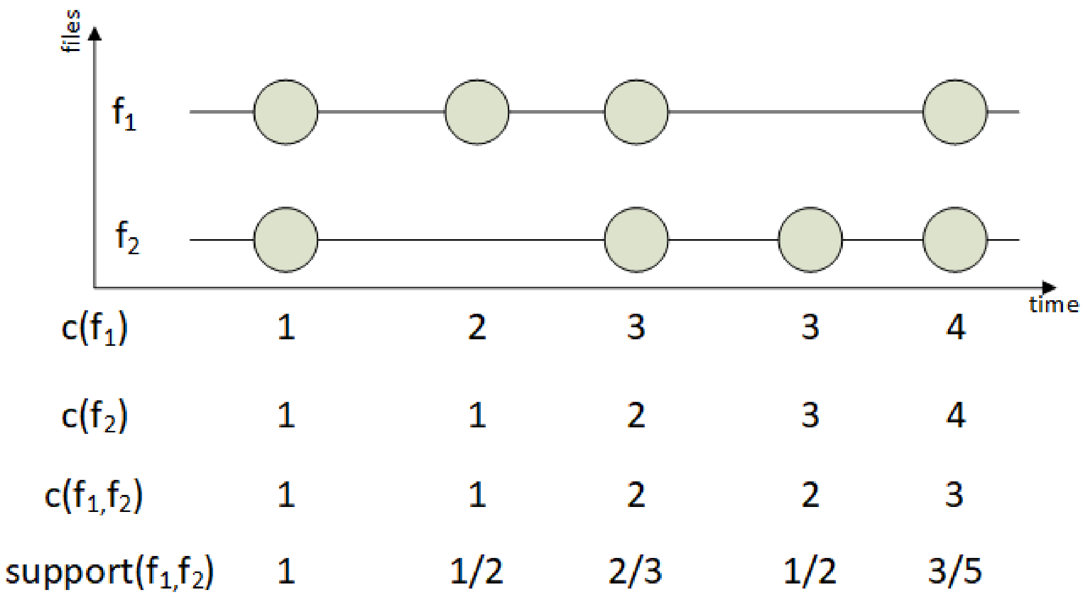
$$support(f_1 \Rightarrow f_2) = \frac{c(f_1, f_2)}{c(f_1) + c(f_2) - c(f_1, f_2)}. \tag{2.1}$$

The support metric is commutative, which means $support(f_1 \Rightarrow f_2) = support(f_2 \Rightarrow f_1)$.

Figure 2.1 shows an example of how the support value of two files is calculated. The light dots on the timeline represent a modification of the corresponding file. If there are two dots above each other, then f_1 and f_2 have been modified in the same change. The values of the formula's variables at that moment are stated below the time bar.

If f_1 and f_2 are always changed together, which means they are completely logical coupled, then $c(f_1) = c(f_2) = c(f_1, f_2)$ and therefore $support(f_1 \Rightarrow f_2) = 1$. On the other side, if they are completely logical uncoupled, then $c(f_1, f_2) = 0$ and $support(f_1 \Rightarrow f_2) = 0$. Therefore, the degree of logical coupling is represented by a value between 0 and 1.

A disadvantage of the formula is that it might miss a dependency relation if one file was changed very often and the other one was not. For example, let $c(f_1) = 1000$, $c(f_2) = 10$ and $c(f_1, f_2) = 10$. Then $support(f_1 \Rightarrow f_2) = \frac{1}{100}$. This value indicates a weak coupling;



c(f₁)	1	2	3	3	4
c(f₂)	1	1	2	3	4
c(f₁,f₂)	1	1	2	2	3
support(f₁,f₂)	1	1/2	2/3	1/2	3/5

Figure 2.1: Example of logical coupling

however, a modification of f_2 will likely cause a modification of f_1 (since f_1 was always modified when f_2 changed), while the opposite is not valid.

However, Oliva et al. also introduced the confidence metric of association rules, which can be used to gain additional insights into the dependency relationship between those files [22]. For example, this means that a modification of f_1 will likely cause a modification of f_2, while a modification of f_2 will not affect f_1. The confidence value of a rule $f_1 \Rightarrow f_2$, calculates the probability of finding the consequence of a rule (f_2), given that the antecedent (f_1) is already existent in a change. The confidence value, which determines how often f_2 changed when f_1 was changed is calculated by

$$confidence(f_1 \Rightarrow f_2) = \frac{c(f_1, f_2)}{c(f_1)} \tag{2.2}$$

In difference to the support metric, the commutative law does not apply to the confidence metric which means that in general $confidence(f_1 \Rightarrow f_2) \neq confidence(f_2 \Rightarrow f_1)$. Considering the example from above, then $confidence(f_1 \Rightarrow f_2) = \frac{1}{100}$, but $confidence(f_2 \Rightarrow f_1) = 1$. These values give more insights into how the two files are related to each other, than the support value.

In the literature, there is no strict definition of logical coupling and even the term itself is often named differently, like co-change, logical dependency or evolutionary dependency [22]. Therefore, there is also no final consensus about the measurement of logical coupling [24]. The most basic metric to determine logical coupling is the absolute amount of changes containing modification of both files, which is $c(f_1, f_2)$. The problem with $c(f_1, f_2)$ is that there can be outliers in the data, which could deform visualizations [25]. D'Ambros solves this problem by dividing the value by the average of the total number of changes of the files, which corresponds to the support metric of association rules from Oliva et al. Another aspect in the calculation of the logical coupling is the time frame, which is used for the calculation [23]. For example, Zimmerman et al. weighted recent changes more important than past ones or ignored all changes older than 180 days. If not stated otherwise, this thesis calculates the logical coupling of two files with the formula of the support metric from Oliva et al. and the calculation is based on all changes, independent of their date.

2.4.3 Semantic Coupling

Semantic coupling, sometimes also referred to as conceptual or textual coupling, is a measurement to determine how closely or loosely two source code parts are coupled by considering the semantic information embedded in the comments and identifiers [26]. Semantic coupling helps to understand the mental model of developers better because they tend to encapsulate the interaction of classes in the vocabulary of the source code. An important aspect of semantic coupling in the scientific literature is that it is used to determine the coupling between source code parts and between source code and features. However, the usability of semantic coupling largely depends on the quality of the domain language used in the source code and the consistent naming of features or concepts.

Like structural coupling, semantic coupling can be calculated by different metrics. Again, these metrics work on different levels of abstraction. For example, Conceptual Similarity between Methods (CSM) computes the similarity between two methods and assigns them a value between 0 and 1 [27]. To calculate the similarity between method m_k and m_j, the cosine between the vectors vm_k and vm_j, corresponding to m_k and m_j in the semantic space constructed by Latent semantic indexing (LSI) is used. Therefore, the formula is

$$CSM(m_k, m_j) = \frac{vm_k^T vm_j}{|vm_k|_2 \times |vm_j|_2} \tag{2.3}$$

Because the cosine can be negative, Poshyvanyk et al. refine the formula with

$$CSM'(m_k, m_j) = \begin{cases} CSM(m_k, m_j) & \text{if } CSM(m_k, m_j) > 0 \\ 0 & \text{if } CSM(m_k, m_j) \leq 0 \end{cases} \tag{2.4}$$

Let there be two methods in a software project, namely getCustomerName ($= m_k$) and getCustomerAddress ($= m_j$). Assume $m_k = \begin{pmatrix} 0.5 \\ 0.75 \\ 0.33 \end{pmatrix}$ and $m_j = \begin{pmatrix} 0.5 \\ 0.75 \\ 0.72 \end{pmatrix}$, which where calculated from the semantic space of the source code of the software project. Then, $vm_k^T vm_j = 0.5 \cdot 0.5 + 0.75 \cdot 0.75 \cdot 0.33 \cdot 0.72 = 1.0501$.

Further $|vm_k|_2 \times |vm_j|_2 = \sqrt{(0.5)^2 + (0.75)^2 + (0.33)^2} \cdot \sqrt{(0.5)^2 + (0.75)^2 + (0.72)^2} = 1.10738$. Therefore, $CSM(m_k, m_j) \approx 0.94$, which means that the methods getCustomerName and getCustomerAddress are semantically closely coupled.

2.4.4 Dynamic Coupling

Dynamic couplings calculate the coupling between two source code parts at runtime [12]. Further, besides the typical abstraction level of analysis like modules, classes, or methods of the other approaches, the dynamic coupling is also applicable to concrete instances of classes [28]. Dynamic coupling is often used to locate features in the code by observing the behavior during the execution.

When computing the dynamic coupling, Arisholm et al. differentiate whether a method executed on an object calls (imports) or is called by (exports) another object's method [28]. For example, the coupling is calculated by counting how many different invocations from method m_s from class c_s to method m_t of class c_t exists. However, there are different granularities. For example, instead of the number of different invocations of a specific method, one is interested in the number of different method invocations from c_s to c_t, or the number of different classes invoked from instances of class c_s in general.

2.5 Development Tools

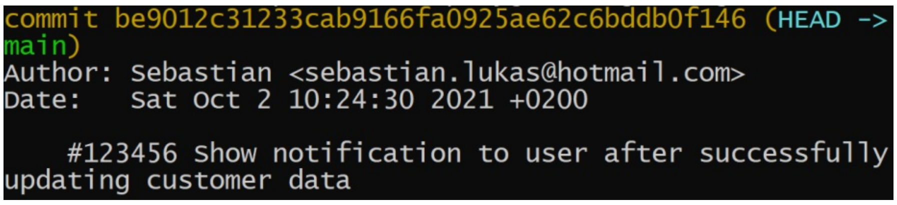
Development tools like the VCS and the ITS not only play an essential role in software maintenance, they also serve as an important source for many feature location and IA approaches. For example, the logical coupling can only be calculated with the information stored in a VCS. The following describes important concepts and features of those tools which are used in this thesis. The described features are not implemented in all existing tools, but many modern and common-used tools implement them.

2.5.1 Version Control System

In a VCS, developers store different versions of source code and project files in a repository [29]. To add a new version to the repository, the users create so-called commits. These commits contain information about the author, the date and time of the commit, a commit message, a hash and a changeset. A changeset is a description of the differences between two versions. By using changesets, developers can recover older versions of the source code. In so-called hunks, changesets contain information about changed files and what lines in those files have been modified.

In VCS developers can work in parallel on different lines of development, so-called branches [29]. With branches, developers can work on new features while keeping the current stable version on another branch. The changes of one branch can later be merged into another branch to apply the changes there as well.

The commit message has an essential role in software maintenance. This message is written by the author of the change and should contain information about its intent. This is necessary to provide context to maintainers who often revisit changes much later. To provide additional context to the commit, many software project teams started to add a reference to an issue from the ITS, which this commit addresses. Figure 2.2 show an exemplary commit message. The rules on how to reference the issue identifier are not standardized but are decided by the project team's compliance. In the example, the issue identifier is referenced at the beginning of the message after the #-prefix. Other projects might have less strict rules which allow referencing the issue anywhere in the message or referencing multiple issues in a single commit message.



```
commit be9012c31233cab9166fa0925ae62c6bddb0f146 (HEAD ->
main)
Author: Sebastian <sebastian.lukas@hotmail.com>
Date:   Sat Oct 2 10:24:30 2021 +0200

    #123456 Show notification to user after successfully
updating customer data
```

Figure 2.2: An example of a commit message with a referenced issue. In this commit message the referenced issue has the ID 123456.

How two versions of a file are different from each other is determined by some differencing algorithm [19]. Currents mechanisms for source code differencing focus either on computing textual differences between files, to which in this thesis is referred to diffs, or by comparing the parse trees generated for the source code. The most popular tools, like the UNIX diff tool, are base on a comparison algorithm called Longest Common Subsequence [30]. This algorithm is relatively efficient and can be applied to any text file, which makes its applicability source code language agnostic. The drawbacks of this approach are that there is no context of the change regarding the source code language, moving source code is not recognized, and changing a single character in one line causes an insert and deletion in the resulting diff. As a consequence of these drawbacks, developers often struggle to understand the diff. To overcome such problems, meta-differencing [19] can be used. In this approach, the source code is transformed into a meta-representation that preserves the textual content of the source code but adds the required abstract syntactic context. By storing the difference between the meta representations of different versions, it is easier for the developer to comprehend the change, e.g., a condition was added to an if-clause).

2.5.2 Issue Tracking System

The ITS contains information about all tasks related to software projects. Usually, an ITS supports the management of different software projects. In the ITS projects, users can create and modify issues. Depending on the complexity of the ITS the issues store a more or less broad range of information. Usually, issues have at least an identifier (issue ID), a title, an author, a creation date, a description, a type and a status. The issue type indicates what the issue is about, such as a bug, a management task, or a new feature. Developers can write comments and update the status during the issue's lifecycle to record the ongoing process. Powerful ITS provide the ability to customize the different types and create custom lifecycles for those types. Thus the lifecycle of a management task might differ from a feature task.

Some ITS also provide the ability to create different relationships between issues. For example, a developer can link the issue of a bug to the issue of the affected feature. By following those relationships, an issue graph can be generated, which is shown in an exemplary way in Figure 2.3. In this example, the parent issue i_1 has two sub-issues, which usually address a specific aspect of the parent issue. By following the path $i_1 \rightarrow i_2 \rightarrow i_5$ i_1 is also related to i_5 although there is no direct connection.

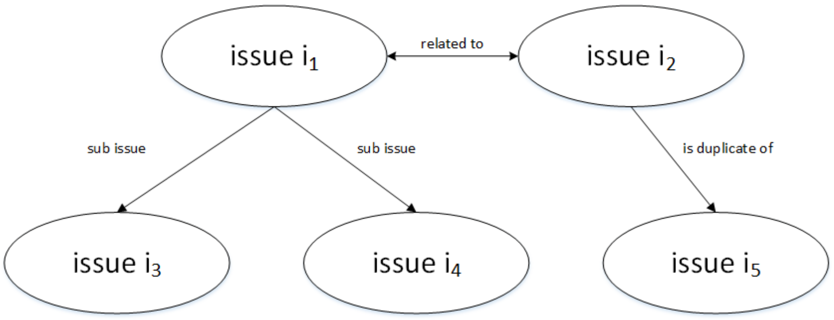


Figure 2.3: Example of an issue relationship graph

Relating issues to features

Modern ITS allows users to manage epics, features, and stories by specifying the ticket type. These tickets can be structured hierarchically; for example, an epic is usually a high-level description of a requirement, while stories and features describe specific details of those requirements in more detail. Sub-tasks can refer to these stories or features, addressing a specific aspect or behavior of a story or feature. By following the relationships between those tickets, a feature can be located in the ITS.

However, the scope and quality of epics, stories and features strongly depend on the software engineering team. Therefore, mapping features to a ticket from an ITS requires defined processes to establish traceability between the feature and the tickets which implement it. Establishing traceability requires the tickets to provide a meaningful description of the expected feature.

Application Lifecycle Management

According to Lacheiner and Ramler, "Application Lifecycle Management (ALM) has been proposed with the objective to provide a comprehensive technical solution for monitoring, controlling and managing software development over the whole application lifecycle" [31].

ALM coordinates activities and artifacts (requirements, source code, test cases) closely related to the software development life cycle and aims to establish traceability between those artifacts [32]. Source Code Management (SCM) is an essential concept for ALM; therefore, the SCM tool builds the foundation of the infrastructure. ALM solutions tend to be complex and require integrating different tools and practices to manage artifacts during the software lifecycle.

ALM tools evolved out of other engineering and management tools, which were used for a broad range of activities, for example, [31]:

- requirement engineering
- design
- implementation
- integration testing
- deployment
- maintenance

There are the different types of ALM tools [32]:

Single-vendor platforms Vendors define the interoperability with the platform and other vendors have to build integrations.

Multi-vendor platforms The framework is developed by an open-source community which is extending the platform.

Single repository Vendors build a complete set of ALM tools using a single repository.

Some examples of commercial ALM tools are codebeamer[2], Azure DevOps Server[3](former Team Foundation Server) and Polarion[4]. Additionally, Atlassian[5] provides services like Jira[6] and Bitbucket[7], which can be combined to build an ALM tool.

2.6 Visualization

When developers investigate the evolution of software, they are confronted by a large amount of heterogeneous data from sources like VCS and ITS [7]. Software evolution visualization helps them to understand the data by providing a graphical representation.

[2]https://intland.com/codebeamer/
[3]https://azure.microsoft.com/de-de/services/devops/server/
[4]https://polarion.plm.automation.siemens.com/
[5]https://www.atlassian.com/de
[6]https://www.atlassian.com/de/software/jira
[7]https://bitbucket.org/

2.6.1 Visual Paradigms

Visual paradigms are used to create the visual scenes presented to the spectator [7]. There are mainly five different approaches:

Pixel-oriented Pixel-oriented techniques map data values to a colored pixel and present values belonging to different attributes in separate windows [33]. These visualizations allow presenting a large amount of data.

Geometric Projection According to Keim et al., techniques related to geometric projections aim at finding useful projection of multidimensional data sets [33]. This includes techniques of exploratory statistics like principal component analysis, factor analysis and multidimensional scaling.

Icon-based The idea behind icon-based visualization techniques is to map each multidimensional data item into an icon. One famous example of the icon-based approach is the Chernoff face visualization [34]. In this approach, the multidimensional data are mapped to properties of faces, like eyes, noses and more. The idea of Chernoff's visualization is that humans can easily recognize differences in faces. Figure 2.4 shows some Chernoff face examples, where each face represents a different data item.

Graph-based The basic idea of the graph-based techniques is to effectively present a large graph using specific layout algorithms, query languages, and abstraction techniques.

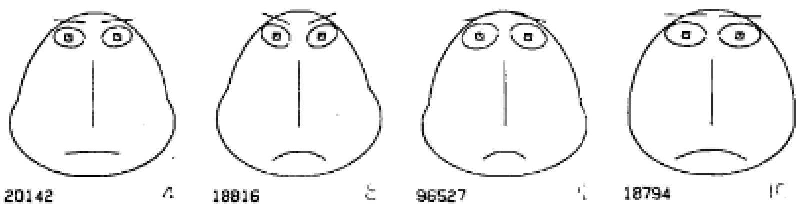


Figure 2.4: Example of Chernoff faces [34]. Each face represents a multidimensional data item.

Hierarchical The hierarchical techniques subdivide the k-dimensional space and hierarchically present the subspaces.

Most of today's software evolution visualizations use the graph-based paradigm, followed by hierarchical and geometric projection [7]. However, many studies in the field of software evolution use multiple techniques to visualize data.

2.6.2 Visual Attributes

Tools use different visual attributes to describe the information in their scenes [7]. Typical visual attributes are form, color, movement or spatial position. They have different effects, depending on the information to visualize and the visual paradigm. For example, color can be used to represent different characteristics of a specific property. However, if there are too many characteristics, many different colors are needed, and users struggle to distinguish them. Therefore, it is important to decide which visual attributes fit best for the data, so the visualized information is easily comprehensible.

2.6.3 Data Sources

Due to the complexity of software evolution, different sources of data are used to visualize [7]. According to Novais et al., most visualizations in the field use SCM systems like CVS, Subversion or Git. They are followed by tools using the source code, bug tracking systems, or a combination of those. Other sources, like documentation and mail data, are only used in very few cases.

2.6.4 Mechanisms of Interaction

Another important aspect of software visualization is the use of interaction mechanisms [7]. Typical mechanisms like zooming, panning and dynamic filtering improve the data analysis and increase user satisfaction. This aspect becomes even more critical when dealing with a large amount of data, which is the case for software evolution.

When dealing with the visualization of evolutionary data, the time aspect plays an important role. Existing tools consider this aspect by either providing a dynamic visualization - a visualization that changes over time - or by using an additional time dimension [35, 36].

CHAPTER 3

State-of-the-Art

This chapter lists and describes the current state-of-the-art from the scientific field of software evolution visualization or feature coupling metrics. The described tools and literature are closely related to this work, either by their purpose or approaches.

3.1 Scientific Works

This section describes related literature in detail and compares them to the approach used in this thesis.

3.1.1 Feature Visualization with CVS and Bugzilla

Fischer et al. [2, 21] linked problem reports from Bugzilla to features and visualized them. The visualization should help developers to reason about future directions of feature implementations and to point out problematic areas. The idea of Fischer et al. is to cluster problem reports related to certain features and find source files that were changed to fix the reported problem. To do so, they mined the modification reports from CVS to find out what changed in order to resolve the problem. Their visualization places problem reports for which the same files have been edited next to each other. Therefore, the user can see which problem reports are related to each other. Figure 3.1 shows the relation of three different features. The nodes (squares, circle triangles) represent problem reports. The form and color of the nodes represent to which feature the report is related.

The red area in the visualization is especially interesting because there are many problem reports from different features. This means that the developers had to modify the same source files to fix the problems related to these features, which indicates a feature coupling. Fischer et al. used Software Reconnaissance to map the abstract concept of feature to concrete source code entities.

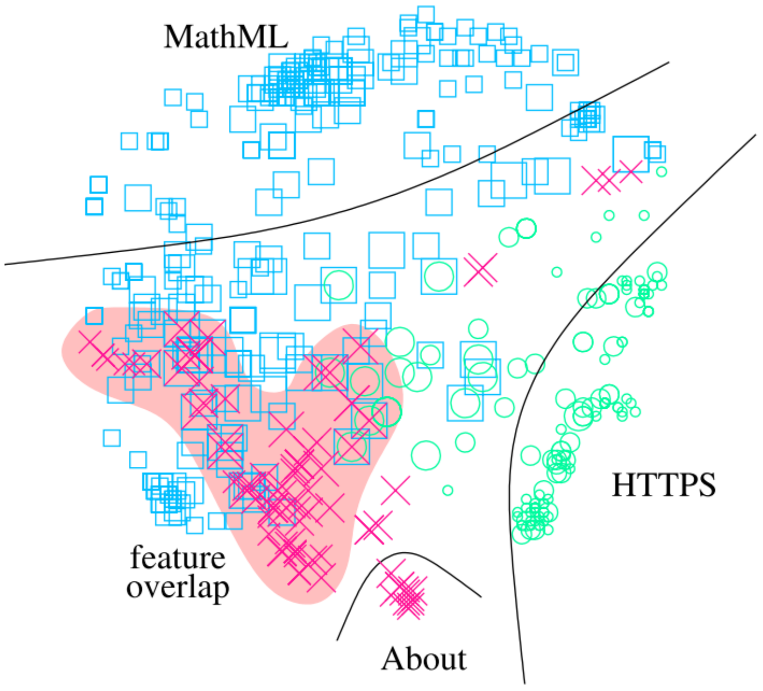


Figure 3.1: Visualization of features from Fischer et al. [2]

One difference between their visualization and the one developed in this thesis is that they visualize problem reports while the developed tool visualizes source code entities like files. Another difference is how files are associated with a feature and how the files' coupling is determined. Fischer et al. use a dynamic analysis approach, while in this work, a combination of different approaches is used to map source code to a feature.

3.1.2 Evolution Radar

D'Ambros et al. developed the Evolution Radar [20]. This tool uses historical information of CVS to inspect the logical coupling relations between files or modules. The tool visualizes the relationship between the source code entities in an interactive radar. Figure 3.2 shows the Evolution Radar. Each blue node represents a file. The file in focus is in the center of the radar. The stronger the coupling between a file and the focused file, the closer the node is to the center. For example, red nodes are strongly coupled with the file in focus because they are positioned very close to it. The evolution radar works on different abstraction levels, but the focus is on modules (architecture level) and the files.

