

Visualizing Historical Ownership with Code City Metaphor

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Enrik Ndou

Matrikelnummer 01426910

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Thomas Grechenig

Wien, 25. Jänner 2025

Enrik Ndou

Thomas Grechenig

Visualizing Historical Ownership with Code City Metaphor

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Enrik Ndou

Registration Number 01426910

to the Faculty of Informatics

at the TU Wien

Advisor: Thomas Grechenig

Vienna, January 25, 2025

Enrik Ndou

Thomas Grechenig

Erklärung zur Verfassung der Arbeit

Enrik Ndou

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.

Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 25. Jänner 2025

Enrik Ndou

Kurzfassung

Software-Projekte haben während ihrer Entwicklung ständig Veränderungen, die für die verschiedenen an der Projektentwicklung interessierten Akteure nicht ganz offensichtlich sind. Studien haben gezeigt, dass es Interesse gibt, um zu wissen, wer an welchem Bereich arbeitet oder wie man den richtigen Entwickler findet, den man für einen Teil des Quellcodes kontaktieren soll. Im Rahmen dieser Forschungsarbeit wird ein 3D-Prototyp vorgestellt, um Ownership Fragen zu beantworten, der auf einer der populärsten 3D-Metaphern basiert ist: „Code City“. Forschungsarbeiten haben eine Vielzahl von Visualisierungen vorgeschlagen, die die Code-City-Metapher verwenden, jedoch fehlt eine Studie, die einen Überblick darüber gibt, wie die Metapher verwendet wurde.

Diese Arbeit wird eine Mapping-Studie liefern, die aufzeigt, wie die Code-City-Metapher in der Softwarevisualisierungsforschung verwendet wurde. Die Mapping-Studie enthält und beschreibt eine Liste von mehr als dreißig verschiedenen Visualisierungen, die ähnliche Konzepte wie die Code-City-Metapher verwenden, zum Beispiel, Gebäude und Städte. Die Studie enthält Tabellen, aus denen die wichtigsten Unterschiede, Gemeinsamkeiten und Besonderheiten zwischen den Visualisierungen leicht ersichtlich sind.

Die Erkenntnisse aus der Mapping-Studie und der Umfrage über den Informationsbedarf von Software-Ingenieuren halfen, die Skizzen für den Prototyp zu verbessern. Die Anforderungen wurden durch semistrukturierten Interviews mit Experten auf diesem Gebiet validiert und nach der Analyse der Interviewergebnisse definiert. Auf der Basis dieser Ergebnisse wurde ein Code-City-Artefakt entworfen und implementiert, das das historische Ownership analysiert. Vier Softwareentwickler nahmen an separaten szenariobasierten Experteninterviews teil, um die Visualisierung hinsichtlich ihrer Zweckmäßigkeit für historische und eigentumsrechtliche Fragen zu bewerten. Die Ergebnisse der Interviews zeigten, dass die Visualisierung nützlich bei der Beantwortung von Ownerships-Fragen ist, insbesondere bei historischen Fragen oder Szenarien, bei denen es darum geht, herauszufinden, wer an bestimmten Dateien oder Ordnern gearbeitet hat. Außerdem gaben sie Verbesserungsvorschläge für Szenarien, in denen der Benutzer sehen und vergleichen kann, wann bestimmte Dateien und Ordner implementiert wurden.

Schlüsselworte: *Slide City, Code City, Visualisierung, Code Ownership, Git, Repository, Mapping-Studie, semistrukturierten Interviews, szenariobasierten Experteninterviews*

Abstract

Software projects face continuous changes during development which are not quite obvious to various stakeholders interested in the project evolution. Studies have shown interest in knowing who is working on what area or how to find the appropriate developer to contact for a source code part. This research will introduce a 3D prototype to answer ownership questions based on one of the most popular 3D metaphors: „Code City“. Research papers have proposed a variety of visualizations using the code city metaphor, however, a study is missing that provides a survey on how the metaphor has been used.

This work will provide a study mapping how the code city metaphor has been used in software visualization research. The study mapping includes and describes a list of more than thirty different visualizations using concepts similar to the code city metaphor, such as buildings and cities. The study contains tables where the major differences, similarities, and unique features between the visualizations can be easily seen.

The knowledge gained from the mapping study and the survey on the software engineer information needs helped to improve the sketches for the prototype. The requirements were validated with semi-structured interviews with experts in the field and defined after analyzing the interview results. Based on these results, a code city artifact that analyzes historical ownership was designed and implemented. Four software engineers participated in separate scenario-based expert evaluation interviews to evaluate the visualization on how purposeful the visualization is for historical and ownership questions. The interview results showed that the visualization is very useful in answering ownership questions, especially historical ones or scenarios that involve finding who has been working in some particular files or folder. Additionally, they provided improvement feedback for scenarios where the user can see and compare when particular files and folders were implemented.

Keywords: *Slide City, code city, visualization, code ownership, Git, repository, mapping study, semi-structured interviews, scenario based expert evaluation*

Contents

Kurzfassung	vii
Abstract	ix
Contents	xi
1 Introduction	1
1.1 Problem Statement	1
1.2 Motivation	2
1.3 Expected Results	2
1.4 Structure	3
2 Methodology	5
2.1 State of the Art Review & Systematic Mapping Study	6
2.2 Requirements	7
2.3 Iterative Implementation	8
2.4 Scenario-Based Expert Evaluation	8
3 State of the Art	11
3.1 Information Needs	11
3.2 Ownership and Authorship Effects on Software Quality	13
3.3 2D Historical Code Visualizations	17
3.4 3D Historical Code Visualizations	21
4 Systematic Mapping of Code City Visualizations	29
4.1 Visualizations Using City Metaphor	29
4.2 Code Cities Tables	51
5 Requirements	55
5.1 Vision & Derived Requirements	55
5.2 Proposed Features (Open questions - Semi-Structure Expert Interviews)	59
5.3 Semi-Structure Expert Interview Results	70
6 Implementation	77
	xi

6.1	Architecture	77
6.2	Repository Metadata	80
6.3	Mining Configuration	80
6.4	Visualization Configuration	83
6.5	Live Configuration	86
6.6	Requirements Deviation	89
7	Evaluation & Results	93
7.1	Test Plan	93
7.2	Demographics	94
7.3	Scenarios	94
7.4	Results	96
7.5	Threats to Validity	104
8	Findings	107
9	Conclusion	109
9.1	Future Work	110
	List of Figures	113
	List of Tables	117
	Bibliography	119
	Appendix	129
	Semi-Structured Expert Interview——Questions	130
	Acronyms	137

CHAPTER 1

Introduction

Since the software coding phase and later during its maintenance, the code faces continuous software changes, whether for fixing bugs or introducing new features [1]. Software change is one of the essential characteristics, and understanding its impact has been a challenge since the first software systems. This chapter will give an introduction to the problem that this thesis is going to tackle and the motivation for the topic.

1.1 Problem Statement

Code evolution and ownership are necessary information in medium and large software [2]. Before developers introduce further changes, they need to understand the current implementation. When there are uncertainties or questions about already implemented features, it is helpful to know beforehand the developers who have worked or are still working on the feature. Looking only at the ownership of the current code state may be misleading. If refactoring, code clean-up, or linting has happened lately, the current code ownership will also change. Code ownership may change even if the order of the functions in a file has changed without changing the functions' content. Another factor that needs to be considered to have the correct overview of who has been working on a particular feature is the code history.

Modern IDEs like IntelliJ provide code annotation of historical changes for a selected file or some selected lines. For example, it shows the author's name and the date the developer introduced the line. Other researchers have proposed visualizations that offer a general view of the historical changes, such as an author's behavior [3] or the committers per file [4]. However, it is still hard to understand where a developer has been working or who and when has contributed the most to a particular part of the project.

Researches for handling source code evolution were done in the past by Girba et al., who implemented a 2D visualization „Chronia“. Researches in 3D tackling the code evolution

are [5, 6, 4, 7], where each of them proposes a variation of the code city. The code city metaphor maps various data sources into city objects, such as streets, flat, districts and in some cases even windows [8, 9]. The advantage of the code city objects is that any user is familiar with the city metaphors. Additionally, a code city object such as the buildings can map multiple data sources in one object, for instance, the building can represent a file, while the height can represent the number of lines and the width can represent the number of functions in that file. These factors made the code city metaphor, such as from Wettel et al. [10], one of the most popular research areas in the visualization research field. The first proposals were in 1999 [8], and later Panas et al. added some other concepts [11]. Lately, only VISSOFT21 had five research studies [12, 13, 14, 15, 16] related to the code city metaphor.

However, due to the abundance of code city research, there is no clear research direction. Moreover, the current visualizations do not show „real“ ownership, for example, who was the most extensive file or package owner during different implementation periods. Another essential factor that is missed in the mentioned research papers for source code evolution is the visualization granularity. The granularity depends on who the stakeholders are and what their interests in data are. For example, developers may need the history of some particular code lines during product development. However, during a more abstract discussion over a feature, even the file-level visualization may cause too much information noise when a more general view is necessary.

1.2 Motivation

Understanding how the code has changed and its impact on the system has been challenging since the first software systems [1]. The motivation for the topic is to make it easier for different stakeholders to find and understand information about a software product. Some of the information needs are to know who has implemented a particular software part or in which parts a developer has been working. The abstraction level should also differ depending on the user's interest. For example, a manager that needs to find the right developer to ask for a feature may need a different abstraction level than another developer that needs to find the correct person to ask for a specific part of code. In the first case, the folder and package level ownership makes it easier for the manager, while in the second case, the file level ownership helps the developer find the right person.

The proposed visualization aims to answer such information needs by combining code history and ownership into one interactive 3D visualization. This visualization's main benefits are comprehending the code's history, finding the implementation expert, and seeing collaboration insights.

1.3 Expected Results

This thesis will evaluate the proposed visualization of code ownership with a particular emphasis on historical ownership using the code city metaphor [10]. Because of the rising

popularity of the code city metaphor without having a clear literature review, the first research question is:

RQ1: How has the city metaphor been used in the software visualization research?

The results of the research question will provide an overview of the current state of the art and different categorizations of the code city visualizations. The result will make it easier to see similarities and differences between different code city visualizations. Some categorization examples are the visualization's features or the software aspect, visualized in the code city metaphor.

The thesis will include the design of a prototype to treat the problem stated in Section 1.1. The intention of this prototype is to simplify getting information related to the history of a source code project, for example, who has been working the most in different periods with a file, folder or package. It should support various software stakeholders, therefore, this thesis will include semi-structured interviews of different selected stakeholders to study their needs and extract knowledge on how to shape the prototype. This research goal is formalized into the second research question:

RQ2: How to visualize the historical code ownership with a city metaphor?

After the requirements have been discussed and validated, the artifact will be implemented and applied to a real-world context. This will help to achieve the third research goal of the "implementation evaluation" to describe, explain, and evaluate the effects of the prototype. This has been formalized into the last research question:

RQ3: How purposeful is the visualization for answering historical and ownership questions?

1.4 Structure

After the Introduction chapter, the thesis includes the State of the Art, which contains a review of the information needs and research related to code ownership and code cities. Chapter 2 will describe the methodology used to answer the research question and construct the visualization. Chapter 4 introduces a mapping study of code cities. Chapter 5 discusses the main requirements for the prototype and an overview of the Semi Structured Interview results. Chapter 6 describes the technical aspects of the artefact implementation. Chapter 7 discusses the scenario-based expert evaluation and Chapter 8 discusses the answers to the research questions. The last chapter concludes the thesis and including discussions for future work.

CHAPTER 2

Methodology

This chapter describes the methodology used to build the prototype and answer the research questions. Initially, the literature was reviewed to describe the state of the art of information needs and historical ownership visualizations. To answer the research questions, the thesis used different methodologies:

- **RQ1: How has the city metaphor been used in the software visualization research?**
The methodology used for the first research question is a systematic mapping of visualizations using the code city metaphor.
- **RQ2: How to visualize the historical code ownership with a city metaphor?**
The initial requirements were derived from the vision and the visualizations' state of the art. After the initial requirements were set, they were validated with semi-structured expert interviews. The requirements validation was followed with an iterative implementation.
- **RQ3: How purposeful is the visualization for answering historical and ownership questions?**
The scenario-based expert evaluation was organized to determine the visualization's purposefulness. They included questionnaires about scenarios related to different project levels.

2.1 State of the Art Review & Systematic Mapping Study

The code city research area has been continuously growing, where only VISSOFT2021 had five papers related to code cities, increasing the need for structuring the research area. The systematic mapping study is an adequate research review for this case [17]. To conduct the systematic mapping, the thesis follows the guidelines of Petersen et al. [17].

The state of the art and systematic mapping research involves finding many research papers using the code city metaphor. The primary method to find these research papers is using keywords in search engines like *Google Scholar*¹, *IEEE*², *ACM DL Digital Library*³, and *Mendeley*⁴.

Many different keywords are used to find state-of-the-art research papers for ownership information needs. The main keywords that were used to find information needs related to ownership questions are: „historical ownership,“ „information needs,“ „software developer comparison,“ and „software evolution“. However, the search keywords were also combined with other keywords: „code,“ „source code,“ „evolution,“ and „quality.“ The formed expressions were combined again with the keywords „visualization“ and „statistics“ to find state-of-the-art visualizations that answered information needs in code ownership.

The main keyword used for the systematic mapping of code city visualizations is „code city,“ but also combined with words like „visualization,“ „3D,“ or „software.“ Many research papers on visualization are found in IEEE VISSOFT⁵, which contains research publications each year. The papers are found in *IEEE*⁶ using keywords like „code city“ and „VISSOFT.“

Besides the keyword search, forward and backward snowballing have been used. Forward snowballing involves searching studies that cite the found ones [18], for example, finding all research papers that reference well-known visualizations, such as Wettel et al. [10]. The backward snowballing involves searching studies cited by the found paper [19]. This method is conducted by reading the related work of the found code city studies. The studies using or extending the code city metaphor were selected. A similar approach was also conducted for the state of the art of historical ownership information needs and visualization.

Fifty-six research papers were selected for the mapping study. These papers either introduce a new visualization based on a „code city“ metaphor or extend a former one, for example adding features like roads or wires or mapping other software data to the city. Papers found while searching with the keywords combinations of „3D,“ „visualization,“ and „software“ were checked to see if the visualizations are related to the

¹<https://scholar.google.at/> version of 11.07.2024

²<https://ieeexplore.ieee.org/> version of 11.07.2024

³<https://dl.acm.org/> version of 11.07.2024

⁴<https://www.mendeley.com/search/> version of 11.07.2024

⁵<https://vissoft.info/> version of 11.07.2024

⁶<https://ieeexplore.ieee.org/> version of 11.07.2024

code city metaphor. Although some papers do not use code city metaphor namings such as buildings, streets, or cities, they are still selected if the visualizations are close to a code city, for instance, [20, 21]. Other papers were discarded.

Papers related to ownership, authorship, or code evolution were selected for the ownership and authorship state of the art. However, papers that only mentioned these concepts but did not provide any results or comparison of ownership or authorship to other software code metrics were discarded.

According to the research age, a different approach was followed in selecting historical ownership information needs research papers and code city research papers. The latest studies on historical ownership information needs were prioritized as they describe the recent stakeholders' needs. However, on the code city systematic mapping, older studies are also prioritized for being the first to introduce concepts and features used in code city visualizations.

2.2 Requirements

Requirements definition involves several steps starting from literature research. The state-of-the-art review on information needs related to code ownership help to understand what the user needs are and how they differ to different stakeholders. This review also inspired the vision that this thesis present. The proposed visualization vision is described in more detail in Section 5.1.

However, the vision alone is not enough for a research process. The second step is validating these requirements and defining the remaining requirements. To gather data for the remaining requirements, semi-structured interviews are conducted. According to Kallio et al. [22], interviews are the most used data collection method, whereas semi-structured is the most used format. They found that one of the reasons for the popularity of the semi-structured interview is because the method has successfully enabled reciprocity between the interviewer and participant and allowed improvisations for follow-up questions based on the participant's responses [22]. The follow-up question can be prepared or spontaneous. In both cases, they help to maintain the interview flow and reduce misunderstandings. Based on Kallio et al. [22] results, the semi-structured interview guide development included five inseparable phases:

1. Identifying the prerequisites for using semi-structured interviews

The selected interviewees are experts in the field of software engineering. The interview results will help define additional requirements and shape the visualization. Firstly, the interviewer will explain the vision, and then the interview will be conducted after the participants understand the research and visualization goals.

2. Retrieving and using previous knowledge

Previous knowledge has been retrieved from state-of-the-art reviews (see Chapter 3).

3. Formulating the preliminary semi-structured interview guide

During this phase, the previous knowledge will be used to prepare the list of questions. The questions may have additional follow-up questions to extract as much information as possible to define the remaining requirements.

4. The pilot test interview guide

Kallio et al. [22] analyzed three different pilot test techniques: *internal testing*, *expert assessment*, and *field-testing*. For this topic, the field-testing technique is chosen. The field-testing technique involves testing the preliminary interview with a potential study participant.

5. Presenting the complete semi-structured interview guide

The interview results will be discussed in Section 5.3, and the interview questions are attached at the end of the thesis. The gathered data will be analyzed to validate the existing features and define the remaining ones.

2.3 Iterative Implementation

Two key points on visualization implementation are the 3D implementation on the front-end part and the mining of the ownership Git data for the visualization.

Git data mining involves a Git analysis to find efficient ways to fulfill the requirements. This process may take a lot of time, and it is inappropriate if it is done in real time when the user is using the visualization. Therefore the data mining is done before the visualization is ready to use. The mined data are organized and stored in the database. An important goal, discussed in the Chapter „5“, is that user interactions should not be followed by major delays. The stored data structure is designed so that it is efficient to load when required from the visualization on the front-end part.

The front-end part of the visualization uses these data and shows the 3D Slide City visualization.

The method to build the visualization is the iterative implementation with agile methodology. After each iteration, a validation of the implementation is done. The agile methodology is appropriate as it is very robust in delivering continuous results and adaptable to changing requirements [23].

2.4 Scenario-Based Expert Evaluation

A scenario-based expert evaluation will be conducted to answer the third research question, i.e., to determine the visualization's purposefulness in practice and to validate it. This step is based on technical action research (TAR). TAR is a single case study to validate experimental artifacts and is used in moving from laboratory experiments to unprotected conditions [24]. TAR is artifact driven where the roles during the research are „technical researcher“, „empirical researcher“ and „helper.“ The scenarios are real-world problems

that the visualization helps to solve. The underlying visualized software project may be a project that the user is already familiar with or an unknown one. Additionally, a prepared set of rating questions for the participants will help determine their experience with the visualization, for example, how useful or helpful it is.

CHAPTER 3

State of the Art

This section discusses information needs about code ownership and visualizations that show code ownership information or have similar 3D features as the proposed artefact, for example, like the code city metaphor.

3.1 Information Needs

Begel, Phang, and Zimmerman [25] surveyed inter-team coordination needs for various software team roles in 2009. The results showed that the top two needs are about finding the people responsible or knowledgeable in a feature, API, product, or service. Furthermore, five more of the top ten are also for finding people to get programming advice or discuss a bug. They found out while interviewing thirteen engineers that these needs were mainly accomplished by asking colleges. Colleagues would then route them to other colleges that may be more appropriate to answer their questions if they could not answer themselves. Begel et al. [25] introduced the „Codebook“ framework to find an automatic solution to help developers with those questions. Codebook mines multiple repositories and generates relations between issues and bug tickets, mined from an issue tracking system to people and source code. From this information, Codebook creates a large graph with different node types and relates them. Each node has a list of metadata, which the user can filter with a search keyword.

Hoozizat (Figure 3.1) is a web search application built on top of the Codebook framework. It does not include a visualization to get a general overview. However, it lists relevant information to the search text. Figure 3.1 shows an example when searching with the keyword „foo.“ in Hoozizat. Each column shows a different type of result, from left to right; people, work items, source code, and files. Small photos next to each result show the associates for people, owners, or artifacts; hovering the mouse cursor over the photos shows a tooltip with contact information for that person. Codebook uses a confi-

3. STATE OF THE ART

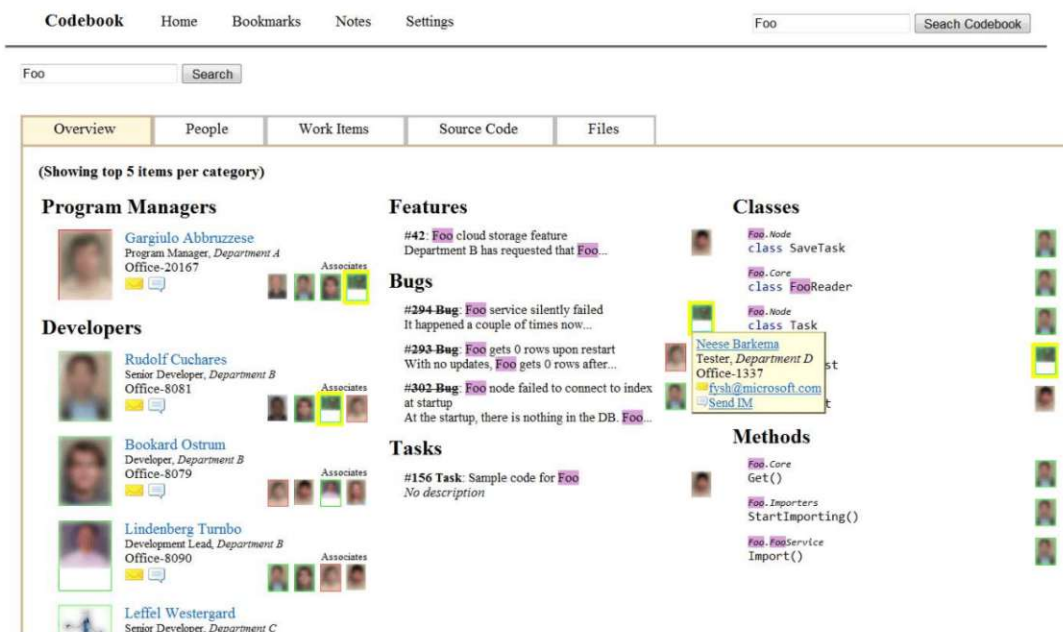


Figure 3.1: Screenshot of Hoozizat [25]

dence score to find the most relevant connection to features or classes containing this word.

In 2014, Begel and Zimmerman [26] did two surveys about questions that software engineers would like data scientists to answer. The first one gathered questions from software engineers about software processes and practices, and the second survey rated those questions. Among the questions for teams and collaborations were those for shared knowledge and code ownership (e.g., „How can I find the best developer to contact for a piece of apparently ‚abandoned‘ code“? [26]).

Fritz and Murphy [2] interviewed 11 developers for questions requiring information from different domains such as source code, change sets, teams, work items, websites, wiki pages, comments on work items, exception stack traces, and test cases. The interviews determined 78 questions and divided them into eight domains. Two of the domains also of interest for the proposed visualization contain questions like „Who is working on what“? (person-specific) and „Changes to the code.“ (code specific).

Fritz and Murphy introduced the information fragment model (a prototype that extends the IBM Rational Team Concept) that composes and shows information from multiple domains (Figure 3.2). However, this model works best for information related to a short period. For example, Fritz and Murphy consider the question „What have people been working on?“ the case where a developer comes back after a week or two of holidays and wants to know what has been happening during this time. Figure 3.2(d) shows how the model answers this question. This approach may be inconvenient if the question is done

for a more extensive time range, for example, for a year or since the project has started.

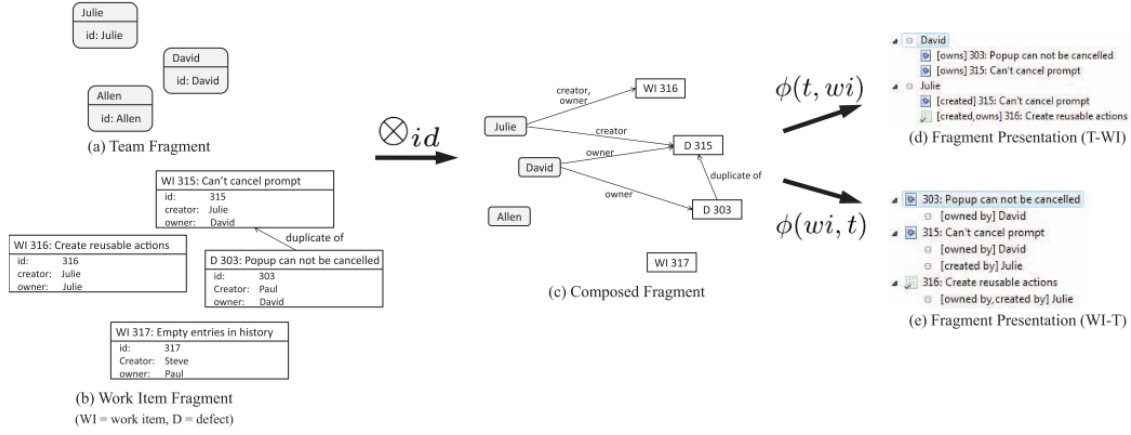


Figure 3.2: Approach to answering the question „What have people been working on?“ [2].

These studies emphasize the need for automated systems to help developers and managers find the relevant people and information quicker on various challenges during software development.

3.2 Ownership and Authorship Effects on Software Quality

Besides answering developers' questions to know who implemented or can share information about a feature, code ownership visualizations may help in defect prediction, such as finding potential bugs or lack of quality code.

Bird et al. [27] examined two Microsoft operating systems, „Windows Vista“ and „Windows 7“, to find out if there exists a relationship between bugs and ownership. They divided the developers of a particular package (binary) into two categories „Major“ and „Minor“ contributors. „Minor“ are the developers who have contributed less than 5% of the commits; otherwise, Bird et al. categorized them as „Major.“ They calculated the „Ownership“ degree of a package by dividing the number of „Major“ by „Minor.“ For example, Figure 3.3 shows the ownership of the example.dll. Top contributor has done 41% of the commits. Five developers are called „Major“ contributor because each of them has contributed more than 5% of the commits. Twelve have committed less than 5% and therefore are considered „Minor“ contributors.

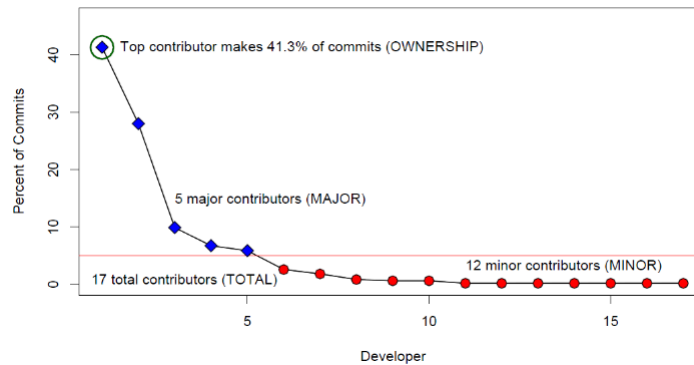


Figure 3.3: Ownership of example.dll by developers [27].

During the interviews, they found that many developers were „Major“ contributors in some binaries but „Minor“ in others. One reason is that the owner of a binary needs to do compatibility changes to another binary where another developer is the owner, thus resulting in the developer being a major contributor to one binary but minor to another. Without proper coordination, this may risk a mismatch between dependencies and a decrease in quality [28], and according to Handerson and Clark [29], minor changes may sometimes result in drastic consequences. Bird et al. [27] found that „Ownership“ relation to code quality was small but had a statistically significant effect in pre- and post-release failures for Vista and pre-release failures for Windows 7. However, including the number of „Minor“ developers in the analysis improved the regression models for both pre and post-release failures to a statistically significant degree. This model supported their hypothesis that the involvement of many „Minor“ contributors in a component resulted in more failures than the components that had fewer. According to Bird et al., using these metrics may help managers make better decisions on the development processes and policies.

On the other side, Foucault et al. [30] tested the same metrics as Bird et al. [27] on seven Java free/libre and open-source software (FLOSS) projects. Unlike Bird et al., they found a weak correlation between code ownership metrics and software quality, and in some projects, non at all. In particular, they found no correlation in the experiments done at the file level; however, they found some correlation at the package level. The package level correlation corresponds to the binaries level used by Bird et al. [27] because the package level is comparable in size to the Windows binaries.

According to Foucault et al., this difference may be due to the differences between industrial and FLOSS projects. One example is the way how developers contribute to the software. In Bird et al. [27] investigation sample, a minor developer of a binary A was also a major developer of another binary B that had dependencies on binary A. However, in open-source projects, a few major developers contribute to all modules, while the minor developers implement only a single feature or fix a few bugs. Foucault et al. conclude that for these reasons, before the release of open-source software, the

contributions were mainly from major developers, or the number of contributions per module was not enough to have minor developers.

Foucault et al. did another research [31] using the Bird et al. metrics on seven open-source projects implemented in different programming languages. Although they confirm the relationship between ownership metrics and software quality, they found the usefulness of ownership metrics debatable. They found that the „Minor“ metric is co-linear with the number of developers. However, they argue that most contributors are „Minor“ contributors in the open-source case. Additionally, they tested the „Most valued owner“ (MVO) metric. A high MVO means that the highest contributor has a high contribution percentage on that module, while a low MVO implies that the highest contributor has a low contribution percentage. Foucault et al. found a negative correlation between MVO and the number of bugs confirming that stronger ownership results in fewer bugs.

Rahman and Devanbu [32] did a fine-grained line-level study of authorship by considering „general“ and „special“ experience on four different medium- to large-sized open-source projects. They define „general“ experience as the author's general contribution to all project files and „special“ experience as the author's contribution at a specific file and time. The research examined, in particular, the implicated code, i.e., the lines of codes that were changed to fix the bug or were responsible for the bug (Figure 3.4). The hypothesis that implication code hunks typically have different authors was not supported; however, they found out that the author of the implicated code resulting in a bug has a lower contribution at the file level and that the file owner is less likely to introduce implicated code. To conduct the research, Rahman and Devanbu used the source code (i.e., the current state of the files) and the feature of the git blame to find the authors with general and special experience.

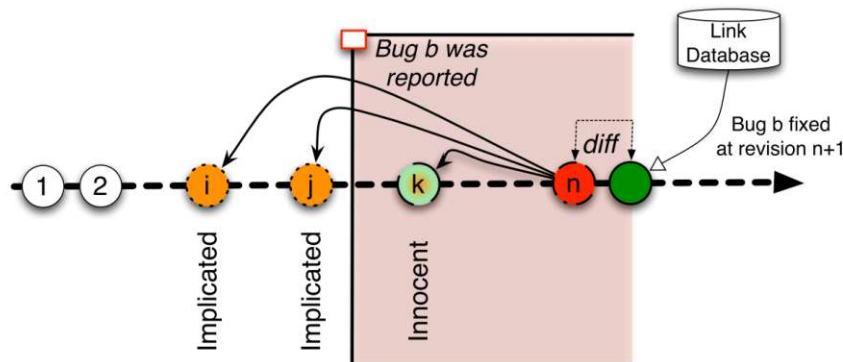


Figure 3.4: Rahman and Devanbu [32] marked as „implicated code“ the code that needed to be modified to fix a defect.