The radar is interactive and allows the user to inspect all visualized entities. For example, users can investigate commit-related data. The radar also allows moving through time and users can select a time frame on which basis the tool calculates the logical coupling of the entities. Further, it is also possible to track files over time to easily see how the

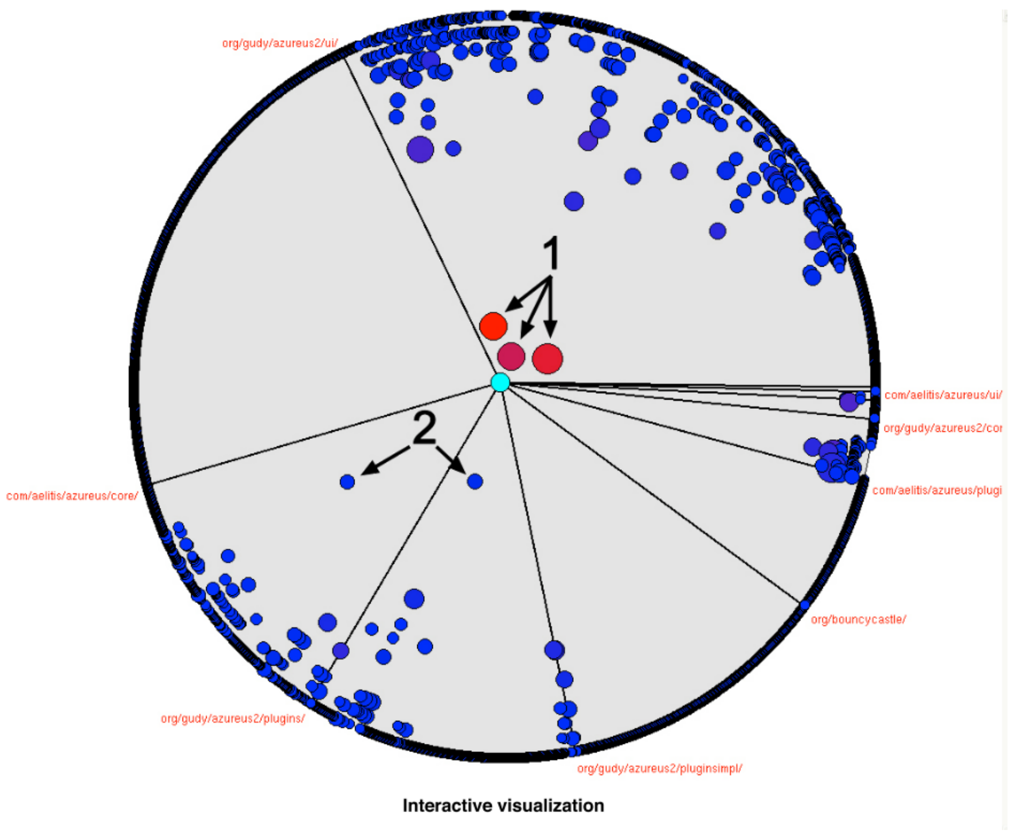


Figure 3.2: Evolution Radar from D'Ambros et al [25]

coupling to the focused file developed.

The most significant difference between the evolution radar and the visualization developed in this thesis is its intent. While the evolution radar focuses on visualizing the coupling of source code entities, this thesis focuses on visualizing features coupling. In this thesis, visualized elements belong to a feature that is extracted from the ITS, while the evolution radar does not investigate abstract features.

3.1.3 Evolution of Features and Feature Dependencies

Steff et al. investigated the role of feature dependencies in release planning [8]. They contributed a new definition of feature dependencies using code and commit-graph analysis. They did a case study on an open-source project on how feature dependencies impact adaptive and corrective maintenance activities.

Steff et al. mine the data of an ITS and a VCS to determine feature dependencies [8]. They determine the logical coupling of change requests from an ITS to define feature dependency. According to their definition, two change requests are logically coupled if there is at least one file that has been changed in commits related to both change requests. Further, they also determine the structural coupling of the changed requests. A structural coupling between two change requests occurs if a class that was modified for one change request has a structural coupling to a file that was modified for the other change request. Based on these definitions, they created two graphs, a semantically coupling graph and a logical coupling graph which contribute to the feature dependency.

Figure 3.3 visualizes a logical coupling graph. The nodes are change requests and their color indicates if it is a feature (green), an improvement (blue), or a bug fix (red). The shape of the node indicates in which version the change request was introduced. An edge between two nodes means that the change requests they represent are coupled.

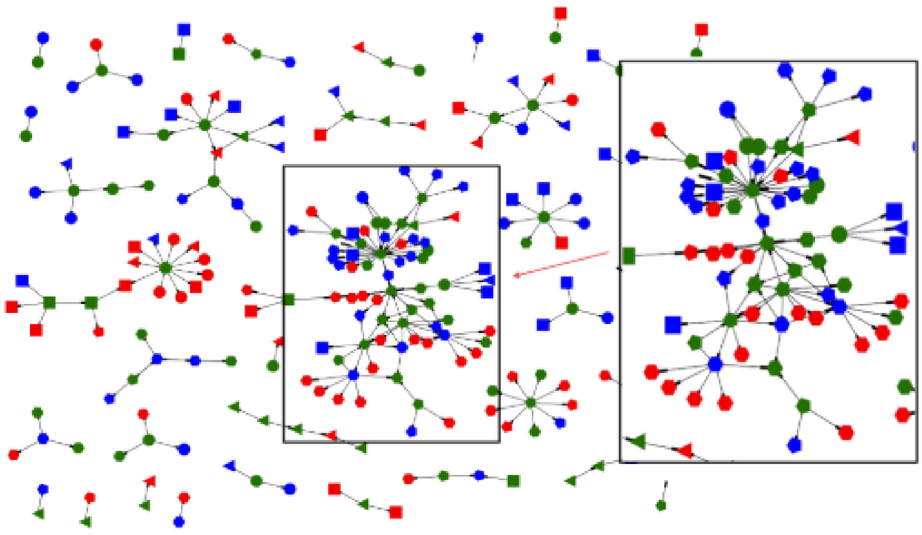


Figure 3.3: Visualization of a logical coupling graph [8]

A difference between the work of Steff et al. and this one is the purpose. Steff et al. determine the coupling of change requests for release planning and therefore, it is of no interest what is causing the coupling, while the purpose of this work is to find out how features are coupled and what source code entities are causing it. Another difference is that their definition of logical/structural coupling considers only absolute values, which means that files/change requests are either coupled or not. However, there is no value indicating how strong the coupling is.

3.1.4 Evolution Storyboards

Beyer et al. [35] developed evolution storyboards. The evolution storyboards visualize a sequence of dependency graphs, which are extracted from the VCS in a dynamic view. The position of nodes depends on their coupling, which means that closely coupled files are located next to each other. A graph, shown in Figure 3.4, visualizes the result. Per default, two different color schemes exist that change the meaning of the color of the nodes. One default color schema indicates the module to which the node belongs. Therefore, the evolution storyboards can not only show which files are tightly coupled but also which modules are coupled. The other color scheme is a heat-based coloring scheme, meaning nodes that moved in many panels are orange, while nodes that rarely move are grey. This scheme helps to identify files that often change their dependencies. Further, the size of a node indicates how often it was changed. The purpose of the visualization is to help developers understanding the evolution of a software system.

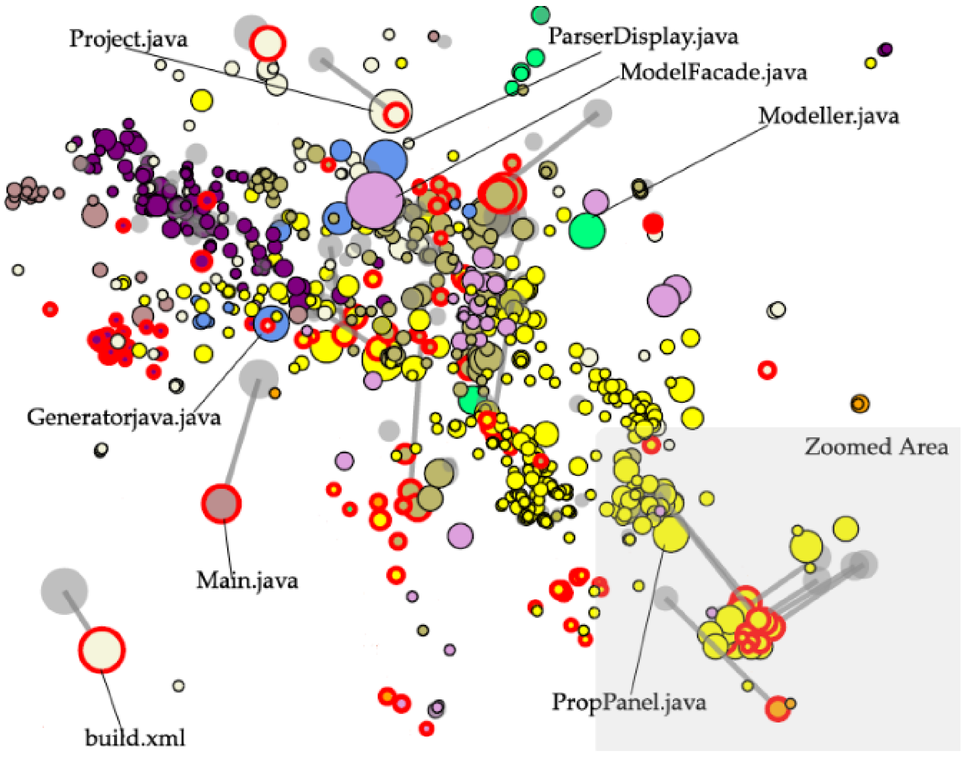
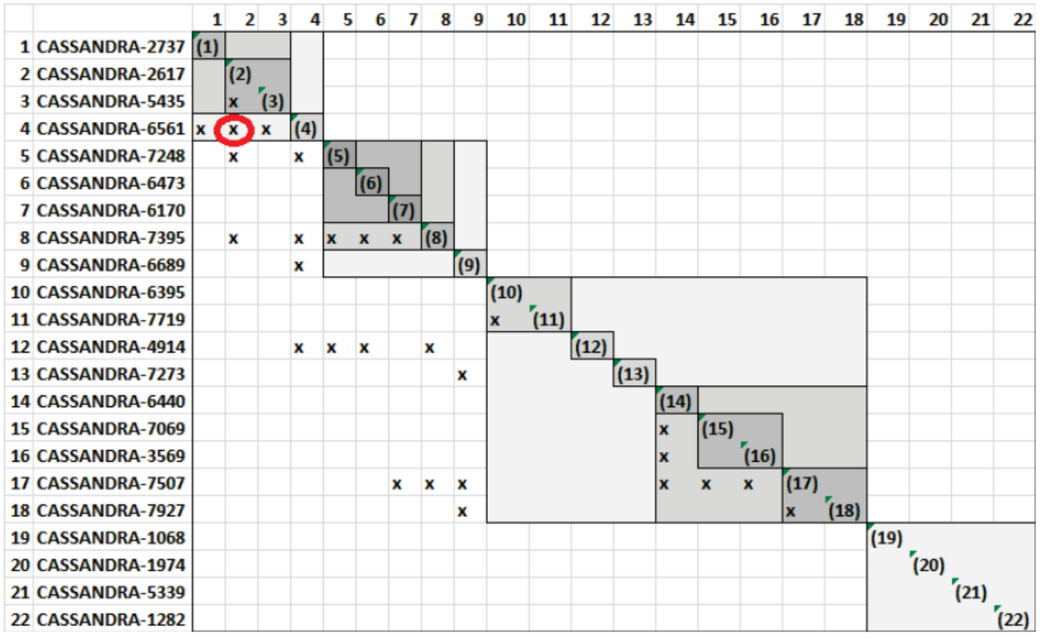


Figure 3.4: Example of a evolution storyboard [35]

Although the evolution storyboards and the visualization in this work are similar (in both cases, the nodes represent source code entities, and some coupling determines the distance between nodes), there are substantial differences. First, in this work, the focus is on visualizing feature-related files and features do not play a role in evolution storyboards. Therefore, there is also a difference in the purpose of the visualization. While Beyer et al. want to help developers comprehend software evolution on a file basis, this work intends to help developers comprehend features and their dependencies. Second, there are differences in the calculation of the node's couplings. Beyer et al. determine the coupling only based on the logical coupling. In contrast, in this work, other factors, like structural couplings and references to issues from an ITS, are considered as well.

3.1.5 Feature Spaces

Mo et al. [37] defined feature spaces, which consist of file sets, which have been changed during the development of a feature over time. These feature spaces are used to capture the dependency relations among features in a feature dependency structure matrix (feature dependency structure matrix (FDSM)). Figure 3.5 shows an example FDSM of some features from the Apache Cassandra project[1]. An x in a cell indicates a feature dependency. For example, the circled cell $(4, 2)$ indicates a dependency from feature CASSANDRA-6561 to CASSANDRA-2617.



		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
1	CASSANDRA-2737	(1)																					
2	CASSANDRA-2617		(2)																				
3	CASSANDRA-5435		x	(3)																			
4	CASSANDRA-6561	x	x	x	(4)																		
5	CASSANDRA-7248		x		x	(5)																	
6	CASSANDRA-6473						(6)																
7	CASSANDRA-6170							(7)															
8	CASSANDRA-7395		x		x	x	x	x	(8)														
9	CASSANDRA-6689				x					(9)													
10	CASSANDRA-6395										(10)												
11	CASSANDRA-7719										x	(11)											
12	CASSANDRA-4914				x	x	x		x				(12)										
13	CASSANDRA-7273									x				(13)									
14	CASSANDRA-6440														(14)								
15	CASSANDRA-7069														x	(15)							
16	CASSANDRA-3569														x		(16)						
17	CASSANDRA-7507							x	x	x					x	x	x	(17)					
18	CASSANDRA-7927									x								x	(18)				
19	CASSANDRA-1068																			(19)			
20	CASSANDRA-1974																				(20)		
21	CASSANDRA-5339																					(21)	
22	CASSANDRA-1282																						(22)

Figure 3.5: Example of a FDSM [37]

The calculation of the feature decoupling level (FDL) metric is based on the FDSM. Mo et al. use the FDL to find out if dependent feature spaces are often changed together and if the feature decoupling level is consistent with architectural maintainability metrics and if well-modularized systems have a low FDL or respectively if seemingly poorly modularized system have a high FDL.

Although there are some similarities in determining the files realizing a feature, Mo et al.'s approach is different from this work in purpose and result representation. Their purpose is to calculate some metric, which indicates how easy it is to change or add new features to a project. Therefore, they do not provide an interactive visualization, which is an essential aspect of this thesis.

[1]https://cassandra.apache.org/

3.1.6 SourceMiner Evolution

Novais et al. [38] developed SourceMiner Evolution, a tool focused on the visualization of software feature evolution. This tool helps developers to comprehend feature's source code across the software history, compare different versions of the program, and to perform maintenance tasks such as refactoring and modifications of existing features. The tool uses feature mappings, which contain the information in which source code parts realize specific features. These feature mappings are based on a seed mapping, which contains source code parts that are already known to realize certain features. Mapping expansion heuristics are used to generate feature mappings for feature versions.

The mappings are then used in different types of visualizations, like a dependency visualization, which represents the coupling among the feature's code elements. Figure 3.6 shows an example of the feature dependency visualization. Users can filter dependencies and assign colors to specific features. The tool also implements structural and lexical filters to allow the user to search for specific classes or methods.

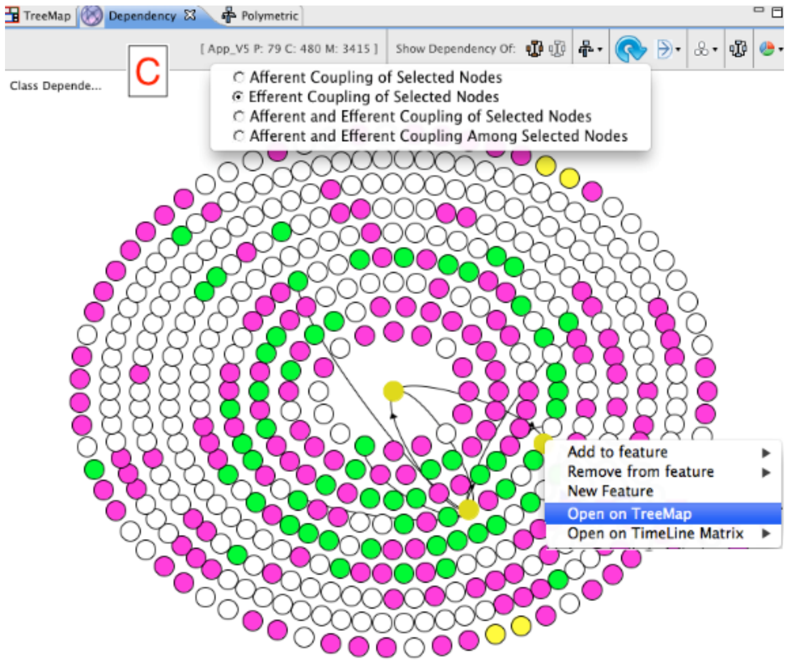

TreeMap
Dependency
Polymetric
[App_V5 P: 79 C: 480 M: 3415]
Show Dependency Of:
Class Depende...
C
Afferent Coupling of Selected Nodes
[✓ Efferent Coupling of Selected Nodes]
Afferent and Efferent Coupling of Selected Nodes
Afferent and Efferent Coupling Among Selected Nodes
Add to feature
Remove from feature
New Feature
Open on TreeMap
Open on TimeLine Matrix

Figure 3.6: SourceMiner Evolution [38]

Intent and result representation of SourceMiner Evolution is similar to the tool developed in this thesis. The main difference between the tools is how they map source code to features. SourceMiner Evolution uses feature mappings, which are XML files containing the mapping between source code and features. In contrast, the mapping in this thesis is determined by different types of coupling in combination with the information stored in an ITS. SourceMiner Evolution uses heuristics to determine the feature mapping for later versions, but there is at least the need for a seed mapping; however, how the data stored in the seed mapping is generated is left open.

3.1.7 Feature Coupling Metrics

In their work, Revelle et al. defined different types of feature coupling metrics, demonstrate the relationship between feature coupling and fault-proneness and evaluated the application of feature coupling to impact analysis [12]. Further, they developed an Eclipse[2] plugin, which allows assigning code to features and calculating their feature coupling metrics for these features.

However, they use structural coupling and textual information to calculate the feature coupling. Therefore, they use the program code as the only source for feature coupling, while the approach in this thesis also uses VCS and ITS to investigate how features are coupled. Another difference is that it is possible to determine the feature coupling with the proposed metrics, but not what source code entities caused it. Further, they do not provide an interactive visualization which is an important aspect of this thesis.

3.2 Literature of Related Topics

This section introduces works from the related topics visualization, feature location and software evolution. The intent of describing these examples is to provide an overview of the broader current state-of-the-art.

3.2.1 Feature Location & Impact Analysis

This subsection lists different works in the field of impact analysis and feature location.

ImpactMiner [6] is an IA Eclipse plugin which was build on top of FLAT[3] [39], a feature location tool using textual and dynamic FLTs. ImpactMiner uses the VCS to analyze co-changes between files to generate association rules which are used to determine possible affected files of changes. JRipples [40] and Chianti [41] are tools which are closely related to ImpactMiner.

Torichiano et al. [42] also use the information stored in bug repositories and VCS for impact analysis. However, they do not investigate co-changes or logical coupling but use a NLP approach to perform impact analysis. Canfora et al. [43] follow a similar approach with their tool Jimpa. Indifference to Torchiano Jimpa uses similar old change requests to link bugs to related source files, while Torichinao uses source code comments.

Zanjani et al. [44] developed an IA approach to find impacted files for a given change request based on a combination of interaction and VCS history. They use IR, machine learning and lightweight source code analysis techniques from source code entities like files and methods. Chochlov et al. developed ACIR, a tool that leverages VCS data to

[2]https://www.eclipse.org/

extract changeset descriptions and use it to cluster source code artifacts which users can retrieve by using IR techniques [14].

Rath et al. [45] developed TraceScore to find source code related to a bug description. Besides source code and its history, their approach leverages requirement artifacts and trace links found in ITS to locate relevant files for a bugfix. TraceScore presents the user a list of files ordered by their relevance for the bugfix. Besides the use case, the main difference between TraceScore and this work is the different usage of information from ITS and VCS.

3.2.2 Visualization

This section describes works in the field of software evolution visualization.

EVA

EVA [46] is a tool that visualizes the evolution of software architecture. It mines the data from a repository and visualizes the software architecture on different levels of abstraction. Figure 3.7 shows the architecture at the release of a software project. Each big circle represents an architectural component, while the small circles inside represent code-level entities.

EVA can also visualize the evolution of the architecture by comparing the source code from different snapshots. Further, it is also possible to investigate changes on the file level, which means that when comparing snapshots, files get annotated which visualizes if they were added or moved.

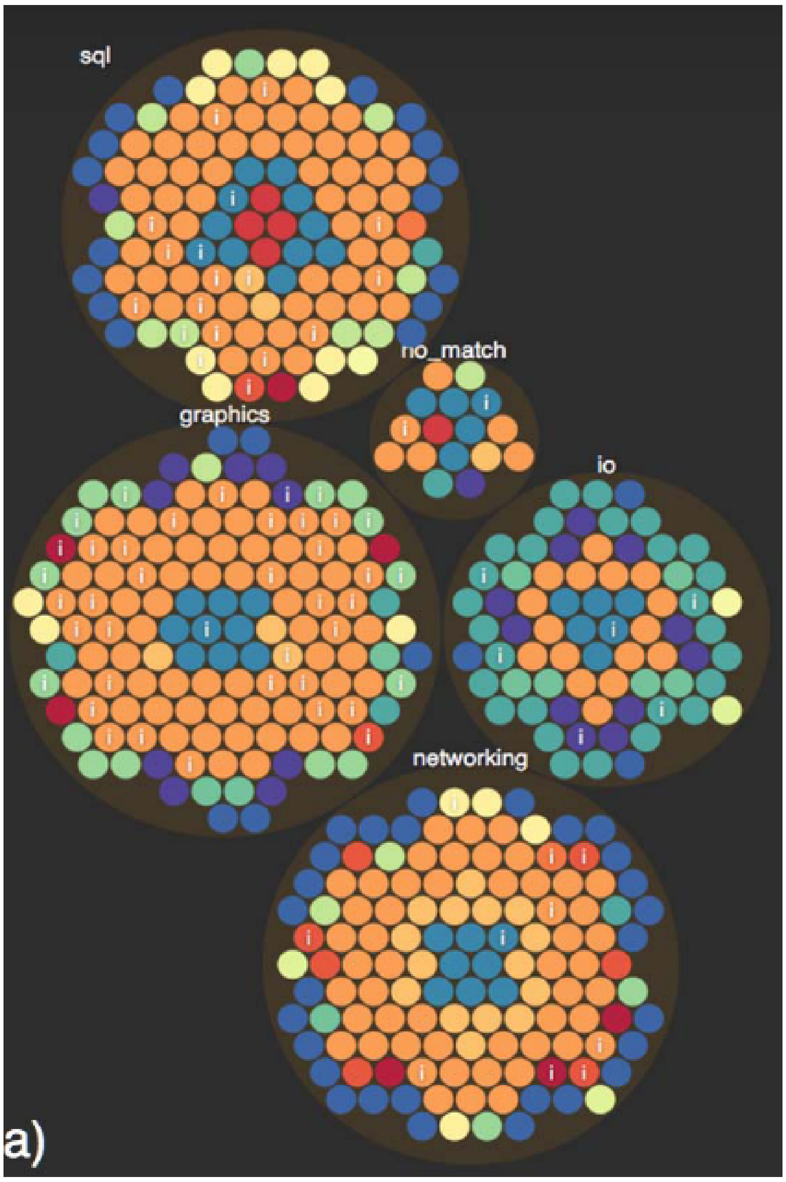


Figure 3.7: EVA - visualization of a software architecture [46]

ChronoTwigger

ChronoTwigger [47] is a tool that visualizes co-changes of source and test files over time. ChronoTwigger is based on the approach of Bayer et al. [48, 35], which determines the co-change of files through mining the data of VCS and represents them in a visualization in which the nodes represent the files. Figure 3.8 shows how ChronoTwigger visualizes the data in either a two-dimensional manner without a time dimension in part *a* of Figure 3.8, or a three-dimensional manner, where the third dimension represents the time, shown in part *b*.