Lately, ownership has become controversial, especially regarding the agile movement [32]. Extreme Programming (XP) involves collective ownership, which encourages each

3. STATE OF THE ART

team member to contribute to any part of the code [33]. Scrum includes collective code ownership and collective product ownership [34]. In Scrum, the ownership is a collaboration between the product owner, who is accountable for achieving business objectives, and the developers, who take responsibility for technical execution. However, Scrum collective product ownership is neither dominated by the product owner nor contains strict boundaries between a team and the product owner [34]. In large software with multiple agile teams, the teams avoid code ownership by pair programming and rotating people between teams [35].

Nevertheless, there are still reasons why code ownership metrics are beneficial in agile environments. Augustine et al. [36] included in their analytic solution implementation Qlik Sense Metrics Portal (QSMP) four metrics where one of which is code ownership (Figure 3.5). The main benefit of the visualization is to make it easier to the developers to find the right person for guidance on a particular unfamiliar part of the code. Another benefit was finding parts of code owned by a developer leaving the project and sharing the information with the team.

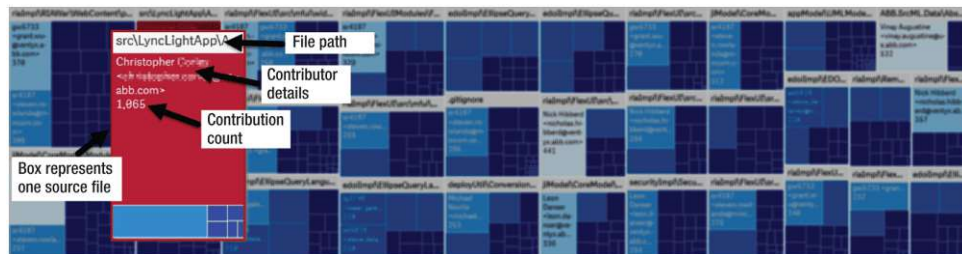


Figure 3.5: The heat map shows each file as a rectangle, where the contributors are displayed based on the commits number. [36]

Orru and Marchesi [37] investigated the relationship between refactoring and code ownership in a preliminary study, which gives further insight to teams that adopt collective ownership (i.e., Agile methodologies). They analyzed the refactoring activity that the developers did on Apache Ant to empirically find if and to what extent relationships exist between different refactoring methods and code ownership. They calculated two ownership metrics similar to Bird et al. [27] by counting commits instead of lines of code. They set the subjective ownership (SO) of a developer on a specific file (F) and commit (C) by dividing the commits amount performed by the developer in F by the commits amount authored by the same developer on other files of the C. For the relational ownership (RO), they divide the commits number the developer has performed in a file by the commits number performed by all developers in that file.

They concluded that developers tend to refactor those files with a high RO (i.e., the files that they have worked with more than other developers). Moreover, the findings show that SO was even more significant, i.e., the developers selected to refactor the files where they have mostly worked. Considering that SO is calculated by counting the number of commits until a specific time, it can imply that developers tend to refactor those files where they have higher historical ownership.

Thongtanunam and Tantithamthavorn [38] compared two different kinds of ownership. They called „commit ownership“ when the high ownership is defined by the higher amount of commits that changed a file (Bird et al. [27] used this technique) and they called „line ownership“ when ownership is calculated based on the lines of code a developer owns from the total lines of code the file has. However, line ownership includes only current ownership and not historical ownership. Historical ownership, also used by Girba et al. [3], can be seen as a combination of both where each committed change includes additionally the line ownership calculated at that time.

Moura et al. [39] used similar historical data to provide equations that help for Software Developer comparison. In their research, they analyze up to line-level granularity, where they check each line if it is newly added modified, or deleted. Their equations provide results on the effort that a developer has contributed and about the code’s survival. The code-survival result indicates the code that a developer has added and was not changed by anyone. Besides the code survival, the introduced equations for finding how often it happens that the code of developer A is modified by developer B. They consider this important for managers because if a manager detects low-performing developers then the equation would help to know who is changing or refactoring their lines. Moreover, they introduced equations, which can compare developers with different commit habits. Some developers commit more often and therefore they may introduce more changes to the same line rather than developers that commit less regularly. For this, they would compress the history of a line if it is modified by the same developer. The case study interview showed the results of the equations were consistent with the perception of the managers.

These studies examined the importance of code ownership and software quality, however, the results were different in different contexts. Bird et al. [27] found a significant correlation between ownership metrics in industrial software. On the other side, Foucault et al. [31] found out weak or no correlation between ownership metrics and quality in open-source projects, indicating that their effectiveness depends on the project type. Additionally, the development model influences the effectiveness of the metrics. For example, Agile methodologies have collective ownership, however, the ownership metrics still proved useful for identifying key contributors and developer behavior (e.g., which files are the developers more willing to refactor.)

3.3 2D Historical Code Visualizations

A variety of visualizations have shown different aspects of code history and evolution. Eick et al. [40] implemented Seesoft in 1992, one of the oldest code history visualizations. The visualization shows columns of files, where the small rows inside each column represent the lines of code. The lines are colored blue if they are the most recently changed rows and red if they are the least recently changed ones. Although it is simple and does not show the whole file history, Eick et al. made an essential contribution to the source code history visualization.

3. STATE OF THE ART

The code history visualization study has been further extended, showing different historical aspects. Some aspects examples are performance evolution in different versions [41]; static code evolution in different versions offered in multiple radial trees [42], with analog clocks by using pie charts [43] or flow graphs [44, 45]; production and test code co-evolution visualization [46, 47]; and as well other evolution visualizations [48, 49, 50, 51]. However, this section will focus on 2D evolution visualization related to code authorship and ownership.

Grabner et al. [52] present the Code Ownership River (Figure 3.6) to assess how much code is currently owned by which developer. The visualization combines all commits over time in a two-dimensional graph. The horizontal axis represents time, and the vertical axis is the aggregated amount of commits per author.

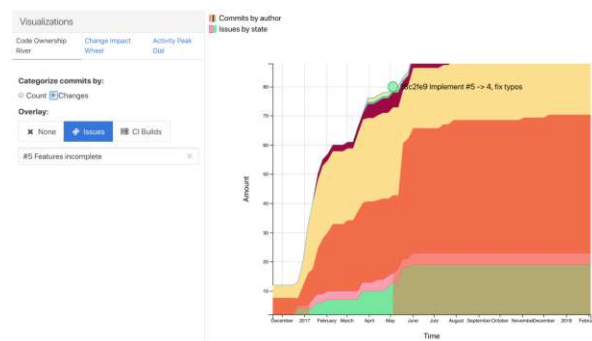


Figure 3.6: Code Ownership River [52]

CHRONOS [53] provides the history of detailed semantic source code changes. The visualization focus is to help developers interested to know more information about some specific lines of code. The user can select a couple of lines of code in one or more files, and CHRONOS will determine and distinguish only the relevant commits. Figure 3.7 shows the history of the selected lines in two different files. Each vertical blue line in the gray row represents all related commits to a particular file. The blue lines are the commits that include changes for the selected line. CHRONOS shows the source code additionally with the changed lines of the selected commit. The source code provides meta information such as date, author, and commit message.



Figure 3.7: Screenshot of CHRONOS [53]

Nowadays, IDEs include features to show the history of a selection, for example, the feature „Viewing the History for a Selection“¹ in IntelliJ. However, CHRONOS can display in one view selection for more than one file. This way, it allows seeing whether they typically were changed simultaneously or if there are other relationships between them by looking at the source code.

Yoon et al. [54] present „Azurite“ (see Figure 3.8), an Eclipse plugin visualization. It has two user interfaces to visualize fine-grained code change history: a timeline visualization and a code history diff view.

The timeline visualization is a 2D presentation of all files. Each row represents the edit history of a single file. The rectangles inside the rows represent the edits, colored red for deletes, green for inserts, and blue for replacements. The horizontal location means the time when the developers performed the change. Its width represents the duration. The vertical location and height represent the position of the changes on the file and the size. There is a fixed minimum height and width not to lose the minor edits. The timeline is interactive, so the user can generally see all files and their history or specific details. Users can zoom, scroll horizontally and vertically, filter, and search changes. The order of the files could have been supported by the drag & drop feature; however, currently, it is sorted by the latest edit file, where the most recently edited ones are on top. When selecting edits, Azurite offers the possibility to see the previous state of the file together with the files that the developer changed simultaneously. Then the user has the opportunity to undo the changes.

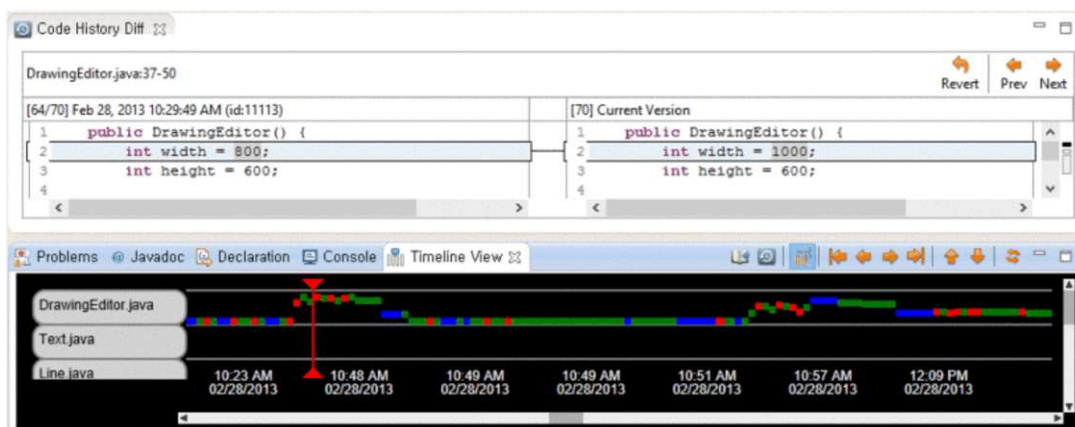


Figure 3.8: Screenshot of Azurite. The above part shows the code history diff of an edit, and the below part is the edits timeline of all files. [53]

When selecting a particular rectangle, the timeline visualization shows the Code History Diff view. The above part of Figure 3.8 shows the lines of code that were changed during a specific period. The red vertical marker on the bottom part of the visualization indicates the code part, which the Code History Diff shows. The user can dynamically move the

¹<https://trunkbaseddevelopment.com/> version of 11.07.2024

3. STATE OF THE ART

marker left or right or click Prev and Next on the top right corner to show different file versions. On the left side of the code history, the diff shows the current state, while the right side shows the state of the code on a selected previous state. The user can click revert to only revert the selected lines to the selected state.

A closely related visualization to the proposed one is Chronia. Girba et al. [3] designed in their research an ownership map visualization Chronia. It shows the evolution of the software by using metrics such as added and removed lines and authorship from the CVS versioning system. Each horizontal line in Figure 3.9 represents a file, and each circle is a commit or a group of consecutive commits that the same author does. The size of the circle depends on the added lines. The color represents the author with the most added lines.

These features combined make it possible to see the behavioral patterns of the developers. Figure 3.9 shows the colors of four different authors, particularly their behavioral patterns, for example, familiarization or the expansion of the blue author, followed by minor edits in multiple files and later a takeover of ownership from the green author. However, it

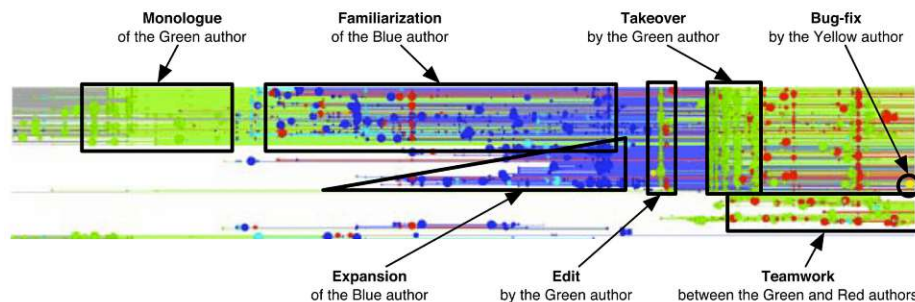


Figure 3.9: Chronia [3]

is not easy to see in Chronia what the developers are working on, which is an essential factor in the ownership maps. To see what a line or a circle represents, one has to first click on it. Then it is possible to see its metadata. Just by looking at the visualization, it is hard to see which features a developer has been working on.

Kuhn and Stocker [55] enhanced the Chronia visualization. They added on top of Chronia a further layer with user-contributed lifetime events and a timeline that marks major releases (Figure 3.10).

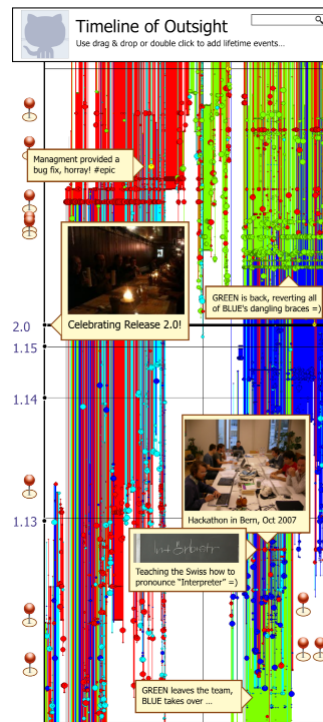


Figure 3.10: Timeline of Oversight [55]

Over time, code history visualizations have evolved to provide insights into software development and ownership. Seesoft [40] is the earliest contribution in 1992, which focused on showing which files had recently changed. CHRONOS [53] refined this by enabling to track of selected line changes across multiple files. Code Ownership River [52] showed how much code is currently owned by each developer. Azurite [54] introduced interactive timelines helping to explore the history of the code changes. Girba et al. [3] introduced Chronia allowing for a deeper understanding of developer behavior and contributions. These studies were the first contributions to 2D visualizations, which improved the way developers and managers understand code evolution and offered insight into the history of team collaborations.

3.4 3D Historical Code Visualizations

The 2D visualizations use only two of the space axes: the x-axis and the y-axis. This is easily supported by computers because also the screens are two-dimensional. Adding a third axis, the z-axis, we have a 3D space, allowing us to map three different variables to the three axis². Using 3D may be very useful when visualizing multidimensional data, to

²Besides the three variables provided by the 3D space, other variables can be mapped using other visualization techniques, such as color, opacity, size, etc.

avoid overlapping and visual noise that can happen when showing the same information in 2D. Additionally, 3D space is naturally tuned to the human perception making it easier to understand the information [56]. Ware et al. concluded in an experimental task to find paths between highlighted nodes with abstract data that the erroneous graphs represented in 2D can be up to 3 times higher than those represented on a virtual reality display [57, 58].

However, there have been pro- and contra-arguments regarding whether 3D visualizations are better than 3D visualization [59]. A 3D visualization requires extra rendering because the third axis does not exist on a computer screen. Although the projection may be transparent to the user, it may be less performant than 2D when rendering a high amount of data. Moreover, to grasp all the information the 3D needs to be interactive[60]. The user may need at least to rotate the visualization to see all the axes. A 3D can have also additional interactions like zooming or rendering different models based on user clicks. On the other side, Wettel and Lanza [61] argue that too much freedom of movement can be an argument against 3D visualization because it leads to disorientation.

The visualization mode depends on which data it will represent and its goals [56]. In contrast to 3D, a 2D visualization does not necessarily require user interaction to understand and see all the presented information, and a 2D visualization may be more performant when showing a lot of data. On the other side, the 3D visualization can support and communicate more data dimensions in a natural way tuned to human perception, however, the visualizations' design, the implemented interactions, and the data mapping are crucial to its success [62].

A popular 3D metaphor for software visualization is the „Code City“. The first proposals were in 1999 [8, 11], and lately, only VISSOFT21 had five research studies [12, 13, 14, 15, 16] related to the code city metaphor.

Wettel and Lanza [10, 61, 63] proposed the city metaphor that could be used for medium and large software named CodeCity. CodeCity shows all files of one project as building blocks (Figure 3.11), while it shows packages as different blue shades on the ground. The parameters mapped to the buildings are the number of methods and attributes of one file. The building height represents the methods' number, and its width and length represent the attributes' number.

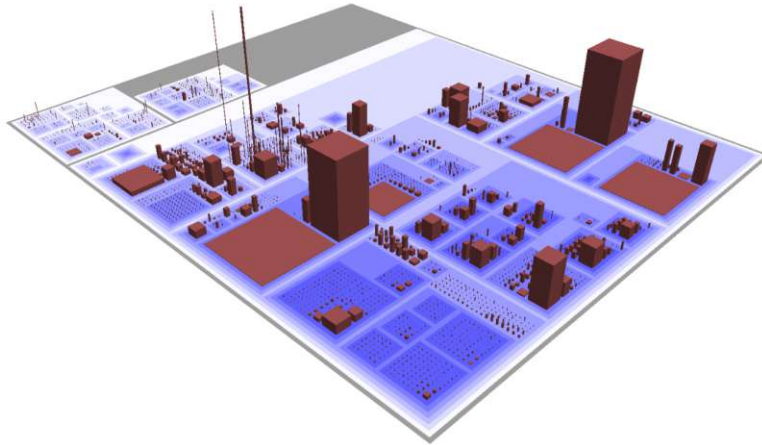


Figure 3.11: ArgoUML as a CodeCity [10]

Wettel and Lanza argued whether the data should be mapped linearly on the visualization or adjusted to avoid extra-large or wide buildings. For example, in Figure 3.11, some buildings are too tall, wide, or small to be seen.

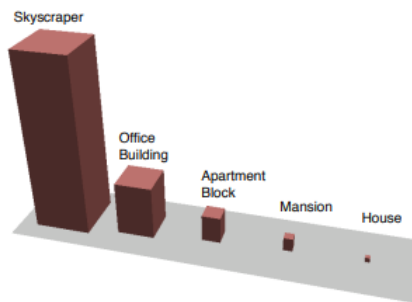


Figure 3.12: CodeCity building types [10]

To adjust the visualization to look more like a compact city with a uniform of buildings, they changed the mapping algorithm from linear to Boxplot-based or Threshold-based mapping [10]. This method will map all the files into one of the building's heights or areas from Figure 3.12. Whether one approach or the other is more effective depends on the question that the user has to answer. A linear method may be more helpful if the user's objective is to find the files with the most attributes or methods. The user may have it easier to see the highest or the most expansive buildings. However, if the users want to compare a district with another, it is easier to compare a group of similar and uniform shapes.

Wettel and Lanza extended the code city metaphor to represent the software evolution. They followed two approaches to show this by using the *time travel* and *age map* methods.

3. STATE OF THE ART

The *time travel* method (see Figure 3.13 left) shows the software in different periods. Building heights show in code city the number of methods. The user can compare the height of the buildings between different cities to see how the number of methods changes. They also introduced the fine-grained method, which shows each method as a small square (see Figure 3.13 right). The fine-grained method helps the user to see which methods were added or removed in different periods.

The second approach, the *age map*, uses colors instead of multiple code cities to show the code evolution. Figure 3.13 right shows the old sections with dark blue and the new ones with green and yellow. The city on the right side of Figure 3.13 contains some small buildings with some yellow squares, which indicates that they were added to the latest version.

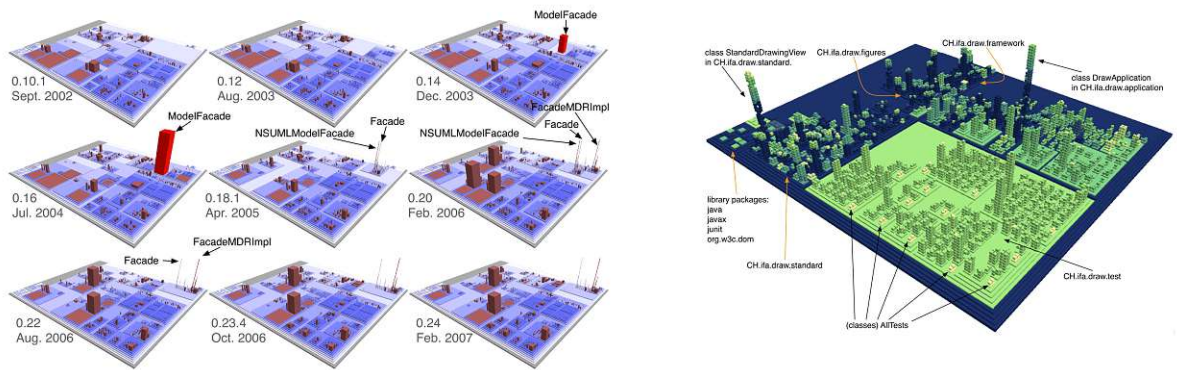


Figure 3.13: Coarse-grained time travel through software history (left) and fine-grained age map (right) [5]

Steinbrückner and Lewerentz [6] used the software city metaphor to represent the development history. They used the street layout where streets connect buildings (Figure 3.14). In comparison, Wettel and Lanza used squares on the ground to represent the packages. When new files are added to a package, the streets representing the package are extended to support more buildings. Steinbrückner and Lewerentz extended the Evo-Streets with the hill metaphor to give more insight into when a file was created. The buildings on a high hill represent an old file. The low hills indicate the continuously grown modules. The classes that were removed or moved to another package left a space in the former district.

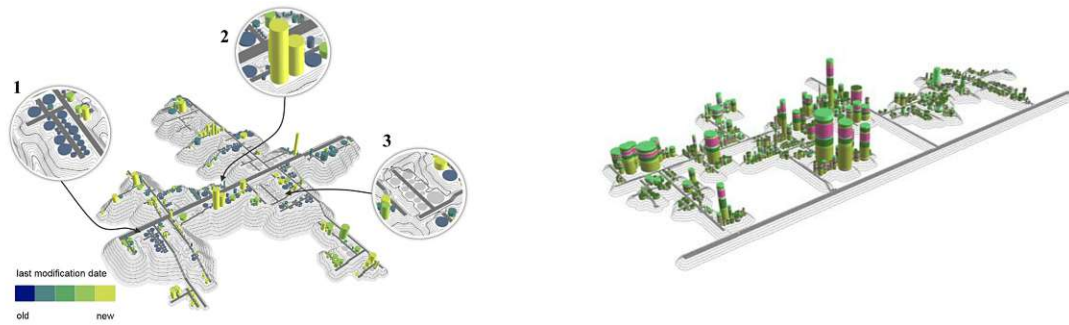


Figure 3.14: Evo-Streets visualization. The modification map (left) and the authorship map (right) [6]

The buildings are constructed by the layer of the modifications mined from SVN that the developers have done to a file since its creation. A high building represents a building that has been changed often. The building colors can be configured. On the left of Figure 3.14, the building is colored by the last modification date. The user can distinguish by color the files that have been lately changed. Figure 3.14 right shows buildings with different layer colors. The layers are colored by the author who implemented the modification.

Liu et al. [4] used the 3D city metaphor of [10] to implement a source code visualization tool named TeamWATCH. Like Evo-Streets [6], its goal is to show the revision history of the files. TeamWatch uses data from Git and builds for each file a cylinder. The cylinder is vertically split into sections where each section references a revision. The cylinder section's color represents the author. By this coloring technique, the visualization shows the historical ownership of the files based on the commits. While the cylinders represent the files, the blue squares on the visualization plane represent the folders. Figure 3.15 (g) shows shades of blue under the folder hierarchy cylinders.

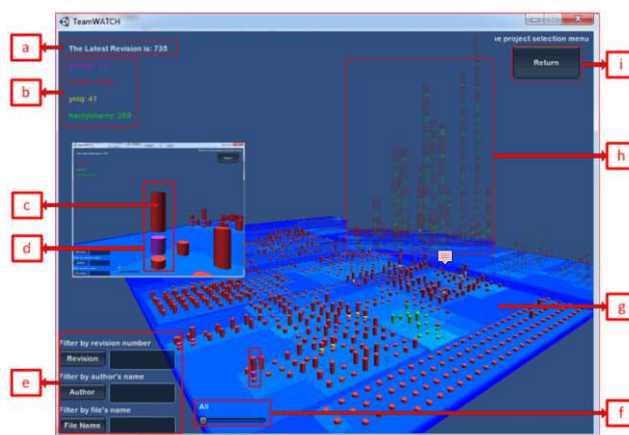


Figure 3.15: „TeamWATCH visualization (a) Revision statistics; (b) Developer statistics; (c) A single cylinder; (d) A stack of cylinders; (e) Filters; (f) Time Slider; (g) Blue district; (h) Transparent cylinders representing deleted files; (i) Return to the main menu.“ [4]

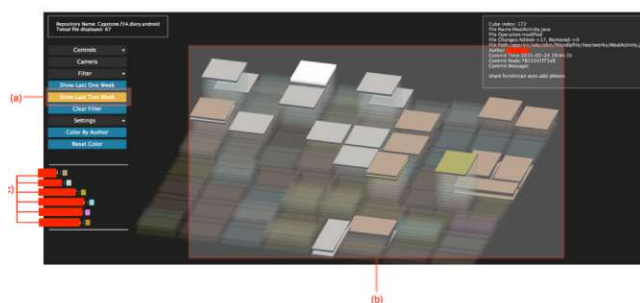


Figure 3.16: Another implementation of TeamWATCH. Commits are filtered by a period and are colored by the author. [64]

TeamWATCH offers interaction for the user to filter the information. One can filter by the author (e.g., in Figure 3.16, revision, or filename). The time slider helps filter and show only the revision from a particular time. Additionally, the user can see more information about a file by pointing to the cylinder with the mouse cursor.

Pfahler et al. [7] present M3triCity, which uses the code city metaphor to show the software evolution (see Figure 3.17). The city allocates enough space to fit every file and folder in history. Compared to the bin-packing layout that allocates as much space as there are currently buildings, the history-resistant layout assures that during the software timeline movement, the buildings will not jump around because of district resizing. The building’s parameters represent static metrics such as lines of code or the number of methods. Edges assist the movement of files to see the movement’s source and target. M3triCity has a timeline at the bottom where users can move to different periods. It also summarizes the changes in time.

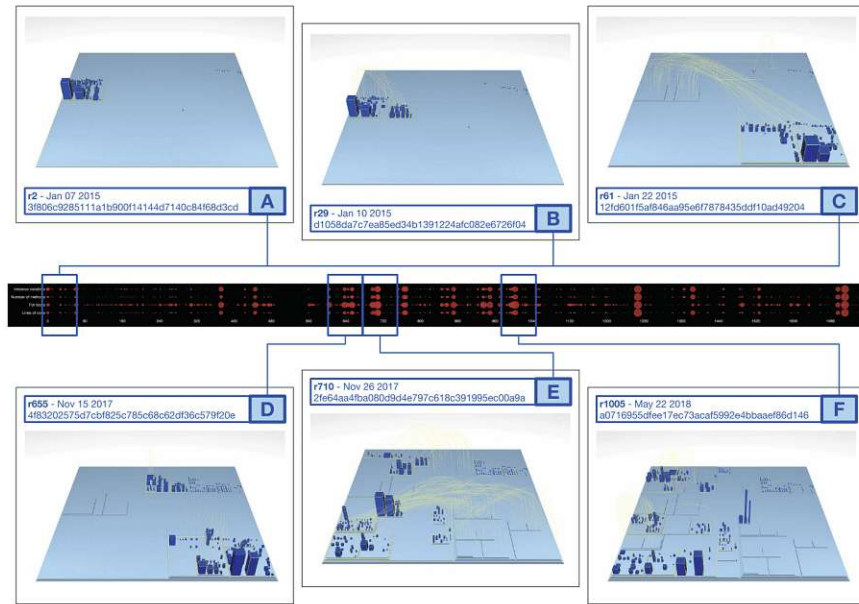


Figure 3.17: M3triCity shows the movements of files. [7]

Evo-Streets [6] and TeamWatch [4] show authorship aspects, such as buildings that represent files layered by the related commits. However, none of the above visualizations shows how the code ownership of the files has changed. The four software evolution visualizations show the city on the file level. However, if the user needs a more abstract overview, then the file-level visualization may be noisy and hard to comprehend or compare. The more general level also correlates with the ownership effects on the software quality discussed in Section 3.3. When Foucault et al. used their research [30] on the ownership metric of Bird et al. [27] at the file level, they got no correlation between quality and ownership metrics. However, the correlation improved when they used the same metrics on the package level. Similarly, Bird et al. [27] used the ownership metrics at the binaries level when testing the metrics on Windows Vista and Windows 7.

The use of 3D visualizations in software development, particularly the „Code City“ metaphor, has been researched to show the structure and evolution of the source code. This approach uses the three axes of space to map more data offering more flexibility and reducing visual clutter compared to 2D visualizations. However, they come with challenges such as rendering costs and the need for interactivity to rotate or zoom, to fully grasp the data. The „Code City“ metaphor, first proposed in 1999, has been applied in several tools, including in visualizations for source code evolution, which help to show software revision histories, authorship, and code ownership. However, the visualizations still struggle to provide a clear overview of changing ownership and how it impacts software quality, especially when data is presented only at the file level, which can become cluttered and difficult to interpret without a higher-level abstraction.

Systematic Mapping of Code City Visualizations

This section presents the mapping study to structure the code city research area.

4.1 Visualizations Using City Metaphor

This sub-section briefly describes all code cities categorized in this mapping study.

1. **FileVis:** Young and Munro [20] presented FileVis in 1998, one of the earliest visualizations that shows a file like a city. The visualization aims to help developers familiarize themselves with the new software. Figure 4.1 shows the representation of a file in FileVis. The orange plane represents a file, and the buildings on the plane its internal functions. Figure 4.1 left shows the zoomed-out mode. The zoomed-in mode in Figure 4.1 right shows more details about the methods (e.g., lines, space attributes).

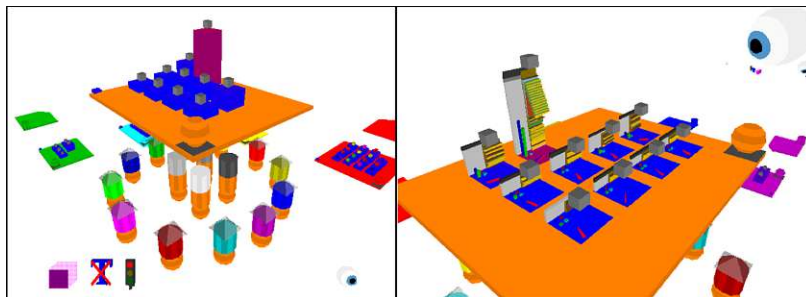


Figure 4.1: FileVis [20]

2. **Software World:** Before the code city metaphor was well known, Knight and Munro [8, 9] introduced the Software World to visualize Java applications. In Software World, the world represents the whole system; the countries represent the directory structure (i.e., the Java packages); the cities represent the files; the districts represent the classes in the file; the buildings represent the methods. The Software World visualization helped to identify the densest cities to indicate the need for refactoring those files. Building attributes such as height, doors, and windows represent details for attributes and the number of code lines inside the method (Figure 4.2).

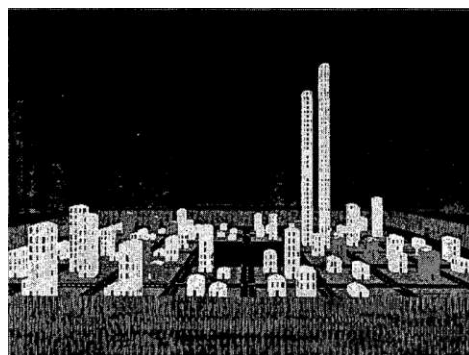


Figure 4.2: Overview of a district (i.e., a class) in Software World [8]

3. **Software Landscapes:** Deussent et al. [65] used spheres and sub-spheres to show the package hierarchy in Java. The visualization shows files of a particular package under their respective sphere as a city. The city, representing a file, contained cuboids symbolizing methods and attributes (Figure 4.3).

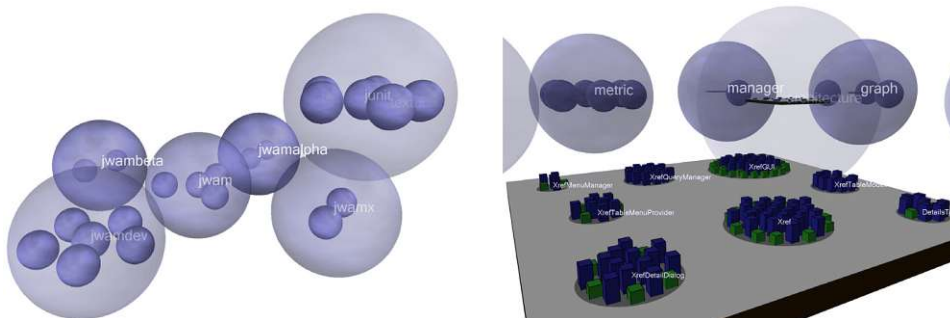


Figure 4.3: Software Landscapes [65]

4. **3D City:** In 2003, Panas et al. [11] proposed a 3D city metaphor concept to visualize packages and files. 3D City represents packages with cities connected by roads (Figure 4.4). The concept is very rich with features. For example, it proposes to use street lamps and trees to map software metrics. However, this concept was not implemented.

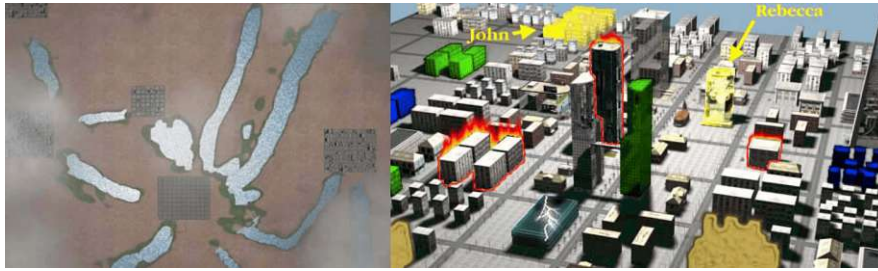


Figure 4.4: 3D City visualization proposal. On the left are the packages as cities; on the right are the files as districts and methods as buildings. [11]

Vizz3D: Panas et al. [66] visualized in Vizz3D their city metaphor in 2005. Panas et al. [67] extended Vizz3D again later in 2007 to the shape in Figure 4.5. Vizz3D shows C++ methods as building. The blue plains displayed as cities represent the files, and the green planes represent the packages. The water towers represent header files.

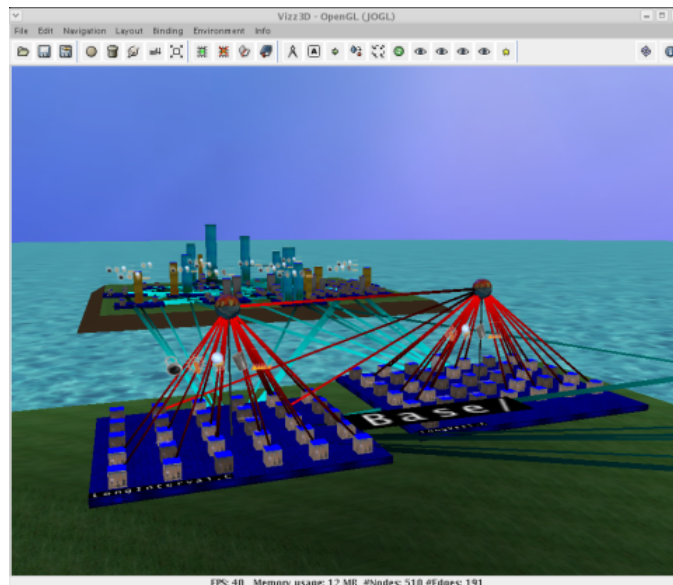


Figure 4.5: Code Cities in Vizz3D [67]

5. **VERSO:** Langelier et al. [68] proposed an approach to visualize metrics such as coupling, cohesion, and complexity (Weighted methods per Class). They later called the visualization VERSO [69]. Cuboid buildings represented classes. They mapped coupling to the building's color and complexity to the building's height. They twisted the buildings from 0 to 90 degrees to visualize the cohesion. Cylinders were used to represent the interfaces [70]. To group the building according to the folder structure, they used the modified Treemap technique and Sunburst technique (Figure 4.6).

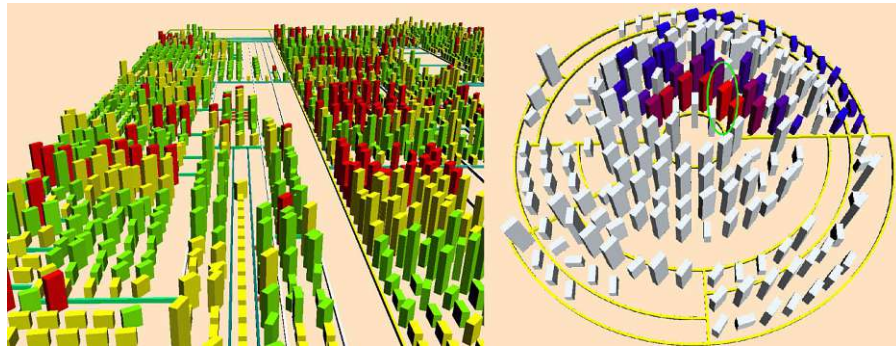


Figure 4.6: VERSO in modified Treemap technique (left) and modified Sunburst technique (right) [68]

Benomar et al. [71] extended Verso with heat maps (see Figure 4.7). Heat maps represent the time dimension of the software, e.g., how often an event has happened or when was the last time a file changed.

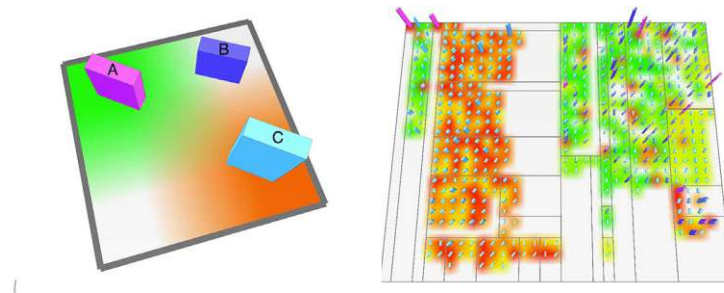


Figure 4.7: VERSO extended with heat maps. [71].

6. **CodeCity:** Wettel and Lanza [10, 61, 63] proposed the first city metaphor that users could use for medium and large software named CodeCity. CodeCity shows all project files as building blocks and the packages as different blue shades of the ground (Figure 4.8). The height represents the number of methods, and the width and length of the building represent the number of variables.

Later they extended the code city to localize design problems [72] or show software evolution [61] using colors. For example, they assign vivid colors to buildings with design problems, such as files with too many attributes or methods.

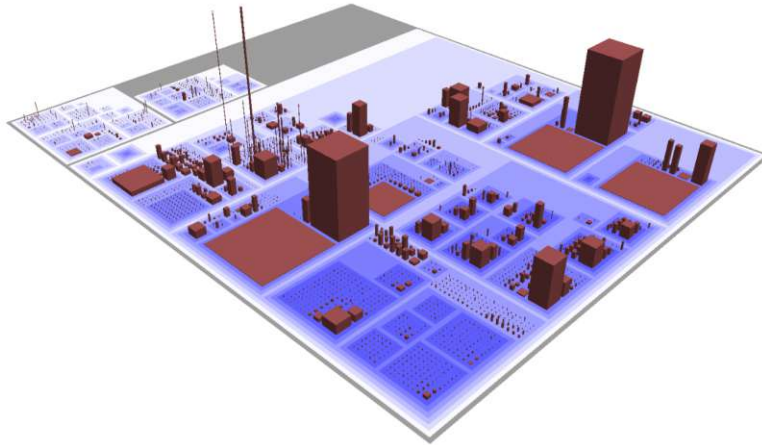


Figure 4.8: ArgoUML as a CodeCity [10]

7. **CocoViz:** Boccuzzo and Gall [73] introduced three metaphors to represent classes: house metaphor, table metaphor, and spear metaphor. The idea behind the three metaphors is that a well-designed software entity will result in a well-designed metaphor. For example, each house represents one class. The house has four parameters: width and height of the body and width and height of the roof. Metrics are normalized to show a well-designed class as a well-designed house. The classes not in this category will result in badly shaped houses (see Figure 4.9). Boccuzzo and Gall [74] extended CocoViz with audio to detect code smells. For example, when clicking a house, it will give audio feedback about the state of the class.

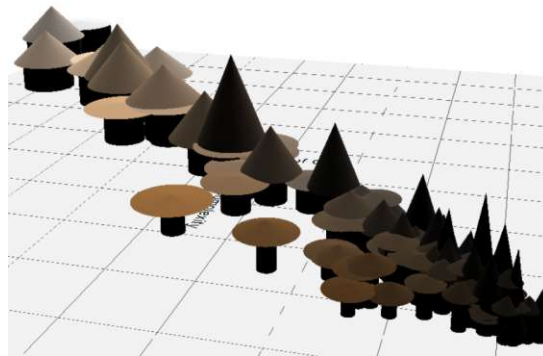


Figure 4.9: House metaphor in CocoViz [74]

8. **UML-City:** Lange et al. [75] present UML-City to show UML aspects by combining two other visualizations, MetaView and MetricView. MetaView visualizes inter-diagram relations in 2D, and MetricView visualizes three different metrics on top of a regular class diagram.

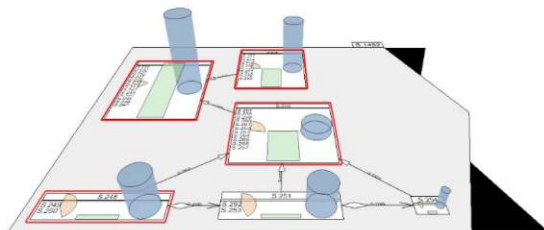


Figure 4.10: MetricView [75]

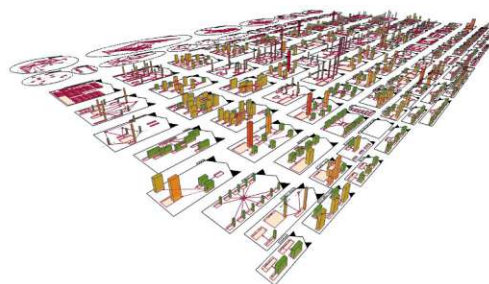


Figure 4.11: UML-City [75]

9. **EvoSpaces:** Alam et al. [76] render JAVA and C/C++ software in EvoSpaces. EvoSpaces shows different types of buildings that depend on the file type (e.g., small hall buildings represent the header files) and the file size (e.g., skyscrapers represent large C/C++ files). The green plains represent packages. The visualization also shows relations between buildings with pipes that connect them. On building zoom, it is possible to see the internals of the file, like methods, macros, and classes divided into different floors.

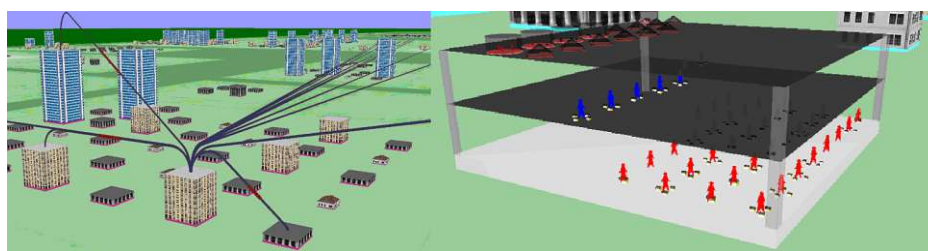


Figure 4.12: EvoSpaces visualization. The left shows the relations between files. The right shows the methods and properties inside a file [76].

10. **Evo-Streets:** Steinbrückner and Lewerentz [6] used the software city metaphor to represent the development history. They used the street layout where streets connect buildings. Additionally, Evo-Streets introduces the hill metaphor. Buildings on higher hills represent the older classes. The buildings in the lower hills indicated the continuously grown modules. The removed or moved classes leave a space in the former district.

The user can configure the building colors. They can represent a period when it was last changed or the author that changed the class (see Figure 4.13). The viewpoints on top of Figure 4.14 connect the buildings which contain the same clone (i.e., copied code). The viewpoint size is related to the size of the copied part [77].

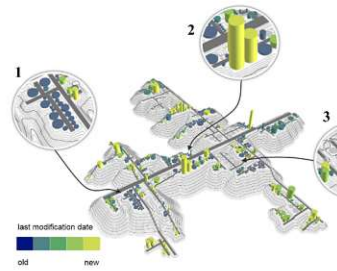


Figure 4.13: The modification map [6].

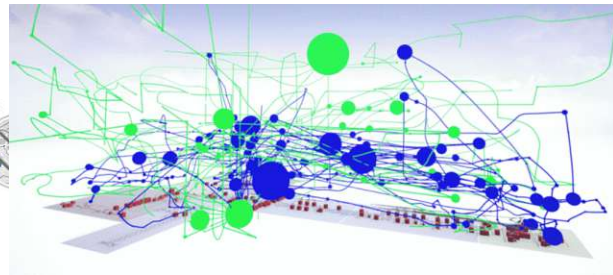


Figure 4.14: The view-points in Evo-Streets [77].

11. **VizzAspectJ City/VizzJava City** Bentradi and Meslati [78] presented a visualization based on CodeCity to visualize an aspect-oriented paradigm (see Figure 4.15). They added two city visualizations. „VizzJava City“ shows the classes, and „VizzAspectJ City“ shows the aspects. The buildings can be different, also the color. The classes are blue, and the aspects are pink. The shades of blue are used to differentiate between abstract classes, classes, and interfaces, and the shades of pink are used to differentiate between the abstract and concrete aspects.

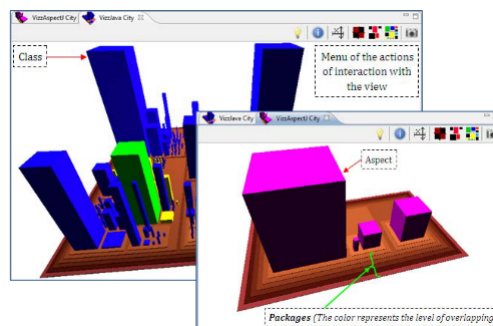


Figure 4.15: VizzAspectJ City and VizzJava City [78]

12. **VITRAIL**: Caserta et al. [79] implemented in VITRAIL the „3D Hierarchical Edge Bundles“ technique on top of two different layouts. The two layouts they used are the city metaphor (Figure 4.16) and the street metaphor (Figure 4.17). The buildings represent classes. However, their size is not related to any attribute. VITRAIL’s main purpose is rendering the edges that represent the call operations.

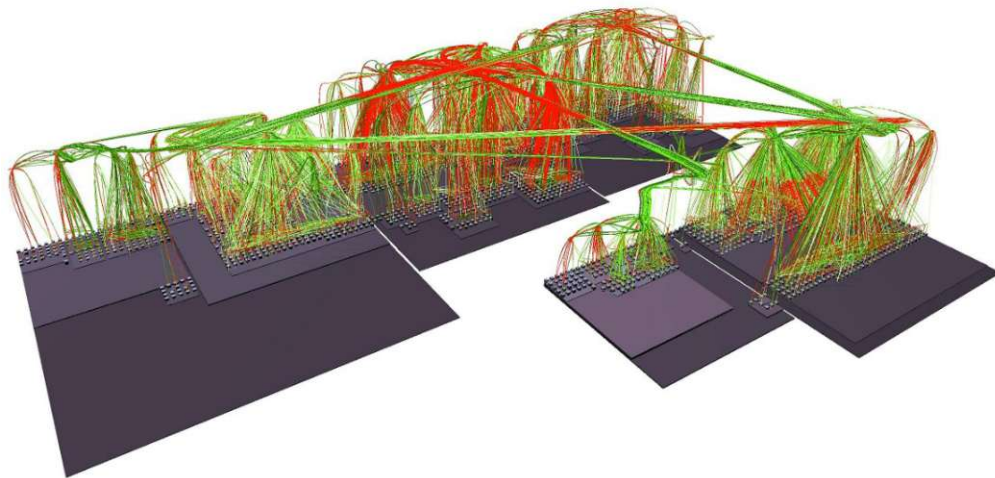


Figure 4.16: The city metaphor from Caserta et al [79].

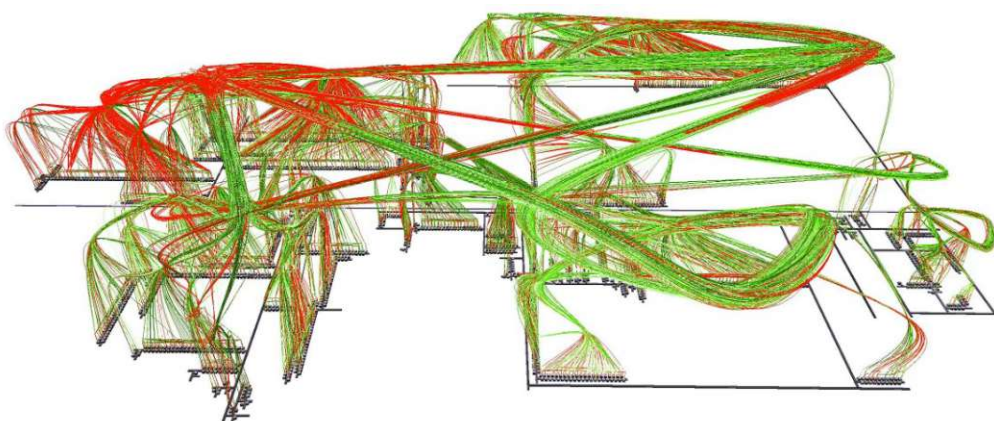


Figure 4.17: The street metaphor from Caserta et al [79]

13. **SkyscrapAR:** Souza et al. [80] introduced SkyscrapAR, an augmented reality visualization for software evolution. It uses districts for packages and buildings for files. The city metaphor is inspired by Wettel et al. [10]. However, the buildings use different metrics. The surface of the building represents the size of the file in lines of code in the first revision, and its height represents the code churns. A code churn represents the changes that have happened after each revision.

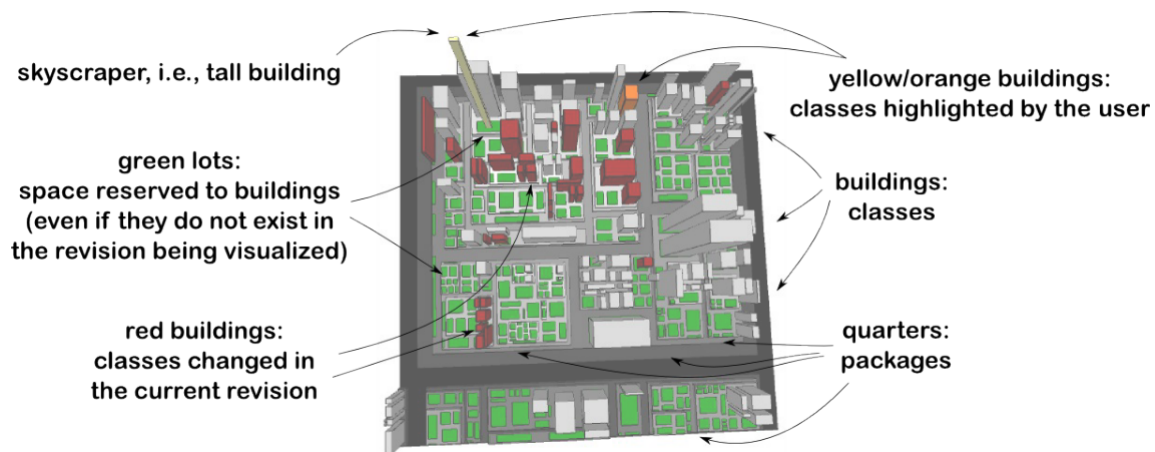


Figure 4.18: The JUnit framework in the SkyscraperAR [80]

14. **SynchroVis:** Waller et al. [81] visualized in SynchroVis dynamic aspects of software during its runtime. SynchroVis helps best to find deadlocks between different threads. Like Panas et al. [67] city metaphor, they showed packages as green planes (Figure 4.19). However, buildings differ from other visualizations. Only the ground floor represents the class, while the other upper floors represent the number of created instances. Edges between floors represent a relation between instances. Waller et al. [81] use the term „street.“ However, the term streets in SynchroVis correlates with the edges used in other visualizations [65, 66, 76] and not with the street layout like in Figure 4.17. Other visualizations like [6] render the street metaphor on the city floor and may connect buildings and other streets. SynchroVis's edges connect floors when the call is executed from one instance to another. If the call is a constructor call, it adds another floor for the newly created instance.

4. SYSTEMATIC MAPPING OF CODE CITY VISUALIZATIONS

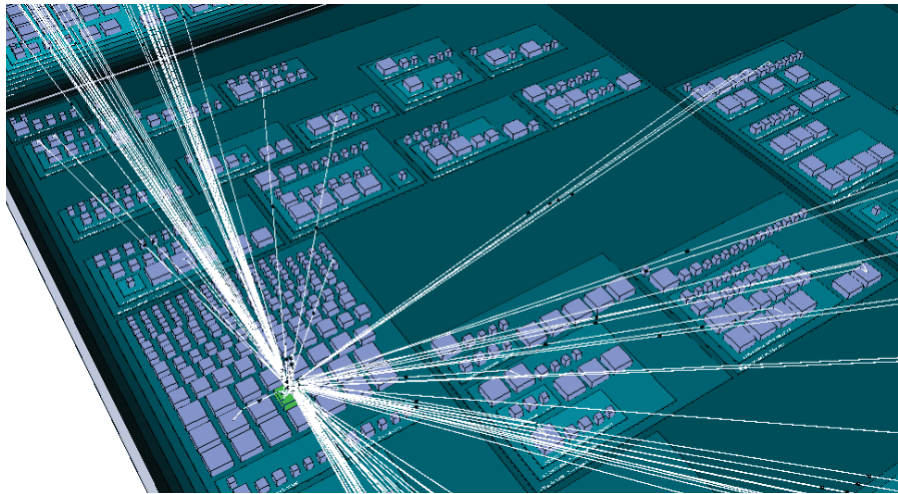


Figure 4.19: SynchroVis visualization [81]

15. **SeeIT 3D**: Sharif et al. [21] introduced an eclipse framework that renders the software in 3D. The poly cylinders shown in Figure 4.20 can represent different things depending on the code granularity the user selects. They can be packages, classes, methods, or lines of code.

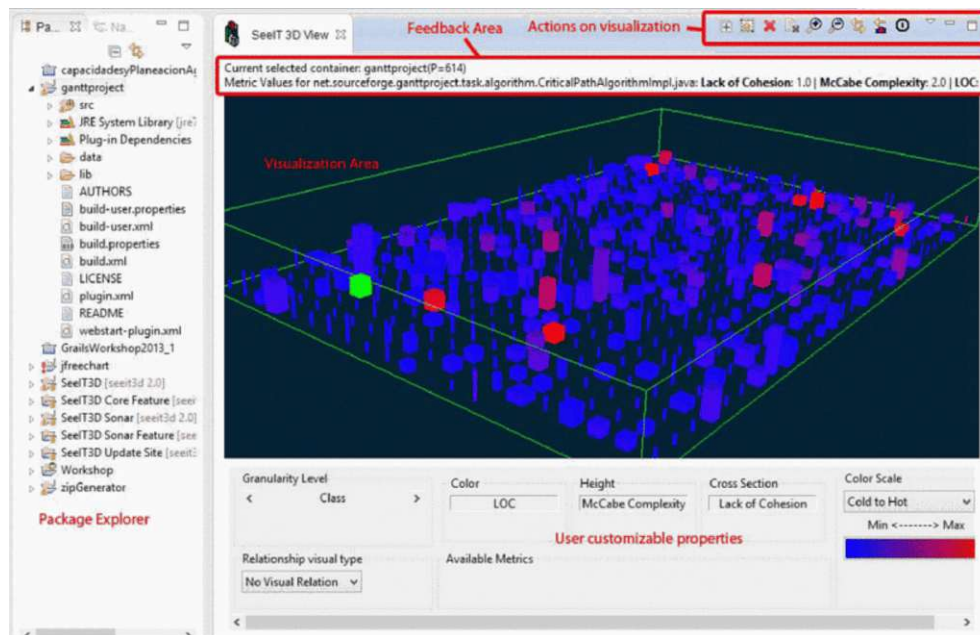


Figure 4.20: SeeIT 3D [21]

16. **ExplorViz**: Fittkau et al. [82, 83] introduced a visualization to show the system behavior on runtime (Figure 4.21). They visualized the replication of nodes in

a cloud computing system [84] and used the code city metaphor to visualize the application's runtime behavior. ExplorViz shows the number of instances created for each class. In their code city metaphor, they show information details on demand. The user can interactively open and close the components/folders when more details are needed. When a component building is opened, it becomes a district.

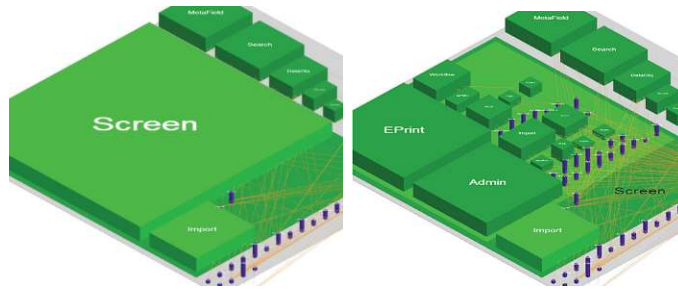


Figure 4.21: The left side shows the Screen as a closed component, and the right shows its details [84]

Fittkau et al. [85] presented a VR approach to explore the ExplorViz. Additionally, Fittkau et al. [86] created a physical 3D-printed model of ExploreViz. The model in Figure 4.22 took 58 hours to print with a budget of 9€ for the materials [86].

Krause et al. designed a heat map for ExploreViz [15] (see Figure 4.23), like Benomar et al. [71] did for the Verso [68] visualization.

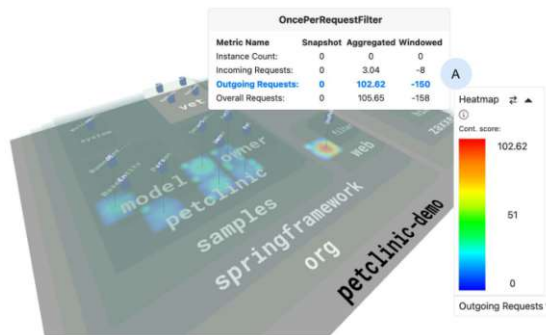
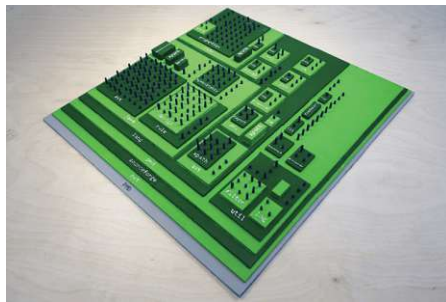


Figure 4.22: ExploreViz 3D printed [86].

Figure 4.23: ExploreViz with heat map overlay. [15].

17. **TeamWATCH:** Using the 3D city metaphor, Liu et al. [4] implemented a source code visualization tool, TeamWATCH, with data from the SVN versioning system. The goal of the visualization is to show the revision history of the files. In TeamWatch, each cylinder represents a file. The cylinder is vertically split into sections where each section references a revision. The cylinder section's color

represents the commit's author. By this coloring technique, the visualization shows the historical ownership of the files based on the commits.

The blue squares on the visualization plane represent the folders. Figure 4.24 shows different shades of blue under the cylinders. For example, (g) in Figure 4.24 represents the folder hierarchy.

TeamWATCH offers interaction for the user to filter the information. One can filter by the author (e.g., in Figure 3.16), revision, or filename. The user can use the time slider to filter and show only the revisions from a particular time or see more information about a file by pointing to the cylinder with the mouse cursor.

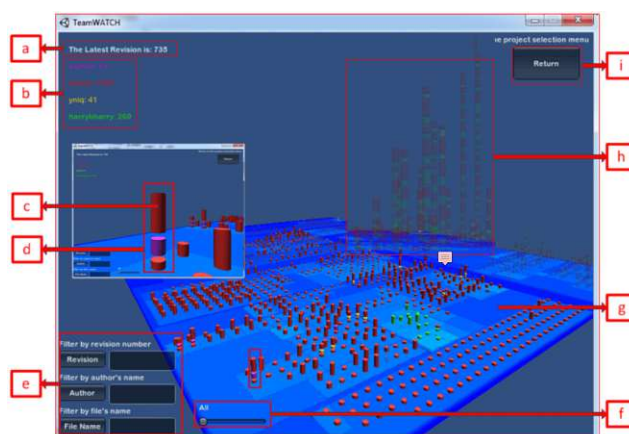


Figure 4.24: „TeamWATCH visualization (a) Revision statistics; (b) Developer statistics; (c) A single cylinder; (d) A stack of cylinders; (e) Filters; (f) Time Slider; (g) Blue district; (h) Transparent cylinders representing deleted files; (i) Return to the main menu.“ [4]

18. **CodeMetropolis:** Balogh et al. [87] used the code city and the Minecraft metaphor¹ to visualize the source code (Figure 4.25). CodeMetropolis allows users to move into the city and visit the buildings to see more information. At a high level, their metaphor assigns classes to buildings and other metrics to length or height, while at the low level, they assign methods to buildings. A flat platform represents the hierarchical structure of packages.

¹<https://www.youtube.com/watch?v=O5Ijvs44vv4> version of 11.07.2024

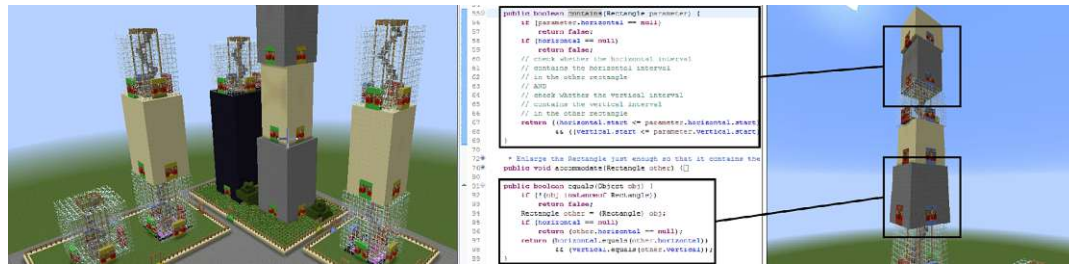


Figure 4.25: CodeMetropolis [87]

19. **Rocat:** Ichinose et al. [88] implemented Rocat, which uses Kataribe - a hosting service for Historage repositories that contain fine-grained source code repositories (Figure 4.26). Kataribe helps Rocat to get incrementally updated on the latest version when developers push new commits. Rocat is based on the code city metaphor of Wettel et al. [10].