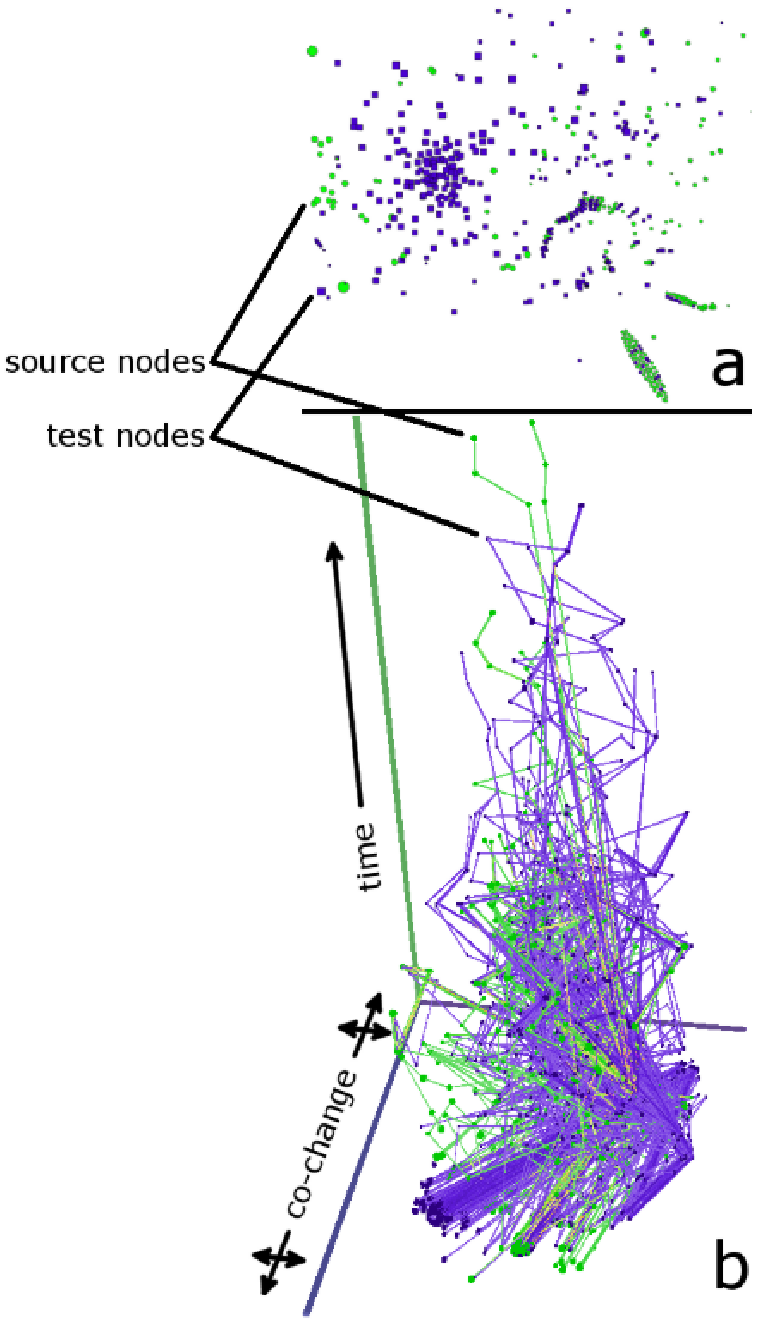


Figure 3.8: ChronoTwigger [47]

CodeCity

As part of the Evospaces project, [49] Wettel [50] developed CodeCity, a 3D visualization using a city metaphor to support tasks related to program comprehension, design quality assessment and evolution analysis. The districts of the city represent packages and its buildings the contained classes. The height of the buildings is based on the number of methods and the base size is determined by the number of attributes in the related class. The project also provides a "time travel" functionality, allowing stepping through different code versions.

CodeCity is not the only example of a software evolution visualization using the city metaphor or similar ones. For example, SkyscraperAR [51] was heavily inspired by CodeCity. However, SkyscraperAR is an augmented reality visualization and uses different metrics for building the city, like lines of code and code churn (total number of changed lines in evolving classes). Another example is Charter et al. [52], who developed Component City, a visualization of reusable software components. Balzer et al. [53] used a landscape metaphor that is similar to the city metaphor but from a broader viewpoint, like using continents, states, cities and districts. However, the tools from Chater et al. and Balzer et al. do not allow navigation through time to visualize the evolution of their cities/landscapes.

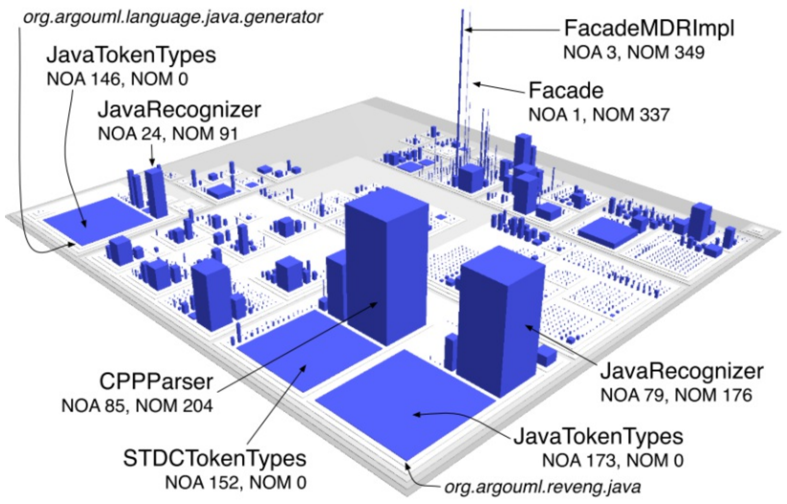


Figure 3.9: Visualization of the ArgoUML[3]project with the city metaphor [50]

[3]https://de.wikipedia.org/wiki/ArgoUML

3.3 Non-scientific Systems

This chapter showcases tools similar to the visualization developed in this thesis but have a non-scientific background.

3.3.1 CodeScene

CodeScene[4] is a commercial tool that calculates metrics regarding source code based on the information stored in the repository and the issue tracking system. The approach taken in this tool is very similar to the one followed in this thesis. In CodeScene two modules are coupled in time,

- if they are modified in the same commit,
- if they are changed by the same developer within a given period
- if they refer to the same issue in commits containing the modules.

However, in difference to this thesis, the tool does not directly focus on feature coupling but on coupling and general technical debt.

Figure 3.10 shows how CodeScene visualizes the logical coupling. The color of the edges indicates the temporal dimension of the dependency; red means the coupling between the connected files gets stronger, while yellow indicates a stable coupling.

3.3.2 CodeMR

CodeMR[6] is a semi-commercial tool to check code complexity, cohesion, coupling, and size metrics. It visualizes these metrics and the relationship between files and modules to provide the users with additional insights. Figure 3.11 showcases the visualization of files and packages within a module. The size of the nodes indicates the size of the source file, while the color indicates the degree of complexity. The shape indicates the degree of coupling; the more edges the shape has, the higher is the coupling to related source files. For example, a small green circle means it is a simple, non-complex file with no significant coupling. On the other side, a big red star is a large and complex file, which is tightly coupled.

CodeMR ships as a plugin for Eclipse[7] or IntelliJ[8] and is based on static information of the source code, which means it does not use data from VCS or ITS.

[4]https://codescene.io/
[5]https://codescene.io/docs/guides/technical/temporal-coupling.html
[6]https://www.codemr.co.uk/features/
[7]https://www.eclipse.org/
[8]https://www.jetbrains.com/de-de/idea/
[9]https://www.codemr.co.uk/

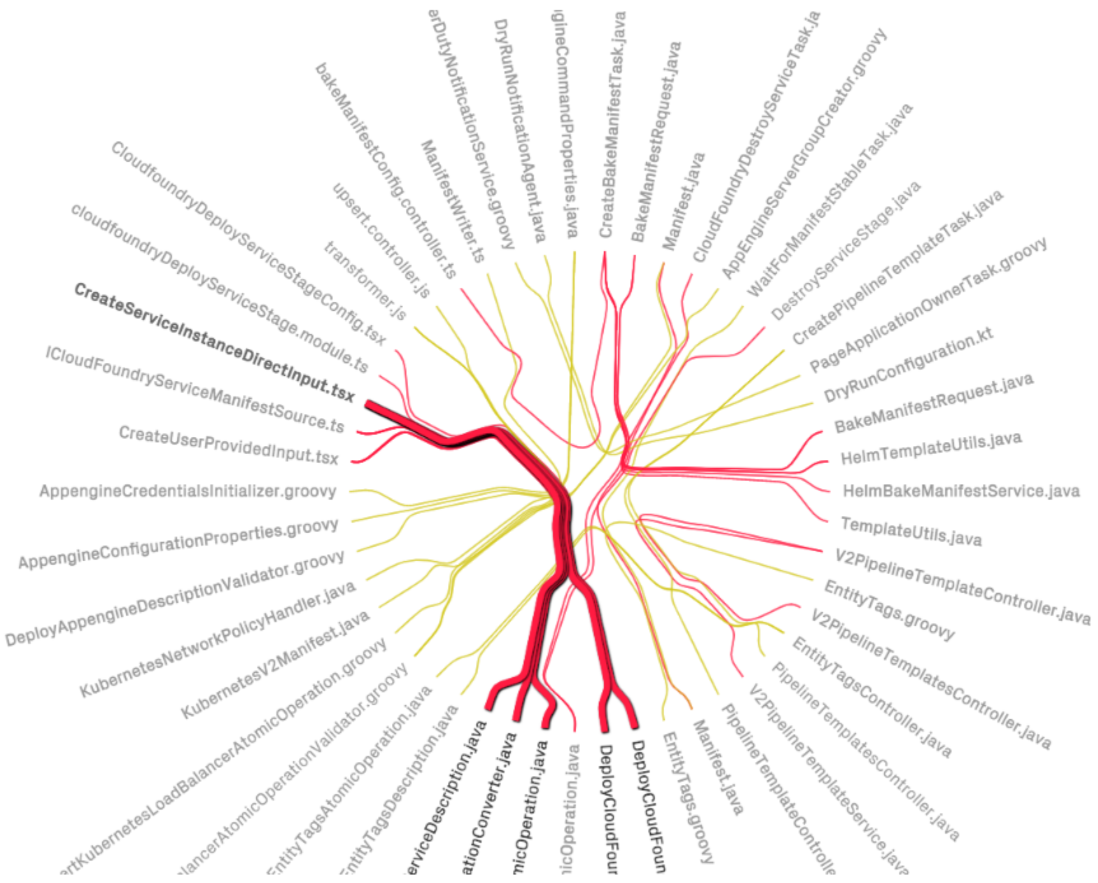

CreateServiceInstanceDirectInput.tsx
ICloudFoundryServiceManifestSource.ts
CreateUserProvidedInput.tsx
AppengineCredentialsInitializer.groovy
AppengineConfigurationProperties.groovy
DeployAppengineDescriptionValidator.groovy
KubernetesNetworkPolicyHandler.java
KubernetesV2Manifest.java
CloudfoundryDeployServiceStageConfig.tsx
cloudfoundryDeployServiceStage.module.ts
bakeManifestConfig.controller.ts
upsert.controller.js
transformer.js
ManifestWriter.ts
DryRunNotificationAgent.java
CreateBakeManifestTask.java
BakeManifestRequest.java
Manifest.java
CloudFoundryDestroyServiceTask.ja
AppEngineServerGroupCreator.groovy
WaitForManifestStableTask.java
DestroyServiceStage.java
CreatePipelineTemplateTask.java
PageApplicationOwnerTask.groovy
DryRunConfiguration.kt
BakeManifestRequest.java
HelmTemplateUtils.java
HelmBakeManifestService.java
TemplateUtils.java
V2PipelineTemplateController.java
EntityTags.groovy
V2PipelineTemplatesController.java
EntityTagsController.java
PipelineTemplatesController.java
V2PipelineTemplateService.java
Manifest.java
EntityTags.groovy
DeployCloudFoun
DeployCloudFour

Figure 3.10: A visualization of logical coupling in CodeScene[5]

Figure 3.11: CodeMR: A visualization of file relations[9]

CHAPTER 4

Requirements Analysis

This chapter describes the challenges and information needs developers have during the maintenance of software projects. Based on these challenges and needs, requirements are formalized, which serve as a foundation for the implementation of the visualization tool. Further, this chapter explains how these requirements were raised and evaluated before they were used in the implementation phase of this thesis.

4.1 Methodological Approach of Requirement Analysis

This section describes how the requirements have been raised and analyzed.

Fundamental problem The starting point of the requirement analysis is the observation that is dealing with software evolution during the maintenance of a project is an existing challenge, as described in Chapter 1.1.

Research information needs of developers With the fundamental problem in mind, research has been conducted to find out what challenges developers face when dealing with software evolution and what information needs they have to overcome those challenges.

Research state-of-the-art Next, broad research about current existing approaches that try to solve those challenges has been conducted. Based on the result, a research gap has been identified for what open challenges have not been addressed sufficiently. Chapter 3 describes the current state-of-the-art. After a gap has been identified, the research in that area has been reinforced.

Definition of requirements Based on the information needs of developers, the challenges they face during software maintenance and the state-of-the-art, the requirements have been defined to satisfy the information needs and help developers overcome the challenges they face. In addition to that, considerations of the thesis' author are also part of the requirements. The origin of every requirement is explicitly mentioned.

Evaluation In order to validate the soundness of the requirements, developers have filled out a questionnaire and have been interviewed. The provided feedback is then worked into the requirements and it is also used to prioritize the requirements.

4.2 Information Needs of Developers

Codoban et al. [54] have surveyed 217 developers in order to find out how they examine software history. 85% of those developers said that software history is essential for their development activities. Codoban et al. asked what motivates developers to examine software history and found out that the reasons to use it mainly depend on the novelty of the changes. As Figure 4.1 shows, what motivates users the most, independent of the novelty of the changes, is debugging.

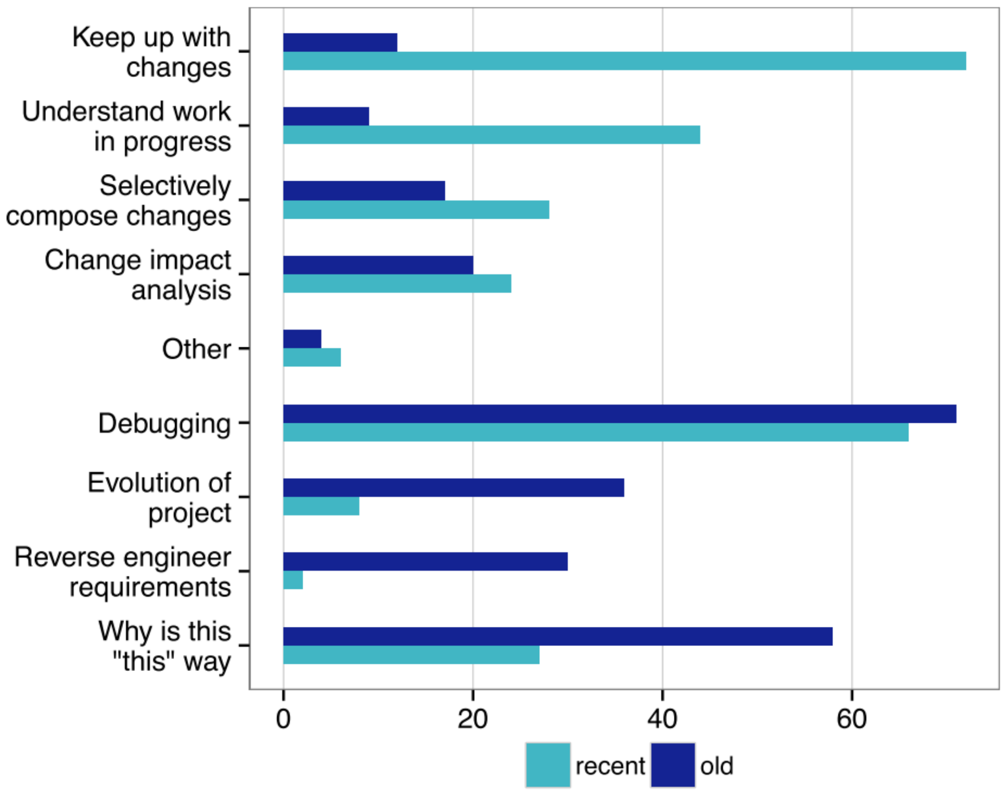


Figure 4.1: Motivation to examine software history. [54]

This means that developers use software history to find specific commits that introduce a bug. After they find the commit, they usually try to understand the change, learn how the requirements evolved and find the commit's author.

The study also shows that developers mostly use recent changes to keep up with changes and understand work in progress while examining old changes to recover the rationale behind some source code. Developers do this in order to verify that their changes will not break any existing functionality. This means they have to reverse engineer the code's requirements to gain a better understanding of what the existing code intends to implement and if the introduced changes affect them.

Fritz et al. [55] interviewed eleven professional developers in order to learn about their information needs. Through their interviews, Fritz et al. identified 78 questions developers wanted to ask, but existing tools do not support them. These questions target a broad spectrum of the project life-cycle, but many questions target the source code. For example, developers asked:

- What is the evolution of the code?
- Why were they [these changes] introduced?
- Who made a particular change and why?
- Who is working on the same classes as I am and for which work item?
- What classes have been changed?
- Which features and functions have been changing?
- What is the collaboration tree around a feature?

4.3 Challenges

Codoban et al. [54] describe the challenges developers face when dealing with software history, how developers are handling these problems and what developers wish tools would provide to them. They categorized these challenges into four themes, which are explained in the following part.

4.3.1 Information Mess

Codoban et al. [54] define information mess as the challenges due to the structure and organization of information. For example, large projects have a VCS, which contains many commits, which makes it hard for developers to find particular commits or to discover undesired changes. Developers also often struggle with so-called tangled commits, which are commits that contain multiple unrelated changes.

To overcome those problems, developers ask for

- Better grouping capabilities for commits

- Better filtering of commits
- Providing filtering capabilities for diffs
- Querying, for example, when was a line added

4.3.2 Knowledge Fragmentation

During software maintenance, developers often have high-level questions regarding architecture-related changes, which requires them to understand the context of previous changes [54]. The problem is that the needed information is fragmented over different systems. For example, developers have issues comprehending how a certain change impacts modules or packages. The developers surveyed in the study of Codoban et al. [54], therefore, wished for a diagram highlighting modules affected by a change.

Another problem maintainers face is to find out what requirements drove a change, which they try to circumvent by adding issue ids to the commit messages or using a tool like Atlassian Stash[1].

4.3.3 Understanding History

For developers, it is hard to understand software history because there is much context that is not documented or hard to find [54]. Most of the time, when developers try to understand commits, they read the related commit message. However, 66% of the surveyed developers said that commit messages are not informative. These problems often occur due to not following commit message guidelines.

Another source used by developers to understand commits are change diffs. However, many participants in the study of Codoban et al. [54] have reported that reading diffs is hard and they often do not provide any insights or the intent of the change is not clear enough. Another problem that makes understanding diffs hard is that they often include noise like added white spaces and changed line endings.

That problem gets worse by so-called tangled commits. Those are commits that do not address a single change but include multiple, often unrelated, changes. This makes it harder for developers to understand the intent of the commit as they have to find out to which change a diff belongs.

4.3.4 Tool Limitations

In the study of Codoban et al. [54] several participants reported they wish for better visualization tools for software history. They also face challenges with the linear organization of history and wish to group history by feature or change similarity. Other challenges regard the usability of existing tools, like setting up search environments or detecting file movements.

[1]https://www.atlassian.com/blog/archives/atlassian-stash-enterprise-git-repository-management

4.3.5 Requirements of Software Visualisations

There are some basic requirements and quality attributes visualizations should meet to provide benefit to users. Kienle et al. [56] provide some baseline requirements and quality attributes that serve as a starting point for requirements elicitation. This section describes those quality attributes and requirements.

Quality Attributes

Rendering scalability In order to meet the user's expectations, the rendering speed of the visualization should scale up to large amounts of data, especially for visualizations that allow direct manipulations, like graphs [56].

Information scalability Visualizing too much data at once can cause a cognitive overload to the user, which is referred to as overplotting [56]. If it comes to overplotting, users cannot extract the relevant information from the visualization, making it unusable. In order to overcome this problem, from which often graphs suffer, the visualization should only provide interesting information, which can be accomplished by using abstraction and the possibility to filter the data.

Interoperability To make tools interoperate with each other, they have to agree on some exchange format [56]. The coupling between the exchange information and the tools should be loose, which means the exchange procedure should be independent of the tools. One example of such an exchange procedure is the Graph eXchange Language (GXL) [57], which is an XML sublanguage, which offers support for exchanging instance graphs and the underlying schema information.

Customizability Tools should provide the possibility to customize it to personal needs by either using configuration files, supporting built-in scripting, or having programmable interfaces [56].

Interactivity Software visualizations should be interactive and explorative [56].

Usability Usability is the "ease of using the tool" and is rated an essential requirement, but also hard to satisfy [56].

Adoptability Different factors influence the adaptability of a tool [56]. For example, the tools should be lightweight and programmable. Another factor is that the tool's benefit must compensate for its adaption (setting it up, learning it) and its usage should positively influence the user's perception. The tool should be compatible with existing processes, users and other tools.

Functional Requirements

Views Typically software visualizations provide multiple views that try to satisfy the different information needs of possibly multiple stakeholders. Different views can also be used to emphasize different dimensions of data, like time or different levels of abstraction.

Abstraction Software visualizations that are based on a graph model tend to become complex very fast. Hierarchical graphs, which allow the grouping of multiple nodes, can create a higher level of abstraction, resulting in a layered view.

Search In a survey by Keller et al. [58] searching for graphical or textual elements was rated as one of the most important functions in software visualization tools. However, many visualization tools have only minimal searching capabilities, so Storey et al. [59] have concluded that visualizations tools need better querying support.

Filters Filtering is a rudimentary form of querying [56] and an effective approach in reducing the data to visualize.

Code proximity Code proximity means the ability to provide easy and fast access to the underlying source code of a visualization entity [56]. For example, when clicking on a node in a graph that represents a source file, that source file is opened.

Automatic layouts An important aspect of visualization is the automatic creation of the visualization's layout [56]. This is especially important for graphs, where nodes should be placed in a manner that avoids intersecting edges as much as possible.

Undo/History When users can perform interactions with the visualization, users should have the possibility to undo their changes and keep a history of past interactions [56].

4.4 Requirements

This section describes stakeholders, use cases and the essential requirements of the developed tool. The author developed the requirements in order to support developers during software maintenance. The requirements are based on the existing literature, state-of-the-art tools, and own considerations.

4.4.1 Stakeholders

The target audience of the proposed visualization are primarily developers. The main goal of the visualization is to support them during the maintenance of an existing software application.

However, other professionals in the software engineering field could benefit as well. The visualization could support software architects to gain an overview of coupled features. This information can be helpful to direct them to coupled entities, which could indicate that these entities should be refactored or are hard to decouple.

The evolutionary aspect of the visualization can support project managers as it shows when features were developed or modified. Further, they could leverage the tool to find out how hard it was to implement a specific feature. This information might be helpful in the future when managers have to allocate resources to change a feature again.

4.4.2 User stories

The requirements of the prototype were elaborated with the following user stories in mind:

User story 1: *As a developer, I want to find out where a feature is implemented in the source code.*

Users should be able to locate the source code that implements a feature without being very familiar with the codebase. Developers need to locate a feature in the code when they need to implement an improvement or fix a bug in that specific feature.

Users select the feature they want to investigate and the tool should visualize the source code entities which implement it.

User story 2: *As a developer, I want to know which features are coupled to each other and why.*

Users should be able to find out which features are coupled and which source code caused the coupling, which supports developers during impact analysis, as it highlights source code entities related to multiple features. A change to a coupled entity could introduce unintended behavior.

Another use case is that developers or software architects might want to find out which entities should be refactored to decouple features.

Users select multiple features they want to check and the visualization should highlight coupled features.

User story 3: *As a developer, I want to find out how features evolved.*

Users can visualize a feature through its whole lifecycle, which can be leveraged for feature comprehension. The evolutionary aspect can also be used to find out since when features are coupled to each other. This helps developers with feature coupling evolution comprehension, which is helpful to plan refactorings or locate bugs as feature couplings could unintentionally introduce unintended behavior.

Users select the features they want to check, and the tool shows the feature's evolution.

User story 4: *As a developer, I want to find out which features are implemented in a certain source code entity.*

When users modify a specific entity, they might not be aware of which features they impact with their change. The tool should enable developers to quickly find out to which features a particular entity belongs, which can be helpful for impact analysis.

Users visualize all features and filter the specific entity they are interested in.

User story 5: *As a developer, I want to know which source code is impacted by a change to a feature/source code entity.*

Like the previous user story, developers want to know which other source code entities might be affected by a change, which helps determine where they have to propagate changes to prevent unintended behavior.

Users visualize all features and highlight the specific entity they are interested in to see how it is coupled to the rest of the application.

4.4.3 Prototype Requirements

ID	R1
Name	Visualization of features and their relationships among each other
Description	Visualization of source code entities which implement a certain feature. Entities that are also used in other features are highlighted; thus, the visualization enables the user to identify source code entities that cause a feature coupling. To identify the relationships, different types of couplings should be used.
Origin	Research gap after state-of-the-art analysis in Chapter 3. Also, based on the author's consideration on how to support developers during software maintenance.