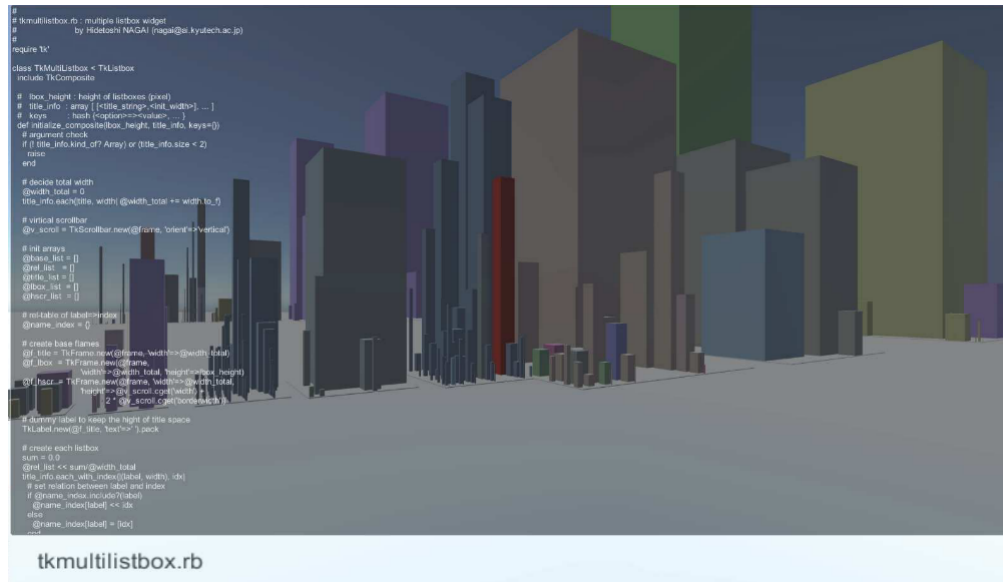


Figure 4.26: Rocat [88]

20. **Origin City:** Ishizue et al. [89] used the code city metaphor of Wettel et al. [10] to create Origin City (OC). According to their findings, software applications changed from different organizations may introduce more bugs. They created OC to visualize the origin of files and their functionality. Each building represents a file. Buildings' horizontal layers represent the functionality of the file. A crucial role in OC is the positioning of buildings. OC places files that belong to the same organization near each other and files that belong to more organizations in a center point of gravity between the organizations.

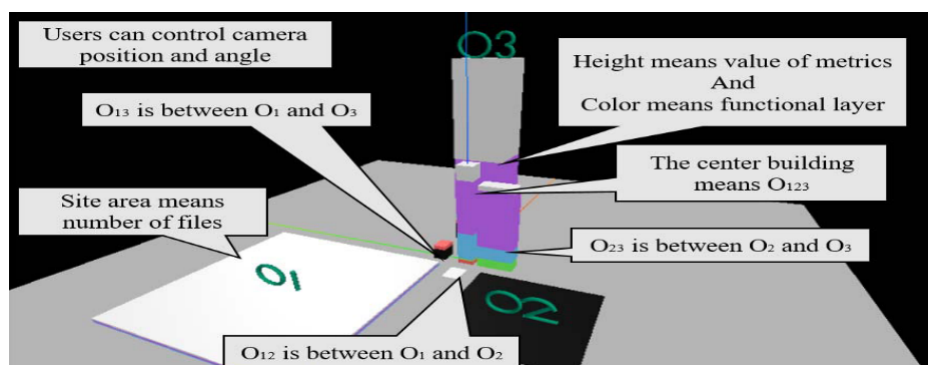


Figure 4.27: Origin City [89]

21. **VR City:** Vincur et al. [90] visualized Java software in their city metaphor VR City. Their buildings represent the Java classes. The floors of the building represent methods. The size of the district is dependent on the size of the method. For example, on the left side of Figure 4.28, the tall skinny buildings represent the interface, while those with a more extensive base but thin on top are abstract classes. VR City uses colors and edges to map more metrics to the visualization; for example, the right side of Figure 4.28 shows two authors' recent activities.

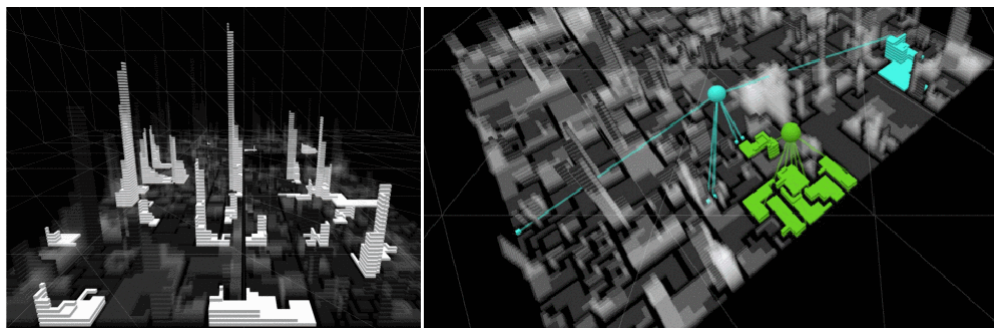


Figure 4.28: VR City visualization. [90]

22. **CityVR:** Merino [91] used the code city metaphor to show the source code in virtual reality (Figure 4.29). They mapped metrics such as lines of code, number of methods, and number of attributes into the buildings' surface, height and color [92].

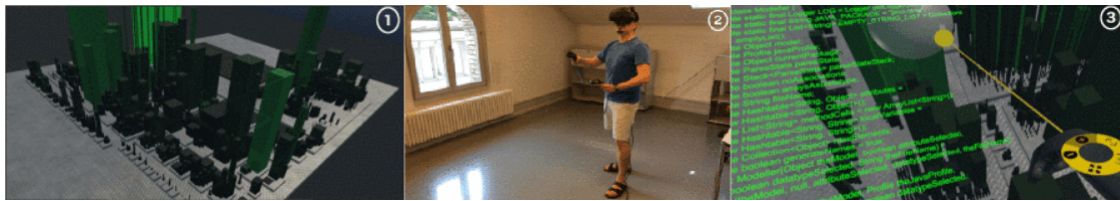


Figure 4.29: (1) City metaphor; (2) The user; (3) The user interaction with the city. [91]

23. **High-Rising Cities:** Ogami et al. [93] visualize the system's performance in their code city metaphor. The districts represent packages or classes. Methods are mapped to buildings. The buildings' height shows the number of events related to the method during runtime. For example, in Figure 4.30, the blue building represents the events that have occurred in a particular fixed time.

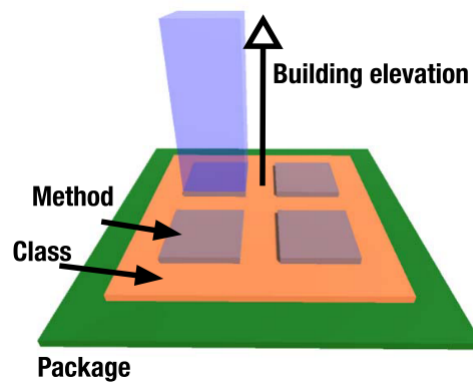


Figure 4.30: High-Rising Cities [93]

24. **Code Park:** Khaloo et al. [94] implemented the „Code Park“ visualization that incorporates the source code into the visualization. It shows the files as 3D cuboids (left side of Figure 4.31). The user, however, can find inside the cuboids the source code on one of the wallpapers. Code park includes interactions to jump from one file, for example, when clicking a variable to see where it was defined.



Figure 4.31: Code Park [94]

25. **Linked Data City (LD-City):** Andries et al. [95] proposed Linked Data City to help users analyze linked data structures and browse linked data repositories. Figure 4.32 shows a screenshot of the LD-City based on DBpedia data retrieved using a SPARQL query. The buildings represent ontology classes, and the buildings' height represents the ontology instances. They used the color to distinguish between internally defined classes in a linked data repository or externally.

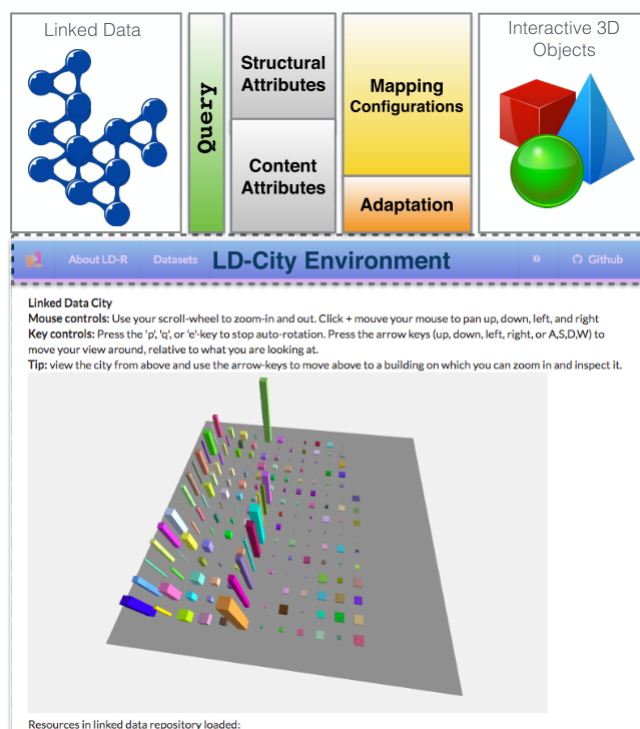


Figure 4.32: Linked Data City [95]

26. **GoCity:** Brito et al. [96] used the code city metaphor to visualize software written in the Go language (Figure 4.33). Unlike Java or C++, Go has struct types containing attributes and methods. However, methods may also be outside of the struct. They adapted the building to represent files and structs. The grey structure represents the files. On top of it, there may be additional sub-buildings that represent the structs inside the file (see Figure 4.33). The large grey building on the right of Figure 4.33 is an example of a large file with two buildings over it which represent two structs in the file. The building parameters are similar as in CodeCity 4.8, i.e., representing the number of attributes, methods, and lines.

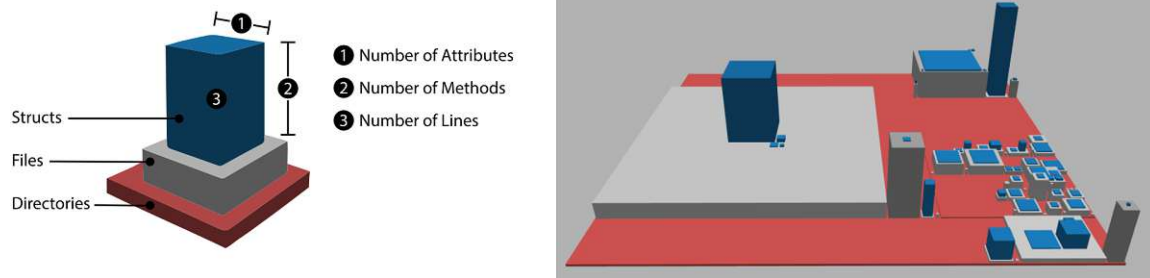


Figure 4.33: GoCity [96]

27. **IslandViz:** Schreiber et al. [97] introduced IslandViz, which visualizes software in Virtual and Augmented Reality. IslandVis is based on islands that represent bundles. The colors of the island's land represent the packages, and the buildings represent the classes. Buildings get an additional floor for every amount n of lines. Each island may have ports representing imports and exports, like the red and green ones on the left of Figure 4.34. IslandViz can also show with edges the dependencies between bundles (right side Figure 4.34).

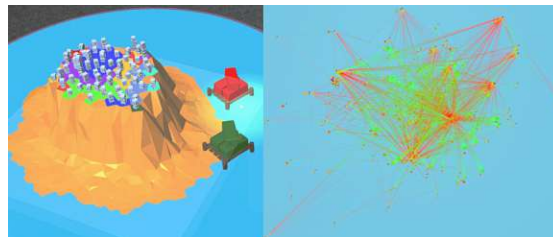


Figure 4.34: IslandViz [97]

28. **PerfVis:** Merino et al. [98] designed Perfvis using the code city metaphor to visualize software performance through immersive augmented reality. Perfis is designed to support dynamic and static data. It can show metrics from the current state of the source code but also data from the live performance of the system. The city visualization is very similar to the CodeCity [10]. The buildings represent classes, and their parameters map metrics such as the number of methods, attributes, and lines of code. The color shows the dynamic aspect of the visualization by encoding the number of times the methods of the class are called.

The scatter plot on the right side of Figure 4.35 helps the developer analyze dynamic aspects missed from the code city itself.

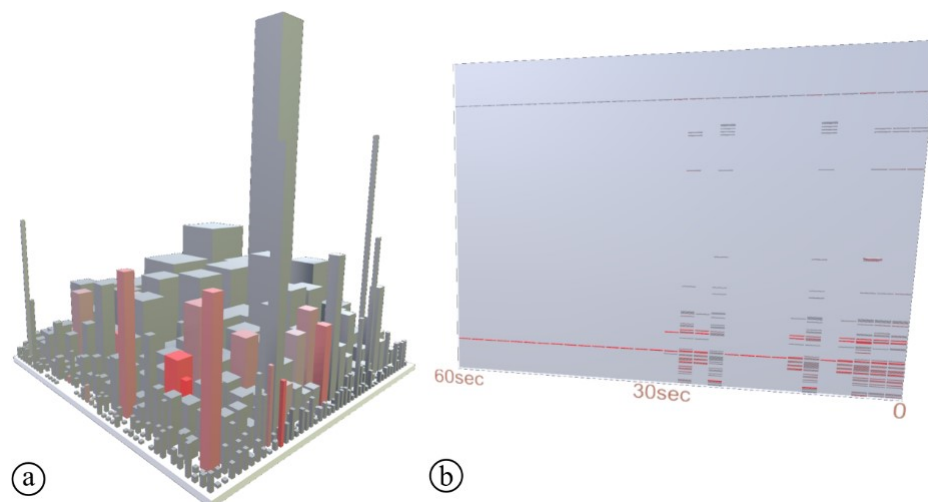


Figure 4.35: PerfVis [98].

29. **Software City in VR:** Jung et al. [99] mapped the packages and files to districts and buildings in their code city metaphor. The buildings could contain courtyards that represent inner classes. The user can dynamically change the metrics used for size and colors. The visualization supports dynamic features such as showing method calls with edges.

The visualization improves the collaboration between multiple users in a virtual mode with an avatar (see Figure 4.36). The avatar is useful to help point out buildings in the virtual mode.

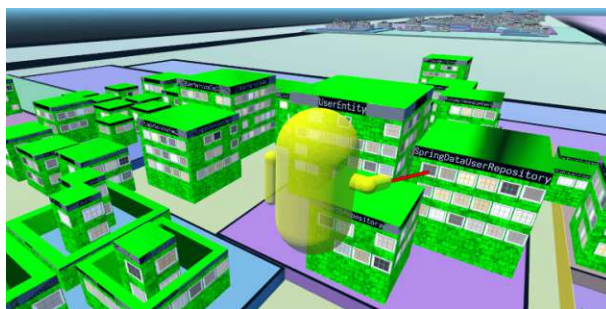


Figure 4.36: An avatar pointing at a building [99]

30. **Memory Cities:** Weninger et al. [24] present Memory Cities to visualize the heap memory in time (see Figure 4.37). Programming languages such as Java or JavaScript use garbage collectors to clean up the memory. The garbage collector finds objects that are not referenced anymore and removes them. However, a long living object may unnecessarily reference an object, blocking the garbage collector from removing it. Memory Cities shows buildings representing heap object groups

and organizes them into districts based on shared heap object properties such as type (e.g., object or list) and their allocation site (the function hierarchy that created the object).

Memory Cities changes constantly during runtime; therefore, the user will need to move the city back in time to analyze the memory. The buildings' height and color help the user to search for heap objects groups that may be growing fast and are part of a memory leak.

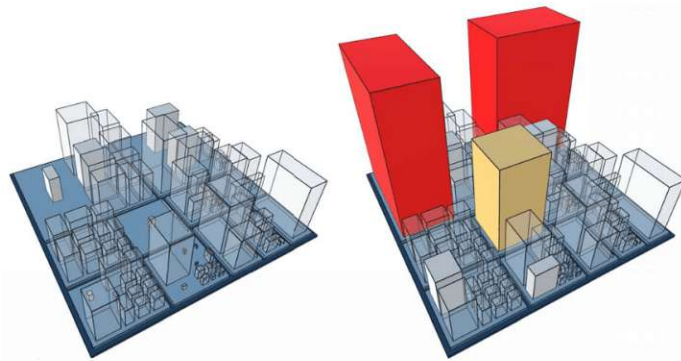


Figure 4.37: Memory Cities [24]

31. **M3triCity:** Pfahler et al. [7] presented a visualization for software evolution called M3triCity (see Figure 4.38). M3triCity buildings represent classes. Its height and depth can be configured to represent static metrics such as lines of code, number of methods, number of for loops, or number of variables. The city allocates enough space for any file or package regardless of when it has existed. The timeline is shown at the bottom of the visualization, which the user can use to move in different periods. The timeline also summarizes the evolution of the building metrics.

4. SYSTEMATIC MAPPING OF CODE CITY VISUALIZATIONS

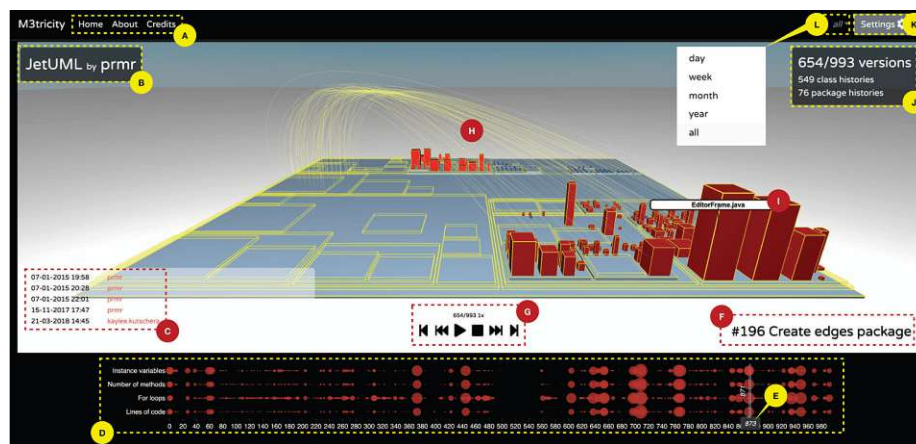


Figure 4.38: M3tricity [7]

32. **M3tricity2:** Ardigo et al. [12] extended M3tricity to include data files in the visualization. They added a special building to represent data files, binaries, and tables (see Figure 4.39).

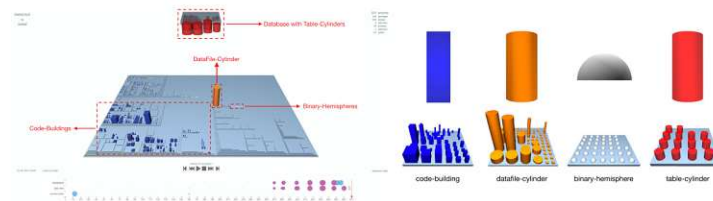


Figure 4.39: M3tricity2 [12]

33. **Layered Software City:** Dashuber et al. [100] used the code city metaphor to show the class dependencies (Figure 4.40). Each building represents a class, and its height represents its dependency number. They designed their visualization into two different layouts.

The TreeMap Layout uses rectangle districts to group the buildings. It shows with edges all the dependencies, which adds a lot of information noise.

The Graph Layout Technique organizes nodes in layers. It positions the buildings in layers to show the dependencies hierarchy from back to front (see Figure 4.40). The Graph Layout Technique uses edges only for dependency cycles with as few crossings in the opposite direction to indicate a potential architectural violation.

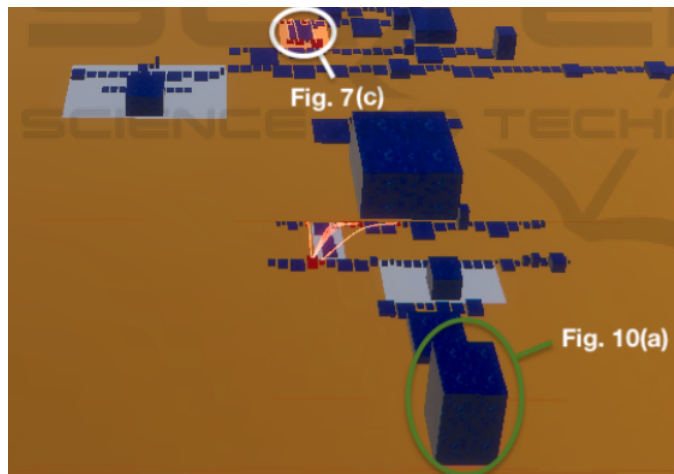


Figure 4.40: The Layered Software City [100]

34. **DynaCity** Dashuber and Philippsen [16] present DynaCity (see Figure 4.41), which extends the Layered Software City. DynaCity shows dynamic data, such as calls between functions. The edges and buildings' brightness represents the number of calls. Buildings represent classes, where its height shows the fan-in (number of calls initiated from the class), and its surface shows the fan-out (number of calls to this class).

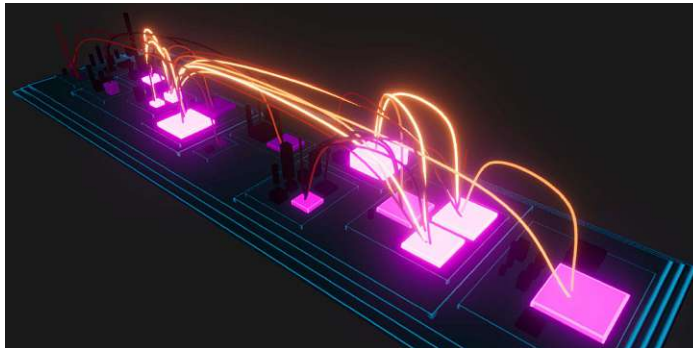


Figure 4.41: DynaCity [16]

35. **BabiaXR-CodeCity** Moreno-Lumbreas et al. [13] reimplemented CodeCity [10] with BabiaXR tools to make CodeCity accessible on-screen and in an immersive VR environment. Buildings represent files. Its surface is proportional to the number of functions, its height to lines of code per function, and its color to the Cyclomatic Complexity Value [101]. The positioning of buildings uses a spiraling algorithm where the first building is positioned in the middle and then the rest in a spiral. The algorithm is applied recursively on the directories.

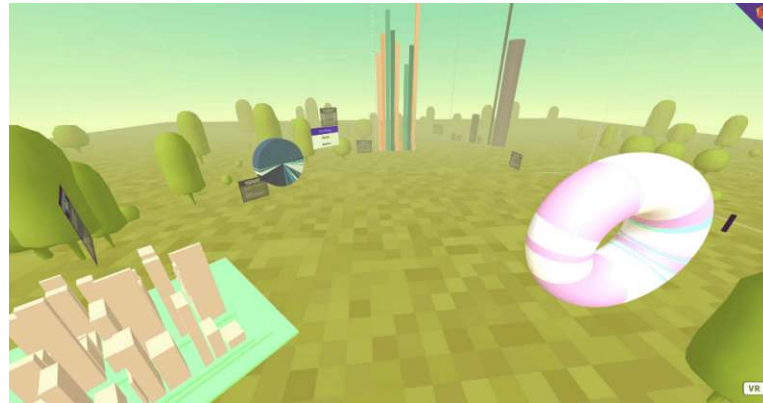


Figure 4.42: BabiaXR-CodeCity [13]

36. **VariCity** Mortara et al. [14] designed VariCity (Figure 4.43) to visualize relationships and design patterns. They calculate for classes variation points (vp-s) and variants. Variation points identify locations where a variation may occur; for example, a constructor of an abstract class is a variation point, while the constructors in the inheriting classes are variations. Buildings represented files, where the height represents the number of variants at the method level, while the building's base surface represents the number of variants at the constructor level. Moreover, the building may have on top structures representing the used design pattern, such as pyramids for an entry point class; domes for the strategy pattern; chimneys for the factory pattern; inverted pyramids for the template pattern; and spheres for the decorator pattern. The city uses streets to show usage relationships and edges to show inheritance.

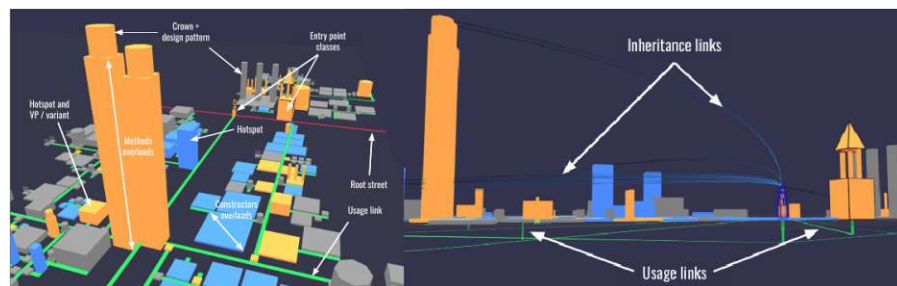


Figure 4.43: VariCity [14]

4.2 Code Cities Tables

The tables below show a summary of the listed code cities to see their differences. From the thirty-six code cities, six of them involve multiple cities in one visualization, where one city can represent a file (three of the visualizations), class, bundle or package. Twenty-eight from the rest of the visualizations render the software source code into one city visualization where CodeMetropolis can switch between showing the whole source code to showing only one file as a city. The most special are UML-City and Linked Data City which show UML and linked data respectively with the code city metaphor.

	Visualization Name	Year	Multiple Cities	Augmented Reality	City Repr- sents	Directories / Pack- ages Representa- tion
1	FileVis [20]	1998	✓		file	n/a
2	Software World [8]	1999	✓		file	country
3	Software Landscapes [65]	2004	✓		class	hierarchical spheres
4	Vizz3D [66, 67]	2005-2007	✓		file	landscapes with dif- ferent depths
5	VERSO [68, 69, 70]	2005-2008			software	hierarchical tree lines over the base
6	CodeCity [10, 61, 72, 63, 102]	2007-2008			software	districts over base
7	CocoViz [73]	2007-2008			software	n/a
8	UML-City [75]	2007			UML	n/a
9	EvoSpaces [76]	2007			software	green shades direc- tory depth
10	Evo-Streets [6]	2010			software	streets
11	VizzAspectJ City / Vizz- Java City [78]	2011			software	districts over base
12	VITRAIL [79]	2011			software	streets, districts over base
13	SkyscrapAR [80]	2012		✓	software	districts over base
14	SynchroVis [81]	2013			software	districts over base
15	SeeIT 3D [21]	2013			software	could be a building
16	ExplorViz [82, 84, 86, 83, 85]	2013		planned	software	buildings, districts over base
17	TeamWATCH [64]	2015			software	districts over base
18	CodeMetropolis [87]	2015			software, file	a district or a build- ing
19	Rocat [88]	2016			software	districts over base
20	Origin City [89]	2016			software	n/a
21	VR City [90]	2017		planned	software	colored base
22	CityVR [91, 92]	2017		✓	software	districts over base
23	High-Rising Cities [93]	2017		planned	software	districts over base
24	Code Park [94]	2017			software	n/a
25	Linked Data City [95]	2017			linked data	n/a
26	GoCity [96]	2019			software	districts over base
27	IslandViz [97]	2019	✓	✓	bundle	district inside the is- land
28	PerfVis [98]	2019		✓	software	districts over base
29	Software City in VR [99]	2020	✓		package	districts over base
30	Memory Cities [24]	2020			software	districts over base
31	m3triCity [7]	2020			software	districts over base
32	m3triCity2 [12]	2021			software	districts over base
33	Layered Software City [100]	2021			software	districts over base
34	DynaCity [16]	2021			software	districts over base
35	BabiaXR-CodeCity [13]	2021		✓	software	districts over base
36	VariCity [14]	2021			software	streets

Table 4.1: Systematic mapping study: First table shows an overview of the city metaphor.

4. SYSTEMATIC MAPPING OF CODE CITY VISUALIZATIONS

		Building			
	Visualization Name	Represents	Height	Color	Programming Language
1	FileVis [20]	method	method length	method complexity	C
2	Software World [8]	method	method length	private, public method	JAVA
3	Software Landscapes [65]	method attribute	n/a (future work: method length)	method vs. attributes	JAVA
4	Vizz3D [66, 67]	function	n/a	n/a	C, C++, JAVA
5	VERSO [68, 69, 70]	class	weighted method (complexity)	coupling between objects	JAVA
6	CodeCity [10, 61, 72, 63, 102]	class	number of methods (configurable)	design problems	JAVA
7	CocoViz [73]	class	methods, lines attributes (configurable)	see god classes	object oriented
8	UML-City [75]	UML-metric	value size	high, low value	Uml
9	EvoSpaces [76]	file	loc (configurable)	header, implementation	C, C++, JAVA
10	Evo-Streets [6]	class	number of modifications	modification time, author	any
11	VizzAspectJ City / VizzJava City [78]	class, aspect	number of methods	the distinction between class and aspect	JAVA, AspectJ
12	VITRAIL [79]	class	n/a	n/a	JAVA (object oriented)
13	SkyscrapAR [80]	class, file	code churn, changes per revision	highlights, current revision	JAVA + (future others)
14	SynchroVis [81]	class	dynamically created objects	floors (instances) colored by thread	JAVA
15	SeeIT 3D [21]	can be a line of code, a method, a class, or a package			JAVA
16	ExplorViz [82, 84, 86, 83, 85]	component, file	number of instances	component - file	(JAVA) Object oriented
17	TeamWATCH [64]	file - commit history	commit number	author	any
18	CodeMetropolis [87]	classes, methods	loc	configurable	JAVA
19	Rocat [88]	file	loc	top contributor	multiple
20	Origin City [89]	file	configurable	horizontal layer by functionality	multiple
21	VR City [90]	class	number of methods	author	JAVA
22	CityVR [91, 92]	class	count lines, method, attributes	number of lines, methods, attributes	JAVA
23	High-Rising Cities [93]	method, classes, packages	dynamic fluctuation of the method execution	n/a	JAVA
24	Code Park [94]	file	class size	class size	C#, can any object oriented too
25	Linked Data City [95]	ontology class	ontology instances	internally and externally defined data	Linked data
26	GoCity [96]	file, struct - GO specific	method number	darker more lines of code	GO
27	IslandViz [97]	class	lines of code	n/a	JAVA
28	PerfVis [98]	class	number of methods	number of called methods	object oriented
29	Software City in VR [99]	class	loc configurable	configurable	object oriented
30	Memory Cities [24]	heap objects	object replications	object replications	JAVA
31	m3triCity [7]	class	lines of code, number of methods, number of for loops or number of variables	n/a	multiple (only JAVA examples)
32	m3triCity2 [12]	class, data, Binaries			
33	Layered Software City [100]	class	number of incoming dependencies (depth outgoing)	red for cycles	JAVA
34	DynaCity [16]	class	fan-in is a measure of the number of functions or methods that call some other function (depth fan-out)	brighter has more calls	JAVA
35	BabiaXR-CodeCity [13]	file	lines of code per function	cyclomatic complexity value	any
36	VariCity [14]	class	method variants	hotspot, high variability	object oriented

Table 4.2: Systematic mapping study: Second table lists how the buildings are constructed in different code cities and the programming languages of the software projects that the visualizations render.

Beside the common features of buildings, districts, floors and colors, some features were used only in a few code city visualizations. Table 4.3 shows how often these feature occurred and Table 4.4 lists the visualizations using these features.

	Feature	Ocurrence
1	Zoom: Rendering other details on zoom	6
2	Wires: mapping data on wires/coords which connects buildings	15
3	Building position: The placement of the building represent a specific metric from the source-data	8
4	Dynamica data: The code city renders live runtime data	7

Table 4.3: Systematic mapping study: Special features occurrence that are not commonly found in other code cities.

	Visualization Name	Special Attributes	Dynamic Data	Static Data
1	FileVis [20]	on zoom: line code information		current state of the code
2	Software World [8]	on zoom: different levels such as countries, cities, buildings, floors		current state of the code
3	Software Landscapes [65]	on zoom: each sphere has cities internally representing files; wires: function calls, inheritance, attribute access		current state of the code
4	Vizz3D [66, 67]	wires: call operation (special color for more calls)		current state of the code
5	VERSO [68, 69, 70]	buildings position: twists the buildings		code quality, cohesion, coupling complexity
6	CodeCity [10, 61, 72, 63, 102]			current state of the code
7	CocoViz [73]	buildings position: maps metrics to x-, y-axis like a graph; interaction: audio feedback on house design state		current state of the code quality
8	UML-City [75]	wires: show UML relationships		UML-data
9	EvoSpaces [76]	on zoom: inner metrics of the file; wires: indicate relationships		current state of the code
10	Evo-Streets [6]	buildings position: central buildings are older		code history svn
11	VizzAspectJ City / Vizz-Java City [78]			current state of the code
12	VITRAIL [79]	wires: call operation (special color for more calls)		files and static calls
13	SkyscrapAR [80]			code churn history
14	SynchroVis [81]	wires: call operations (multiple edges for more calls)	objects on runtime	
15	SeeIT 3D [21]			current state of the code
16	ExplorViz [82, 84, 86, 83, 85]	wires: function calls; interaction: open close folders	objects on runtime	
17	TeamWATCH [64]			all commits splited
18	CodeMetropolis [87]	on zoom: go inside the building to see more metrics; interactions simulates Minecraft		current state of the code
19	Rocat [88]	interactions change code dynamically		current state of the code
20	Origin City [89]	buildings position: groups buildings by organization		functionality
21	VR City [90]	wires: implemented by same author		current code
22	CityVR [91, 92]			current code
23	High-Rising Cities [93]	interaction: life fluctuation of method execution	method execution fluctation	
24	Code Park [94]	on zoom: shows code inside the room,		source code
25	Linked Data City [95]		-	data
26	GoCity [96]			current state of the code
27	IslandViz [97]	wires: show dependencies; buildings position: group buildings in islands		current state of the code

4. SYSTEMATIC MAPPING OF CODE CITY VISUALIZATIONS

28	PerfVis [98]	interaction: runtime method calls	color represent number of method calls	current state of the code
29	Software City in VR [99]	wires: method calls; buildings position: attraction force by calls	calls methods	current state of the code
30	Memory Cities [24]	buildings position: distributed by type and then by allocation	heap objects	
31	m3triCity [7]	wires: file movements		current state of the code
32	m3triCity2 [12]	wires: file movements		current state of the code
33	Layered Software City [100]	buildings position: based on dependency hierarchy; wires: indicate dependency cycles;		dependencies
34	DynaCity [16]	wires: function calls - brighter colors indicate more calls	show the runtime amount of calls	
35	BabiaXR-CodeCity [13]			current state of the code
36	VariCity [14]	wires: inheritance relationships; interaction: show design patterns		current state of the code

Table 4.4: Systematic mapping study: Third table lists special features that are not commonly found in other code city visualization and the kind of data mapped to the city.

CHAPTER 5

Requirements

This chapter describes the requirements that the visualization implementation will follow next. The vision and state of the art derive the initial requirements, as described in Section 5.1. Then the requirements will be validated by the semi-structured expert interviews described in Section 5.3. The questionnaires may include questions to define further requirements.

5.1 Vision & Derived Requirements

The proposed visualization Slide City is a software evolution visualization based on the code city metaphor. The back-end side of the software mines all the commits of a selected Git branch to show ownership information, such as the highest owner. The highest owner of a file or folder is the developer with the highest authorship of the file or folder in terms of lines of code. Depending on the added and removed lines, the highest owner may change with every commit.

Slide City maps the historical data mined from Git into the height of the respective buildings they relate to, starting from the earliest commits at the bottom to the latest commit on the top of the building (see Figure 5.2). The user can slide up the buildings' floors representing commits to see the timeline (see Figure 5.1).

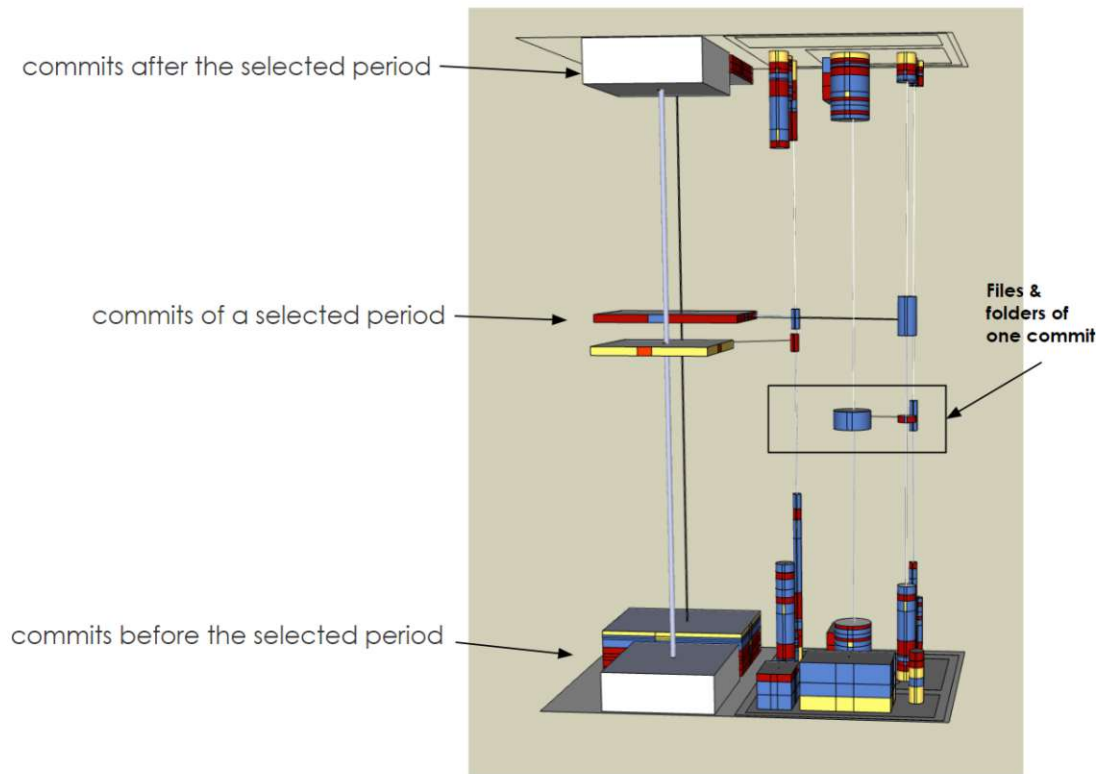


Figure 5.1: A sketch of slide city.

5.1.1 Slide City Buildings

The city base consists of cylindric buildings representing files and cuboid buildings representing folders. Folder buildings can become districts when the user opens them to see their internal files and folders as buildings in a recursive order. The sketch in Figure 5.2 shows the two different buildings that Slide City contains.

The building's surface shows the current file or folder size in lines of code, i.e., a wider building contains a larger code base. In the folder's case, its surface represents the source code of all the files inside, making the whole city surface relative to the source code size.

The building floors represent the commits. Each floor's volume is related to the commit size. The visualization calculates the height of the floor by dividing the floor volume (i.e., the size of the commit) by the building's surface (i.e., the current size of the building).

Slide City colors the floors according to the highest owner at that specific commit. New commits from new authors may not necessarily imply that the new author also has the highest code ownership. Therefore, the highest owner color may be combined with the commit's author color. For example, Figure 5.2 has colored stripes in the middle of the buildings that represent the committer. The floors' colors on the left side of Figure 5.2

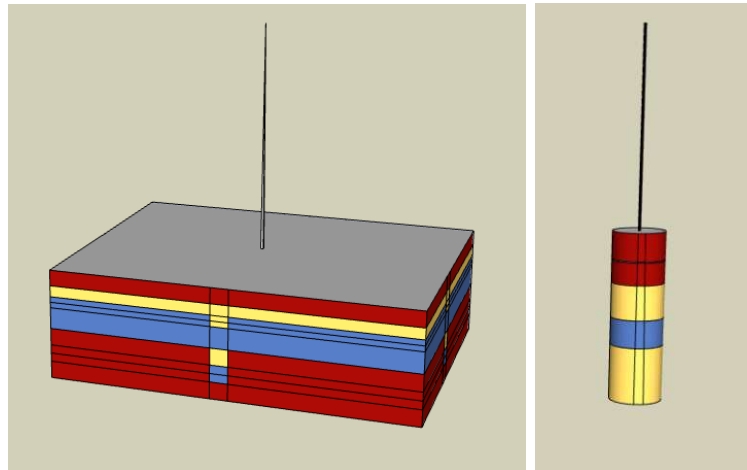


Figure 5.2: Slide City buildings: The left building represents a folder. The building on the right represents a file.

shows that the first commits of the „blue“ author (the first and second floor from the bottom) are too small to take ownership of the „red“ author.

5.1.2 Opened & Closed Folder

Liu et al. [4] code city metaphor show all the project files as cylinders, while the districts on the floor represent the folders (see Figure 3.15). However, different stakeholders may be interested in various levels of the project. For example, if a manager wants to see the historical ownership of a large folder or compare it to another large folder, then showing all the inner files may bring too much noise. Turning districts into buildings interactively may help the user to pick out the needed information efficiently. Fittkau et al. [83] implemented a similar approach for ExplorViz, where the number of created instances could be mapped per file or package into buildings. The user could manually turn the package buildings into districts to see more details.

Figures 5.3 and 5.4 show a simplified example of folder X in two modes. Figure 5.3 shows folder X as a district containing one file and two other sub-folders. While Figure 5.4 shows folder X as a building. Both figures have red lines that tell the commit's hash mapped to the floor. The hash are simplified for readability into names from r1 (the earliest commit) to r7 (the latest commit) and colored according to their author. The label's color corresponds with the stripes in the middle of the buildings because both represent the committer.

The perspective of Figure 5.3 is helpful if the user is interested in a detailed history of the sub-parts of the selected folder. However, commits can change several files simultaneously. Without the labels telling the commits, it would be impossible to figure out how many commits are involved in the X folder. In Figure 5.4, it is immediately visible that the X

folder has only seven commits. Additionally, it is easier to figure out the order of the commits and how the ownership of the X folder has changed.

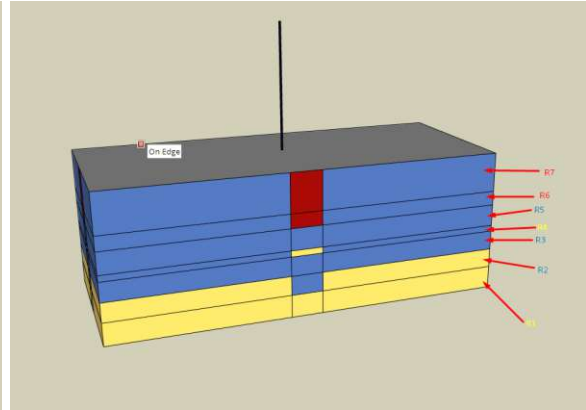
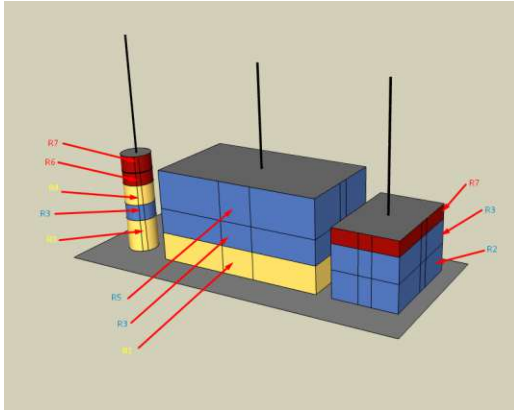


Figure 5.3: The detailed level of folder X. Figure 5.4: The general level of folder X.

Figure 5.4 shows that even though there have been commits from three different authors, there are only two authors that had the largest ownership during the folder X implementation; the „yellow“ had the largest ownership on the first commits; the „blue“ author had the largest ownership on the latest commits. The top floor may look intuitive that the „blue“ author is the highest owner even when looking at Figure 5.3 because the blue author keeps the highest ownership of the largest sub-folder. However, this can also be misleading. Another case would be that the „blue“ author owns the largest sub-folder in Figure 5.3 only by a slight degree higher than the „red“ author¹. While the „red“ author overtakes with the commits R6 and R7 a high degree of ownership of both the file and the smaller sub-folder resulting in an overtake of the ownership of X. This example shows that analyzing the rich information of the sub-parts of X may be misleading to have exact ownership information for a folder. The general level may be more helpful when seen as a part of the city (see Figure 5.5).

5.1.3 Slide Feature

Commits can be related to multiple files and folders. Each building floor can be only a part of such commits. Using edges to connect the floors that belong to the same commit may risk covering the whole city with wires. Slide City allows moving the floors up regardless if they are closed folders or files and automatically connects the floors that belong to the same commit. The user can see from a horizontal point of view the list of commits going up from the latest to the earliest (see Figure 5.1).

¹The supposed case example might not be quite exact because the „red“ author has pushed no commits in the largest sub-folder and therefore has an ownership degree of zero

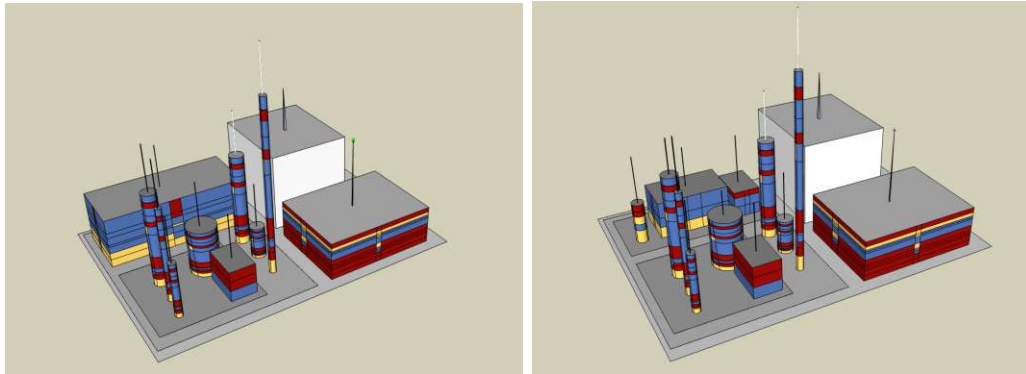


Figure 5.5: Slide City sketch. The folder on the top-left corner is shown on a general level on the left and on a detailed level on the right.

5.2 Proposed Features (Open questions - Semi-Structure Expert Interviews)

The „Information Needs“ described in Section 3.1 and the vision from Section 5.1 gives insight into a large part of the requirements. However, there are still open questions on how the visualization should behave or look. This section describes the proposed features that will be defined and validated with semi-structured interviews.

5.2.1 Committer Color Mapping

The visualization colors each building floor according to the highest owner. However, the floor may also show the committer. Figure 5.6 shows different cuboid and cylindric building options, combining the committer color with the highest owner color. For example, the first options represent the committer with stripes in the middle of the buildings, while the color surrounding the floors represents the highest owner. However, looking from a general level (e.g., Figure 5.1), the stripes may need to be clearer. Figure 5.6 shows further options to tackle this problem, for example, small circles in the middle of the floor or multiple diagonal stripes.

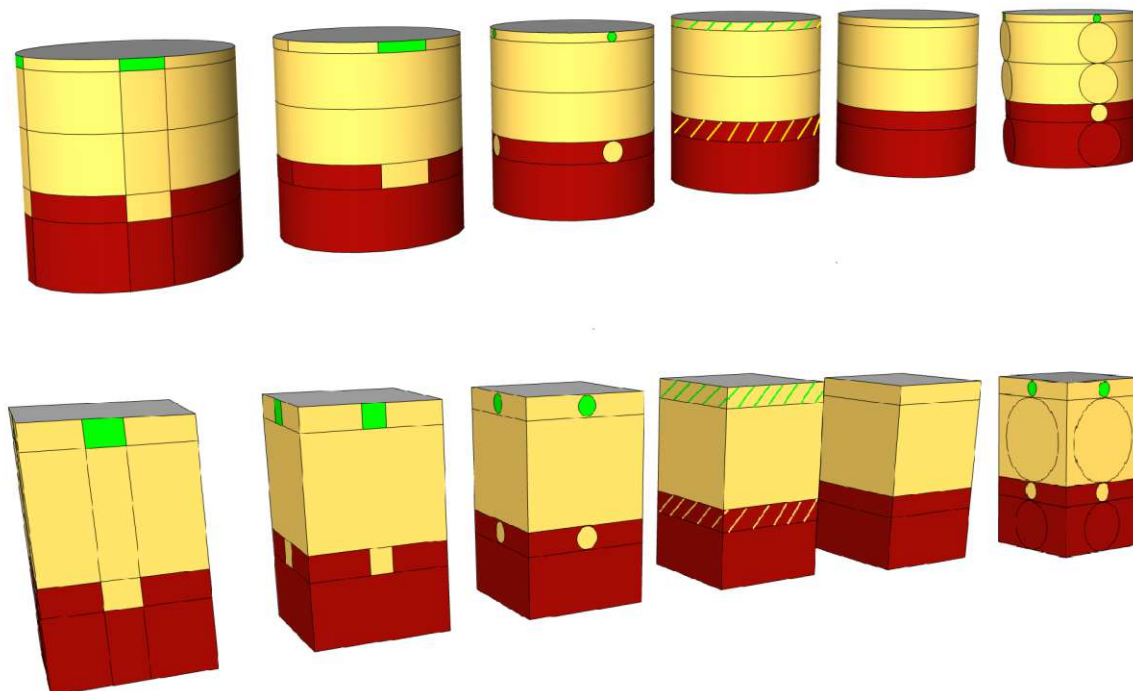


Figure 5.6: Different possibilities to color the committer.

5.2.2 Mapping Colors to Multiple Authors

A large project may have many authors. Especially open source projects have a lot of minor developers who commit small fixes [30]. Having colors with shades close to each other mapped to authors may make it difficult to differentiate them. Some options on how to tackle this problem are:

- Assign all authors random colors.
- Let the user assign the colors. This option can be selected combined with other options.
- Assign to minor authors only shades of a particular color (e.g., red) and assign the rest of the colors to the primary authors. When many developers have contributed, for example, more than twenty, the user may be interested in those who have contributed the most. The minor developers, in this case, can be those who have contributed less than 20% combined.
- Assign to authors only dark colors. The user can then manually switch authors of interest to a bright color. The bright color can be selected automatically as the

opposite of the dark or a random bright one. The chosen authors (those with a bright color) will be distinguished easier than the rest.

5.2.3 Height of the Building

The volume of the floor can represent only the added lines or both added and deleted lines. An indication that many lines were deleted is when the building is tall but has a small surface, i.e., several lines were added but also deleted. However, when the deleted lines are not included in the floor volume brings the drawback that the user cannot see with which commit the code was deleted, and commits that have only deleted lines will not be shown at all.

5.2.4 Metrics Mapping Technique

Wettel et al. [61] used two methods to map the attributes of the buildings. The linear mapping (Figure 5.7-left) renders into the buildings the exact proportional size of the metrics. It gives a precise overview of the metrics. However, it comes with the cost that there will be a few buildings too high and a few others too small. Mapping the attributes to a prepared set of buildings using the box plot and threshold-based mapping techniques renders a more uniform view of the city (see Figure 5.7 right).

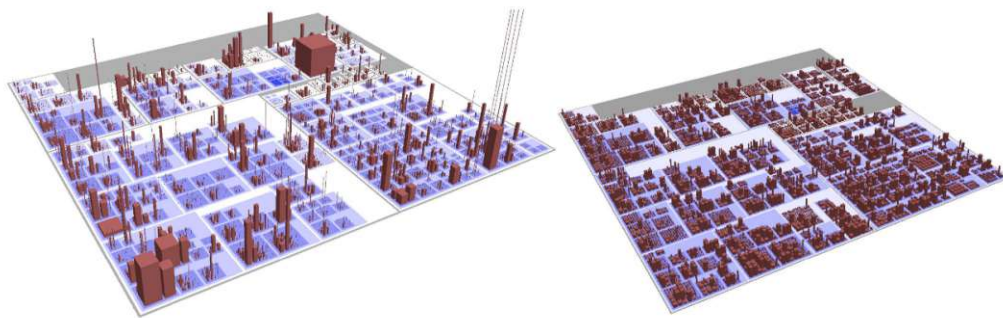


Figure 5.7: Linear and box-plot mapping [61]

Another anomaly may happen when the visualization shows a few very large (e.g., with 50000 lines of code) and a few very small files; then the 99% of the files that are in the range of 100 to 200 lines of code may risk looking equal if they are mapped into a range from 0 to 50000 lines. Some techniques to handle mapping the metrics to the visualization are listed below:

- **Linear mapping:** Maps attributes one to one from Git-data to slide city.
- **Box-whisker plot:** Maps data into four quartiles and shows data between a range as equal (see Figure 5.8 b).

- | | | | | | | | | | |
|----|----|-------|-----|----|-------|------|----|-------|--|
| 1 | -> | 0 | | | | | | | |
| 10 | -> | 1 | 100 | -> | 2 | 1000 | -> | 3 | |
| 20 | -> | 1,301 | 110 | -> | 2,041 | 1010 | -> | 3,004 | |
| 50 | -> | 1,699 | 500 | -> | 2,699 | 5000 | -> | 3,699 | |

(a) Histogram equalization.

(b) Box-whisker equalization.

Figure 5.8: Histogram & Box-whisker equalization [56]

The Slide City buildings look like a solid construction where each commit comes after the other. However, this is not quite the reality of how it looks in Git history. Developers can update the same files and folders in their feature branches and later merge them at the main branch. Figure 5.9 shows a part of the file history. The branch on the left is the main branch. Commit A and E are part of a feature branch. The second feature branch is shown on the right with commits K, L, and B. Commit F may be from another author who helped on the second feature branch and then merged it to the feature branch with commit B. Note that the commit H changes the file directly in the main branch.

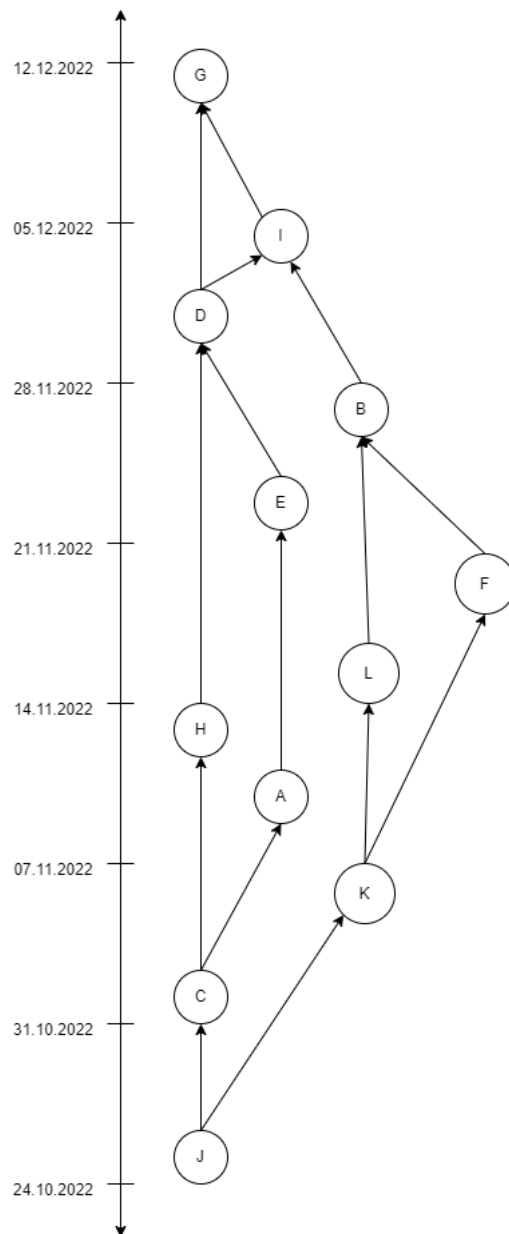


Figure 5.9: The Git history of a file changed in parallel.

The first feature branch is merged with the commit D. The developer implementing the second feature branch pulls the main branch and resolves conflicts with commit I. Then the developer merges the second feature branch with main by creating the commit G. There are different ways how the commits in Figure 5.9 can be sorted:

1. **By date:** This sorting ignores the branches and simply lists them by date. In this

case, the order would be J, C, K, A, H, F, E, B, D, I, and G. This sorting also correlates with the slide feature (see Section 5.2.9).

2. **The branch with the first merge commit comes earlier:** This sorting prioritizes the merge commits. The commits before the earliest merge commit are sorted into the front. In Figure 5.10, the earliest merge commit from commit J is B. Therefore, the branch starting in K is sorted first: K, L, F, and B. The next merge commit from commit J is D. Commit A from the first feature branch is before commit H, so this branch gets priority. However, it is arguable if the first feature branch should be sorted after H because it contains commits after H (e.g., commit E). The sorting would then be C, A, E, D, and D or C, H, A, E, and D. At this point, the algorithm has sorted J, K, L, F, B, C, A, E, H, and D. The next merge commit I has all its previous commits sorted, so the algorithm adds it at the end of the list. It follows the same logic also with G. The result would be: J, K, L, F, B, C, A, E, H, D, I, G or J, K, L, F, B, C, H, A, E, D, I, G.

5.2.6 Consecutive Commits

Consecutive commits from the same author can be merged and visualized as a single floor. Girba et al. [3] used a similar feature in Chronia visualization. However, the colored lines in Chronia (see Figure 3.9) represent only the committers. Slide City shows the code ownership, which may or may not be changed by the commit of a new author (e.g., the first floor in Figure 5.10 represents a commit from a new author but not as large to overtake the ownership). The third and the fourth commit come from the same committer and have the same ownership. As they follow each other, they are shown as merged in the second building of Figure 5.10. However, there are also other possibilities to combine the floors. For example, the third building in Figure 5.10 merges all consequent commits from the same author and calculates the largest ownership. The fourth building merges those commits with the same author as the highest owner, regardless of who the committer is.

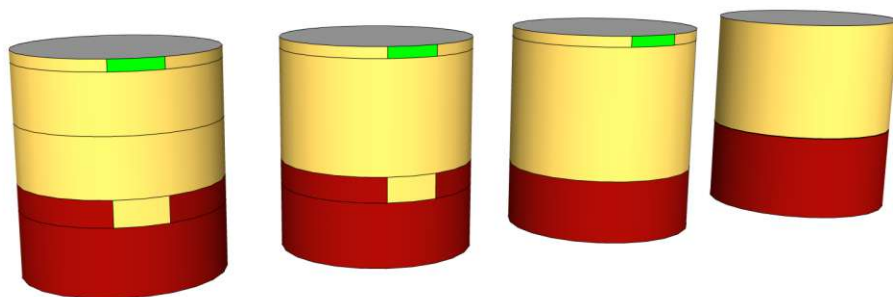


Figure 5.10: Different visualization of merged commits.

5.2.7 Merge Commits

Merge commits are the commits that have one or more parents. Git flow workflow suggests that features should be done in a separate branch and later merged into the main or develop branch with a merge commit². Although a merge commit may have more than two parents, according to Git-Workflow², this is not a good practice and is rarely used. The merge commits should introduce no changes. However, merge conflicts may happen. The conflicts require the developer's attention to resolve and push the conflict's solution with the merge commit. The merge commit diff in Git shows all the differences with any parent branches (i.e., it lists all the changes that are not contained, at least in one of the parent branches). However, in the visualization, this may duplicate the previous commits because every change from one parent branch is not present on the other. Another approach, except for ignoring the merge commits, is to show only when a change differs from all the parent branches. This change happens when the developer has manually resolved a conflict.

5.2.8 Building construction strategy

During project implementation, the structure of folders and files may change several times. Depending on the situation, the developer may rename files or folders or move them to other places. Developers may remove folders and split their content into current or new folders. The files themselves can be split, removed, or maybe added again later. These structures can become even more complex when considering branches, like in Section 5.2.5. Files and folders can be changed, moved, and deleted differently and simultaneously in different branches. Slide City must consider these special cases when rendering the software history in its buildings.

Showing the whole history of the files and folders may sound intuitive, but it includes several trade-offs. The following example will be used to illustrate the different strategies:

1. **Commit 1: Red author** creates **Folder A** and **Folder B**;
creates **File C** in **Folder B**.
2. **Commit 2: Yellow author** commits changes to **Folder B**;
moves **File C** to **Folder A**;
deletes **Folder B**,
3. **Commit 3: Green author** commits changes to **Folder A**;
creates **Folder D**, and
moves **File C** to **Folder D**.

²<https://www.atlassian.com/de/git/tutorials/comparing-workflows/gitflow-workflow> version of 11.07.2024

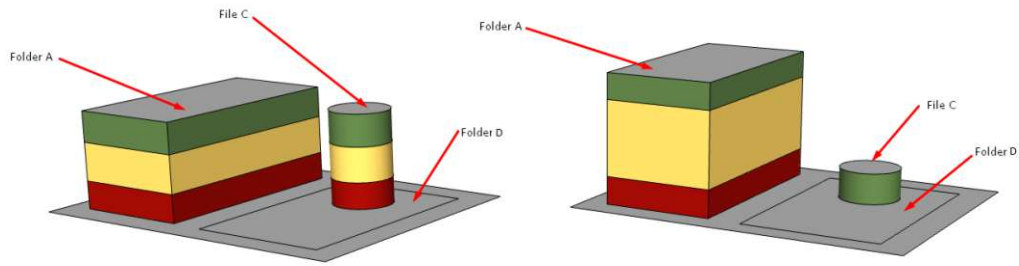


Figure 5.11: Intact File history (left) and Split File History (right)

Figure 5.11 shows two slide city bases options of how the example can be visualized. It shows the history of folder A from a general level and the history of file C, which is currently in folder D. For simplicity, the example assumes that the committer also overtakes the ownership; therefore, the committer color is the same as the highest owner color. The main point of the example is how to show the history of file C. File C is initially created in folder B, then moved to Folder A, and lastly moved to folder D. The movement of files and the restructuring of folders often happens during refactorings. The following two options discuss the techniques for handling file movements and the pro and contra arguments.