ID	R2
Name	Support different abstraction levels
Description	The user should be able to investigate feature coupling evolution on different levels of abstraction. The supported abstraction levels are package, file and method.
Origin	Challenge:Information mess 4.3.1, Challenge: Knowledge fragmentation 4.3.2

ID	R3
Name	Evolutionary aspect of features
Description	The user should be able to investigate the evolution of a certain feature which means the visualization has to support a time dimension. The user should be able to easily navigate for-and backward in the visualization's time dimension in order to gain a better understanding of the evolutionary development of the feature. Unimportant changes for the inspected feature should be filtered.
Origin	Challenge:Information mess 4.3.1, Challenge:Understanding history 4.3.3, Challenge:Tool Limitations 4.3.4, interactivity of visualizations 4.3.5

ID	R4
Name	Filtering of source code entities
Description	The user should be able to filter the source code entities which are presented to the user. The different filter options are dependent on the selected abstraction level. For example, the user should be able to filter for file names or file extensions on the file abstraction level, and method names on the method abstraction level.
Origin	Challenge:Information mess 4.3.1, filter functionality of visualization 4.3.5

ID	R5
Name	Filtering of commits
Description	The user should be able to filter relevant commits based on different filter options. The user should be able to filter the time frame, commit author, commit messages and contained files of the commit.
Origin	Challenge:Information mess 4.3.1

ID	R6
Name	Search and filtering (related) issues
Description	The user can investigate features by selecting the issues which address the particular feature. In order to improve this process, the user can search and filter the issue list. For example, the user can filter for different properties of issues (issue type, issue title, author, time frame) or perform a full-text search on the title or issue description. Further the tool should provide related issues to the user as well.
Origin	Challenge:Knowledge fragmentation 4.3.2, research gap after state-of-the-art analysis 3. According to the author's knowledge using related issues to analyze feature-related source code is a novel approach that is evaluated in this thesis.

ID	R7
Name	Showing source code
Description	Users can easily navigate from a visualization node to the source code of that node (assuming the node represents a file or a method).
Origin	Visualization requirement:code proximity 4.3.5

4.5 Requirement Evaluation

The prototype's requirements were evaluated by conducting an interactive questionnaire with experienced software engineers. The questionnaire aims to evaluate the requirements from Chapter 4.4.

4.5.1 Interview Questionnaire

A pilot session was conducted to evaluate the interview process and the questions. The insights of that session were used to redesign the procedure and questionnaire. For example, some questions got rephrased because they were hard to understand or too abstract. Another insight from the pilot session was that sketches of the discussed concepts should be provided throughout the interview. These sketches helped the participants to understand the different coupling types better and remember them when the concept occurred later in the interview.

4.5.2 Questionnaire Structure

The questionnaire consists of two parts.

The first part consists of demographic background questions and asks the participants how long they work in the software engineering field and about their experience with VCS and ITS. These questions aim to understand the developer's experience with those tools and if they use them for software maintenance purposes.

The second part asked the developers how purposeful a visualization of different coupling types is to them and what kind of abstraction levels they prefer. The questionnaire can be found in the appendix 9.1.8. When answering the questions, the participants should also explain their answers orally.

The questionnaire does not contain any questions regarding R4 and R5 (filtering) because filtering is necessary to avoid information mess. Additionally, [56] also stated that filtering is a requirement for valuable visualizations. So it is assumed that filtering is a mandatory feature and was therefore not addressed in the questionnaire.

4.5.3 Interview Sessions

In total, three sessions were conducted; each one lasted around 60 minutes. Unfortunately, not many participants took part in this interview, which would have significantly increased the results' validity. This issue is further discussed in Section 7.4.1.

The sessions were held remotely using the video conferencing tool Zoom[2]. During the sessions, the interviewer shared the screen showing the questions and example sketches next to it. The participants gave and explained the answers and the interviewer filled

[2]https://zoom.us/

out the questionnaire. With the participant's permission, the interviewer recorded the sessions and made notes.

The interviewer started the session by briefly introducing the challenges developers face during software engineering maintenance. After the introduction, the participants filled out the demographic background questions and described their experience with VCS and ITS.

In the second part, the interviewer introduced the different couplings types to the participants. After an introduction, the interviewer encouraged the participant to answer the questionnaire regarding the explained concept. During the interview, the participants could ask questions at any time.

4.5.4 Results

This section describes the results and insights gained from the requirement evaluation sessions.

Demographic Background

Three developers (two male, one female) practicing software engineering between 12 and 15 years participated in the interviews. For all of them, VCS and ITS play an essential role in their work.

When asked if VCS are easy to use, they agreed for base cases, but more advanced use cases, like advanced merging strategies, are challenging. Finally, they all agreed when asked if they use the VCS for maintenance purposes, but not daily.

For the ITS, they said that the ease of use is very dependent on the tool itself and the provided features. When asked if they use ITS for maintenance purposes, two participants neither agreed nor disagreed. They said they would use it; however, they usually face tickets with insufficient information and do not rely on the ticket's information.

4.5.5 Questionnaire Results

Figure 4.2 shows how much participants agreed when asked if a visualization of the specif coupling would help them during software maintenance. The result shows that participants believe that visualizing all different types would help them. They expect the most significant benefit from the feature relation.

Average strength of agreement To quantify the results from questions using a Likert scale, the metric *average strength of agreement* is used. This metric is determined by calculating the average value from all user responses for that question. Participants expressed their opinion on a discrete scale from 1 (strongly disagree) to 5 (strongly agree); therefore, the metric's value also ranges from 1 and 5.

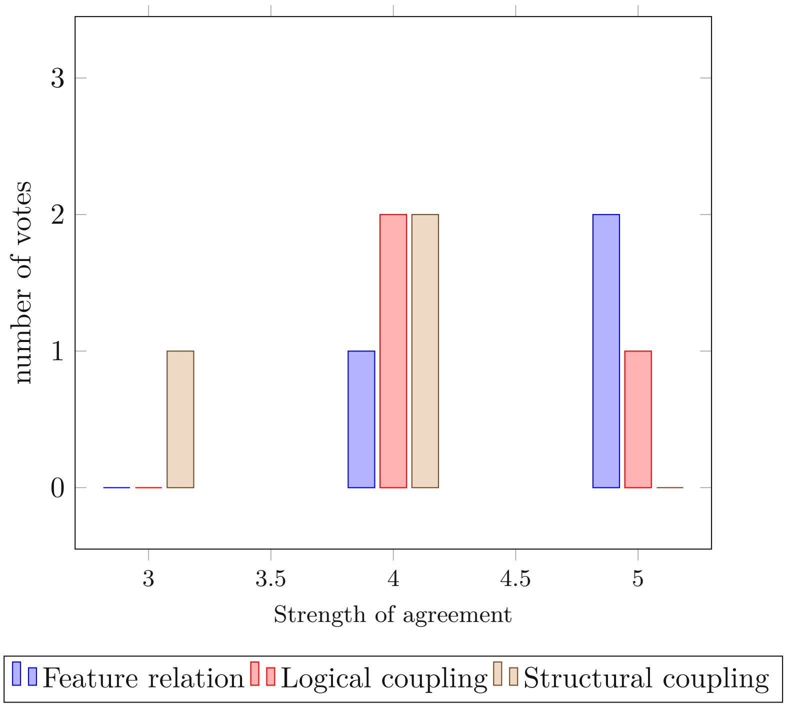


Figure 4.2: How much participants would benefit from a visualization of a specific coupling type

Ranking

The interviewer also asked the participants to rank the couplings. The result of the ranking, shown in Table 4.1, is that logical coupling is ranked as the most preferred visualization, followed by feature-relation and structural coupling.

The reasons why the participants ranked structural coupling after the others are diverse. They mentioned the following reasons:

- Overplotting due to too much information
- The information is directly available in the source code, so there is no need for visualizing it
- They already knew a tool that visualizes structural coupling and it was hard to understand

Evolutionary Aspect

Another question was if the participants are interested in the visualization's evolutionary aspect. The questionnaire asked if the evolutionary aspect would help them understand

Rating	Coupling type
8	Logical coupling
7	Feature relation
6	Structural coupling

Table 4.1: Ranking of preferred coupling type

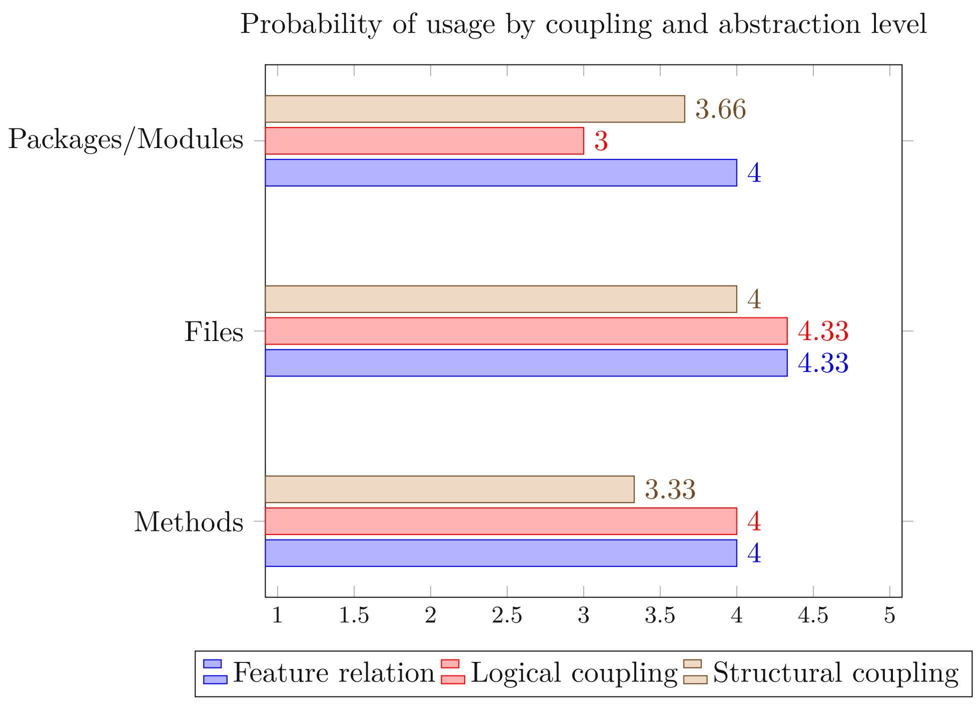


Figure 4.3: How likely participants would use a coupling visualization based on the displayed abstraction level

the feature's evolution and how it could help them identify feature coupling evolution.

The participants said that the evolutionary aspect is more interesting for investigating feature couplings (average strength of agreement: 4,66) than for a single feature (average strength of agreement: 3,66). It could help them find out when features are coupled by showing the visualization at different commits.

Abstraction Level

The result in Figure 4.4 shows that participants preferred the file abstraction level to investigate feature coupling evolution. Participants down-voted the method abstraction level because they believed that this information might already be too fine-grained to gain relevant insights quickly. The results are ambivalent for packages because one participant would use it for architectural considerations, as another said he is not very interested in the evolution of packages, but only for the current state.

All participants pointed out that it is also essential for them in the visualization to find out what changes cause a feature coupling at a current snapshot. Thus, they want to see the responsible diffs of the changeset.

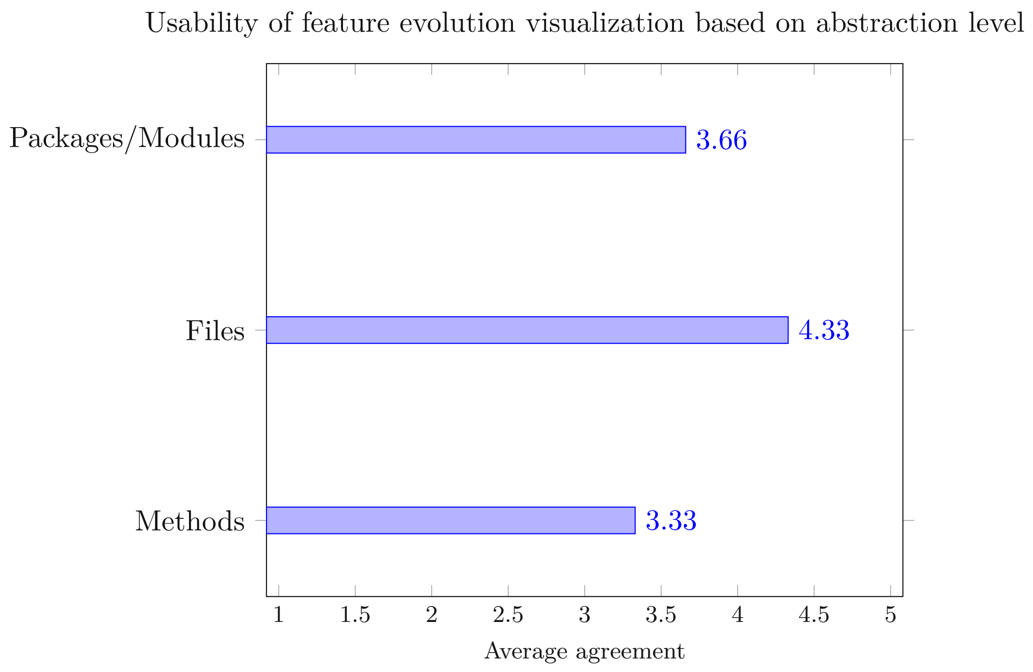


Figure 4.4: Participants answer on what abstraction level they would investigate feature evolution

Using Ticket Relations for Feature Scope Determination

The questionnaire also asked the participants if the tool should also consider related issues to determine the feature scope to help them understand the impact of a maintenance task on the feature.

The results are unambiguous, as one participant strongly agreed while one disagreed (average strength of agreement: 3,33). One participant said he would use it but believe that the tickets are not maintained in practice, and the information cannot be relied on.

4.6 Implications for Implementation

The results show that developers are interested in a visualization tool for features. During the interviews, the participants were presented the structural, the logical, and feature-relation coupling. Feature-relation coupling means that source code, which was changed for issues of the same feature, is related. The participants considered all types useful but preferred feature-relation and logical coupling more than structural coupling. Participants said they could investigate structural coupling directly in the source code by using the Integrated Development Environment (IDE). Therefore, structural coupling was excluded from the visualization, which allowed the author to focus more on feature-relation and co-changes during the implementation phase.

The results suggest that developers prefer to use the proposed visualization on the file level, which will be the default setting. However, they would use the method or package level for other use cases. Participants rated the package abstraction level as the least relevant one.

CHAPTER 5

Implementation

This chapter describes the technical aspects of the implemented visualization tool. It will explain the selection of technologies used for the implementation and give an overview of the prototype's architecture. Further, it explains what technical challenges were faced and how these challenges were addressed.

5.1 Technical Considerations and Technology Stack

The prototype is a web application running in the user's browser. The programmer of this tool chooses this type of application due to the broad availability and support of required technologies to develop visualizations. Another consideration is that web applications are well-known by most users, which do not need to install the application because it runs in any modern web browser.

For simplicity, the architecture is a client-server architecture. The client, the HTML frontend, was developed as a Single-Page-Application (SPA) with Angular[1]. Angular was selected over alternatives like Vue[2] or React[3] because it comes along with Typescript, which should prevent programming errors and provides better maintenance capabilities. Additionally, the tool developer is more familiar with Angular than the other frameworks, which results in a faster development process.

In the front end, users can filter issues and commits, manage the features they are interested in, and investigate the features' visualization. The GUI is implemented by

[1]https://angular.io/
[2]https://vuejs.org/
[3]https://reactjs.org/

using the component library PrimeNG[4]. The visualization graph is generated with d3.js[5] using a force-layout[6].

The backend is written with Java and uses Spring Boot[7]. Spring Boot was selected because it is easy to set up, requires only minimal configuration for setting up a server and other used Spring modules, like Spring data[8], are easy to integrate. The backend accesses VCS repositories by using JGit[9]. The source code is analyzed by generating an AST with srcML[10]. The source file's AST, stored in an XML file, is queried with Xpath.

Data is stored in an ArangoDB[11] instance. ArangoDB is a multi-model data store, which includes a document store and supports a graph model. The tool developer decided to use ArangoDB because it supports joins for documents, provides a SQL-like query language called AQL and supports a graph model. The graph model has a considerable benefit on the application performance because many different relations, like logical couplings, have to be processed. Details on the data schema and its benefits from the graph model can be found in Section 5.2.1.

5.2 Architecture

The following subsections will explain the tool's building blocks in detail and their connection to the rest of the application. Figure 5.1 gives a general overview of the prototype's architecture.

5.2.1 Backend

The Java Spring Boot backend is responsible for mining the data from a remote ITS and VCS and stores relevant data in an ArangoDB instance. Further, the backend provides REST endpoints where the clients can access that data to render the visualization.

The backend is structured as a 3-layered architecture. The top layer is the controller layer, which provides the REST endpoints. The controllers map the HTTP request data to Java objects and pass it to the service layer. The service layer might modify the data if necessary and pass it to the data layer. The data layer implements the Data Access Object (DAO)s as Spring Data repositories, which query the data from the ArangoDB.

[4]https://www.primefaces.org/primeng/
[5]https://d3js.org/
[6]https://github.com/d3/d3-force
[7]https://spring.io/projects/spring-boot
[8]https://spring.io/projects/spring-data
[9]https://www.eclipse.org/jgit/
[10]https://www.srcml.org/
[11]https://www.arangodb.com/

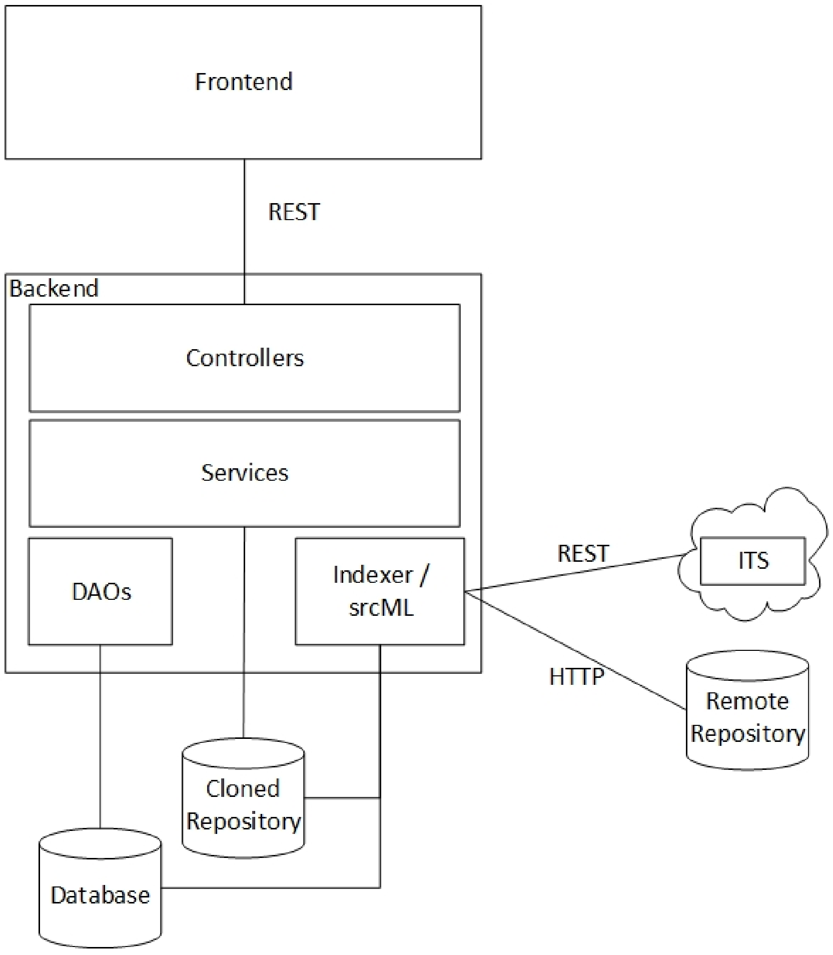


Figure 5.1: An overview of the prototype's architecture

Data Store

Database The tool uses ArangoDB, a multi-model database for persistence. Figure 5.2 shows the implemented Entity-Relationship model . Regular entities, like commits and issues, are implemented as document collections, while associative entities are realized using edge collections. In ArangoDB, an edge collection is a collection that represents edges in a graph; to do so, an entry in the edge collection contains two references (called "_from" and "_to") to entries from document collections. The attributes in the diagram represent additional properties, which are stored at the edges. For example, for related issues, it is stored what kind of relation the issues have.

Feature As features are not created automatically from the data, the user has to assign

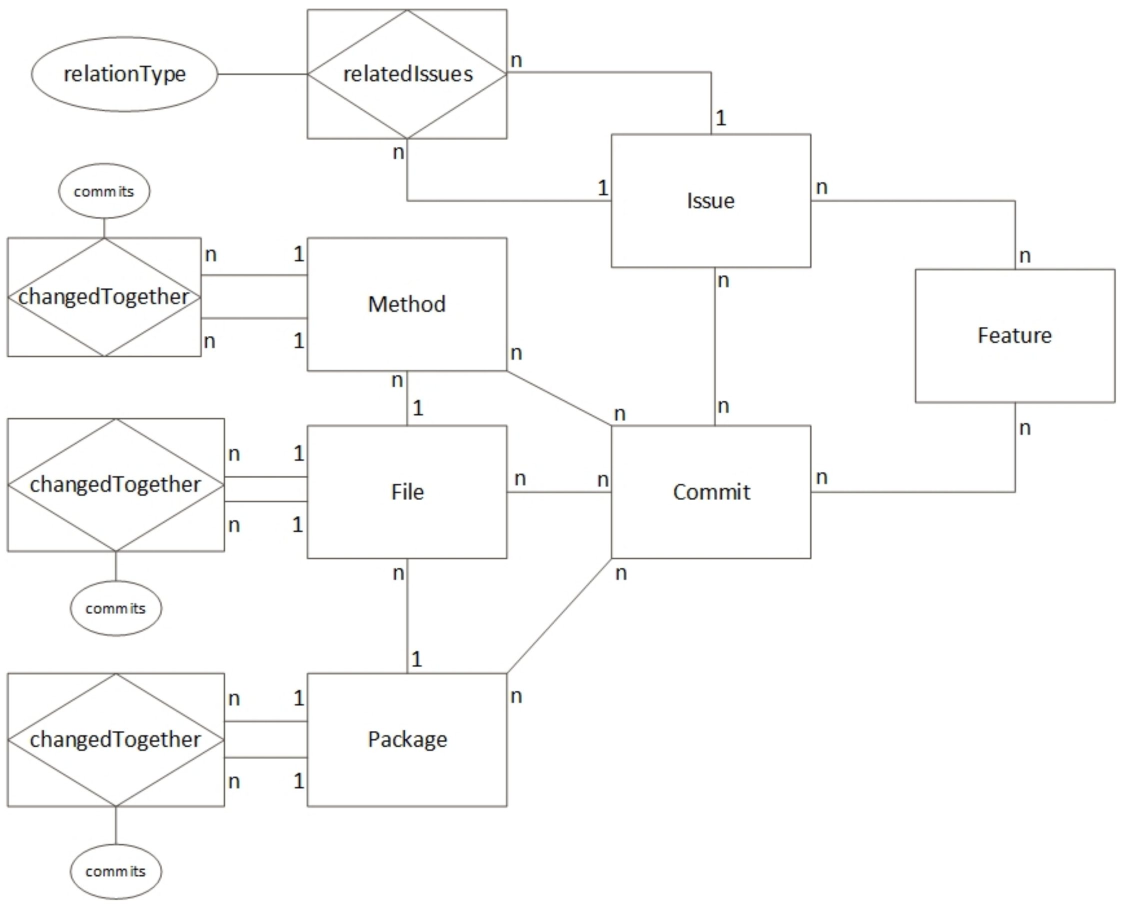


Figure 5.2: The entity relationship diagram shows how the entities of the prototype are connected to each other

issues and commits to features. To avoid re-creating features, they can be stored in the database. Users can assign a name and a color to a feature to recognize it in the visualization. When running the visualization, the user can select which features he or she wants to inspect.

Commit Represents the metadata from the repository's commits, such as author, date and commit message, but additionally stores references to issues and source code entities.

Issue Stores the most important fields from the ITS issues, like title, description, tracker, status, date and author. Issues can be used to efficiently query all issue-related commits and add them to a feature.

Related issues Stores the relationship between issues, as well as the type of relationship. The data is persisted in an edge collection, and users can query the path to get a relationship graph, as described in 2.3. This relation is used to find related issues and their commits quickly.

Method Represents a method in a file. It stores the method's name and a reference to the file to which it belongs.

File It stores the file's path and the file type. Source files also store a reference to the package to which they belong.

Package Represents a package and store its name.