1. **Intact File history:** This technique keeps every commit related to a folder intact in one building. If the file has moved, its whole history moves together with it. This solution may be the most intuitive one for Git users. Git tracks only files instead of folders. It considers folders only as a path of the file that it is tracking. When moving a file, Git considers it a rename of the file (i.e., a rename of the path). Figure 5.11 left shows this solution. Although file C was part of folder A in the second commit, file C is currently part of folder D. Therefore, all the changes to file C are shown inside folder D (i.e., in the history of file C).

The drawback of this strategy is that it ignores the history of the folders. The strategy considers the folders as a grouping of files in the current state (i.e., in the last commit). This folder structure, even when shown as a building, shows the history of the files it contains instead of its history itself. For example, on the left of Figure 5.11, folder A does not include file C in its commit even though it was part of it in the second commit, and Folder B is not shown at all because it does not have any files.

2. **Split File History:** This technique shows each file modification inside the folder where it was changed. In contrast to the „Intact File History“ strategy, the latest folder where the file was moved includes only the changes of the file that happened in that folder. Figure 5.11 right shows this solution. The second commit of folder A is larger than on the left Figure because Folder A contains the code from file C at that commit.

This approach has the drawback that the whole file's history may not be shown. Folder D now includes only the latest commit of file C (i.e., the commit when file C was moved to folder D). The second commit of File C is integrated into the second commit of Folder A. However, if the folder is opened to a district, this commit cannot be shown until the timeline returns. Showing the proper folder's history comes with the disadvantage of splitting the file's history.

The second drawback is that the visualization loses the first commit because it belongs to a previously removed folder. The floor can be reshown when the user slides the city towards the past, which will be discussed in Section 5.2.9. However, much of its history will be hidden at the initial view of the code city.

5.2.9 Slide feature

One of Slide City's most innovative code city features is the slide feature. The user can slide up the building's floors representing the pieces of the commits. While the floors slide up, the visualization connects the floors representing pieces of the same commit by wires. The connected floors slide vertically in parallel to make it easier to identify the relation.

Moreover, while the city slides to the top, it should keep the detailed level the user has set. For example, the user can leave opened a couple of folders and others closed. After some of the commits have moved up, the city base shows a paste state of the software.

Commits contain information about renames or movements of files and folders besides the added and deleted code lines. Folders and files can be renamed, split, merged, or deleted. Therefore when the commits move up, besides reversing the changed lined numbers, the city base has to handle the revert of the folder structure, which in large refactorings can change drastically.

However, the slide feature depends on how the buildings are structured, discussed in Section 5.2.8. The strategy followed to construct the buildings influences how the buildings will be deconstructed when the commits slide up. Below are described both construction techniques with slide strategies and how the example from Section 5.2.8 would look like when sliding up:

1. Intact File history:

This technique ignores the movement of files when creating the buildings. Figure 5.12 shows in four timestamps how the floors would move in this case. From the example File C was created inside folder B the first commit. However, the file C is together with its history in folder D, therefore it remain in all sliding sequences in the same folder. Folder B remains hidden in the visualization.

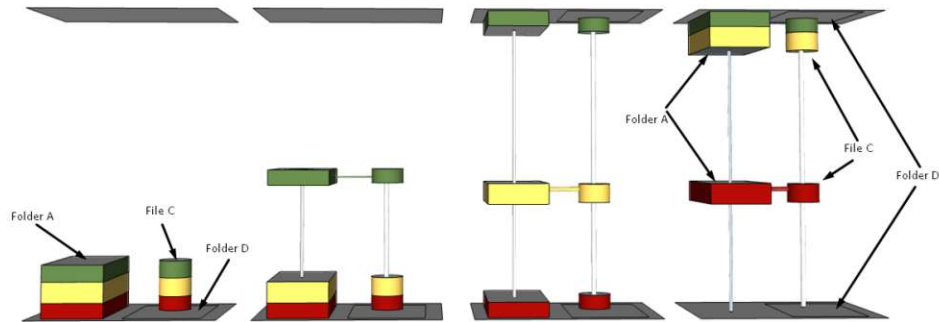


Figure 5.12: Slide City, where file buildings always contain full history.

2. Split File History:

This strategy considers additionally to the file history also the folder history. Therefore, a file building has only part of its history intact as long as it is part of the same folder's history.

Figure 5.13 shows the slide city in different timestamps following this strategy. In the first stage, file C has only the commit that is also part of folder D. When the yellow commit moves up at the third timestamp, folder D, which the green commit had created, disappears from the base with its content the file C. This timestamp shows Folder B, created during the first commit and has the first commit of File C. The second commit of file C is inside the second floor of folder A.

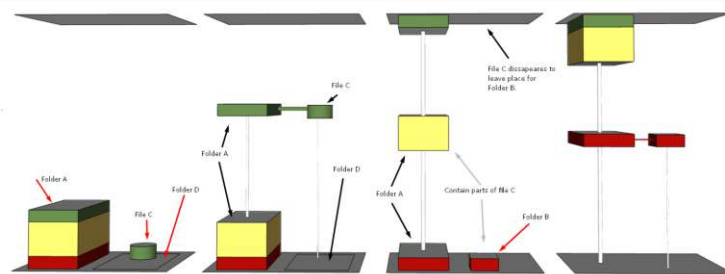


Figure 5.13: Slide City, where buildings consider file and folder history.

5.2.10 Apply generality level

Slide City provides the possibility to close and open folders manually. The user can open a folder to see its content, and Slide City renders its internal folders and files as buildings. However, it can be very cumbersome to open the details level of multiple folders. This concern can be solved by a feature that allows opening and closing all folders at a specific level of generality.

For example, if the user selects the fourth level, all the folders, from the project's root folder to depth four, will be shown as districts on the city base. The rest of the folders will be rendered into cuboid buildings.

After applying the generality level, the user can still manually close or open other folders. If the user opens a folder manually, it remains to decide whether the visualization should keep the state of this folder after the user re-applies the generality level.

5.2.11 Filter by Authors, Files and Folders

To answer questions like „Where is someone working?“, it is helpful to filter and spotlight the commits related to the author. Slide City may offer two filtering methods. One is by commits, and the other by authors. The user may select more than one author to make a comparison.

Slide City can render the city when filtering by author in two possibilities. It can show only the buildings and floors related to the author and hide the rest, or the whole city but color only the floors related to the author. A similar decision can also be taken for filtering by files and folders.

5.2.12 Blocklist

Usually, developers leave artifacts and generated files and folders out of the repository and do not commit them³. However, they do not always apply this rule to any file, especially in the project start phase. For example, package-lock.json⁴ from npm projects written in JavaScript or TypeScript projects is usually kept in a repository. This file may become very large in comparison to others. While the largest implemented files may be a few hundred lines of code, package-lock.json can reach a hundred thousand lines. Files like package-lock.json can make some commits look very large when only a dependency version is changed. Slide City can offer the possibility to blocklist files to deal with the noise of generated files kept in the repository.

A similar solution can also be done for some particular commits. For example, Slide City can exclude from rendering some blocklisted commits. However, the effects of one commit are not shown only on the floors representing the commit but also on the floors above. For example, a file contains two commits from different authors: Author A pushes the first large commit, and author B the second smaller one. The first commit is added to the blocklist, which results in Slide City rendering only the second commit. The real largest owner of the file is author A, even after the second commit, because the first commit is much larger. In this case, it can be possible that the user not only blocklists the commit not to be shown as a floor but also from the ownership calculation and the files' surface.

³<https://www.atlassian.com/git/tutorials/saving-changes/gitignore> version of 11.07.2024

⁴<https://docs.npmjs.com/cli/v8/configuring-npm/package-lock-json> version of 11.07.2027

5.3 Semi-Structure Expert Interview Results

Semi-structured interviews will be conducted to gather data for the proposed features in Section 5.2. According to Kallio et al. [22], interviews are the most used data collection method, whereas semi-structured is the most used format. They found that one of the reasons for the popularity of the semi-structured interview is that the technique has successfully enabled reciprocity between the interviewer and participant and allowed improvisations for follow-up questions based on the participant's responses [22]. The full questionnaire is in Appendix 9.1.

5.3.1 Pilot Phase

The pilot phase was conducted to find any issues with the questionnaire before the interviews. The pilot phase helped identify too complex questions. Those questions were improved further with sketches, graphs, and terms descriptions. During this phase, we concluded that a handout that contains a summary of sketches and terms would be helpful to the interviewee. The introduction and questions with long descriptions, such as the first question in Section 4 or the question about scaling the metrics, were copied to the handout. Additionally, we decided to allow voting options with similar ranks for the ranking questions.

5.3.2 Questionnaire Phase

The interviews were held in online meetings, which lasted around 1 hour. The interviewer shared their screen and sent the handout so each interviewer would receive the same introduction. Further explanations in cases where necessary were handled with short questions and answers. Most of the questionnaire is composed of ranking questions; therefore, the interviewer asked questions about the reasons why a particular ranking was preferred. The ranking questions are followed by an optional question where the interviewee can suggest further options. The full questionnaire can be found in Appendix 9.1.

5.3.3 Results

The results of the interviewee provided an important starting base to figure out which features and how they should be implemented. However, during the implementation, some options showed that they did not result as expected, and sometimes they would hinder the performance. This section summarises the interviewees' answers and preferences. The Implementation chapter (Chapter 6) discusses cases where deviation from the requirements was necessary.

Demographics

Two of the interviewees were two male junior software engineers under thirty years old with five to ten years of experience with Git, and the third interviewee was a senior

female engineer aged thirty to forty years old with more than ten years of experience in Git and having leading roles in software engineering.

Information needs

The pre-given options for the first question, „Who has been working on a feature?“ were ranked differently (see Figure 5.14), indicating that it is unclear how the visualization would help answer the question. A high ranking received the „Last committer,“ the „Present ownership at the file level,“ and the optional suggestion to use the issues and their relation to commits with commit id instead of the given options.

However, the ranking of the second question, „Which features has a developer implemented?“ resulted in more equal answers (see Figure 5.15). The interviewees gave a high rank to all the options and uniformly ranked option „Historical ownership at file level“ as the most important one.

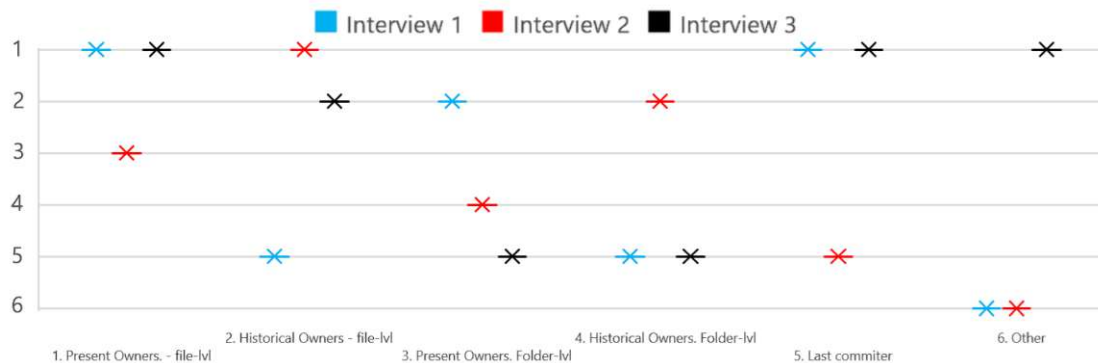


Figure 5.14: Graph for „Who has been working on a feature?“

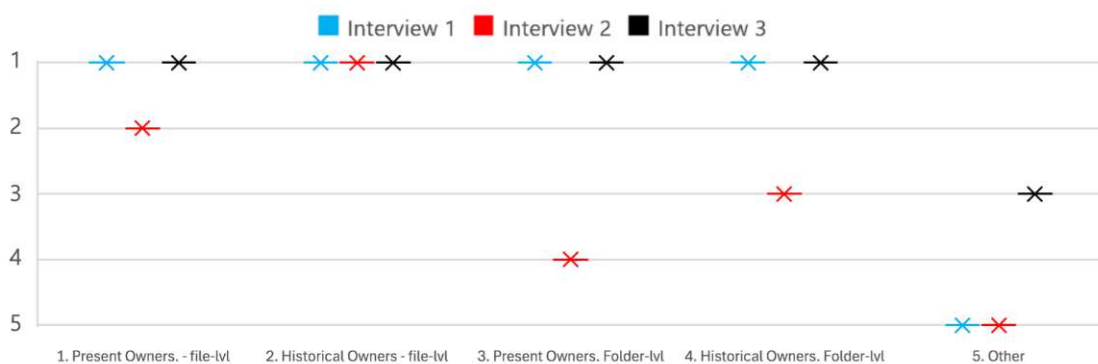


Figure 5.15: Graph for „Which features has a developer implemented?“

Slide City Features

This interview section included questions about preferences for visualization details.

- **How should the committer color be shown along with the highest code owner color?**

The preferences for showing the committer and the owner color varied highly between the three interviewees. For example, the two options to show the committer as a rectangle or circle for each floor were voted the best choice by the first interviewee. The second interviewee also ranked them high and suggested as a more favorable option to combine both, such as showing circles for files and rectangles for folders, would be better. However, the third interviewee ranked both with the lowest score, considering them confusing. The third interviewee preferred the striped design; the first two interviewees ranked it the lowest. However, they all ranked the option to have it configurable in the top two ranks.

- **How to map colors to multiple authors?**

The most preferred option was to assign all authors random colors, and the second most preferred one was to assign authors dark colors and let the user manually switch them to bright colors.

- **What should the height of a floor represent?**

All interviewees preferred including the added- and deleted lines into the building's height instead of just the added lines. One of the interviewees suggested as a third option to identify and calculate the modifications, and another suggested designing different floors for added and deleted lines.

- **How should the visualization map the metrics?**

The histogram mapping was the most ranked option (see Figure 5.16). The third interviewee suggested a configurable option between linear, histogram, and logarithmic mapping.

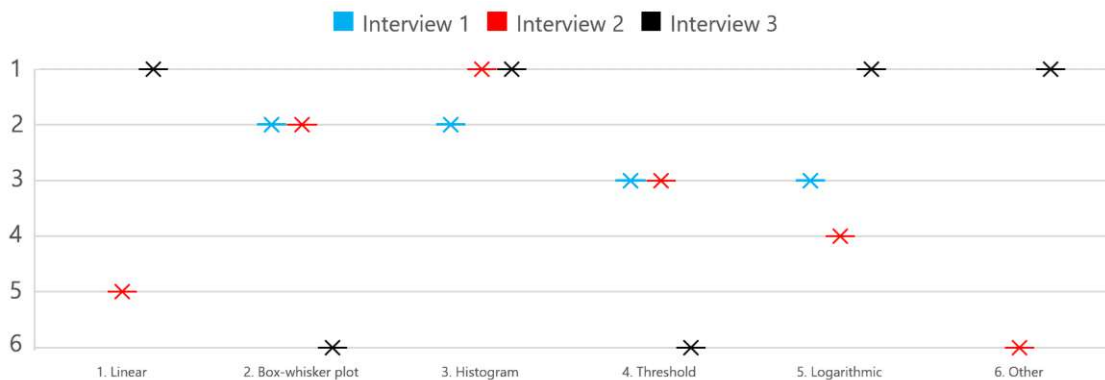


Figure 5.16: Graph for „How should the visualization map the metrics?“

- **How should the commits be sorted?**

The first interviewee found that if the commits were sorted by the date, they would be more understandable. The second one found the sort-by date valid; however, he preferred seeing the alternative option where branches are considered. The third interviewee ranked the sort based on branches as more helpful.

- **How should consecutive commits be shown?**

The interviewees did not prefer combining commits into one floor because the information would be lost. The second interviewee agreed to combine for performance reasons only if two consecutive commits had the same author and owner.

- **How should merge commits be shown?**

The first two interviewees ranked the highest the option to show only changes different to every parent commit, otherwise to hide them. The third interviewee found both options equally valid depending on the situation; therefore, she suggested making it configurable for the user instead.

Building Construction and Deconstruction

Building construction and deconstruction is the feature that affects the visualization the most. Predominately, the interviewees preferred the option „Intact File History“ over „Split File History.“ The third interviewee suggested having it configurable, as both methods show different aspects. For example, „Intact File History“ tells compact information about the code history, while „Split File History.“ shows more information about the recording. However, she finds that the user mostly would need to see the compact information rather than refactorings, which makes the „Intact File History“ method more useful.

Filter Features

- **Should the generality level affect folders opened or closed manually by the user or only those not touched?**

The first interviewee found the generality level useful. However, they did not prefer any of the options. He suggested changing the level only for the files and folders the user had already changed. In contrast, the second interviewee preferred the second option, i.e., to change the level only for the files and folders the user had not changed. The third interviewee suggested that the user should be allowed to choose between the first two options, i.e., to change the generality level for all files and folders or only for those files and folders not touched by the user.

- **How should the visualization be filtered by the author?**

The most preferred option was „Show the whole city but color only the commits related to the author,“ followed by „Show only the buildings related to the author“ (see Figure 5.17).

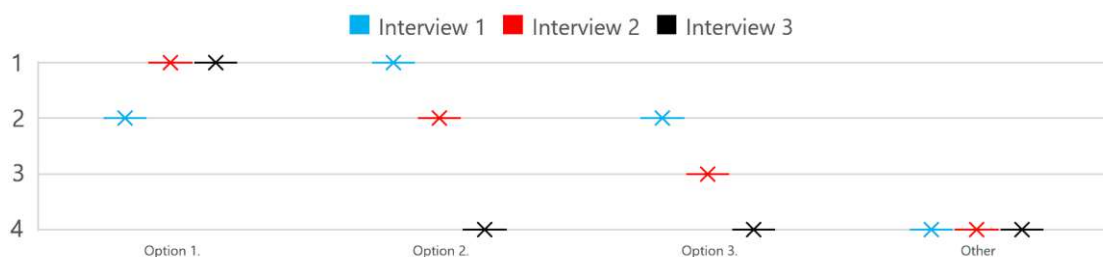


Figure 5.17:

Graph for „How should the visualization be filtered by the author?“

Option 1: Show the whole city but color only the commits related to the author.

Option 2: Show only the buildings related to the author.

Option 3: Show only the floor buildings related to the author.

Other options: -

- **How useful is it to exclude files?**

The interviewees found excluding the file before construction helpful for performance reasons (see Figure 5.18).

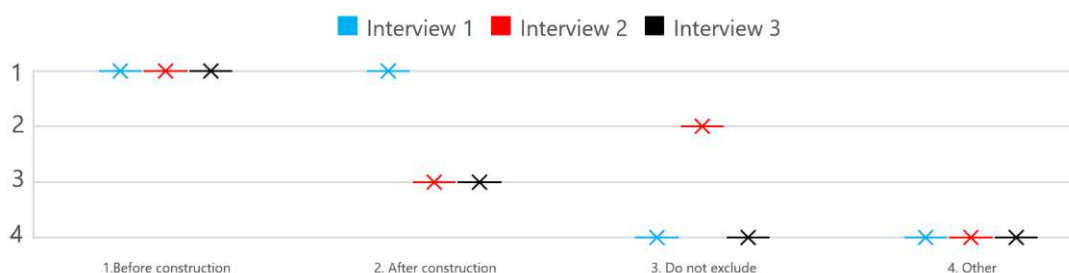


Figure 5.18: Graph for „How useful is it to exclude files?“

- **How should the blocklisted commits be excluded?**

All the interviewees found blocklisting commits helpful and not showing the floors representing them. Two interviewees preferred not to include lines from the blocklisted commits in the ownership calculation, and only one preferred excluding the lines from the building's surface.

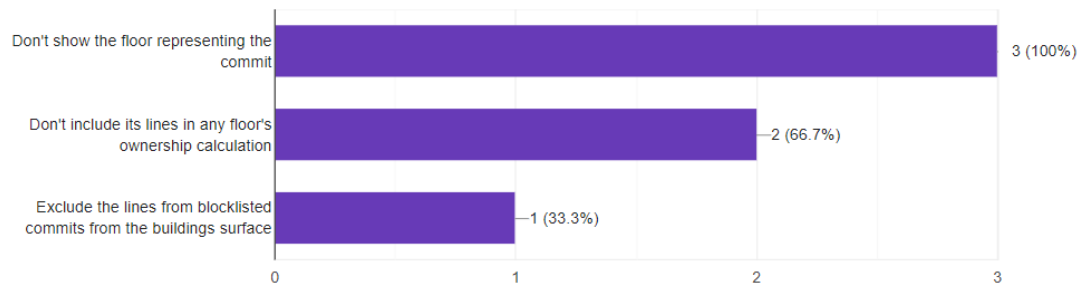


Figure 5.19: Graph for „How should the blocklisted commits be excluded?“

- **Which features are more important?**

Showing ownership and committer was chosen as the most important feature for the visualization. „Exclude commits“ and „Change generality level for all folders“ received the lowest rank.

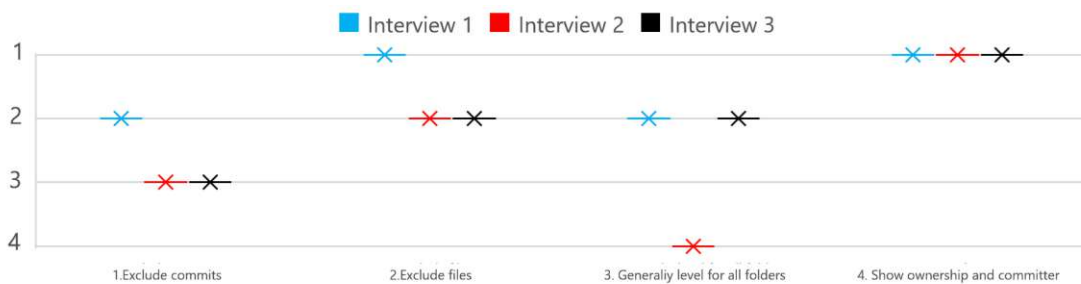


Figure 5.20: Graph for „Which features are more important“

5.3.4 Threats to Validity

One of the main threats to the validity is the number of participants. Three participants are only a few to identify software engineers' general opinions. Secondly, only one of the interviewees was a senior software engineer.

A major threat is also the questionnaire's complexity. Some of the questions were too complex to understand and to decide on a ranking in a short amount of time. For example, scaling techniques need a separate introduction before deciding how and where to use them.

The main tools to help the interviewees were textual description, sketches, and the possibility to rank given options. However, it is hard to foresee the pitfalls of the visualization, for example, how a particular feature may look in small or large software.

Because of the many features the visualization offers, some of them may conflict. For example, merging consecutive commits into one may conflict with the feature of commits'

5. REQUIREMENTS

sliding as separate commits. All this information may be too much for a participant to process in one hour, understand the main concepts, and determine the inconsistencies.

CHAPTER 6

Implementation

A few configuration steps must be followed to show the Slide City visualization. This chapter describes the technical steps and the background of the software to configure and show the visualization. Firstly, it describes the technologies used and the architecture of the software. Then it shows the steps and the involved modules that provide the functionality.

6.1 Architecture

The visualization has a simple architecture (see Figure 6.1). The front end is implemented in Typescript language using the Angular 15 framework. The 3-D visualization uses a library called angular-three¹, which internally uses ThreeJS. ThreeJS uses the low-level graphics APIs of WebGL and provides a higher-level API for JavaScript. Angular-three wraps the ThreeJS functionality by providing prepared components for the ThreeJS concepts, such as Meshes, Geometries, and Materials.

¹<https://www.npmjs.com/package/angular-three> version of 11.07.2024

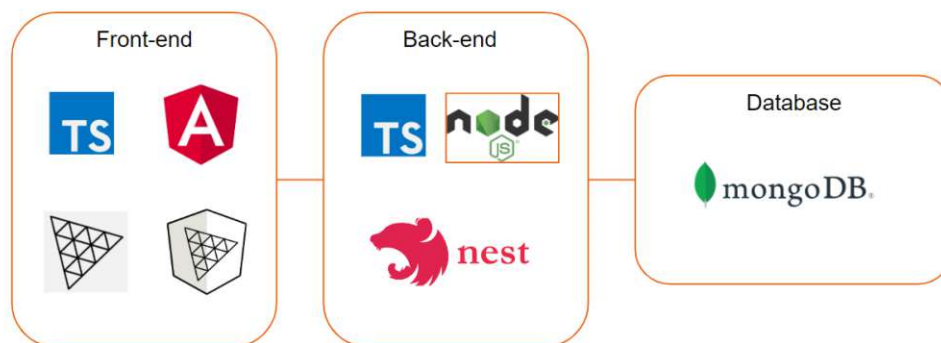


Figure 6.1: Architecture

The back end is also implemented in Typescript using the NestJS language. NestJS is a framework built on top of NodeJS that provides in the back end a similar functionality as Angular in the front end. It gives the possibility to inject services and modularize the application with modules. By using MongooseJS², NestJS makes easier the ORM mapping. The back end uses a No-SQL database MongoDB to store metadata about repositories and the mined data from Git.

6.1.1 Backend Architecture

The backend is split into three NestJS modules: „GitRepoModule,“ „RepoMetaModule,“ and „RepoFilesModule.“

„GitRepoModule“ does not contain any controllers or database repositories. Its goal is to manage the downloaded Git-repositories and provide services to the other modules, such as prepared Git functions for cloning and removing repositories or getting the commits list. The module uses the simple-git³ library to access the repositories.

„RepoMetaModule“ provides API for the repository metadata and the commits' lists. The clients can clone the repositories by sending with a POST-Request the name, URL, branch name, and credentials to access the Git-repositories. These metadata are stored in the database to be accessed later. The „RepoMetaCommitsController“ provides metadata for the commits. The metadata includes the hash, date message, and author information.

„RepoFilesModule“ is the module that does the repository mining and prepares the visualization data. The module uses the provided „GitRepoModule“ functionality and the „simple-git“ library for special Git commands such as diff, log, and blame. Section 6.3 explains in more detail the mining process.

²<https://mongoosejs.com/> version of 11.07.2024

³<https://www.npmjs.com/package/simple-git> version of 11.07.2024

6.1.2 Front-end Architecture

The Front-end contains standalone components, which behave as a single module, services to retrieve data from the backend or do front-end business logic, and stores for data management. The main component groups are „repo-management,“ „medium-configurator,“ „dashboard,“ and „visualization“ set inside the „dashboard“ folder. Each group may have one of the following folders:

- **data-access:** contains the related service to get data from the backend or business logic done on the front-end side. However, they do not cache data in Subjects. The Stores cover data management.
- **feature:** contains components aware of the services. They are usually called „Smart Components“⁴.
- **ui:** contain components that do not know about services and the components' relations. They are usually called „Presentation Components“⁴.

The Front-end uses NgRx⁵ stores, which provide reactive state management for Angular apps inspired by Redux. Each store has its actions, effects, reducers, and selectors. Front-end contains the following stores:

- **repo-meta** contains the metadata of the current repository that is or will be shown.
- **medium-configurator** contains the configuration that the user applies during the „Visualization Configuration.“ Section 6.4 explains the applied configuration in more detail.
- **repo-files** contain the repository data received from the backend, which will be rendered in the visualization.
- **dashboard-details** are updated dynamically when a commit is selected or a commit slides. It stores metadata about the commit, the file, and information about the commit's changes in the file.
- **hoover** stores the last hovered FileCommit. The visualization containers get this information to display the commit hash, the author, and the current owners of the hovered FileCommit.
- **light-configurator** stores the configuration, which the user can change on the top left side of the UI without reloading the visualization. Section 6.5. shows in more detail the data stored in the light-configurator.

⁴<https://blog.angular-university.io/> version of 11.07.2024

⁵<https://ngrx.io/> version of 11.07.2024

6.2 Repository Metadata

The website's home page starts with the list of mined repositories and the possibility of mining a new one. To mine a new open source repository it is required an arbitrary name, the repositories clone HTTP-URL without the „https://“ prefix and the branch name. For private repositories, the user must have a read access token. After adding the required data, the front end sends a post request to the back end to create the metadata for the repository. The back-end clones the repository to return feedback to the user on whether the given repository details are correct. The information that the back-end will mine from the repository is bound to the repository URL and the selected branch. Therefore, those two inputs cannot be updated when reselecting the same repository metadata.

6.3 Mining Configuration

Upon applying the new repository meta-data, the back end confirms whether the repository could be cloned. On success, it sends back metadata for all the repository commits. On the next page, the front end shows the mining configuration options.

Data mining involves the longest process, where the back end iterates all the commits of the selected branch and creates an object from the `RepoFiles` class. Therefore it is done only once when a new repository is applied, and the `repoFiles` object is stored in the Mongo database. The `RepoFiles` schema includes an array of metadata for each commit sorted by date, the list of the blocklisted commits, and the file list. The file list is an array of objects representing the files that will be rendered as cylinders in the front end. Each file contains its list of partial commits, called `FileCommit`, related only to that file. The back end stores in `FileCommits` information such as the added and deleted lines, ownership data, and further metadata from the commit.

FileHistoryBuilderService is a back-end service that has only one public function called **buildFileHistory()**, which receives as arguments:

- *repoId*: *string* - the ID of the repository metadata is also used as the folder name of the cloned project.
- *commitsToBeHandled*: *CommitDto[]* - the list of all the commits sorted by date.
- *repoFiles*: *RepoFiles* - the `repoFiles` object where the service will store its result.
- *configuration*: *MiningConfigurationDto* - the mining configuration received from the front end.