Changed together These relationships store which source code entities have changed in the same commit. For every abstraction level, there is a dedicated edge collection. The edges keep track of all commits in which the source code entities have changed. Because commits already contain a reference to the source code entities, this information is saved redundantly. However, keeping track of commits in the edges allows better query performance when calculating the logical coupling at any given commit. These collections usually contain a large amount of data due to the many relationships. Example: For a single commit with n files, $\frac{(n)(n-1)}{2}$ edges have to be stored or updated. For a commit that modifies 25 files, 300 edges are needed to store all relations. For that reason, the relations of commits which modify more than 50 files or methods are not indexed. However, commits of that size usually implement a source code restructuring or cross-cutting concern, which does not influence the feature coupling. Additionally, it was taken care that participants did not need to investigate those commits in the evaluation.

Cloned Repository The cloned repository is a copy of the investigated source code repository. Due to the limitations of scope, the tool only supports Git repositories and is only capable of working in a monorepo setup. The repository is used to extract the commit metadata of each commit and store it into the database, where the tool benefits from better performance and querying possibilities.

JGit uses the cloned repository to generate diffs for specific commits because diffs are not indexed. The diffs are shown to the users, so they find out what changed in a specific commit.

Indexers

Indexers fetch the data from a (remote) resource and store it locally in an accessible manner. The tool contains three indexers, the JiraIndexer, the GitIndexer and the SourceCodeIndexer.

JiraIndexer The JiraIndexer uses the official JiraRestClient[12] to fetch all issues via REST from a remote Jira system. Relevant data of the issues are stored in an ArangoDB document collection.

Determining Issue Relations The relationships between issues are stored in an edge collection to query the relationships more efficiently via a graph model. Besides the better query performance, this also allows configuring the maximum allowed path length to be considered related. Per default, the queried path length is set to 3 to also find further related issues. However, a too-long path length could result in finding unrelated issues.

GitIndexer The GitIndexer uses JGit to clone a Git repository from a remote location to a local folder. After cloning the repository, all commits are iterated, starting with the first commit.

Determining Commit-Issue Relations The GitIndexer checks for every commit if it references an ITS issue using regular expressions representing its identifier. If a reference has been found within the commit message, the issue gets linked to the commit. The regular expressions have to be defined before starting the indexing process in the application configuration.

SourceCodeIndexer The SourceCodeIndexer iterates over all commits and uses the diff set of the changed files to store which source code entities (packages, files, methods) have changed together.

Determining Commit-Source Code Relations The SourceCodeIndexer checks which source code entities (file, methods, packages) have been changed for every commit. If two source code entities a and b have changed in a commit c, an edge e between those entities is saved in an edge collection. The edges also keep references to the commits in which the nodes have been changed. If another commit d also changed a and b, e keeps references to c and d. The edges and the references will be used to determine the logical coupling; details regarding calculating the logical coupling can be found in 5.2.2.

One challenge the SourceCodeIndexer has to overcome was to determine which methods have been changed within a commit.

To determine the changed methods of a commit, the SourceCodeIndexer restores the files at the time of that commit, and with srcML, it generates the AST of those files. The SourceCodeIndexer queries the AST for all methods and stores starting and ending lines. Then the SourceCodeIndexer iterates over all hunks of the commit. A hunk includes a list of so-call edits representing the modified region between two versions of roughly the

[12] https://docs.atlassian.com/jira-rest-java-client-parent/4.0.0/apidocs/com/atlassian/jira/rest/client/api/JiraRestClient.html

same content[13]. The SourceCodeIndexer then checks if the edit's start or end of the new version is between the starting or ending line of one of the file's methods.

In the prototype, a method is identified by a file and a method name. However, some programming languages support overloaded methods or inner classes, meaning that a file's method name is no longer unique. Overloaded methods could increase the number of nodes in the visualization but often do not provide an additional benefit because they are often semantically related. Therefore, it was decided to compose the logical coupling for such methods in a single entry.

Indexing Process The indexers must be triggered manually and run sequentially, starting with the JiraIndexer, followed by the GitIndexer and the SourceCodeIndexer. When the indexing process starts, all collections are created in the database. The duration of the indexing process depends on the number of commits, issues, and codebase size. For example, mining the ActiveMQ Git repository[14] and Jira system[15] took multiple hours. This repository is 16-year-old, contains over 10 000 commits and has a codebase of 950 000 lines of code. Their Jira system contains 7 200 tickets.

SrcML SrcML is a command-line tool which parses source code into an XML format [60]. The parsed XML represents the AST of the file.

As srcML is a dedicated tool for which no Java API exists, a wrapper has been implemented. The wrapper takes a file path as input and generates an XML file containing the AST of that input source. The backend queries the XML file with Xpath.

Currently, srcML can only parse C, C++, C# and Java files, which means it limits determining the logical coupling of methods to those programming languages. However, in 2020 the authors of the tools awarded a grant which allows them to add additional languages[16]. Because the XML elements are different from each used programming language, the prototype implementation only supports the indexing of Java files. However, the tool can be extended to work with the other supported languages of srcML.

5.2.2 Implementations of Couplings

Feature-Related Source Code

Packages, files and methods are considered feature-related if they have been changed in one of the feature's commits before or at the currently investigated commit. The calculation to which features a source code entity is related to is performed in the frontend because this can change dynamically and is highly dependent on the user's configuration like inspected features, filters, and the currently selected commit.

[13]https://archive.eclipse.org/jgit/docs/jgit-2.0.0.201206130900-r/apidocs/org/eclipse/jgit/diff/Edit.html
[14]https://github.com/apache/activemq/graphs/commit-activity
[15]https://issues.apache.org/jira/projects/AMQ
[16]https://www.srcml.org/

Logical Coupling

The logical coupling is determined and calculated directly in the database. The query in 5.1 shows how to determine all logical couplings of a file. The query has the following input parameters:

@fileId The id of the file for which logical coupled files should be found

@commitTime The timestamp at what time to get the logical coupling. Only earlier commits are considered for the calculation.

@minCount An positive integer value representing the minimum amount of commits containing both files. This value is used as a filter to avoid returning results for files, which have not often changed together. This value is used to avoid returning too many results because files might have been changed along with many other files, but only once.

@minLC A floating number between 0 and 1, which represents the minimum degree of support value between files. This filter should be used to avoid returning low-coupled files.

@fileIds This is a list of all in the feature included file ids. It is used in the query to avoid dropping logical couplings between feature-related files.

The response of the query is a list of coupled files and the confidence and support values. The query works similarly for packages and methods.

Listing 5.1: Query to calculate the logical coupling of a file

```
1  FOR fileId in @fileIds
2  LET c0=(FOR c IN INBOUND fileId GRAPH 'commitToFile'
3  FILTER c.commitTime <= @commitTime
4  RETURN c._id)
5
6  LET couplings = (FOR vertex, edge IN ANY fileId GRAPH 'filesChangedTogether'
7
8  LET filteredChangedIn = (FOR changedInCommitId IN edge.changedIn
9  FILTER changedInCommitId IN c0
10 RETURN changedInCommitId)
11
12 LET c1=(FOR c IN INBOUND vertex._id GRAPH 'commitToFile'
13 FILTER c.commitTime <=  @commitTime
14 RETURN c._id)
15
16 FILTER (LENGTH(filteredChangedIn) >= @minCountFilter AND
17 LENGTH(filteredChangedIn) / COUNT(c0) >= @minLCFilter) OR
18  vertex._id in @fileIds
19 SORT LENGTH(filteredChangedIn) / COUNT(c0) DESC
20 RETURN {
21  file: vertex,
22  countChangedTogether: LENGTH(filteredChangedIn),
23  countRoot: COUNT(c0),
24  countFile: COUNT(c1),
25  confidence: LENGTH(filteredChangedIn) / COUNT(c0),
26  support: LENGTH(filteredChangedIn) / (COUNT(c0)  COUNT(c1) −
27 COUNT(filteredChangedIn))
28  })
29 RETURN DISTINCT {root: fileId, couplings: couplings}
```

Line 1-3: Determines all commits which were committed at or before a given time (@commitTime) and were the commit includes the file with a given file id (@fileId)

Line 5: For every file which has changed in the same commit as the file with id @fileId

Line: 7-9: Get the commits in which both files have changed together before @commitTime

Line 11-13: Get all commits before @commitTime in which the other file has changed

Line 15-19: Filter and order the results by support value. Line 18 is responsible that the other filters do not drop logical couplings between feature-related files. This might be necessary because it could confuse users that feature-related files are not logically coupled.

Line 25-27: Calculate the support and confidence value of each relation

5.2.3 Frontend

In the front end, users can filter issues and commits, manage the features they are interested in, and investigate the features' visualization.

Interface

The interface is based on Angular and consists of four views, which are described below.

Feature View In the feature view, users can add new features, assign them a name and change the color, which represents the feature in the visualization. In this view, users have the option to enable or disable every feature for the visualization. This is necessary because visualizing all features would quickly lead to an overplotted visualization and increase the mental workload on the users. Users can also persist features to the backend, so created features can still be investigated later.

This view also lists all files and methods which were modified in one of the feature's commits. Because this list might become relatively large, users can query it for file or method names.

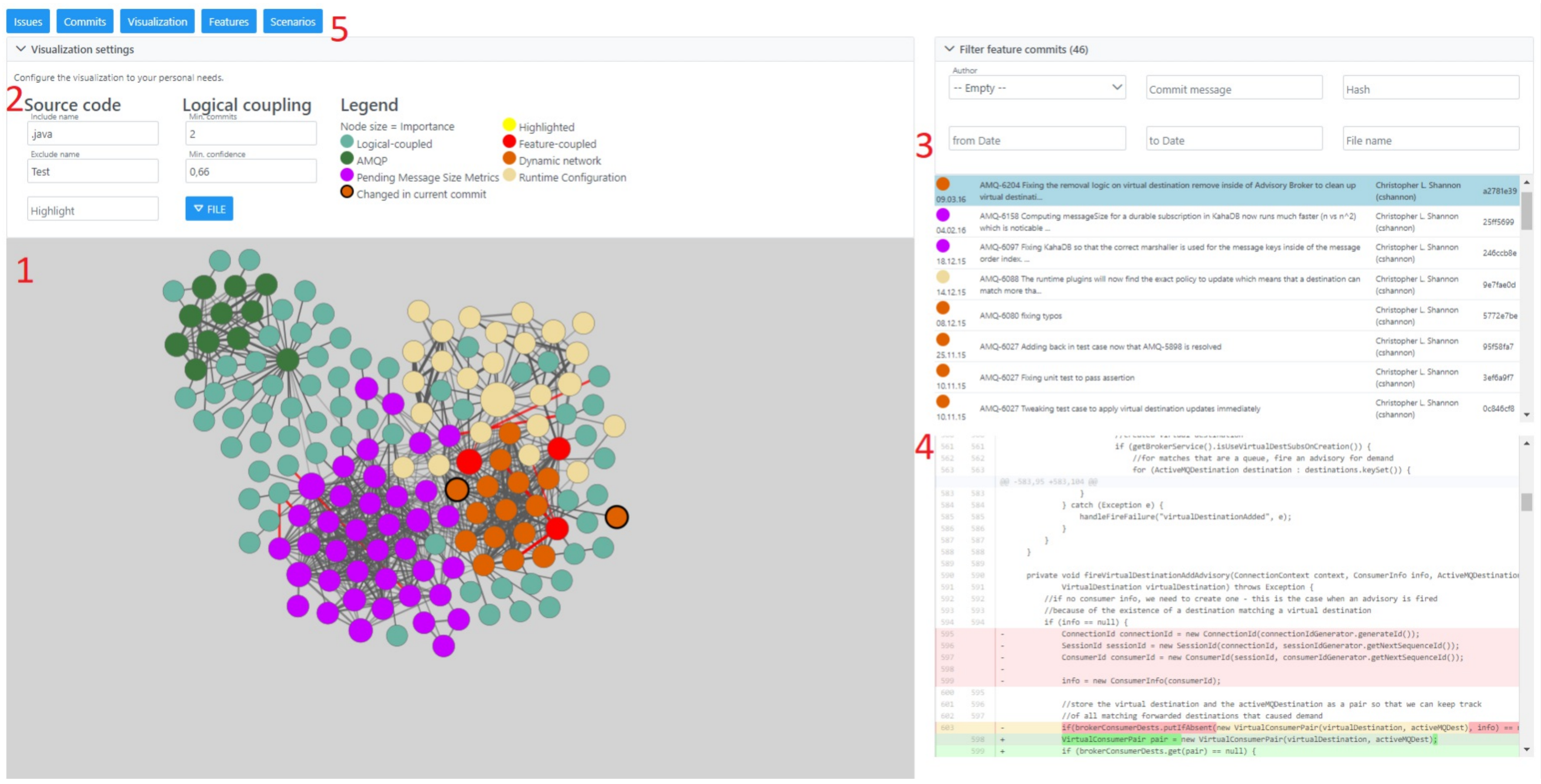
Issue View The issue view shows a list of all indexed issues from the configured Jira system. It also includes filters for date, issue type, status, or the issue description. From the result list, users can see issue details and add the issue to a specific feature. Additionally, users can add related issues and commits to the feature, which is necessary to link features to the source code.

Commit View The commit view shows a list of all indexed commits in the system. Besides some metadata like commit date and author, the filters allow searching for commits containing a specific file or related to a certain issue.

Further, users can inspect commit details like the changeset and add commits to features. This way, it is possible to assign commits with no related issues to a feature as well. Additionally, this allows the user to create features to investigate all commits containing a particular file quickly.

Visualization View Figure 5.3 shows the visualization view of multiple features. It contains the visualization (1) and navigation to investigate features through time. Section 5.2.3 described the visualization in more detail. The visualization view also allows changing the abstraction level or filtering the data (2). Users might want to change the abstraction level, dependent on their use case. For example, if they want to gain an overview, they could use the package level. However, if they try to find the cause of a coupling, they might be interested in the file or even the method abstraction level. The navigation contains a list of all feature commits, where the currently investigated commit is highlighted in blue (3). Users could leverage this list to comprehend the evolution of a feature or find out when features became coupled because it enables them to visualize features throughout the lifecycle. This commit list can be filtered as well, which might be needed to find specific entries. For example, it is possible to only consider commits from a specific time or with a certain file. The users investigate the changeset and diffs of the current commit to find out which files and lines of code have been changed (4). They could use this information for different purposes, such as finding out which change caused a coupling or fixed a bug.

After changing the commit, the visualization is updated and renders the features' state at the selected commit. At the top, users can switch to the other views at any time (5), which might be needed to configure the visualized features accordingly.



Visualization settings

Configure the visualization to your personal needs.

Source code

Include name	.java
Exclude name	Test
Highlight	

Logical coupling

Min. commits	2
Min. confidence	0,66

Legend

Node size = Importance
● Logical-coupled
● AMQP
● Pending Message Size Metrics
● Changed in current commit
● Highlighted
● Feature-coupled
● Dynamic network
● Runtime Configuration

Filter feature commits (46)

Author	-- Empty --	
Commit message	Hash	
from Date	to Date	File name

Date	Commit message	Author	Hash
09.03.16	AMQ-6204 Fixing the removal logic on virtual destination remove inside of Advisory Broker to clean up virtual destinati...	Christopher L. Shannon (cshannon)	a2781e39
04.02.16	AMQ-6158 Computing messageSize for a durable subscription in KahaDB now runs much faster (n vs n^2) which is noticable ...	Christopher L. Shannon (cshannon)	25ff5699
18.12.15	AMQ-6097 Fixing KahaDB so that the correct marshaller is used for the message keys inside of the message order index. ...	Christopher L. Shannon (cshannon)	246ccb8e
14.12.15	AMQ-6088 The runtime plugins will now find the exact policy to update which means that a destination can match more tha...	Christopher L. Shannon (cshannon)	9e7fae0d
08.12.15	AMQ-6080 fixing typos	Christopher L. Shannon (cshannon)	5772e7be
25.11.15	AMQ-6027 Adding back in test case now that AMQ-5898 is resolved	Christopher L. Shannon (cshannon)	95f58fa7
10.11.15	AMQ-6027 Fixing unit test to pass assertion	Christopher L. Shannon (cshannon)	3ef6a9f7
10.11.15	AMQ-6027 Tweaking test case to apply virtual destination updates immediately	Christopher L. Shannon (cshannon)	0c846cf8

```
                    if (getBrokerService().isUseVirtualDestSubsOnCreation()) {
                        //for matches that are a queue, fire an advisory for demand
                        for (ActiveMQDestination destination : destinations.keySet()) {
@@ -583,95 +583,104 @@
                        }
                    } catch (Exception e) {
                        handleFireFailure("virtualDestinationAdded", e);
                    }
                }
            }

    private void fireVirtualDestinationAddAdvisory(ConnectionContext context, ConsumerInfo info, ActiveMQDestinatio
            VirtualDestination virtualDestination) throws Exception {
        //if no consumer info, we need to create one - this is the case when an advisory is fired
        //because of the existence of a destination matching a virtual destination
        if (info == null) {
-           ConnectionId connectionId = new ConnectionId(connectionIdGenerator.generateId());
-           SessionId sessionId = new SessionId(connectionId, sessionIdGenerator.getNextSequenceId());
-           ConsumerId consumerId = new ConsumerId(sessionId, consumerIdGenerator.getNextSequenceId());
-
-           info = new ConsumerInfo(consumerId);

            //store the virtual destination and the activeMQDestination as a pair so that we can keep track
            //of all matching forwarded destinations that caused demand
-           if(brokerConsumerDests.putIfAbsent(new VirtualConsumerPair(virtualDestination, activeMQDest), info) == 
+           VirtualConsumerPair pair = new VirtualConsumerPair(virtualDestination, activeMQDest);
+           if (brokerConsumerDests.get(pair) == null) {
```

Figure 5.3: Visualization view

Visualization

The visualization is implemented with the d3-force[17] module from d3.js. An undirected graph is created with this module, where nodes represent source code entities like packages, files, and methods. Figure 5.4 shows the visualization of two features on the file abstraction level. With the help of the visualization, developers can figure out which files implement the feature and how the files are logically coupled to each other. Additionally, the red node highlights an entity modified by both features and an edge connecting entities of different features. These elements indicate how these features are probably related to each other.

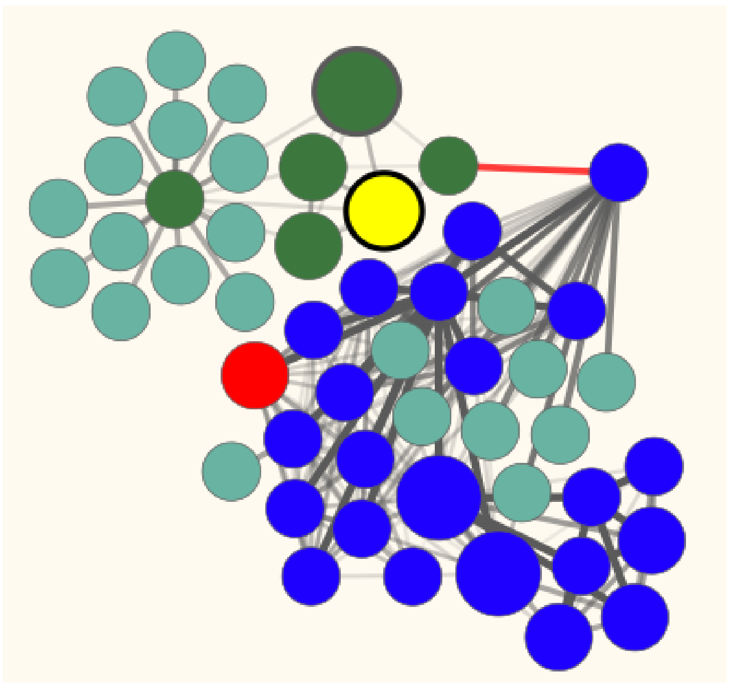


Figure 5.4: Visualization of files from two features with a file changed in both features (red node) and two highly logical-coupled files (red edge).

Color The color of a node represents the feature to which the source code entity belongs; therefore, nodes of the same color are feature-related. Users could use this visualization to locate the entities which implement a specific feature. Entities that are related to multiple features are red. This information indicates that this is a possible candidate for a feature coupling. Thus, if the developers plan to change these entities, they could influence another one as well. Additionally, if two features have many red nodes that

[17] https://github.com/d3/d3-force

connect them, the features are probably tightly coupled and it could be hard to change one without unintentionally modifying the other one. The red color is used as a signal color, which should draw the user's attention.

Nodes of the currently investigated commit are also highlighted with a black border, so users see what nodes have changed at the current time, which gives the user the capability to see which parts of a feature have changed. Additionally, it can help developers identify the cause of a feature coupling because if a red node has a black border, it might indicate that this commit introduced a new coupling.

If the user searches for specific elements, the element is highlighted in yellow, which helps them quickly identify the entities.

Size The node's size represents how often the entity has been changed in the feature, which provides additional information for developers during feature location. As larger nodes are entities that were changed more often for the feature, this could indicate that these nodes are more critical than others, providing developers a good starting point to investigate the feature on the source code level.

Edges Edges represent the logical coupling of entities, where the opacity visualizes the strength of the coupling. If an edge connects two different features, the edge is red to indicate that it might be a critical coupling. Edges connecting entities from different features could indicate that if one feature needs modification, the change might also need to be propagated to the other feature. Additionally, edges with a high logical coupling are red if at least one edge's node is related to a feature. Again, the color red should make users more aware of the high relevance of those relations.

If the coupled entities do not belong to an investigated feature, they are cyan. These entities might not belong to a visualized feature; however, it might still be required to modify them.

The visualization of logical coupling should support developers with impact analysis, as it shows them which entities are coupled to each other. When developers modify a certain entity, they might want to check out the logical couplings, as it shows them possible candidates which the change might affect. Additionally, the logical coupling can be leveraged to unveil dependencies between entities that the software maintainer is unaware of.

Interaction The user can interact with the visualization in multiple ways. Users can zoom in and out or pan within the visualization frame. By dragging a node, users can change a node's position; connected edges are updated accordingly. When clicking a node, a tooltip presents information about the source entity. For example, it shows more details about the node, like the full path of a file, the features the node is related to, and feature commits in which this node has changed. This information is necessary to provide the user more information. For example, the visualization shows the user that two entities are highly logically coupled. Now, he/she wants to investigate how these entities were changed. Then they could use the list in the tooltip to quickly find the commits which contain the changes responsible for the coupling.

When clicking on an edge, the connected nodes are highlighted yellow, and the tooltip shows the support and confidence value of the related entities. Additionally, the tooltip of logical coupled entities shows a list of commits in which both entities were modified.

Filtering Users can configure the visualization to show features on the package, file, or method abstraction level. Users can reduce the number of nodes and focus on only relevant entities by filtering for entity names. Additional input fields allow the users to search and highlight nodes or to exclude specific nodes. This is highly necessary, as the visualization quickly contains many elements, and users would suffer from overplotting.

It is also possible to specify the minimum required logical coupling for edges to be presented in the visualization, preventing overplotting. However, visualization always includes the logical coupling of feature-related source code entities, so these edges cannot be excluded. Additional filters would improve the utility of the tool more, though the scope of this work limited the implementation to the most essential filters.

CHAPTER 6

Evaluation

In order to evaluate the prototype, a scenario-based expert evaluation has been conducted. During these evaluation sessions, participants should use the prototype to solve different tasks and give feedback. This section describes the goal and scope of the evaluation sessions and gives detailed insights into the process.

6.1 Goal

The goal of these sessions is to elaborate if the prototype and the implemented approach satisfy the information needs developers have. The result of the evaluation sessions should provide answers to the research questions, which are described in Chapter 1.3.1.

6.2 Scope

The scope of the evaluation is bound to the visualization in the front end. Participants were therefore not required to set up the project or index the repositories on their own.

Due to the limited scope, features are assumed to be well-defined, documented, and already configured in the prototype. The scenarios were prepared and set up to serve as a starting point in the sessions. Therefore, the evaluation excluded the management of the features.

The evaluation setup excludes the Feature-, Issue- and Commit view from the user interface to reduce the mental workload on participants during the sessions. Additionally, the pre-configuration guarantees that each participant has the exact prerequisites to solve the tasks.

6.3 Session Process

In each session, one participant and two instructors attended a remote Zoom meeting. Figure 6.1 gives an overview of the meeting's course. Process steps with a red background indicate that the instructor is in control of the test environment. Accordingly, blue steps indicate that the participant has remote access to the test environment. However, the instructor could always help the participants if they had issues with the controls.

In general, the sessions lasted between 70 and 90 minutes, dependent on how long participants needed to solve the scenarios and how detailed they elaborated their feedback.

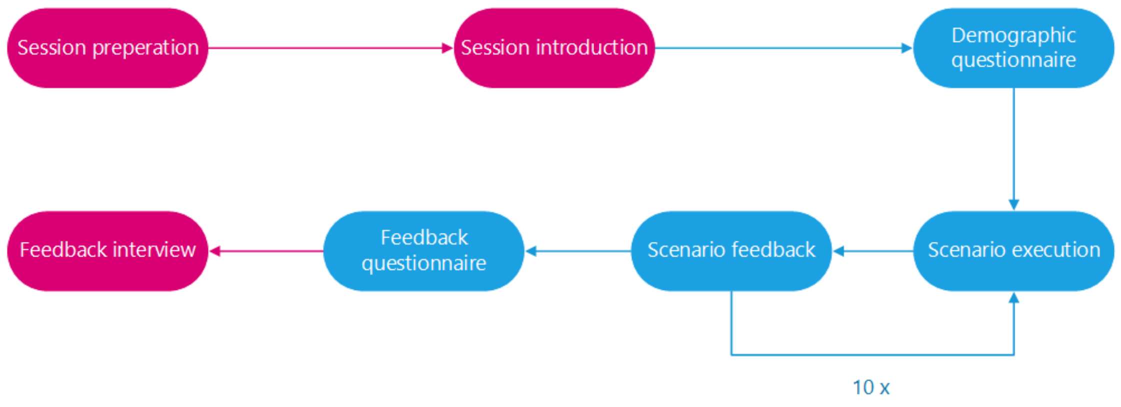


Figure 6.1: Evaluation session overview

6.3.1 Roles

Three people, an instructor, a supervisor, and a participant, attended an evaluation session. Their duties are described below:

Instructor

- Setup test environment on local environment
- Shared screen and gave remote control to the participant
- Introduced participant to the tool and process
- Supported participants with usability issues like zooming, which might occur due to the remote control
- Checked correctness of participant answers
- Answered participants questions
- Wrote notes during the sessions

Supervisor

- Moderator of the sessions

- Prepared Zoom meetings and managed recordings
- Answered participants questions
- Supervised the meeting

Participant

- Solved scenarios
- Answered questionnaire
- Switched between scenarios and questionnaire
- Provided general feedback

6.3.2 Session Preparation

The prototype was set up in a local environment on the instructors' machine. Before the session, the instructor started up the local environment and opened the prototype and the questionnaire in the web browser. The instructor set up the start state of the evaluation session and hid interface elements that were not part of the evaluation, e.g., the Feature-, Commit- and Issue view.

6.3.3 Session Introduction

At the beginning of the session, the instructor explained the topic and the prototype to the participant. Then, the instructor showcased the prototype's interface and how users can configure the visualization to their needs. The instructor demonstrated the prototype's functionality by visualizing two features that did not occur later in the evaluation scenarios.

The introduction took 15 to 20 minutes per session, and participants had the chance to ask questions all the time but could not use the prototype on their own in that time. After the introduction, and when the participants had no more questions, the remote control was hand over to the participant.

6.3.4 Demographic Questionnaire

Before performing the evaluation tasks, the participants filled out a demographic questionnaire regarding how long they work in the software engineering field and their experience and usage of VCS and ITS. The questionnaire can be found in the Appendix 9.1.8.

This information is helpful to understand why participants have a particular opinion on the tool. For example, if they do not know VCS or ITS at all, they might not understand what kind of information the tool presents them. If they do not use a VCS or ITS in their work, they might not see a benefit of the visualization and the approach at all.

As this questionnaire contained only five questions, this step took the participants less than five minutes.

6.3.5 Scenario Execution

After the participants filled out the demographic questionnaire, they performed ten evaluation tasks which are described in the Appendix 9.1.8.

Each scenario is independent of the others so that they can be performed in random order. However, the order was fixed in all sessions, which guaranteed that more complex tasks were performed later when the participants were more familiar with the concepts and the tool. Additionally, as the primary goal of the evaluation session is neither finding out if users could solve all tasks correctly in a minimal time nor investigating the usability of the interface, one can argue that there is no need to permute the scenario order.

To start with the following scenario, participants proceeded in the questionnaire, which showed them the next task to solve. After reading the scenario instruction, the participant switched back to the prototype's user interface and selected the scenario they needed to work on. Figure 6.2 shows the specific user interface to setup the scenario. The prototype then configured the application state to match the scenario's start state and the participant could investigate the visualization to solve the task.

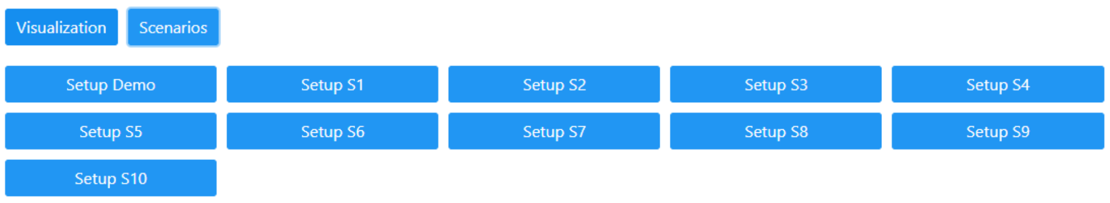


Figure 6.2: By using these buttons participants could easily setup the required scenario

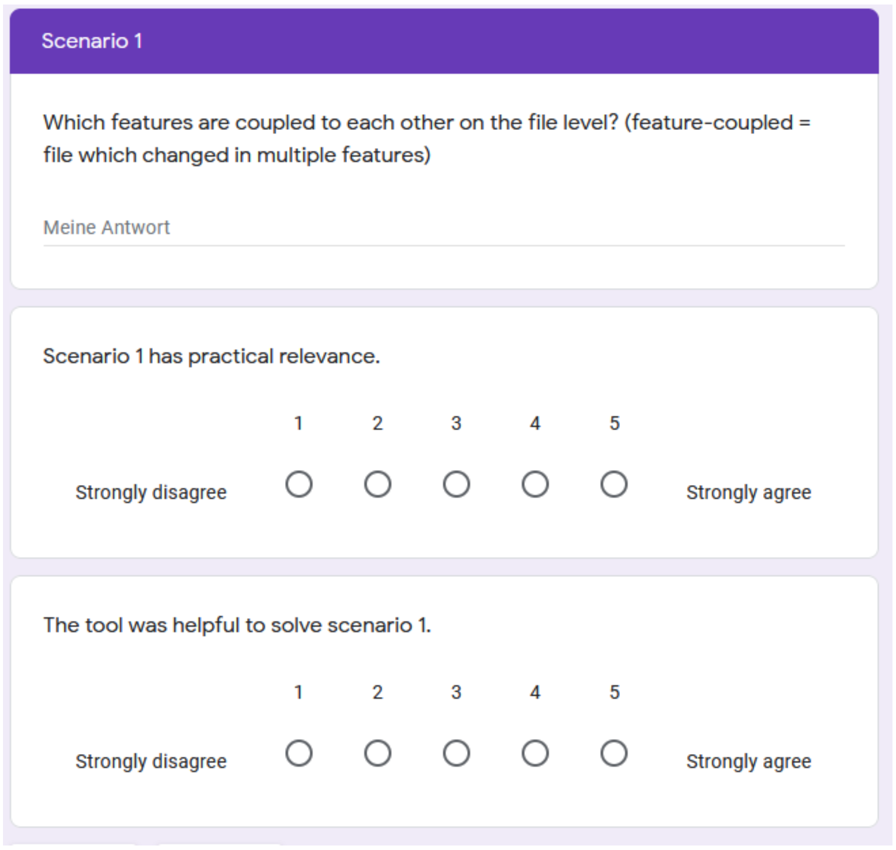
The participants could always switch back to the task description or ask the instructor for help when they had questions. When the participants thought they found the right solution, they communicated it to the instructor, who then confirmed the correctness.

6.3.6 Scenario Feedback

After finishing a scenario, the participants gave feedback on the relevance of the scenario and how much the tool helped them to solve it. Figure 6.3 shows how a scenario is presented to the participant in the questionnaire. The participants should give feedback to the scenario by using a Likert scale from 1 to 5 for the following questions:

- Scenario X has practical relevance.
- The tool was helpful to solve scenario X.

Besides that, the instructor and supervisor encouraged the participants to elaborate their answers in more detail verbally. The instructor took notes of verbal answers as well.


Scenario 1

Which features are coupled to each other on the file level? (feature-coupled = file which changed in multiple features)

Meine Antwort

Scenario 1 has practical relevance.

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

The tool was helpful to solve scenario 1.

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

Figure 6.3: Task and feedback of Scenario 1

Feedback Questionnaire

After the participants finished the last scenario, they filled out a questionnaire, asking for general feedback. The questionnaire consists of eleven questions which can be found in the Appendix 9.1.8. Participants explained their answers verbally and could also switch back to the visualization to show what they meant.

Answering these questions took around 15 minutes, dependent on how much participants elaborated their opinion.

6.3.7 Feedback Interview

After the participants answered the feedback questionnaire, the instructor asked them for their general opinion about the prototype and approach and if they want to add something. This interview aimed to receive qualitative feedback and address feedback that is uncovered by the questionnaire. During the interview, the instructor took notes of the participant's statements. This interview lasted between 5 to 10 minutes, and the session ended afterward.

6.4 Test project

As the evaluation should be based on real-world data, the software project had to meet specific requirements. The following list describes these requirements:

Open-source The project must operate under an open-source license. This is required to provide traceability to the data used in the evaluation and not run into any legal issues.

ITS The project is required to have an actively used ITS, where it is possible to set relations between tickets. The tickets and their relations are necessary to cluster issues to certain features easily.

Git repository The project must use a Git repository. Most commit messages must include a reference to the addressed ITS ticked ID. This reference is required to relate source code to issues and features.

Java The primary programming language of the source code must be Java. This is necessary in order to parse source files to an AST to extract the changed methods of commits.

Documentation The software's features must be documented, so it can be tracked to the issues that implement those.

ActiveMQ (AMQ)[1] from the Apache Foundation[2] was selected because they met all requirements and set issue relations in their Jira[3] tickets regularly. They also provide issue references in almost all commits messages for several years, simplifying tracking source code back to issues and features.

Most of the scenarios use features that were introduced or changed in the release version 5.13[4]. However, it was impossible to use features only from this release because the data could not be used for specific scenarios.

[1]https://activemq.apache.org/
[2]https://www.apache.org/
[3]https://issues.apache.org/jira/projects/AMQ
[4]https://activemq.apache.org/new-features-in-513

CHAPTER 7

Results

This chapter presents the outcome from the evaluation, which is described in Chapter 6 and consists of the qualitative questionnaire results and the quantitative feedback from the interview. It also describes threats to validity in Section 7.4 in the end.

7.1 Demographics

Six software developers participated in the evaluation, where three of them also participated in the requirement evaluation from Chapter 4.5. As Figure 7.1 shows, the participants have experience in software engineering between 7 and 18 years, which also includes education. The average experience of the participants is 12.67 ($\sigma = 3.67$) years and the median is 13 years.

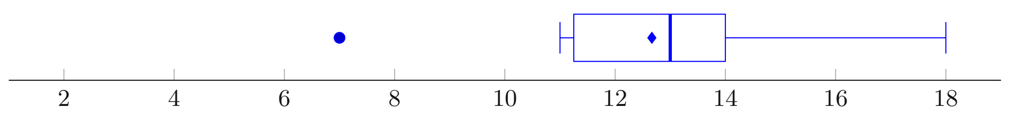
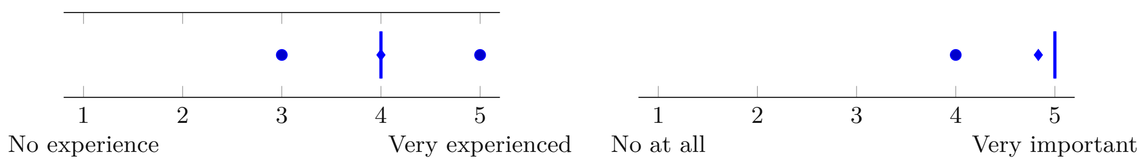


Figure 7.1: Participants' experience in software engineering

The participants should rate their experience with VCS and how essential VCS are for their work from 1 (not at all) to 5 (expert/essential) in the participant questionnaire. The results are visualized in Figure 7.2. They rated their experience as 4 ($\sigma = 0.62$) on average, which means they are familiar with such systems and understand the most concepts of VCS. On average, they rated the importance of VCS for their work as 4.83 ($\sigma = 0.4$).

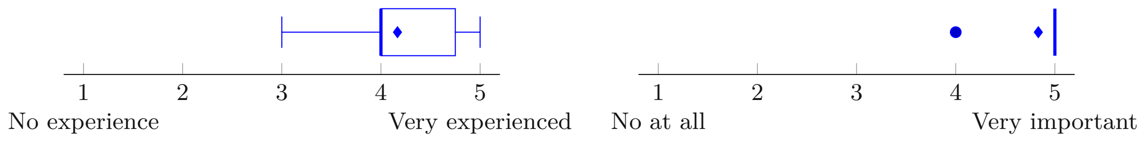
Similarly, the participants should rate their experience with ITS and how important those are for their work. Figure 7.3 show the result from the participants. They rated their experience with ITS 4.17 ($\sigma = 0.76$) on average, implying they are familiar with the



(a) Experience with VCS (b) Importance of VCS

Figure 7.2: Participant's answers regarding VCS

concepts used in those tools. Some of the participants also mentioned they have already used multiple different ITS. They rated the importance of ITS for their work with 4.83 ($\sigma = 0.41$) on average.



(a) Experience with ITS (b) Importance of ITS

Figure 7.3: Participant's answers regarding ITS

7.2 Scenario Evaluation

This section describes the quantitative and qualitative feedback the participants provided for the scenarios.

Scenario 1

Which features are coupled to each other on the file level? (feature-coupled = file which changed in multiple features)

	∅	σ	median
Rated scenario relevance	4.17	0.75	4
Rated tool utility	4.17	1.33	5
Success rate	1	0	1
Completion time [sec]	99.7	39.9	93.5

Addressed user story: 2

All participants could solve Scenario 1 in 99.7 seconds on average. The participants could quickly identify the feature-coupled files by clicking on the red nodes. However, they had to click on each node individually to find out which features were coupled.

Therefore, some participants wished for another representation of coupled entities, which showed them which features are involved in the coupling without any user action. For example, they suggested using colors of all involved features instead of red because it would also show if more than two features are coupled through that entity. These participants rated the tool's utility worse than participants who were satisfied with the visualization. On average they rated the utility with 4.17 ($median = 5$), with a high standard deviation of $\sigma = 1.33$.

They rated the relevance to 4.17 ($\sigma = 0.75$), which indicates that developers see a specific purpose of visualizing feature couplings. However, one participant questioned the relevance of the visualization if there is a long temporal distance between the causing commits. Some participants also questioned if the visualization would work for visualizing more features.

Other participants even wished for a more abstract visualization that shows them directly the couplings of features and not the files.

Scenario 2

Since when are the features "Runtime configuration" and "Dynamic network" coupled?

	∅	σ	median
Rated scenario relevance	4.17	1.17	4.5
Rated tool utility	4.67	0.52	5
Success rate	1	0	1
Completion time [sec]	108.17	19.15	105.5

Addressed user story: 3

The participants solved this scenario in two different ways. In the first approach, some participants navigated to the latest commit and checked the commits in the tooltip of the coupled files. Because they had again to click on all nodes and remember the commit date, this approach required more mental workload.

The other approach was to use the commit list in the sidebar, select a commit, and see if the visualization includes a red node. Most of the participants using this approach figured out the first possible commit, which can cause a feature coupling. The participants using this approach were slightly faster than the others.

The participants rated the relevance with 4.17 ($\sigma = 1.17$) on average. This scenario was less relevant for a few participants because they were not interested in the past or a particular point when a coupling was introduced, but what happened afterward.

The participants rated the tool's convenience to solve the scenario with 4.67 ($\sigma = 0.52$). Some people criticized that they had to click through the visualization or the commit list to get this information. They said the tool should easily provide that information and wished for more aggregation in the commit list. However, the participants using the commit list were generally more satisfied with the tool to support them.

Scenario 3

There is a file in the "AMQP" feature which has a logical coupling to another file with a confidence > 0.85. Please name the related files.

	∅	σ	median
Rated scenario relevance	4	0	4
Rated tool utility	3.33	1.51	3
Success rate	1	0	1
Completion time [sec]	252.3	82.95	251

Addressed user story: 5

All participants could solve this scenario, but it took them longer than most other scenarios. This was because the participants were unfamiliar with the tool's interface and did not know when the tool would highlight edges. They also struggled with the user interface. The first problem was that the tooltip of the relation was not intuitive enough, or it was hard to detect the desired edge. The second problem was that participants struggled to detect which confidence values were relevant to them and asked for help. Finally, the scenario data also had another pitfall because the relevant files were also feature-coupled, which caused some confusion among the participants.

Another problem was that the visualization displayed a lot of less relevant edges. Some participants wished for visualizing only the most important edges or wished for more filters. For example, one participant wanted to filter only a specific interval of logical couplings and wished for a minimum and maximum value filter. Therefore, the participants, rated the tool's utility to 3.33 ($\sigma = 1.51$), with a median of 3.

However, they find the general idea of visualizing logical coupling useful and rated the relevance of the scenario to 4 ($\sigma = 0$). They liked that it could support them in finding couplings between source code and configuration files, which cannot be easily detected with structural coupling. One participant said he/she liked the idea, but given that a software project is well maintained, unit tests can also help detect important relations in the source code.

Scenario 4

You have the feature-coupled features "Runtime configuration" and "Dynamic network" in your system. Please name one of the files which caused the coupling.

	∅	σ	median
Rated scenario relevance	4	1.26	4.5
Rated tool utility	4.5	1.22	5
Success rate	1	0	1
Completion time [sec]	42.2	32.3	26

Addressed user story: 2

This scenario, which is similar to Scenario 1, could also be solved by all participants. In general, they could solve this scenario a little bit faster than Scenario 1 because the participants were more familiar with the visualization at this point.

Except for one participant, all were completely satisfied with the tool support ($\varnothing = 4.5$, $\sigma = 1.26$, $median = 5$). The one wished for more aggregation for the visualization nodes. Another participant liked the visualization's provided insight but also wished to highlight if features are coupled due to high logical couplings. The problem with the existing implementation is that there are too many edges to find the most relevant ones.

They rated the relevance of the Scenario with 4 ($\sigma = 1.26$) on average. Participants mentioned that the relevance is dependent on the particular source files, but the information can be helpful when they plan to refactor a feature. For example, if there are many red nodes, this could indicate that some entities should be refactored to decouple highly coupled features.

Scenario 5

How many packages have changed in the "Dynamic network" feature?

	∅	σ	median
Rated scenario relevance	3.17	1.17	3
Rated tool utility	4.33	0.82	4.5
Success rate	1	0	1
Completion time [sec]	32	15.1	27.5

Addressed user story: 1

This scenario could be solved by all participants in a short time frame. The participants considered the practical relevance as 3.17 on average, but the standard deviation of $\sigma = 1.17$ is also high. Some of the participants saw a purpose of visualizing feature-relation on the package level because it can support them to review the software architecture. These people rated the relevance higher than the other participants, who did not see a use case for the visualization on package level. They toughed it would not support them for typical maintenance tasks, like resolving bugs or improving features. Therefore, they rated the scenario less relevant.

Overall the participants rated the utility of the tool 4.33 ($\sigma = 0.82$). Some criticized that the visualization should show the number of packages for each feature. One participant said he/she preferred the visualization more if it would also visualize the package hierarchy.

Scenario 6

You have two feature-coupled features "Runtime configuration" and "Dynamic network" in your system. There is a method which caused the coupling. Please show the diff / hunk, which caused the coupling.

	∅	σ	median
Rated scenario relevance	3.83	0.98	4
Rated tool utility	4.17	0.75	4
Success rate	1	0	1
Completion time [sec]	204.7	34.52	204

Addressed user story: 2

Scenario 6 was the most challenging task for most participants, and due to some usability issues, they needed multiple minutes to solve it. One problem was the slow rendering performance of visualizing features on the method level. The second problem was that they often investigated the wrong commit, where the instructor had to point the participant in the right direction. The third problem was that they had to review a large commit, and it was hard for them to navigate to the correct diff. Some participants did not memorize the file's name, which they had to check, so they had to close the diff and check for the node's name again.

The participants rated the relevance of the scenario to 3.83 ($\sigma = 0.98$) on average. For one participant, this scenario was already too fine-grained and said he is not interested in the methods which caused the coupling, just in the files. However, one participant was very enthusiastic about the scenario and how such a tool can help to solve it. He/she said that this feature could drastically decrease the time to find bugs.

In general the participants rated the utility with 4.17 ($\sigma = 0.75$). Almost all participants criticized that they had to find the required diff on their own. They hoped the tool would present only the diff of the method they clicked on, or the tool would jump to the diff of the clicked method.

Scenario 7

Which feature(s) might be affected when editing MessageDatabase.java?

	∅	σ	median
Rated scenario relevance	4.5	0.84	5
Rated tool utility	4.67	0.52	5
Success rate	1	0	1
Completion time [sec]	59.8	35.7	39

Addressed user story: 4

The participants solved this scenario by using one of two different approaches. After searching for the entire file name, some participants used the "include"-filter, which showed them only one node. Some were not sure if the visualization was working correctly because there was only one node. After the instructor confirmed everything was fine, they quickly could solve this scenario.

Other participants used the highlighting functionality, which changed the color of nodes that matched the search term. All participants could solve this task quickly and rated the tool's utility for such scenarios with 4.67 ($\sigma = 0.52$). Some participants suggested using a different approach to highlight the nodes because when the node's color becomes yellow, which indicates the node matched the search query, they lose the information to which feature the node belongs, and the user has to open the tooltip to find it out.

The participants rated the practical relevance of the scenario with 4.67 ($\sigma = 0.52$).

Scenario 8

Which file has a high (> 0.75) logical coupling with PendingMessageCursor.java?

	∅	σ	median
Rated scenario relevance	3.67	0.82	3.5
Rated tool utility	4.67	0.52	5
Success rate	1	0	1
Completion time [sec]	78.7	38.1	66.5

Addressed user story: 5

Like the previous scenario, the participants either used the include-filter or the highlighting functionality to find the relevant file. However, the participants using the highlighting had more difficulties. After these participants entered the file name, they were confused because the visualization highlighted four nodes instead of one, which they expected. The reason for that behavior is that the filter is implemented without an exact match, which caused other files, like *AbstractPendingMessageCursor.java* to be highlighted as well. Nevertheless, the participants rated the tool's utility to 4.67 ($\sigma = 0.52$) on average.

However, they do not find this scenario as relevant as the others, so the average rating was 3.67 ($\sigma = 0.82$). That is because participants think that relationships would be obvious or they could find the relation in the source code quickly. The scenario data might influence their opinion because the searched coupling was between *PendingMessageCursor.java* and *AbstractPendingMessageCursor.java*. For some participants, the coupling between those files is too apparent, so they might not see the benefit of logical coupling.

Scenario 9

Which methods have changed to fix the bug described in issue AMQ-8097?

Addressed user story: 5

	∅	σ	median
Rated scenario relevance	4	1.55	4.5
Rated tool utility	4.5	0.84	5
Success rate	1	0	1
Completion time [sec]	43.5	26.8	38

Some participants used the commit filter to search for commits related to the ticket *AMQ-8097*, while the others directly investigated the visualization. However, on average, all participants could solve this scenario quickly and rated the utility with 4.5 ($\sigma = 0.84$).

When it comes to relevance, the participants rated 4 on average with a high standard deviation of $\sigma = 1.55X$. The people who were not convinced by the scenario's relevance said that they either were not interested in the methods which fixed a bug or that other tools provided similar functionality, and therefore they would not need a visualization to help them.

Scenario 10

Which features have changed in Nov 2015?

	∅	σ	median
Rated scenario relevance	3.67	4	1.51
Rated tool utility	4.33	0.82	4.5
Success rate	1	0	1
Completion time [sec]	80.8	36.7	75.5

Addressed user story: 3

All participants solved this scenario by using the commit filters. Even if they did not know the time format to use in the filters, they could solve it quickly. The participants rated the tool's utility to solve it with 4.33 ($\sigma = 0.82$). Some participants wished for better aggregation in the commit list.