The method processes all the commits from the first to the last from the selected branch, including the unselected commits to find the file movements. For each commit, it runs the „git log,“ and the „git diff“ command (see Section 6.3.1 and 6.3.2). Several Git

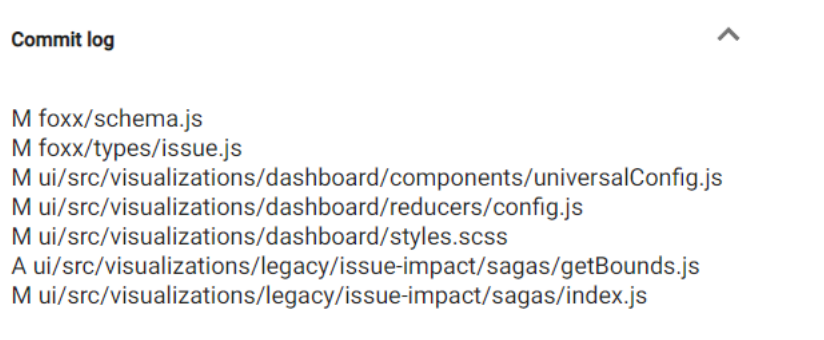
commands are executed during this process to mine the necessary data; however, this section will discuss the most important ones: „git log,“ „git diff,“ and „git blame.“

6.3.1 Git Log

Upon starting with the first commit, it logs all the changed files by calling the following command in the repository path:

```
git log -c -1 --name-status --pretty=format: commit-hash
```

This command lists the file path and names together with their file status⁶. The result is stored in the commits meta-data to be displayed in the UI under commit details (see Figure 6.2).



```
Commit log
M foxx/schema.js
M foxx/types/issue.js
M ui/src/visualizations/dashboard/components/universalConfig.js
M ui/src/visualizations/dashboard/reducers/config.js
M ui/src/visualizations/dashboard/styles.scss
A ui/src/visualizations/legacy/issue-impact/sagas/getBounds.js
M ui/src/visualizations/legacy/issue-impact/sagas/index.js
```

Figure 6.2: Example of Commit log

6.3.2 Git Diff

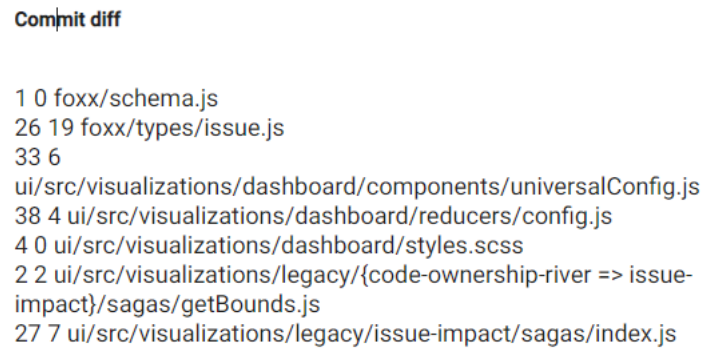
The „git diff“ command lists for each file changed from the commit the number of added and deleted lines. The command also contains custom flags that the user can define during mining configuration, for example, „--ignore-all-space“ and „--find-copies-harder“. The „git diff“ command is shown below, where the „diffConfiguration[]“ array is replaced with the custom flags.

```
git diff --numstat diffConfiguration[] parent-hash commit-hash
```

This command is executed only for the commits which have one parent. When more parents are involved, the git diff command shows, besides the new changes, also the differences between branches, which are irrelevant to the visualization. Showing differences to each parent was explicitly discouraged during the semi-structured interviews in favor of showing only the resolved conflicts for merge commits. The added lines for merge

⁶https://git-scm.com/docs/git-status#_short_format version of 11.07.2024

commits are set in a later step. The result of the „git diff“ command is stored in the commit object so the user can see the result for any commit besides the visualization (see Figure 6.3).



```
Commit diff
1 0 foxx/schema.js
26 19 foxx/types/issue.js
33 6
ui/src/visualizations/dashboard/components/universalConfig.js
38 4 ui/src/visualizations/dashboard/reducers/config.js
4 0 ui/src/visualizations/dashboard/styles.scss
2 2 ui/src/visualizations/legacy/{code-ownership-river => issue-
impact}/sagas/getBounds.js
27 7 ui/src/visualizations/legacy/issue-impact/sagas/index.js
```

Figure 6.3: Example of Commit Diff

6.3.3 Git Blame

After the log and diff command, a sub-iteration starts for each file from the „git log“ result. During this sub-iteration, the process creates the FileCommit object, which has the fundamental data on rendering visualization floors and buildings. The FileCommit contains the information that a specific commit has affected on one specific file. The FileCommit receives the relevant result from the „git log“ and „git diff,“ such as the status, added and deleted lines that the commit changed in the file, path changes, holds a flag whether the commit was blocklisted during configuration, and the „owners“ object.

The „Owners“ object is a map object where the key is the author, and the value is the number of lines the author has in the file after the commit, i.e., it contains the ownership of each author that has changed the file. The following Git command is executed to calculate the ownership for a FileCommit:

```
git blame -l -e blameConfiguration[] commitHash filePath
```

The „git blame“ command is executed for each commit and file. The „blameConfiguration[]“ array contains the custom flags the user can add during the mining configuration. The result contains a list of all the file’s lines combined with the line owner.

The FileCommits are added into separate lists grouped by files. This list also contains flags to check at which FileCommit the name or the path of the file has changed.

6.4 Visualization Configuration

After mining the repository data or selecting an already mined repository, the front end gets the RepoFiles from the backend and is routed to the visualization configuration. This configuration is not stored in the backend and is applied to RepoFiles as soon as the frontend receives them. The main advantage of splitting the mining and visualization configuration is that the second one does not need to send further requests to the backend. The visualization configuration can be redone several times; however, it will rerender the visualization from the start, which may take time and a lot of processing power on the client. This section lists the options that the user can choose.

6.4.1 Visual Material

The user can select visualization aspects such as the material of the rendered objects. The „Phong“ material offers a better 3D user experience, however, with more performance costs. The „Basic“ material does not render shadows, which sometimes makes it difficult to see object edges, but it offers a more performant and convenient environment for large repositories.

6.4.2 Deleted Lines

The user can select whether the floor height shall contain only the added lines or additionally the deleted lines.

6.4.3 Exclude & Include Files

The user can exclude files or folders or include only some of them. The algorithm checks for each file or folder if it contains at least one exclude line in its path. Then if an input is added to the include section, the algorithm checks if the file or folder contains at least one of the include lines in its path. For a file to be rendered, it should not match any of the excluded lines; if there are any included lines, it must match one.

6.4.4 Commit Mapping

The top surface of a building is defined to show the latest size of the file in LoC. The height of the floors represents the size of the commit in LoC. The user can select if the commit sizes should be mapped into the volume or the surrounding floor area. The floor height is calculated accordingly.

6.4.5 Commit Height Scaling:

The interviewees in Section 5.3. preferred linear, histogram, and logarithmic mapping. The Box-Whisker plot mapping can also be achieved with the histogram, while threshold mapping was not preferred. The visualization allows the user to choose between the first three mappings (i.e., linear, histogram, and logarithmic mapping), and it allows

the user to set a coefficient that further decreases (if the coefficient is smaller than one) or increases (if larger than one) the height. Before mapping the height, the front end creates objects for each possible floor that may appear, for example, a folder or a file floor. Below is described how the calculation of each mapping is done.

- **Linear mapping:** The linear is the simplest one. After all floors (for files and folders) have been defined, each of their heights is multiplied by the selected coefficient. If the coefficient is one, then all buildings will represent one-to-one the sizes of files and folders (see Figure 6.4).

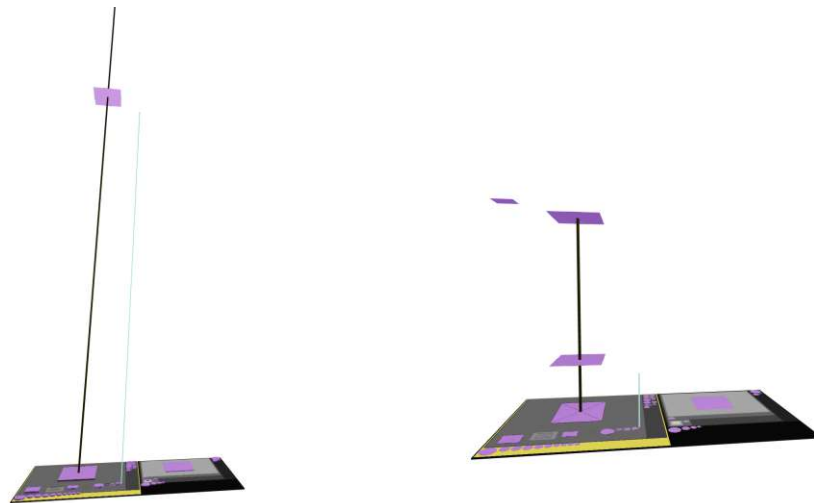


Figure 6.4: Project rendered with linear mapping. On the left, the coefficient is 1, while on the right, the coefficient is 0.3. The thin blue line is the tallest floor in the visualization. On the first commit, it contained more than 400 lines. However, on a second commit, it was reduced to 1 line. The second commit shapes the building with a top surface of 1 and a first floor of size 400.

- **Histogram mapping:** The floor heights are sorted by size and split into buckets. The user can define the bucket amount during the mapping selection. Eventually, the algorithm maps the same height to each element in the bucket. The heights range from the lowest to the tallest bucket. However, the differences between each bucket and its successor are the same. Each floor height is multiplied by the coefficient, similar to linear mapping (see Figure 6.5).

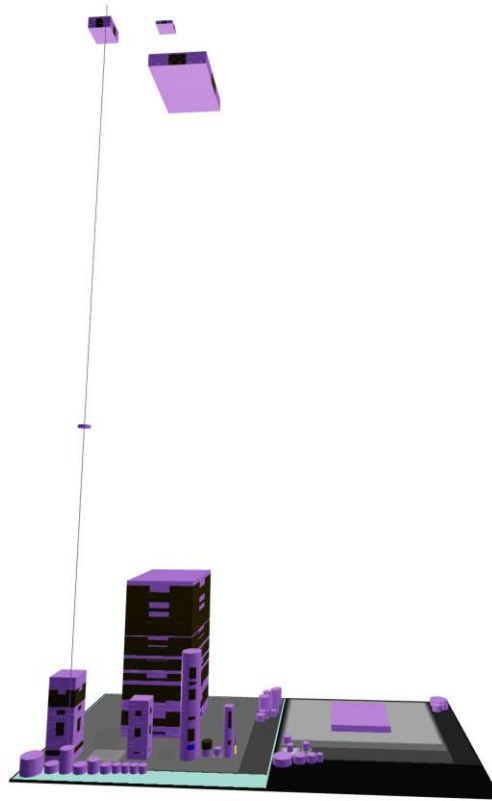


Figure 6.5: Histogram mapping with coefficient 0.1 in 20 buckets.

- Logarithmic mapping:** The visualization calculates the logarithm with base two for all the heights to reduce the extreme height differences between different floors. However, usually, the commit size of a file is smaller than the size of the file. Thus, regardless of whether the commit size is mapped to the volume or the area of the floor, the height will be smaller than one. Logarithms calculated for numbers from 0 to 1 result in larger numbers that may increase the height differences. Therefore before calculating the logarithm, each height is first multiplied with a small coefficient to make each height larger than one. Then after the logarithm is calculated, the floor height is multiplied by the coefficient, similar to linear mapping (see Figure 6.6).

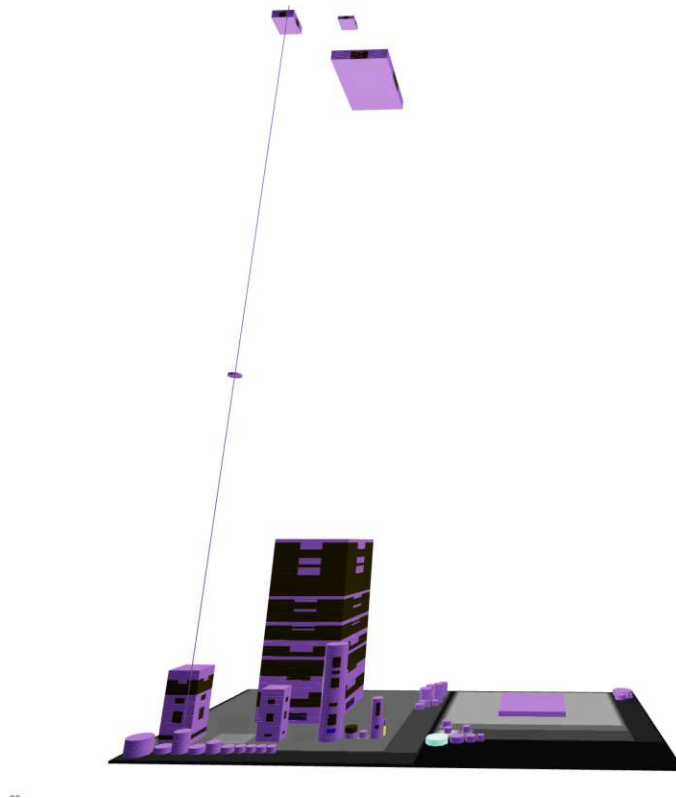


Figure 6.6: Logarithmic mapping with coefficient 0.2.

6.5 Live Configuration

The section on the visualization's top left corner provides features to update the visualization on the runtime (see Figure 6.7). The configuration is grouped and stored in the „light-configuration“ store in the front end. In contrast to the visualization configuration, the live configuration does not trigger the rendering of the floors and allows the user to change the visualization dynamically. Similar to the architecture of the whole front-end application, the responsible components listen to the events to execute the expected changes upon configuration change.

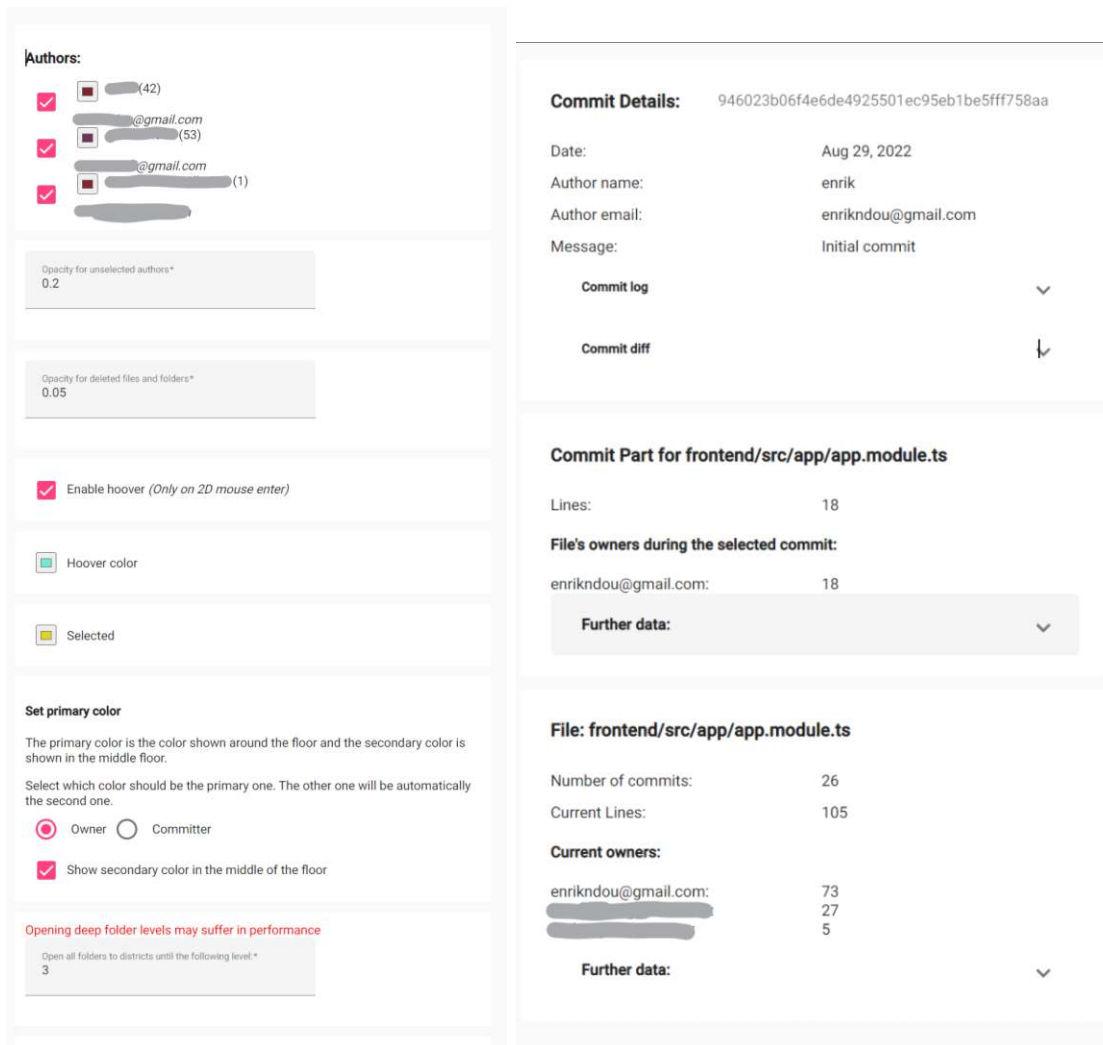


Figure 6.7: Live configuration section. Figure 6.8: Commits and files details section.

6.5.1 Colors & Filtering

The main feature is related to the authors' colors. The functionality is based on interview results. The colors are distributed randomly. Each author is assigned a random hexa code where each of the six digits is less than „A.“ Although the user can set any color for any author, the light colors are free to select. Additionally, the checkbox allows the user to deselect the author. Deselecting will not automatically hide everything related to the author but will turn the buildings into wireframes. Two other inputs below can dynamically change the visibility of deleted files and folder and wireframed structures.

6.5.2 Hoover & Selection

The next feature is the Hoover and selection. Hoover makes it possible to see a few pieces of information inside the visualization frames. For example, upon hovering a floor, on the top left corner is shown the path with the name of the file or folder; on the top right corner is shown the commit hash; on the bottom left corner is shown all the author with the ownership that they had then. These three make identifying and getting brief information about the floors easier. However, the Hoover feature slows down the performance in large software, for example, with more than 700 commits. Therefore, it is possible to disable it. The user can change the hovered floor's color.

Likewise, the user can change the color of the selected file or folder. Upon selecting one floor, three sections are shown on the right side (see Figure 6.8).

- The first gives general information about the related commit, such as the author's name, date, and message. Additionally, it contains also the commit's log and diff information which were stored during the mining.
- The second shows information about the commit part related only to the selected file or folder, such as how many lines were added and the ownership during that period. This ownership is the same that is shown on Hoover at the bottom left of the visualization.
- The third section shows the current state of the file in the folder, including how many commits are involved, how many lines are currently, and who is the present owner.

All three sections' data are stored in „dashboard-details“ and used by the component with the same name-prefix.

6.5.3 Primary & Secondary

By default, the primary color (i.e., the floor surrounding color) represents the ownership, and the secondary color (i.e., the color in the floor middle) shows the committer. This feature allows swapping the colors depending on which is more important for the use case. Additionally, it makes also possible to hide the second color so the user can see either the ownership or the committers only.

6.5.4 Generality Level

The interviewees of the semi-structured interview showed a preference for changing the generality level (i.e., at which depth should the folders open automatically). However, every interviewee expressed a different opinion on what they expected from it. The first interviewee expected that the generality level would change the generality level only to the folders that the user has opened and closed. The most common case is when the

user changes a few folder-building into districts. During the implementation, another feature already nearly covered this aspect. Upon double-clicking on a district, it would unite all its sub-buildings and become a single building, regardless of how many levels the sub-district was open. During the implementation, another challenge was observed. Changing the generality level requires all the folder floors shown on the visualization to be deconstructed and create new floors for all their sub-buildings. Therefore, because of the prolonged time for the process, the initial idea to set the generality level in a slider is not user-friendly. Secondly, the implementation showed that it is not easy to differentiate between the folders that the user has or has not touched. A simple example is when the user opens the root folder-building. In this case, if the algorithm marks the district and the shown folders as touched by the user, then the generality level would no longer affect any building. If the algorithm would mark only the root folder as touched, it is still to be determined when these buildings should become unmarked.

Because of the above mentioned issues, the generality level affects all files and folders and is presented by input to enter the generality level (i.e., the depth level where the folders should be opened). Instead of sliding on several time-consuming generality levels, the user has to write one depth level. Additionally, it includes a warning above it that the process may take a prolonged time.

6.6 Requirements Deviation

The implementation followed the described requirements and the semi-structured interview results. However, it also showed that some requirements could be inconsistent. For example, the interviewees preferred to keep consecutive commits as separate floors. However, combining them into one floor would make it incompatible with the slide feature because the commits move one by one on the slide; if some partial commits are „squashed“ and slid together, then the user can not determine the commit size. Moreover, it would also force floors in other buildings to be combined, which would not always be meaningful. Another inconsistency is the slide feature and the commit sort. If the slide should be based on time, then it would not fit with the commit sort by considering the branches.

The second challenge is the visualization performance, which relies heavily on the ThreeJS library and its wrapper Angular Three. The circle geometry is constructed from several triangles⁷. Although a perfect circle is impossible, the more triangles are added to the geometry, the more it looks like a perfect circle. However, this comes with a large performance cost. In a large software visualization, there are many files, and each is represented with cylinders. Each file in itself has cylinder floors for each FileCommit. For performance reasons, the circle contains a small number of triangles where it is possible to distinguish it from the rectangles, and it is possible to differentiate in its side the primary from the secondary color. Figure 6.9 shows files as cylindrical buildings. They have nine segments (i.e., nine triangles), which make them distinguishable from

⁷<https://threejs.org/docs/#api/en/geometries/CircleGeometry> version of 11.07.2024

the cuboid buildings, and the primary color is set thrice into two consecutive segments in contrast to the secondary color set thrice into one segment. Conversely, cuboids are more performant because a square can be built with only four triangles, unlike a circle with nine triangles. Therefore, where possible, squares are prioritized against circles; for example, the secondary color is shown as a rectangle instead of a circle, regardless if it is a folder or a file.

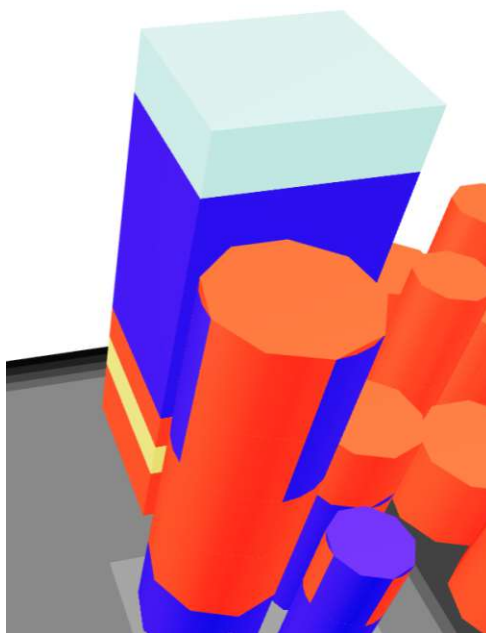


Figure 6.9: Cylindrical building.

The semi-structured interview showed different preferences in many features; therefore, many features have more than one option and are configurable. The user can configure the options before mining the data, before rendering the visualization, or after the visualization has been generated. For instance, the author's colors are distributed randomly to the authors. However, in the top left corner, the user can change the colors automatically without restarting the visualization. During the visualization configuration, the user can select one of the three most preferred options to map the data (i.e., logarithmic, histogram, and linear) or if the deleted lines should be included. During the mining configuration, the user can select which commits to be shown by deselecting one by one, using the time range, or generally deselecting merge commits.

The testing phase showed that almost every project had developers using multiple accounts. This made it more difficult to show correct ownership because the visualization would favor the developers with one account. For example, a floor would show the color

of a developer with 50 lines instead of another developer, which has 60 lines split into two accounts. Therefore a new feature was added to merge the accounts. As soon as the visualization has started, below the live configuration is the Section „Author mapping“. The user can select another target account for each account to which the ownership will be merged. For example, Figure 6.10 shows that the account „1226762@student.tuwien.ac.at“ is going to be merged to „e1226762@student.tuwien.ac.at“. This will result in combined ownership in the visualization, and the committer color of the mapped account will be replaced with the target account color.

Author mapping

Map the following account to another email. The emails used as target cannot be mapped to another target.

Roman Decker - roman.decker@gmail.com	▼ null
Johann Grabner - johann.grabner@inso.tuwien.ac.at	▼ null
Julian Kotrba - me@juliankotrba.xyz	▼ null
Alexander Nemetz-Fiedler - alexander.nemetz-fiedler@outlook.com	▼ null
THMv1TU - 1226762@student.tuwien.ac.at	▼ e1226762@student.tuwien.ac.at

Figure 6.10: Cylindrical building.

CHAPTER 7

Evaluation & Results

This chapter describes the scenario-based expert evaluation process to validate the prototype. This process aims to answer the third research question „How purposeful is the visualization for answering historical and ownership questions?“. The scenarios were selected based on real-world use cases where the visualization would answer ownership questions. After the pilot phase feedback, the process including the questions was reformed. This chapter will first describe the pilot phase, the scenarios, and the results.

7.1 Test Plan

The goal of the scenario-based expert evaluation is to test the implemented prototype with real-world problems. The approach is based on technical action research (TAR), an artifact-driven single case study that helps to evaluate laboratory experiments. This phase gathers and analyzes the participant’s feedback to find a conclusion about the RQ3. The test includes eight scenarios where for each scenario the participant will answer two questions about the visualization’s purposefulness for the scenario and how clearly it shows the results.

The scenarios were planned to be conducted online over Zoom with a timeslot of around one hour per participant. During the pilot phase, we decided, that the interviewer start with an introduction of the visualization and its most important features, which came across as more appropriate than prepared recordings of the introduction. For the introduction, a PowerPoint presentation with five slides was planned to be held in under 15 minutes.

Next, the scenarios were conducted. The interviewer read the problem that should be solved and prepared the prerequisites for the scenario. Initially, it was planned that the participant find the solution, however, during the pilot phase, this seemed infeasible to do. The first reason is that some scenarios required a large amount of data and conducting the

process online over Zoom made this more unpleasant. Another aspect that contributed to the complexity was the difficulty of combining all the secondary features, such as mining the correct commits, filtering, and the time slide. The available time was not enough for the participants to familiarise themselves with all the features and learn to combine them to find a solution. Therefore, we decided in the pilot phase that the interviewer would show how the problem of the scenario could be solved and, then the participant was asked the two following questions:

- **How purposeful is the visualization idea for this scenario?**

The question aims to gather feedback on whether the visualization idea fits to give answers for similar scenarios. The answer can be given in 5-point scale from 1 (not purposeful) to 5 (purposeful).

- **How clearly does the visualization show the expected result?**

This question seeks to assess if the prototype clearly presents the expected outcomes. The answer can be given on a 5-point scale from 1 (not clear) to 5 (clear).

The participants discussed the questions about the solutions for each scenario while the interviewer took down notes. Additionally, the meetings were also recorded for later processing.

7.2 Demographics

Similar to the semi-structured interviews, the first section's questions aim to gather demographic data for the participants. The questions focused on their age, gender, experience with Git, general experience with software engineering, and their roles in projects, such as whether they held leadership positions. Due to the limited number of participants, not all possible demographic variations were covered. However, the collected data might help explain differences in test results across similar cases.

It was possible to re-interview only one of the semi-structured expert interview participants. All participants were male. One of them had less than five years of experience with Git, two of them had more than five years and one had more than ten years. Two participants had leading roles in software engineering projects; one in teaching and another one worked once as a tech lead. While the other two were more familiar with software visualizations.

7.3 Scenarios

The listing below contains the scenarios and a short explanation of the benefits of using such scenarios.

- **Which developers have currently more ownership in frontend source code (i.e. in frontend/src)?**

The first question is also the easiest to familiarize the user with the prototype. On the other side, it is an example of a scenario where a manager or a developer wants to know who is the user that owns most of the lines in a particular file or folder.

- **Which developers had historically more ownership in frontend/src?**

This is similar to the first scenario, however, this relates to the use cases when someone wants to know how a feature has evolved until now. In these cases, a developer who has the highest ownership for a longer time may have more information about the feature that has evolved.

- **Who are the main developers that added the most changes and the developers that have the highest ownership on files related to „user“ (frontend & backend)?**

This question is related to use cases where the user needs to know ownership and code contribution information only about a particular feature. The prototype supports, however, only search and filtering by file and folder name. Usually, files containing the core features include that part in the name.

- **When were the files related to „user“ implemented (backend & frontend)?**

The use cases when this feature is necessary is if the user is interested to know in which periods a particular feature was implemented.

- **Which developer currently owns more services (i.e. it has the highest ownership on those services) and which developer is currently the highest owner of all services combined?**

The visualization provides information in different levels such as project, packages, or file levels as long as they are in one repository. This scenario shows the difference when observing a particular folder at different levels.

- **Which developers have added consistently unit tests for the services and which do not?**

This question is relevant especially in the academic context when the tutors need to check the consistency that the students follow when implementing a project. This particular question is about the unit tests.

- **Which were the developers that changed more code in the service folder in the last two weeks? Who owns most of the code added in the last two weeks?**

The metrics comparison between developers that join the project in different periods is more difficult than the usual academic projects where students start and end the project at the same time. This scenario, although using an academic project, shows the comparison of developers' contributions only for a particular period. For example, if a developer has joined the last two weeks, only the developers' contribution of the last two weeks is compared.

- **Which angular components (typescript files) have the highest owners switched the most?**

Bird et al showed in their research [27], that packages that have a lot of major owners (i.e. developers that own more than 5% of the code) may have a higher risk of bugs. This scenario goes a bit further to find files, which have switched the highest owner more than once.

7.4 Results

The participants expressed different opinions for different scenarios. Figure 7.1 and Figure 7.2 show the results for each scenario and each interview. Figure 7.1 shows the result of the first question „How purposeful is the visualization idea for this scenario?“ and Figure 7.2 shows the result of the second question „How clearly does the visualization show the expect result?“

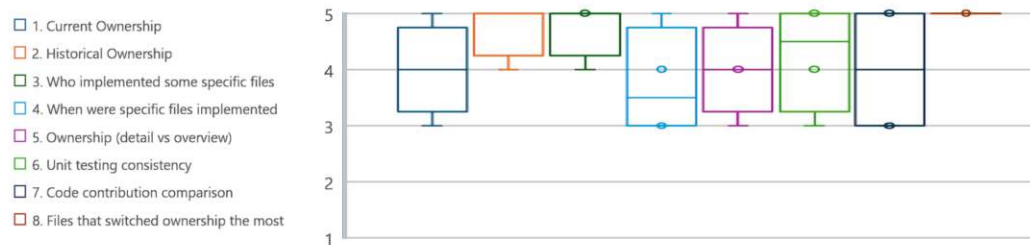


Figure 7.1: Results for each scenario of the question: „How purposeful is the visualization idea for this scenario?“ (1 not purposeful - 5 purposeful)

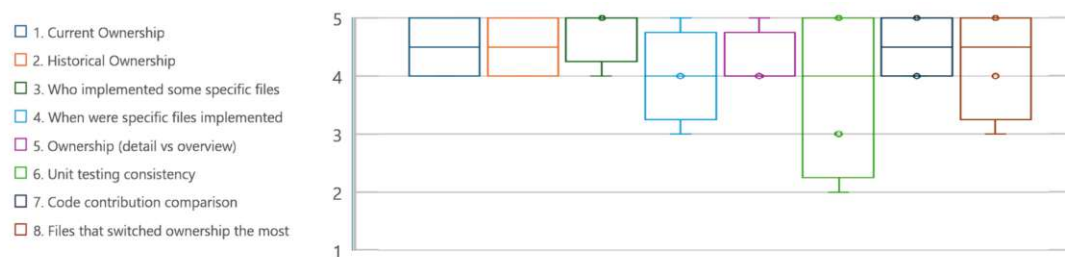


Figure 7.2: Results for each scenario of the question: „How clearly does the visualization show the expect result?“ (1 not clear - 5 clear)

Below is described the solution and the results for each scenario.

1. **Which developers have currently more ownership in frontend source code (i.e. in frontend/src)?**

The solution is to find and click the frontend/src folder that is by default opened as a district and then see on the left side the calculated ownership (see Figure 7.3).