Most participants were not interested in such information and consider this scenario as less relevant. On average they rated the relevance 3.67 ($\sigma = 1.51$). One participant said that the scenario has no relevance for developers but considers it very relevant for project managers. Also, some participants mentioned that they do not need this information in the visualization because they can extract it directly from the ITS.

7.3 Participant Feedback

This section describes the results of the twelve questions from the general feedback questionnaire. The feedback is not specific to a specific scenario but asks for general feedback on the approach and the prototype. Table 7.1 shows the analysis of the participant's answers for the feedback questionnaire, which is described in the Appendix 1.

ID	Question	∅	σ	median
FQ1	It was easy to identify features.	4.5	0.82	5
FQ2	It was easy to identify feature couplings.	4.67	0.52	5
FQ3	The visualization helped me to find out what caused a feature coupling.	4.5	0.55	4.5
FQ4	The visualization helped me to find out since when features are coupled.	4.33	0.82	4.5
FQ5	The evolutionary aspect of the visualization is important.	4.17	0.98	4.5
FQ6	The visualization is too complex.	2.67	1.63	2.5
FQ7	Visualizing feature-related source code provided meaningful insights.	4.5	0.84	5
FQ8	Visualizing logical couplings provided meaningful insights.	4	0.89	4
FQ9	The visualization of packages is useful.	3	1.55	3
FQ10	The visualization of files is useful.	4.67	0.52	5
FQ11	The visualization of methods is useful.	4.5	0.84	5

Table 7.1: Qualitative feedback from the participants

Most participants found it easy to identify features in the tool (∅ = 4.5). However, they wished for a better legend. The legend described the features and the color of feature couplings, logical couplings, and highlighted nodes. Therefore, participants got easily confused, especially initially, when they did not know if a particular color represents a feature or some coupling. Another problem was that the legend was not always visible due to being placed in an accordion UI element. The participants wished that the legend was always visible in the visualization.

The participants could discover feature couplings quickly (∅ = 4.67). However, they wished for more aggregation in the visualization. Many participants wished for another abstraction level, which visualized the feature on a top-level. They suggested that an edge or a Venn diagram could visualize a feature-coupling between features, and when they zoom in, they see what caused the coupling. Some people were also questioning if the implemented approach would work for multiple features. They argued that the color represents a feature coupling but does not say anything about the involved features.

Most participants thought the visualization helped them find the cause of a feature coupling (∅ = 4.5). However, some people also said that it could be better as it does not scale well. For example, if multiple features are visualized, the user would have to click through all red nodes to determine if and how certain features are coupled. It would be better if the visualization would present only relevant source code entities and relations.

Most participants felt supported by the tool to find out since when features are coupled (∅ = 4.33). However, some people criticized that they still had to go through the project's history to find the relevant commits. They wished for better aggregation in the commit list (combining multiple commits) or highlighting commits that introduced a coupling.

For most participants, the evolutionary aspect of the visualization is important, though a few people said that they are only interested in the current situation, e.g., if a coupling currently exists in the code, but not when it got introduced or how it evolved. Those people concede that the feature evolution could be purposeful for retrospective, but they do not see it necessary for developers' daily work.

When asked if the visualization is too complex the results were diverse as Figure 7.4 shows. It was easy for some participants to identify the important parts of the visualization,

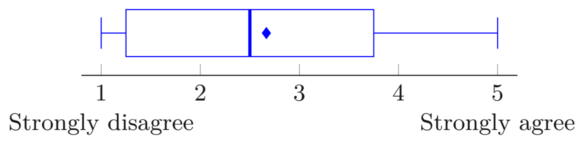


Figure 7.4: The visualization is too complex.

while others said the visualization should be reduced. They disliked, for example, that too many edges were visualized and that there was no aggregation. It was also not transparent enough to them when a logical coupling was included in the visualization, or an edge was highlighted. They wished for more filtering and aggregation options. For example, they wished to visualize logical couplings within a certain confidence interval.

For most participants visualizing feature-related source code provided meaningful insights (∅ = 4.5). Some participants find the tool theoretically useful but questioned whether the developers regularly link the issues and commits together.

Finally, the questionnaire asked the participant how useful they find the different abstraction levels. The result, shown in Figure 7.6, indicates that participants find the visualization for files (∅ = 4.67), more useful than for methods (∅ = 4.5) and packages (∅ = 3). They said the file level is the most useful to them because it is the most natural entity to them. In their opinion, methods are already very fain-grained, and they prefer investigating those directly in the IDE. The least helpful abstraction level is package because it strongly depends on the quality of the software projects, and developers do not see a benefit from this. However, some participants think it can help them to plan or review the software architecture. Those people rated the usefulness of the package abstraction level higher than the rest.

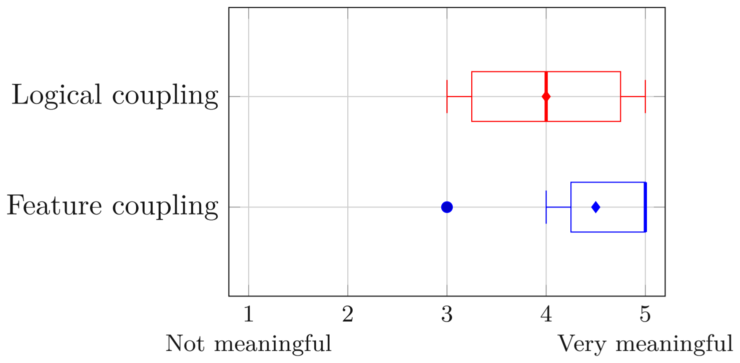


Figure 7.5: Rated meaningfulness of different coupling types.

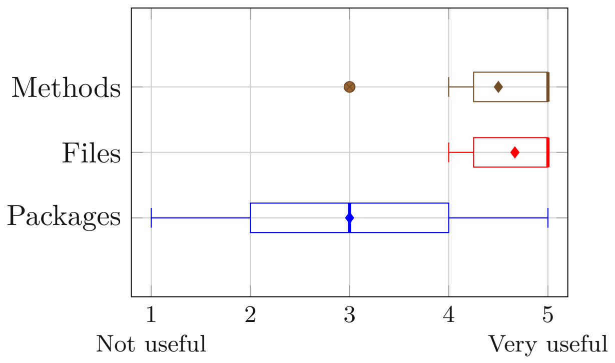


Figure 7.6: Rated meaningfulness of different abstraction levels

7.4 Threats to Validity

This chapter describes the threats to validity of the performed evaluation. The listed points could have influenced the result of the evaluation.

7.4.1 Number of Participants

Only three participants took part in the requirement engineering interviews and six developers participated in the evaluation of the tool. The problem with such a small sample size is that it cannot cover all the different opinions and experiences in the diverse audience group of software developers.

Additionally, single participants have a significant impact on the overall quantitative feedback. Conducting this evaluation with more participants was not possible due to the limited scope of this work. However, in combination with the qualitative feedback, the evaluation still provided valuable insights and can serve as a starting point for a more extensive survey.

7.4.2 Scenario Setting

A scenario-based expert evaluation has been conducted to evaluate the approach. Within this evaluation process, a real-world software project was used in which participants solved artificial scenarios. Although these scenarios are related to real-world tasks, like finding which files implement a feature, these scenarios cannot cover all the different challenges developers usually face.

Another problem with the scenarios is that they are intentionally kept simple because the duration of the sessions is limited. In the real world, developers can face much more complex challenges.

Also, some scenarios were formulated or structured differently than developers would face them in the real world because scenarios need to be verifiable based on the project's provided data. For example, when testing the logical coupling, the participants had to filter for specific confidence values. In the real world, they would have to come up with their own value or try out different values.

Another aspect, which might have influenced the result regarding the benefit of different abstraction levels is that most scenarios use the file abstraction level. In total, six scenarios used the file abstraction level, two used the method abstraction level and only one used the package level. This imbalance is because the file level is the most natural one in this context; for example, commits usually list the changed files and not the changed methods or packages. Additionally, scenario data might not include changed packages or methods as other files were edited in commits.

7.4.3 Applicability on other Software Projects

The evaluation is based on the AMQ project from the Apache Foundation, which is a well-maintained software project. As the scenarios are based on this project's ITS, VCS, and source code, the result is influenced by the discipline and experience of the project's maintainers.

Though the concepts of feature coupling and logical coupling are programming language agnostic, it might be that the evaluation of other projects could lead to different results, dependent on the structure of source code and how much the maintainers use the ITS and VCS. However, evaluating with different projects was not possible due to the scope of this work.

Another influence might be the programming language. For example, for technical reasons, the prototype is only able to index Java source code. Therefore, visualizing packages would not make sense in other projects, as the concept of packages might not exist in the used programming language. Therefore, the visualization might be more beneficial for a Java developer than for a developer who is not familiar with packages because the concept does not exist in his/her preferred language.

Also, the logical coupling on the package level was very low in the AMQ project and was not significant. The problem with the logical coupling on the package level is that it highly depends on the packages' size, as larger packages are likely affected by more commits. Thus, based on how the logical coupling is calculated, it is less likely that packages will have a high logical coupling value because it is related to the number of commits containing the package.

7.4.4 Configuration of Features

In the scenarios, not all features were visualized to keep the scenarios simple and decrease the rendering time of the tool. However, in the real world, the participants would probably investigate more features at once.

Not all AMQ features have been configured because this had to been done manually and was not possible due to the project's size. In a real-world project, the developers would have to configure all features manually.

In the evaluation, it is assumed that features are well-defined and documented. Additionally, the evaluation assumes that tickets in the ITS elaborately describe features and that commits are always linked to the issues they address. However, these assumptions cannot generally be made in real projects where these best practices might not be applied accordingly, resulting in incomplete results.

Setting up and configuring the features could be done by an algorithm, but this is out of the scope of this work and can be addressed in future research, as Chapter 9.1.6 describes.

7.4.5 Questionnaire Formulation

Some questions of the feedback questionnaire allowed a broad range of understanding. For example, the participants were asked if a visualization *provided meaningful insights* or *was helpful*. Some participants said that this depends on the user's role. For example, they considered the visualization of packages more useful for a software architect but not for developers. When a feature was changed the last time might be more interesting for a project manager than for a developer.

When participants communicated that they would rate differently for other roles, the instructor encouraged them to rate from the perspective of a software developer and took notes on what they would rate for different roles.

CHAPTER 8

Discussion

This chapter interprets the results of the conducted scenario-based expert evaluation to finally answer the research questions stated in Chapter 1.3.1.

RQ-1: How purposeful do experts rate a visualization concept based on the combination of co-changes from VCS and ITS data regarding

- *feature location*
- *impact analysis*
- *feature evolution*

The results of the requirement questionnaire indicate that participants have a high interest in the proposed visualization to support them during feature location and impact analysis. In that regard they rate the purpose of visualizing feature-relation the highest ($\varnothing = 4.67$), logical coupling second ($\varnothing = 4.33$) and structural coupling last ($\varnothing = 3.67$). Participants showed little interest in structural coupling as this information is directly available in the source code. In practice, participants would most likely use a visualization utilizing logical coupling and feature relation.

Developers prefer visualizations on the file abstraction level, independent of the coupling because it is the most natural unit. Developers would use other abstraction levels for different purposes. For example, some participants said the package abstraction level might be interesting for reviewing the architecture, or the method level might indicate what test cases they should add.

Participants said a visualization of feature evolution could help them determine when a feature coupling was introduced ($\varnothing = 4.67$); however, they are more interested in the current state of features. Participants would investigate the evolution of features most likely on the file level ($\varnothing = 4.33$) then on the package ($\varnothing = 3.66$) or the method ($\varnothing = 3.33$) level.

Most participants do not see a meaningful purpose on utilizing issue relations for feature location ($\varnothing = 3.33$) or impact analysis ($\varnothing = 3.33$). Participants were not convinced that this approach would work in practice, as according to their statements, issues are often maintained poorly, and therefore they would not trust this kind of information.

RQ-2: How to visualize feature coupling evolution by utilizing source code co-change and issue tracking data?

A visualization tool has been implemented by using incremental prototyping to answer RQ-2. Before this process started, technical considerations were made based on the requirements of the visualization. The chosen technology stack was a good choice, as all requirements could be implemented without significant problems. However, participants mentioned that the tool would be more valuable if it was integrated into an IDE as a plugin. However, this was not possible due to the limited scope of this work, and a web frontend could also be used to evaluate the implemented visualization.

The main challenges during the implementations were the indexing process and the data retrieval performance. The indexing process was complex to implement because it required mining the whole software history and storing which changes were made. During this process, ASTs had to be calculated to determine which packages and methods have been changed. This process could last multiple hours, dependent on the size of the project. To circumvent this issue to a certain degree, large commits were not processed. It was taken care of during the scenario selection that participants had not to investigate those commits. The data retrieval performance was the second challenge, as there were many relations to be queried. This problem could be circumvented by a certain degree by limiting the number of features to visualize and using a high confidence value as a filter, as this reduces the number of edges that have to be queried.

The implementation had some usability issues. However, the instructor encouraged the participants to ignore those in their rating, except they significantly impacted solving the given tasks.

These issues sometimes lead to increased completion times, for example, in Scenario 6, where participants faced poor loading performance for loading many methods and also struggled with the navigation to find the correct diff.

Another limitation of the provided tool is that it only works for projects with a single source code repository due to the prototype character. However, many real-world projects consist of a multi-repository setup. In a multi-repository project, the logical coupling cannot be determined across different repositories. Still, it would be possible to use issue relations to determine feature relations.

Considering multiple repositories in the tool could require additional filters or even a new abstraction level for repositories in the visualization. However, these aspects were not covered in this thesis but should be addressed in future work.

RQ-3: How does the proposed visualization idea support developers with

- *locating features*
- *analyzing feature change impact*
- *comprehending feature evolution*

In order to answer RQ-3, the sub-research questions are answered first.

RQ-3.1: How do developers benefit from visualizing feature-related source code entities for feature location and impact analysis?

The average completion time of scenarios where participants have to find out which files belong to a feature or which features are implemented in a specific file was relatively short. For example, participants solved Scenario 5 in 32 seconds on average and Scenario 7 in 60 seconds. In FQ8, participants said that visualizing feature-related source code entities provided meaningful insights to them ($\varnothing = 4.5$). They appreciated that it could support them to identify non-source files which are part of a feature; for example, they mentioned SQL files or configuration files.

In Scenario 4, participants had to find the cause of feature couplings, which can be important files for impact analysis as these files address multiple features. Participants could solve this scenario in 42 seconds, which is relatively fast compared to the overall average completion time of 100 seconds.

Feature relation was easy to understand, and the prototype visualized this information sufficiently. However, the visualization could improve the color-coding for feature-coupled entities because coupled entities were just red nodes, not indicating which features were affected. In that regard, the visualization shows potential for improvement.

Another benefit is that the visualization can be used to determine which features or source code should be refactored. For example, if there are many red nodes between two features, this could mean that these features are highly coupled, and it could make sense to decoupled them with a refactoring.

RQ-3.2: How do developers benefit from visualizing logical-coupled source code entities for feature location and impact analysis?

Like feature coupling, also logical coupling provided meaningful insights for participants ($\varnothing = 4$) as it showed which entities were related to each other. Participants mentioned that visualizing logical-coupled source code entities could help find couplings between configuration files and source code entities.

According to the results, feature relation provided more meaningful insights than logical coupling. This outcome is interesting as participants rated both coupling types similarly in the requirement interviews. This could have multiple reasons. First of all, the sample size is meager, as only three people participated in the requirement interviews. Therefore, the outcome of these interviews is less meaningful than the results from the evaluation sessions.

Secondly, some participants said that if they are already familiar with the source code, the logical relation would be obvious to them anyway. This circumstance might be an outcome of the performed scenario, where the investigated coupling was based on an inheritance relation which was obvious already from the file names. If the participants had to investigate less obvious relations, e.g., between a configuration and a source file, participants might rate the logical coupling even more beneficial.

Finally, it was harder to investigate logical coupling than feature relation due to the visualization's interface. Also, Scenario 3, which addresses logical coupling, had some pitfalls because the files which participants had to investigate were not only strongly logical-coupled but also feature-coupled, which confused.

RQ-3.3: What are the advantages and drawbacks of different abstraction levels of source code entities for feature location and impact analysis?

According to the results and feedback, the most significant benefit provided the file abstraction level ($\varnothing = 4.67$). This abstraction level was the most intuitive one to all participants, and they said this was the most important one.

Participants said that also the method abstraction level provided meaningful insights ($\varnothing = 4.5$). They liked that they could pinpoint why a coupling exists in the source code with the help of that abstraction level. However, some participants said that the visualization became overplotted and should be filtered. For some participants, this abstraction level is also already too fain-grained because they just want to find out which files they should address but would find the methods on their own in the IDE.

On the other side, one participant said that the method abstraction level could drastically reduce the time to find bugs. Therefore, the method abstraction level can be used for specific use cases, like finding bugs or finding out what causes a feature coupling, but it is less relevant due to the increased complexity it adds.

The least preferred abstraction level is package ($\varnothing = 3$). Most participants said they do not have a use case for that kind of information. Still, some participants mentioned they would use it as software architects to review the architecture or plan a more extensive refactoring. However, these use cases are a minority.

Another problem with the package abstraction level was that it does not show how many files within each package are coupled. Also, the logical coupling between packages is depended on the packages' size and it was always very low in the data.

The results of the evaluation sessions are in alignment with the outcome of the requirements interview, where participants were also most interested in the file abstraction level.

RQ-3.4: How does visualizing feature evolution support developers with feature comprehension?

Participants could solve Scenario 2, in which they had to find out since when a coupling occurred rather quickly ($\varnothing = 105.5sec$). They rated the scenario relevance with 4.17 on average, and the tool supported them well ($\varnothing = 4.67$).

Also, Scenario 10, where participants had to find out when certain features were changed, could be solved quickly ($\varnothing = 75.5sec$). However, they rated this scenario less relevant ($\varnothing = 3.67$) then the average ($\varnothing = 3,92$).

Participants mentioned that they would use the tool to determine which features were changed during specific periods like previous Scrum sprints. They would use it to investigate how the software architecture has evolved.

However, a few participants mentioned that the evolutionary aspect is not important because their primary interest is in the current state of the software. These participants only want to know if features are coupled and which parts are coupled. However, it was not relevant for their work when the coupling was introduced or how it evolved.

Developers rated the tool support to find out since when features are coupled with $\varnothing = 4.33$. However, the tool could still be improved in that regard. One problem was that the visualization re-rendered every time the user changed the investigated commit, which caused the participants could not track the entities they were interested in. The rendering of the visualization took some time, which made it tedious to investigate history. Some participants also wished for better aggregation in the commit list to better understand which feature was modified. For example, they suggested composing commits together.

Summarizing the insights from the sub-research questions, the visualization supported developers locating features because they could easily find out which source code entities belong to visualized features without being familiar with the source code. One advantage over typical approaches, like investigating structural coupling, is that developers can also identify which configuration files belong to specific features or are highly logically coupled to other source code entities.

The visualization supported developers with impact analysis because it highlighted feature-coupled files or high logical couplings between nodes where changes might have a significant impact. The results indicate that the tool helped to identify those entities; however, the visualization could be improved in that regard. For example, the visualization could use better color codings and reduce noise by only visualizing essential couplings.

Regarding feature evolution comprehension, the visualization supported the developers by showing to which feature a commit belongs. The tool provided a commit list, where users can see when commits for certain features were made, who made them, and how the source code was changed. They could also visualize a feature at any given commit, which participants used to find out since when features are coupled (Scenario 2) or what concrete change caused a coupling (Scenario 6).

Participants used the commit list efficiently to determine which features were changed in a specific period or who made changes to specific features. However, this kind of information was less relevant for most participants. Scenario 10, which addressed such use cases, was rated less relevant $\varnothing = 3.67$). Participants said they could use the VCS and ITS directly to get this information when needed and therefore do not see a benefit for the tool, except it is integrated into these tools.

Additionally, participants were less interested in such information but mentioned it might be helpful for project managers.

CHAPTER 9

Conclusion

This chapter concludes the results of this thesis and elaborates where future work can build on.

This thesis hypothesizes that a visualization of features on the source code level supports developers during software maintenance with feature location, impact analysis, and feature evolution comprehension. In that regard, the existing state-of-the-art provides visualizations to developers, but these tools either visualize features, not on the source code level, or visualize source code but not in the context of features. Therefore, the approach followed in this thesis leverages the data from VCS and ITS to visualize features on the source code level. The proposed visualization aims to answer the information needs regarding feature location and impact analysis developers have during software maintenance. The elaborated approach uses issues and commits to establish a connection between features and the source code and uses logical coupling to emphasize important relations between source code entities and features.

Before the implementation of the prototype started, the approach was evaluated in expert interviews. The result of those interviews was that developers rated this approach purposeful and would benefit from visualizing feature-relation and logical coupling. The outcome also indicates that developers are less interested in a visualization of features using structural coupling because they could investigate these kinds of relations directly in the source code using their IDE. Another reason was that participants already knew similar visualizations and thought that structural coupling was hard to understand as this visualization often suffered from overplotting. The visualization of structural coupling was therefore not implemented in the prototype.

After the visualization was implemented with incremental prototyping, it was evaluated in scenario-based expert evaluation sessions. In these sessions, participants had to solve software maintenance-related tasks by utilizing the visualization.

After the evaluation sessions, participants answered a questionnaire, in which they rated the prototype. Participants rated the ease to locate feature-related source code with 4.5 on average on a scale from 1 to 5. They could also quickly identify coupled features ($\varnothing = 4.67$), what caused the coupling ($\varnothing = 4.5$), and find out since when features were coupled ($\varnothing = 4.33$). These results indicate that the visualization of features on the source code level based on VCS and ITS data supports developers with feature location, impact analysis and feature evolution comprehension.

The visualization helped participants with feature location because they could locate them in the source code quickly without being familiar with the codebase. Additionally, the visualization highlighted source code entities that were changed in multiple features or had a high logical coupling. This information was helpful for participants to be aware of the impact changes might have on other features. Another advantage of that proposed approach is that developers can discover dependencies between source files and non-source files like configuration files. These kinds of dependencies are hard to identify by using structural coupling. However, the tool could be improved in visualizing those dependencies by improving the representation of those change-prone entities. For example, at first glance, participants could only see that a feature coupling exists but not which features were coupled or what caused the coupling. They had to investigate further to gather the desired information. Another problem participants faced was that the visualization suffered from overplotting in some cases and they wished for better aggregation and filtering functionalities.

The tool allowed developers to visualize features throughout the project's lifecycle, which helped participants determine since when features were coupled with each other or how a feature has evolved. However, some participants showed less interest in the evolution than others, as they were only interested in the current state of the application. Additionally, one problem with the visualization in this regard was that visualizing a feature at a different commit caused a new data retrieval and re-rendering of the graph. This made the investigation of evolutionary information more tedious and could cause that participants could not spot the difference between snapshots.

One open question is how to configure the software features in the visualization in a scaleable manner. For the evaluation, the author of this thesis configured features beforehand, which means the issues and commits were related to the features they addressed. The tool supports this process by leveraging issue references in commit messages and utilizing issue relationships, but still requires manual work. In the real world, developers would have to set up the features on their own, which can be tedious if the VCS and ITS suffer from poor data. Future work could elaborate on how this process can be improved.

9.1 Future work

This chapter describes which parts were not addressed by this work and where future research can build upon.

9.1.1 Extend Evaluation Scope

Due to the limited scope of this work, the evaluation had different shortcomings, which could be addressed by future research to increase the validity of the outcome.

For example, the evaluation omits the configuration of features, which had to be done manually by selecting issues and commits related to them. Therefore, this works assumes that the scope and definition of features are known.

Future research could include this process in the evaluation, as it would be required in real maintenance scenarios. Alternatively, the research could also focus on how the assignment can be automatized.

Another weakness of the result is the low number of developers who participated in the evaluation sessions. Future work could conduct the evaluation with more participants to increase the significance of the results. Additionally, the evaluation should investigate different software projects because, in this work, only one software project was used in the evaluation process.

9.1.2 Utilizing Different Coupling Types

This work determines feature relation based on issue references from commit messages and logical coupling. However, other types of coupling, like structural, semantical, or dynamical coupling, could be used to determine the feature relation of source code entities. Future research could investigate how purposeful developers rate other couplings and how this data can be adequately visualized.

9.1.3 Support for Multiple Repositories

As the current implementation only supports projects with a single repository, future work can investigate how the approach can be applied to projects with multiple repositories, which are commonly used in real software projects, for example, in micro-service architectures. Open questions are how features can be located across different repositories and how source code couplings can be determined across those boundaries.

9.1.4 Better Aggregation and Filtering

Multiple times in the evaluation sessions, participants mentioned they would prefer better aggregation and filtering capabilities. Further research can investigate which improvements add value for users without making the tool too hard to understand. Such research could investigate how much customization developers need.

A good starting point for this research is the wishes participants expressed during the sessions. For example, some participants wished for an additional abstraction level of features, which gives participants a quick overview.

9.1.5 Discovery of Hotspots

Participants mentioned that the visualization could be improved by only showing them the most important entities. For example, the visualization should be configurable to show only coupled features or coupled entities.

Another improvement is to find the most coupled features, which requires sophisticated data analysis. Future work can investigate the most interesting hotspots for participants and research how this information can be extracted from the given data in a reasonable time.

9.1.6 Programmatic Composition of Features

In the current implementation, the features need to be configured manually. The tool supports the developers with maintaining the features by adding all commits of issues and related issues to features. The disadvantage of this process is that it is very time-consuming and error-prone, as the developers need to find the issues which implement features on their own and also link issues together in the ITS. Additionally, it assumes that the feature scope is completely defined and known by the developer and that it is accordingly described and linked in ITS tickets and commits.

9.1.7 Improved Visualization Usability

The usability of the implemented visualization can be improved in multiple ways. The following sub-chapters describe different usability issues participants faced during the sessions and how they can be fixed.

Rendering Performance

The rendering performance of the visualization was slow, which could cause participants to lose focus on the task they wanted to solve. Especially when navigating through different commits, this could cause participants to reject the tool due to usability issues, even if they would benefit from the provided information.

The rendering performance is dependent on two parts. The first part is loading the data from the database. Future work could investigate better data structures or different database systems to speed up the query performance of large data sets. The second part is the rendering performance in the frontend. The current implementation uses a force layout of d3.js. Another algorithm to render the graph might perform faster or result in a better graph layout.

Tooltip

The tooltip was often misplaced, requiring the users to pan in the visualization to see the entire content. Additionally, users often had to close the tooltip because it covered some important elements and could not be dragged.

Another problem with the tooltip was that the content was hard to understand. Participants struggled to determine to which features source code entities belonged and often got confused by the different directions of the confidence values. Future work should improve the representation of the information to be easier to understand and still provide insights.

Zooming

The prototype provided zooming in the visualization. However, there was no minimum or maximum zoom level, and users got easily lost due to the remote control. Future work could improve the zoom usability by providing a better zoom control and leveraging the zooming to change the abstraction level or aggregation level.

For example, the visualization could show only features when zooming out, and the more the user zooms in, the more fine-grained the visualization gets. As an alternative, when zoomed out, only the most essential entities and edges could be visualized. On a higher zoom level, more and more entities could be visualized. With that functionality, users could quickly investigate the data in their preferred degree of detail.

9.1.8 Multi-color Nodes

One problem participants faced to find out which entities caused feature couplings was that all coupled entities were shown as red nodes, which did not indicate which features were coupled. Participants had to open the tooltip to determine which features were involved in the coupling, which could be tedious if there are many red nodes in a visualization. Therefore, one improvement could address how the visualization presents feature-coupled entities visually. For example, multi-colored nodes could be used to indicate which features are involved in the coupling. However, these important nodes must still leap out of the rest of the application, and users should detect those entities quickly. Future work could elaborate on how this could be accomplished.

Edges

Participants often struggled when they had to investigate logical coupling due to a large number of edges. Even if they chose a high confidence value, the filter did not remove enough edges to avoid overplotting. It was also unclear why edges were kept in the visualization, even if the confidence value was below the threshold or why an edge was highlighted as important. As edges in the visualization refer to logical couplings, they could have different meanings dependent on if they are directed or bi-directional.

Future work could investigate more on the visualization and semantics of edges. For example, directed edges could be leveraged to represent the confidence value in a particular direction, while undirected edges represent support values. Additionally, future work can elaborate on how much customization users prefer to customize the edges, e.g., showing direction or highlighting an edge.

Legend

One problem was that the legend described colors which either represented features or contextual information, like feature-coupling, highlighted nodes or logical-coupled entities. The legend should distinguish between features and contextual information and might include additional information (like how many entities there are per feature). Further, the legend was placed in an accordion above the visualization, which caused the participant to expand the accordion to view it every time. It would have been better to move it to a position where it is always available.

Visualization Settings and Filters

As for the legend, the filters were also placed in the accordion, which was closed per default. In some cases, this caused problems, as participants could not immediately find the abstraction level switch. Therefore, the filters and the general visualization settings should always be visible.

Additionally, the filters could be extended to provide more powerful options. For example, the filters are implemented by checking if the node names contain the search term. More powerful implementations would allow the user to specify how the filtering should be applied, e.g., participants could choose where the search term must be placed (beginning/end), choosing between case-sensitive and case-insensitive, or using regular expressions.

Further, filters for the logical coupling could be added as well. The tool provides a minimum confidence value filter; however, the prototype always included edges of feature-related source code entities in the visualization. A more sophisticated tool would empower the users to customize the behavior of the logical coupling filter to their needs. For example, it would also allow a maximum filter, consider only co-changes within a specific period, or filtering for support value instead of confidence. There are many different options, and it would be interesting what benefits such powerful filtering capabilities offer to developers and how they would leverage them.

List of Figures

List of Tables

Acronyms

AMQ ActiveMQ. 74

AST Abstract Syntax Tree. 10, 54, 59, 74, 90

CBO Coupling between Objects. 9

CSM Conceptual Similarity between Methods. 14

DAO Data Access Object. 54

FDL feature decoupling level. 26

FDSM feature dependency structure matrix. 26, 101

FLT feature location techniques. 8, 9, 28

GXL Graph eXchange Language. 41

IA Change impact analysis. 2, 9, 15, 28

ICP Information flow-based coupling. 9

IDE Integrated Development Environment. 51, 84, 90, 92, 95

IR Information Retrieval. 8, 28, 29

ITS Issue Tracking Systems. 4, 15–18, 23, 25, 27, 28, 33, 47, 54, 71, 74, 82, 98

LSI Latent semantic indexing. 14

NLP Natural Language Processing. 8, 28

RFC Response for a Class. 9

SCM Source Code Management. 18, 20

VCS Version Control Systems. 4, 8, 12, 15, 18, 24, 28, 30, 33, 39, 47, 54, 71

Bibliography

[1] Renato Novais, Camila Nunes, Caio Lima, Elder Cirilo, Francisco Dantas, Alessandro Garcia, and Manoel Mendonça. On the proactive and interactive visualization for feature evolution comprehension: An industrial investigation. *Proceedings - International Conference on Software Engineering*, pages 1044–1053, 2012.

[2] Michael Fischer, Martin Pinzger, and Harald Gall. Analyzing and relating bug report data for feature tracking. *Proceedings - Working Conference on Reverse Engineering, WCRE*, 2003-Januar:90–99, 2003.

[3] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: A taxonomy and survey. *Journal of software: Evolution and Process*, 25(1):53–95, jan 2013.

[4] Martin P. Robillard. Topology analysis of software dependencies. *ACM Transactions on Software Engineering and Methodology*, 17(4):1–36, 2008.

[5] Jaejoon Cafeo, Bruno B.P. and Cirilo, Elder and Garcia, Alessandro and Dantas, Francisco and Lee. Feature Dependencies on Evolving Software Product Line : An Exploratory Study on Change Propagation. *Inf. Softw. Technol.*, 69:37—-49, 2016.

[6] Bogdan Dit, Michael Wagner, Shasha Wen, Weilin Wang, Mario Linares-Vásquez, Denys Poshyvanyk, and Huzefa Kagdi. ImpactMiner: a tool for change impact analysis. *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 540–543, 2014.

[7] Renato Lima Novais, André Torres, Thiago Souto Mendes, Manoel Mendonça, and Nico Zazworka. Software evolution visualization: A systematic mapping study. *Information and Software Technology*, 55(11):1860–1883, 2013.

[8] Maximilian Steff, Barbara Russo, and Guenther Ruhe. Evolution of features and their dependencies - an explorative study in OSS. *International Symposium on Empirical Software Engineering and Measurement*, pages 111–114, 2012.

[9] Adam Vanya, Rahul Premraj, and Hans Van Vliet. Interactive exploration of co-evolving software entities. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, pages 260–263, 2011.

[10] Robert Galliers. Choosing appropriate information systems research approaches: A revised taxonomy. *The Information Research Arena of the 90s*, page 155–173, 01 1991.

[11] D.R. Graham. Incremental development and delivery for large software systems. In *IEE Colloquium on Software Prototyping and Evolutionary Development*, pages 2/1–2/9, 1992.

[12] Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Using structural and textual information to capture feature coupling in object-oriented software. *Empirical Software Engineering*, 16(6):773–811, 2011.

[13] Nemitari Ajienka, Andrea Capiluppi, and Steve Counsell. An empirical study on the interplay between semantic coupling and co-change of software classes. *Empirical Software Engineering*, 10 2017.

[14] Muslim Chochlov, Michael English, and Jim Buckley. Using changeset descriptions as a data source to assist feature location. *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation, SCAM 2015 - Proceedings*, pages 51–60, 2015.

[15] Iso/iec/ieee international standard - systems and software engineering–vocabulary. *ISO/IEC/IEEE 24765:2017(E)*, pages 1–541, Aug 2017.

[16] Michael D. Feist, Eddie Antonio Santos, Ian Watts, and Abram Hindle. Visualizing Project Evolution through Abstract Syntax Tree Analysis. *Proceedings - 2016 IEEE Working Conference on Software Visualization, VISSOFT 2016*, pages 11–20, 2016.

[17] Vipin Balachandran. Query by example in large-scale code repositories. *2015 IEEE 31st International Conference on Software Maintenance and Evolution, ICSME 2015 - Proceedings*, pages 467–476, 2015.

[18] Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *2013 IEEE International Conference on Software Maintenance*, pages 516–519, 2013.

[19] Jonathan I. Maletic and Michael L. Collard. Supporting source code difference analysis. *IEEE International Conference on Software Maintenance, ICSM*, pages 210–219, 2004.

[20] Marco D Ambros, Michele Lanza, and Mircea Lungu. Visualizing Co-Change Information with the Evolution Radar. *IEEE Transactions on Software Engineering*, 35(5):720 – 735, 2009.

[21] Michael Fischer and Harald Gall. Visualizing feature evolution of large-scale software based on problem and modification report data. *Journal of Software Maintenance and Evolution*, 16(6):385–403, 2004.

[22] Gustavo Ansaldi Oliva and Marco Aurélio Gerosa. On the interplay between structural and logical dependencies in open-source software. *Proceedings - 25th Brazilian Symposium on Software Engineering, SBES 2011*, (May 2014):144–153, 2011.

[23] A. Zeller, S. Diehl, P. Weissgerber, and T. Zimmermann. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.

[24] Marco D'Ambros, Michele Lanza, and Romain Robbes. On the relationship between change coupling and software defects. *Proceedings - Working Conference on Reverse Engineering, WCRE*, pages 135–144, 2009.

[25] Marco D'Ambros, Michele Lanza, and Mircea Lungu. The evolution radar: Visualizing integrated logical coupling information. *Proceedings - International Conference on Software Engineering*, (January):26–32, 2006.

[26] Nemitari Ajienka, Andrea Capiluppi, and Steve Counsell. *An empirical study on the interplay between semantic coupling and co-change of software classes*, volume 23. Empirical Software Engineering, 2018.

[27] D. Poshyvanyk and A. Marcus. The conceptual coupling metrics for object-oriented systems. *IEEE International Conference on Software Maintenance, ICSM*, pages 469–478, 2006.

[28] E. Arisholm, L. C. Briand, and A. Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, 30(8):491–506, Aug 2004.

[29] Nayan B. Ruparelia. The history of version control. *SIGSOFT Softw. Eng. Notes*, 35(1):5–9, January 2010.

[30] James W. Hunt and Thomas G. Szymanski. A Fast Algorithm for Computing Longest Common Subsequences. *Communications of the ACM*, 20(5):350–353, 1977.

[31] Hermann Lacheiner and Rudolf Ramler. Application lifecycle management as infrastructure for software process improvement and evolution: Experience and insights from industry. pages 286–293, 08 2011.

[32] Jukka Kääriäinen. *Towards an Application Lifecycle Management Framework: Dissertation.* PhD thesis, University of Oulu, Finland, 2011. CA2: TK804 CA: Cluster9 OH: Väitöskirja SDA: ICT Project code: 24506 PGN: 103 p. + app. 81 p.

[33] Daniel A. Keim and Hans Peter Kriegel. Visualization techniques for mining large databases: A comparison. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):923–938, 1996.

[34] Herman Chernoff. The use of faces to represent points in k-dimensional space graphically. *Journal of the American Statistical Association*, 68(342):361–368, 1973.

[35] Dirk Beyer and Ahmed E. Hassan. Animated visualization of software history using evolution storyboards. *Proceedings - Working Conference on Reverse Engineering, WCRE*, pages 199–208, 2006.

[36] Michael J. Decker, Christian D. Newman, Michael L. Collard, Drew T. Guarnera, and Jonathan I. Maletic. A timeline summarization of code changes. *Proceedings - 3rd International Workshop on Dynamic Software Documentation, DySDoc3 2018*, pages 9–10, 2018.

[37] Ran Mo, Yuanfang Cai, Rick Kazman, and Qiong Feng. Assessing an architecture's ability to support feature evolution. *Proceedings of the 26th Conference on Program Comprehension*, pages 297–307, 2018.

[38] R. L. Novais, C. Nunes, A. Garcia, and M. Mendonça. Sourceminer evolution: A tool for supporting feature evolution comprehension. pages 508–511, 2013.

[39] Trevor Savage, Meghan Revelle, and Denys Poshyvanyk. FLAT3: feature location and textual tracing tool. *2010 ACM/IEEE 32nd International Conference on Software Engineering*, 2:255–258, 2010.

[40] Jonathan Buckner, Joseph Buchta, Maksym Petrenko, and Václav Rajlich. JRipples: A tool for program comprehension during incremental change. *Proceedings - IEEE Workshop on Program Comprehension*, pages 149–152, 2005.

[41] Xiaoxia Ren, Barbara G. Ryder, Maximilian Stoerzer, and Frank Tip. Chianti: A change impact analysis tool for programs. *Proceedings - 27th International Conference on Software Engineering, ICSE05*, pages 664–665, 2005.

[42] Marco Torchiano and Filippo Ricca. Impact analysis by means of unstructured knowledge in the context of bug repositories. *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 1, 2010.

[43] Gerardo Canfora and Luigi Cerulo. Jimpa: An eclipse plug-in for impact analysis. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, (June 2014):341–342, 2006.

[44] Motahareh Bahrami Zanjani, George Swartzendruber, and Huzefa Kagdi. Impact analysis of change requests on source code based on interaction and commit histories. *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 162–171, 2014.

[45] Michael Rath, David Lo, and Patrick Mäder. Analyzing requirements and traceability information to improve bug localization. *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 442–453, 2018.

[46] Daye Nam, Youn Kyu Lee, and Nenad Medvidovic. EVA: A Tool for Visualizing Software Architectural Evolution. *40th International Conference on Software Engineering*, pages 53–56, 2018.

[47] Barrett Ens, Daniel Rea, Roiy Shpaner, Hadi Hemmati, James E. Young, and Pourang Irani. ChronoTwigger: A visual analytics tool for understanding source and test co-evolution. *Proceedings - 2nd IEEE Working Conference on Software Visualization, VISSOFT 2014*, (September):117–126, 2014.

[48] Dirk Beyer and Andreas Noack. Clustering software artifacts based on frequent common changes. *Proceedings - IEEE Workshop on Program Comprehension*, pages 259–268, 2005.

[49] Michele Lanza, Harald Gall, and Philippe Dugerdil. EvoSpaces: Multi-dimensional navigation spaces for software evolution. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, pages 293–296, 2009.

[50] Richard Wettel. Visual exploration of large-scale evolving software. *2009 31st International Conference on Software Engineering - Companion Volume, ICSE 2009*, (June 2009):391–394, 2009.

[51] Rodrigo Souza, Bruno da Silva, Thiago Mendes, and Manoel Mendonça. Skyscrapar: An augmented reality visualization for software evolution. 01 2012.

[52] Stuart M. Charters, Claire Knight, Nigel Thomas, and Malcolm Munro. Visualisation for informed decision making; from code to components. *ACM International Conference Proceeding Series*, 27:765–772, 2002.

[53] Michael Balzer, Andreas Noack, Oliver Deussen, and Claus Lewerentz. Software Landscapes: Visualizing the Structure of Large Software Systems. 2004.

[54] Mihai Codoban, Sruti Srinivasa Ragavan, Danny Dig, and Brian Bailey. Software history under the lens: A study on why and how developers examine it. *2015 IEEE 31st International Conference on Software Maintenance and Evolution, ICSME 2015 - Proceedings*, pages 1–10, 2015.

[55] Thomas Fritz and Gail C. Murphy. Using information fragments to answer the questions developers ask. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pages 175–184, 2010.

[56] Holger M. Kienle and Hausi A. Müller. Requirements of software visualization tools: A literature survey. *VISSOFT 2007 - Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, (July 2007):2–9, 2007.

[57] Andreas Winter, Bernt Kullbach, and Volker Riediger. An overview of the GXL graph exchange language. *Lecture Notes in Computer Science (including subseries Lecture*

Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2269:324–336, 2002.

[58] S. Bassil and R. K. Keller. Software visualization tools: survey and analysis. In *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001*, pages 7–17, 2001.

[59] Margaret Anne D. Storey, Davor Čubranić, and Daniel M. German. On the use of visualization to support awareness of human activities in software development: A survey and a framework. *Proceedings SoftVis '05 - ACM Symposium on Software Visualization*, 1(212):193–202, 2005.

[60] M L Collard, M J Decker, and J I Maletic. srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration. In *2013 IEEE International Conference on Software Maintenance*, pages 516–519, sep 2013.

Appendix

Requirement Questionnaire

Gender				
☐ Female		☐ Male		
For how long do you work in the software engineering field?				_____ years
Version Control Systems are essential for my work.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
Version Control Systems are easy to use.				
☐ 1 (No experience)	☐ 2	☐ 3	☐ 4	☐ 5 (Very experienced)
I often use Version Control Systems for maintenance purposes, like finding relevant changes or finding bug-introducing code.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
Issue Tracking Systems are essential for my work.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
Issue Tracking Systems are easy to use.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
I often use Issue Tracking Systems for maintenance purposes, like finding the cause of a code change.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)

A visualization of feature-related source code would help me during software maintenance.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
The visualization of feature couplings would help me to identify important source code for maintenance tasks.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
A visualization of logical coupled source code would help me during software maintenance.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
The visualization of feature couplings would help me to identify important source code for maintenance tasks.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
A visualization of structural-coupled source code would help me during software maintenance.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
Please rank what kind of visualization you would most likely use during maintenance tasks.				
Feature relation	Logical Coupling			Structural Coupling
For maintenance tasks, I would use a visualization of feature-related packages/modules.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
For maintenance tasks, I would use a visualization of feature-related files.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
For maintenance tasks, I would use a visualization of feature-related methods.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
For maintenance tasks, I would use a visualization of logical coupled packages/modules.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
For maintenance tasks, I would use a visualization of logical coupled files.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
For maintenance tasks, I would use a visualization of logical coupled methods.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
For maintenance tasks, I would use a visualization of structural coupled packages/modules.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
For maintenance tasks, I would use a visualization of structural coupled files.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
For maintenance tasks, I would use a visualization of structural coupled methods.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
Besides software maintenance, do you see other use cases for one of the above combinations of abstraction level and coupling? Like for architectural considerations or code reviews. If so, please explain what combination (coupling/abstraction level) you would use and for what use case.				

I would use a visualization showing the evolution of a feature to understand how it evolved.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
Visualizing the evolution of feature couplings would help me identifying important source code parts.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
I would use a feature evolution visualization on the level of packages/modules.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
I would use a feature evolution visualization on the level of files.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
I would use a feature evolution visualization on the level of methods.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
Seeing the concrete changes (as diffs) of a commit is essential for the visualization.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
During a maintenance task, I would use related issues together with their changes to know of what I need to be aware of.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
During a maintenance task getting commits of related issues would help me understand the impact of my task.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)

Evaluation Questionnaire

Participant Questionnaire

For how long do you work in the software engineering field?				
Rate your experience with Version Control Systems.				
☐ 1 (No experience)	☐ 2	☐ 3	☐ 4	☐ 5 (Very experienced)
Version Control Systems are essential for my work.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
Rate your experience with Issue Tracking Systems.				
☐ 1 (No experience)	☐ 2	☐ 3	☐ 4	☐ 5 (Very experienced)
Issue Tracking Systems are essential for my work.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)

Scenarios

ID	S1
Description	Which features are coupled to each other on the file level? (feature-coupled = file which changed in multiple features)
Start state	• Selected Features: – AMQP – Dynamic Network – Pending Message Size Metrics – Runtime Configuration • Default filters • Selected latest feature commit
Solution	Runtime configuration & Dynamic network

ID	S2
Description	Since when are the features "Runtime configuration" and "Dynamic network" coupled?
Start state	• Selected Features: – Dynamic Network – Runtime Configuration • Default filters • Selected latest feature commit
Solution	20.10.15 cc81680e

ID	S3
Description	There is a file in the "AMQP" feature which has a logical coupling to another file with a confidence > 0.85. Please name the related files.
Start state	• Selected Features: – AMQP – Single port for all wire protocols • Default filters • Selected latest feature commit
Solution	AmqpNioTransportFactory.java - AmqpTransportFactory.java (confidence: 0,857)

ID	S4
Description	You have the feature-coupled features "Runtime configuration" and "Dynamic network" in your system. Please name one of the files which caused the coupling.
Start state	• Selected Features: – Dynamic Network – Runtime Configuration • Default filters • Selected latest feature commit
Solution	BrokerService.java, VirtualTopic.java or UpdateVirtualDestinationsTask.java

ID	S5
Description	How many packages have changed in the “Dynamic network“ feature?
Start state	• Selected Features: – AMQP – Dynamic Network – JMX Query API • Filter: No include filter, exclude xml; test • Selected latest feature commit
Solution	6

ID	S6
Description	You have two feature-coupled features "Runtime configuration" and "Dynamic network" in your system. There is a method which caused the coupling. Please show the diff / hunk, which caused the coupling.
Start state	• Selected Features: – Dynamic Network – Runtime Configuration • Default filters • Selected latest feature commit
Solution	Source code lines (59-79 & 85-104) in the diff of file UpdateVirtualDestinationsTask in commit cc81680e

ID	S7
Description	Which feature(s) might be affected when editing MessageDatabase.java?
Start state	• Selected Features: – AMQP – Dynamic Network – Pending Message Size Metrics – Runtime Configuration • Default filters • Selected latest feature commit
Solution	Pending Message Size Metrics

ID	S8
Description	Which file has a high (> 0.75) logical coupling with PendingMessageCursor.java?
Start state	• Selected Features: – AMQP – Dynamic Network – Pending Message Size Metrics – Runtime Configuration • Default filters • Selected latest feature commit
Solution	AbstractPendingMessageCursor.java

ID	S9
Description	Which methods have changed to fix the bug described in issue AMQ-8097?
Start state	• Selected Features: – AMQ-8097 – Dynamic Network • Default filters • Selected latest feature commit
Solution	MessageDatabase.resolveClass and SubQueueSelectorCacheBroker.resolveClass

ID	S10
Description	Which features have changed in Nov 2015?

Start state	• Selected Features:
	  – AMQP
	  – Dynamic Network
	  – Pending Message Size Metrics
	  – Runtime Configuration
	  – Single port for all wire protocols
	• Default filters
	• Selected latest feature commit

Solution	Runtime configuration & Dynamic network

Feedback Questionnaire

FQ1: It was easy to identify features.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
FQ2: It was easy to identify feature couplings.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
FQ3: The visualization helped me to find out what caused a feature coupling.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
FQ4: The visualization helped me to find out since when features are coupled.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
FQ5: The evolutionary aspect of the visualization is important.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
FQ6: The visualization is too complex.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
FQ7: Visualizing feature-related source code provided meaningful insights.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
FQ8: Visualizing logical couplings provided meaningful insights.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
FQ9: The visualization of packages is useful.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
FQ10: The visualization of files is useful.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)
FQ11: The visualization of methods is useful.				
☐ 1 (Strongly disagree)	☐ 2	☐ 3	☐ 4	☐ 5 (Strongly agree)

Table 1: Feedback questionnaire