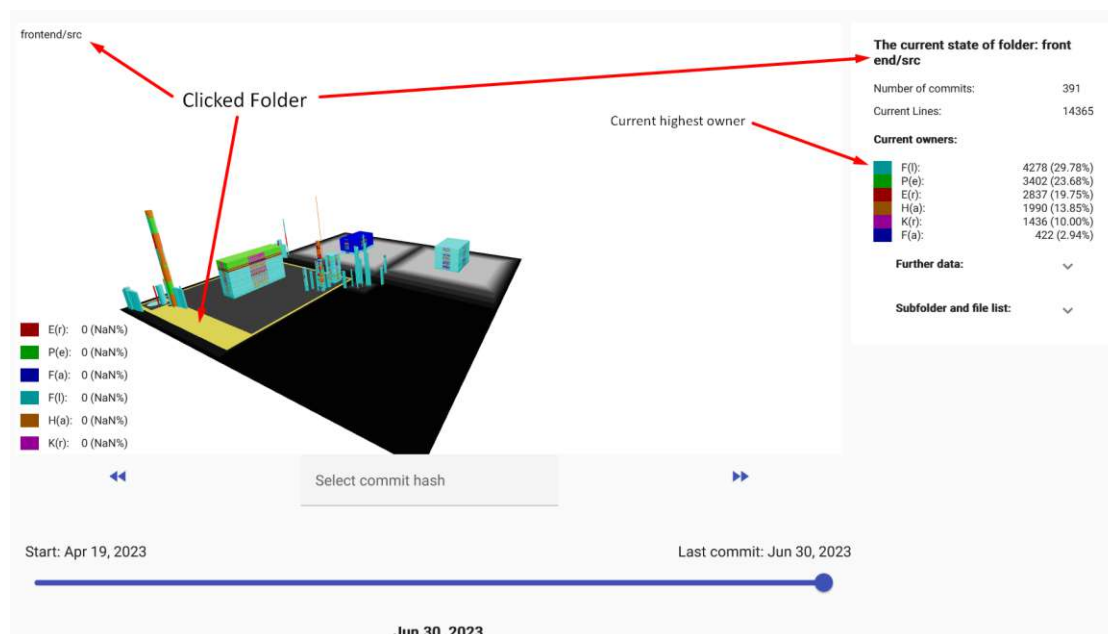


Figure 7.3: The simplest way to see the current ownership of a folder.

The participants rated the idea and the clarity between four and five. The main observed issue is that the user would need to hover in different folders to find the correct one. This could be improved by adding labels to the districts.

The fourth participant valued the purpose of the idea with three. The argument was that the buildings shown were not representing the result. This could have also been achieved if the district had collapsed into a building because the question was generally for the src/ instead of its inner details.

2. Which developers had historically more ownership in frontend/src?

The solution is to show the frontend/src building, which by default needed to be collapsed from a district. The highest owner is directly visible on every commit while the second and the rest of the owners can be seen on the bottom-left while hovering on the floors (see Figure 7.4).

7. EVALUATION & RESULTS

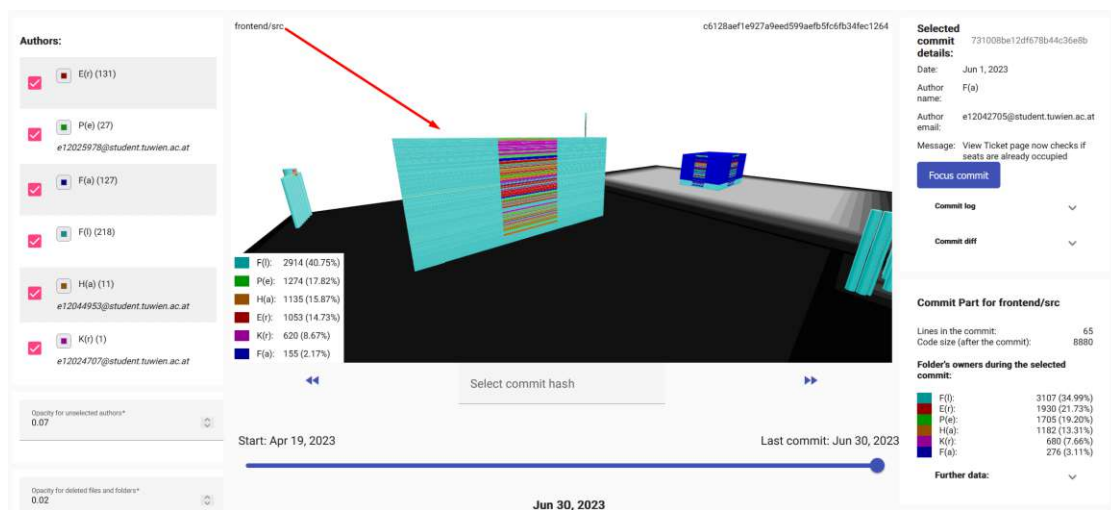


Figure 7.4: The hovered building in the middle, shows from bottom to the top the turquoise color indicating that F(1) has been the highest owner from beginning up to now.

The participants rated this visualization also between four and five.

- Who are the main developers that added the most changes and the developers that have the highest ownership on files related to „user“ (frontend & backend)?

The solution is to filter only the files and folders that have the keyword user in the file path and then see by color who are the committers and largest owners (see Figure 7.5).

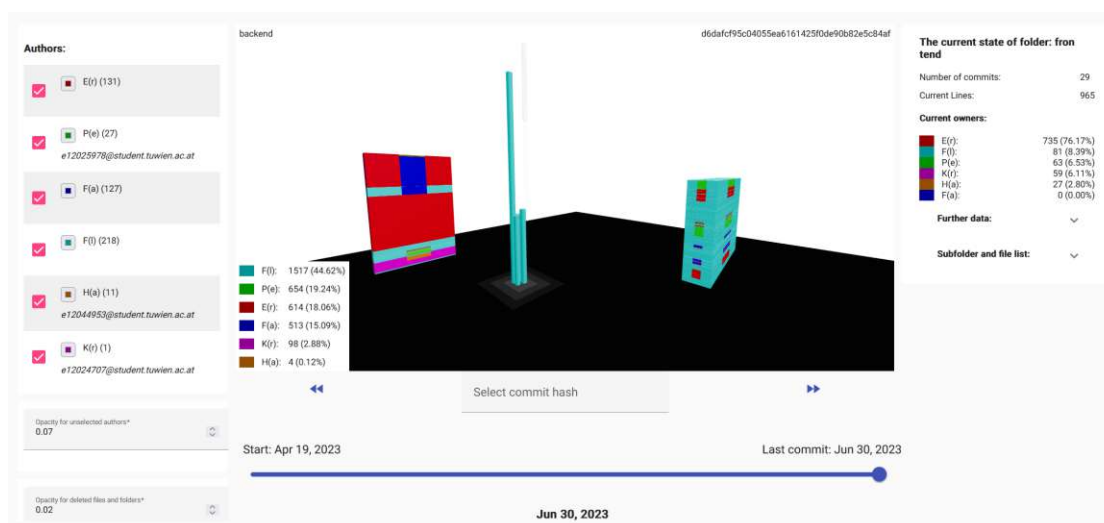


Figure 7.5: The building on the left represents the frontend. The stripes in the middle show the commits which indicate that the red and the blue author have added the most changes. However, when looking the current ownership on the details on the right, we see that the blue color comes last. This happens because the code added by the blue author was removed again. The building on the right is backend which is predomenly tourquoise followed next by red and green commits.

The participants rated both the idea and the clarity with five except the first participant rated the idea with four and the third participant rated the clarity with four.

4. When were the files related to „user“ implemented (backend & frontend)?

The solution is to use the slide feature on the user-filtered folders and see the periods when the frontend and backend are implemented (see example in Figure 7.6).

7. EVALUATION & RESULTS

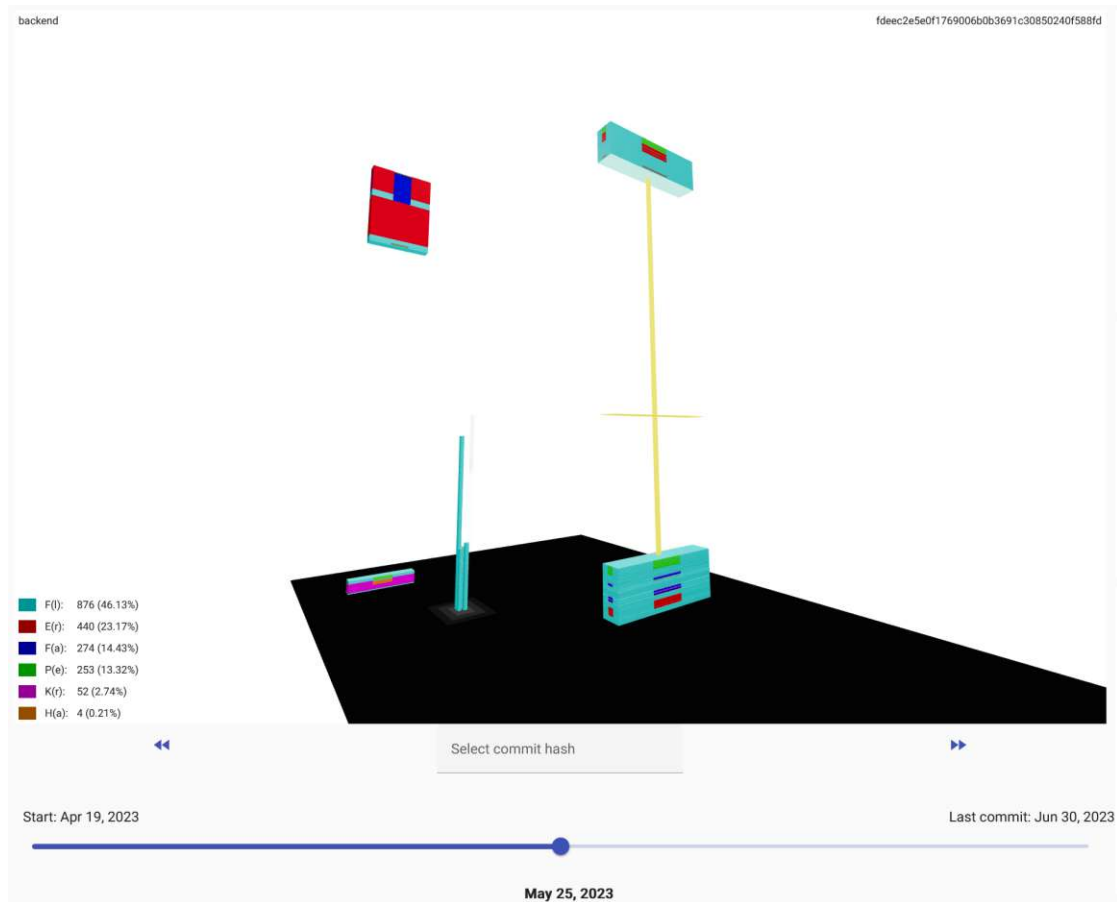


Figure 7.6: The time slider is set to 25th of May which is almost the middle of the project-timeline. We can see on the top the commits that come after this date and at the bottom the commits that come before. It can be observed, that most of the user related files in backend were implemented before and in frontend after the 25th of May.

This scenario got different ratings from the participants. The first participant rated with three both the idea and the result clarity; the second rated with three and four; the third both five and the fourth one both four. A suggestion to improve the visualization for showing the implementation period is that the visualization would be capable of showing all the floors distributed by time at once, where floors of different buildings can be vertically compared to each other.

5. **Which developer currently owns more services (i.e. it has the highest ownership on those services) and which developer is currently the highest owner of all services combined?**

The solution is to show only the service implementation folder as a building and see who is the largest owner from the ownership color. Then, upon double-clicking the building, the user can count for each user how many buildings (i.e., services)

they own. Figure ?? shows the difference between the folders largest owner and the author who owns more services.



Figure 7.7: On the left, we can see that the blue and as well the violet author own 3 services. However, on the right, we see all the services combined and it appears that overall the turquoise author has the highest ownership.

The participants gave a different rating for the idea and the result clarity. The first participant found it as a good use case for visualization and rated both with five. The second participant did not like the fact that the size of the building would influence the service counting, for example, when two users own three services then the one who owns larger services may be selected first. Therefore he rated the idea with three and the clarity with two. Contrary, the fourth participant would rate both five for counting which developer owns more services but with four and three for the combined ownership. The third participant rated both with four because of the minor difficulty in knowing what you should do to find which developers own more services. Overall, the idea was rated with 5, 3, 4, and 4.5; the result clarity was rated with 5, 4, 4, and 4.

6. Which developers have added consistently unit tests for the services and which do not?

The solution is to filter and show only the two relevant folders. The user can filter a few developers to see if their implementation commits are followed by testing commits (see Figure 7.8).

7. EVALUATION & RESULTS

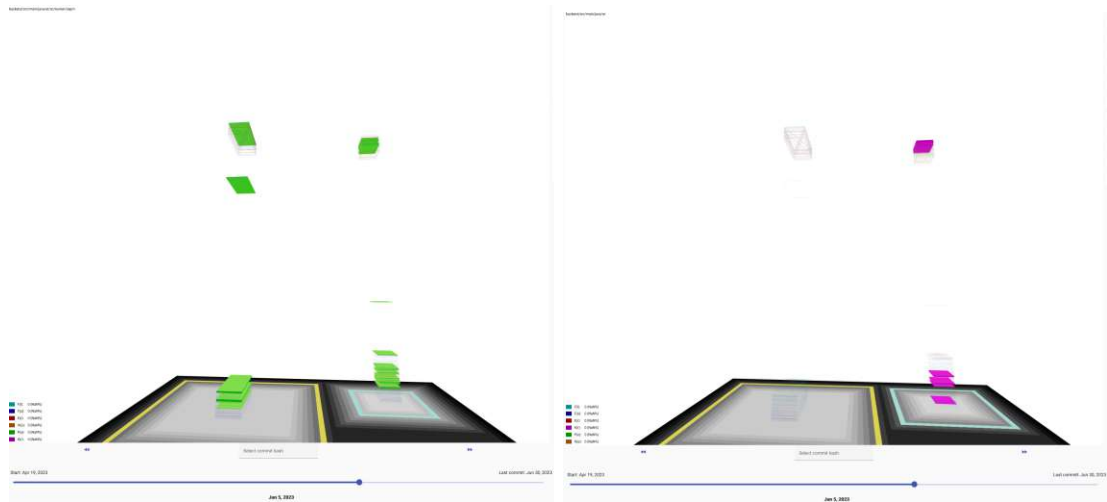


Figure 7.8: Both images have on the right the service implementation and on the left the service unit tests. We can see in this comparison that there is a consistency of unit test commits in regard to the implementation changes from the green author but not from the violet author which is shown in the right image.

The first participant rated with four and three, with the argument that there could be a better comparison mode between the two folders. Perhaps an example could be the similar option as for the fourth question, where more than one commit is distributed in a time axe (preferably all of them for this use case). The second participant argued that the visualization helps to see if the developer wrote tests and how much however is hard to see if they wrote tests for their features. He rated the idea with three and the clarity of the result for this scenario with two as this visualization would not fit these use cases. The third and the fourth participants found the differences between users clear and therefore rated both with five. The differences in rating also depend on how much and what kind of information the user expects from the visualization.

7. Which were the developers that changed more code in the service folder in the last two weeks? Who owns most of the code added in the last two weeks?

The solution: This scenario needs to reconfigure the mining process, where only the commits of the last two weeks will be included. Anything implemented before this date will be ignored not only as commit changes but also when calculating the ownership after each commit. Then, the user has to filter only the service/impl folder or simply find it in the visualization and show it as a building to see how the ownership color has changed. Figure 7.9 shows that the blue author had been working at the beginning of the last two weeks, but then was continued by the green and the violet author. On the right side in the selected commit details, we can see that in the end 64% of the last two weeks' code belong to the violet author.

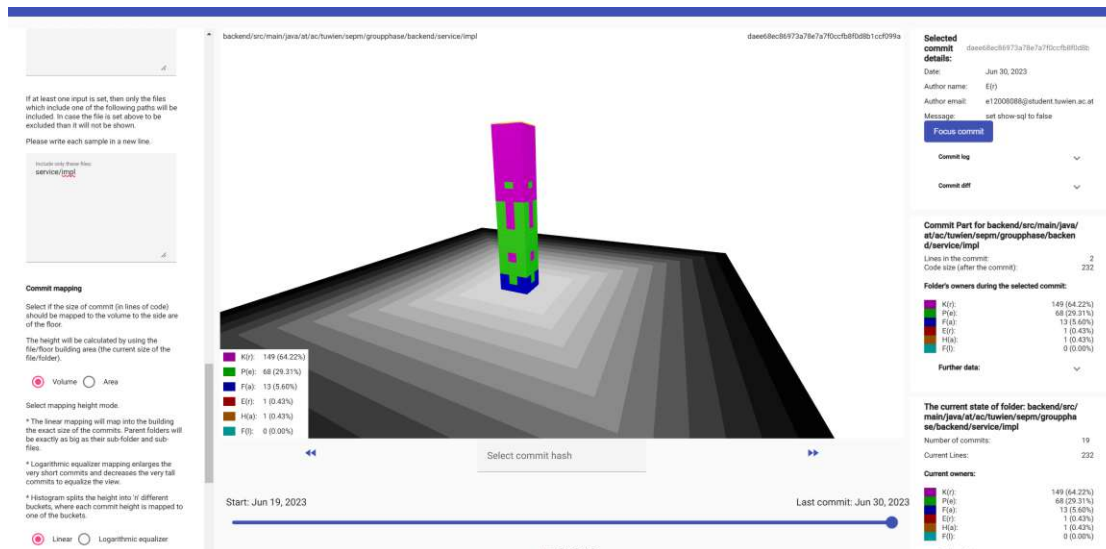


Figure 7.9: Service implementations folder of the last two weeks.

The first and the second participants voted for the clarity of the result with two but three for the idea. The first participant found that the purpose about the above scenarios was less relevant while the second participant would rather rely on numbers and also seeing the code rather than relying on visualizations. Overall the clarity of the result was rated with 4, 4, 5, and 5, while the purpose of the idea was rated with 3, 3, 5, and 5.

8. Which angular components (typescript files) have the highest owners switched the most?

The solution is to filter only the „component.ts“ files and show them at the file level, by using the „Open folder level“ feature. Hiding the stripes and showing only the ownership simplifies distinguishing the ownership changes. Figure 7.10 shows each component as a separate building. With a few rotations of the view, the user can find the two buildings pointed by the arrows that the ownership switched the most.

7. EVALUATION & RESULTS

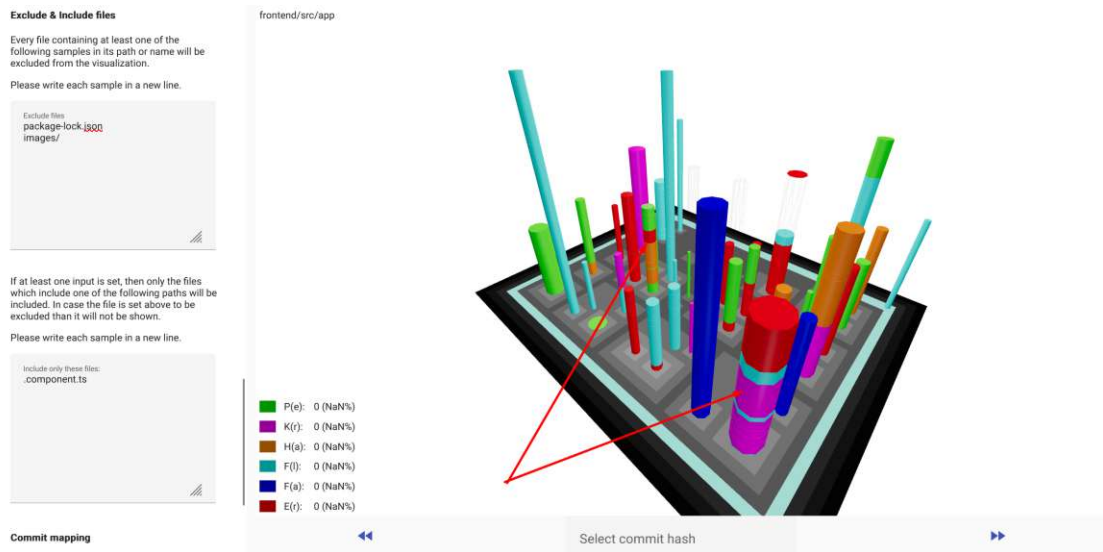


Figure 7.10: Each component is shown as a separate building. The marked files have the ownership changed the most.

All participants rated the idea with five. The second and the third participants rated also the result clarity with five. The first participant rated the result clarity though with three. He argued that the example is clear but if there would have been more files then the answer would have been more difficult to find with the visualization. He proposed that a different coloring scheme would have helped in this case, for example, a coloring scheme that shows the hotspots. The fourth participant was between four and five but decided on four as there was a difficult factor in finding the files.

7.5 Threats to Validity

The environment where the scenario evaluation interviews were done is very similar to the semi-structured interviews. Therefore most of the risks are similar. The interviews were done over Zoom. During the pilot phase, it was observed that it was too difficult to let the user take control and use the prototype.

The sample of the interviewees was small. Only one of the semi-structured interviews was available for the scenario evaluation. Overall, four interviewees with Git experience from less than five years up to more than 10 years participated in the interviews. Some of the questions got very opposite opinions on some feature usefulness in the same scenarios. This may come due to the demographics, of the participants' previous experience with the software visualizations. However, a larger sample would give better results on the scenario's purpose- and usefulness.

Some of the implemented features were removed from the questionnaire because of their complexity. For example, the mapping of the entities into the city mapping was available as linear, logarithmic, and histogram. The mapping was preselected as linear with a factor scaling of 8 on the commit height. The semi-structured interviews showed that explaining the differences between mapping methods would take too much time.

Findings

Besides the threats to validity, the scenario-based expert evaluation gave enough information to answer the third question. This chapter discusses the research question results starting with the first one:

- **RQ1: How has the city metaphor been used in the software visualization research?**

Chapter 4 describes extensively in a study mapping on the question of how the code city metaphor has been used and improved for various research goals. The chapter lists thirty-six visualizations that relate to the concept of a building or a 3D building-like structure to visualize and show information about a software project. The listed visualizations are introduced in research papers. They may have been improved by further research to add features or to use the same visualization to show different aspects of the visualization. For example, CodeCity [10, 61, 72, 63, 102] and ExplorViz [82, 84, 86, 83, 85] have been researched and extended in five research papers. The study mapping showed how and what aspects of the software have been visualized with the code city metaphor. The tables at the end of the chapter help to differentiate and categorize the visualizations easily. The first columns show meta information such as the name and the date when the visualization was first introduced. The next columns list how the city concepts have been used. Most of the research papers introduced visualization where the software was mapped into only one city metaphor, similar to Slide City. The next columns show the data that are mapped to the code city elements, for example, whether the height of a code city building represents the size of a file in lines of code, the number of functions or methods in a file, or runtime data such as the number of instances of a particular class. The last table shows the attributes and features that make the visualization different from the rest and shows the main

static and dynamic data represented in the visualization.

- **RQ2: How to visualize the historical code ownership with a city metaphor?**

The survey on information needs (discussed in Section 3.1) showed that software project stakeholders had different interests in the source code including questions about who has been working on some files or folders. Section 5.1 describes the vision of the proposed artifact, which is based on the code city metaphor using ownership data. The hypothesis over the proposed features was validated with semi-structured interviews. A sample from a different pool of software engineering roles were interviewed to give their opinion and arguments on how the artifact could help them solve their needs. The results were gathered and set as requirements for the Slide City prototype, which was then later implemented on an agile approach (see Chapter 6).

- **RQ3: How purposeful is the visualization for answering historical and ownership questions?**

During the implementation were found and resolved some inconsistencies in the requirements (see Section 6.6) and some features were left out due to their complexity for this thesis. Nevertheless, a new set of hypotheses and questions were prepared to validate whether the prototype was helpful for the users. The semi-structured expert interviews included eight scenarios where the participants rated the idea of the prototype solving the scenario and they rated how clear the visualization showed the expected result. During the pilot phase, it was observed that the implementation had usability issues due to the high workload to finish all the details and because the interviews were conducted online. Therefore, the interviewer executed the solutions with the visualization, and the participants rated and argued their opinions. All the scenarios received different positive ratings from three to five (see Section 7.4), except the scenario where unit testing consistency was evaluated. This clarity of the expected result clarity for this scenario was evaluated twice with five, once with three, and once with two. The participants suggested improvements that may help further the user to see and compare in the Slide City when particular files or features were implemented and also see their behavior, for example, how often is the implementation combined or followed by unit tests. The improvements are summarised in the Future Work chapter (see Chapter 9.1). For an overall view of the both questions results for each scenario see Figure 7.1 and Figure 7.2.

CHAPTER 9

Conclusion

This thesis conducted two main types of research on the code city visualization area. The first contribution is the „Systematic Mapping Study for Code Cities“, which answered the first research questions „How has the city metaphor been used in the software visualization research?“. The research showed how the metaphor was used in software engineering to visualize the projects' source code. Key factors to group the visualizations are the features they used and what the visualization represents. The common attribute of the visualization is its three-dimensional state and the objects where the source code was mapped to represent buildings. Different visualization mapped different aspects of the code into the building such as classes, files, packages, or even more granular aspects like methods, functions, or structs. Most of the visualizations used the height and color aspect to represent code metrics, which were also different such as the number of lines of code or methods in a file, number of dependencies, or mapping dynamic aspects during runtime as such object replication from a class. Some of the visualizations included also special features such as wires, interaction by zooming, or use the building location to further show more information about the project.

The second part of the research question deals with researching the preferences of software stakeholders on how a code city visualization can help them get historical ownership information about a software project. The thesis includes a survey on the information needs of different stakeholders related to the source code and how ownership is used to help a part of these needs. The prototype was adjusted based on the survey and the findings of the mapping study. Then, to answer the RQ2, its requirements were validated and prioritized with semi-structured interviews. After the implementation phase, the scenario-based expert evaluation helped to answer the RQ3 on finding out how purposeful is the idea and how clear the results were to the participants. In both interviews, the semi-structured and scenario-based expert evaluation included ratings from one to five, where one is the lowest and five is the highest rating. All the scenarios received different positive ratings from three to five except the scenario where unit testing consistency was

evaluated. This visualization result clarity for this scenario was evaluated twice with five, once with three, and once with two. The participants who ranked it with two and three added their feedback on how this can be improved which is discussed in Section 9.1.

9.1 Future Work

The need to improve further the prototype came during different phases of the research. The incremental implementation approach showed after the semi-structured interviews that a few requirements needed to be adjusted to be consistent with each other. The adjustments and added requirements that were applied in the visualization were discussed in Section 6.6. However, during the implementation, it was observed that due to the large amount of data in some repositories, the visualization needs to be further simplified. The visualization suffers in performance in particular when all the history is displayed and also all folders are opened up to the file level. Additionally, further improvements were discussed during the scenario-based expert evaluation which will be listed below.

- **Performance**

From the mining up to interacting with the visualization, some separate processes could be more performant such as mining, loading, and rendering the data. The mining part can be considered the least problematic one because it is done only once per project. Additionally, the implementation and the data structure are designed in a way that if new commits were added to the project, then only the new commits are required to be processed. The loading of the data from the backend to the frontend is done upon visualization initialization. One reason is that initially the whole project history is rendered and this requires data from all the commits stored in the database. However, most of the folders are shown initially collapsed without all the internal file details. Therefore for a design that loads only the overview information of a folder and then later the file information on demand may be more performant. The drawback of this approach is that it may require, for example during the mining process, to have the overview data result of the folder saved additionally in the database.

Although the data loading part may take a few seconds, the most process-intensive phase is rendering the visualization initially. By default, the visualization renders the whole history of the project which means that there are at least as many floors as there are commits. Besides this, by default is set to open the folders up to the third level as districts, i.e. the cuboid buildings represent a fourth-level child folder and the cylindric buildings represent a file that can be found from the first to the fourth level. It is not expected that every commit to have affected every file or folder instance up to the fourth level, however, the permutation results in large numbers of floors, where each represents a part-commit that has changed a specific file or folder. Depending on the project's size, this requires a lot of GPU power to render. An improvement here would be to merge floors, especially those that

can be hardly visible. For example, when showing the history of a folder without the internal information, consecutive commits that have changed only a few lines may not be of importance in a list of hundred commits that have changed a large amount of the folder. For example, the Figure 9.1 has a building in the middle of the yellow district which represents almost all the frontend source code. The building has 404 commits resulting in 404 floors, which makes it very difficult to render inside the given height but still to be visible by the human eye. Another case when merging consecutive commits is if they come from the same author in one branch.

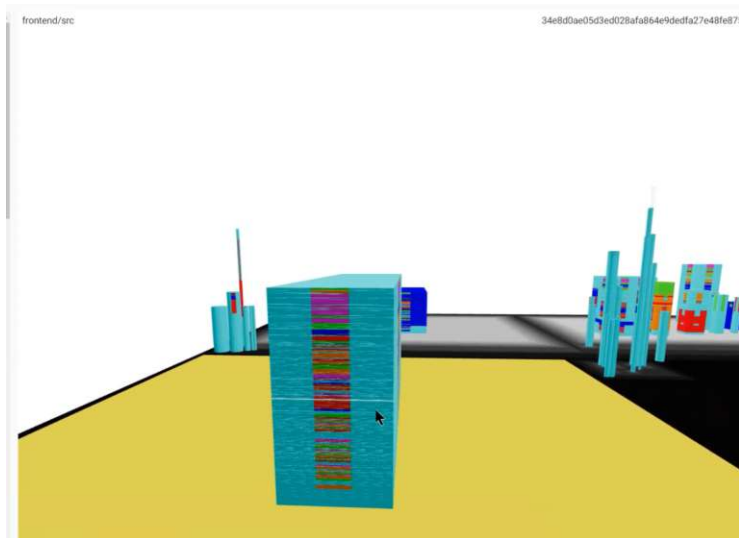


Figure 9.1: Building with many floors

Another option to improve the performance is by constraining the user to the number of floors that can be visible at once in the visualization. For example, if a folder is opened to see its inner details then another one is automatically collapsed to have fewer buildings in the visualization. If the user needs both folders at the same time, then the application should instruct to use the time filter or only filter to show the folders and files of interest.

- **Slide multiple commits at the same time**

The current implementation supports sliding only one commit at once. The vertical slide is split into three parts and each functions differently. The initial state of the parts of a commit can be in different high levels because the buildings have different highest and commits. The bottom part of the slide is used to get the next commits parts into the same level. Then the middle part is where the commit moves up and the third part is where the commit parts go to their upside-down buildings place. During the slide each of these parts has one commit moving. However, the middle part of the slide can have more than one commit sliding at the same

time, while still keeping one at the bottom and one at the upper part. A reason why this was not covered during the thesis is that in some mapping modes (for example linear mapping), some floors may be too high and when a lot of floors with a large height are sliding simultaneously then a very large amount of light needs to be reserved initially. This however can be further studied and researched to find better solutions on how to show multiple commits sliding. The suggestion to slide multiple commits simultaneously came also during the sixth scenario, which involves checking the developer's inconsistency in adding unit tests.

- **Coloring schemes**

The colors of the „Slide City“ are bound to the author. The user can change the color that represents the author, but not the meaning of the color. During the last scenario, which involves finding a file or folder where a lot of developers have switched ownership, one of the interviewees suggested using the colors differently in a way that color does not show anymore the author but rather has different shades of the same color depending by how many different developers switched the ownership. This configuration can help also in other scenarios where the user should find something that can be calculated and found by the visualization logic.

- **Commit sorting**

During the semi-structured interviews, the participants expressed curiosity about sorting the commits not by date but by considering the branches. Sorting by date may list two different branches by combining and mixing them. In some special cases, it may result in having the ownership being switched multiple times during the history of a folder, whereas in reality, the ownership belonged to one developer in one branch and the other developer on the other branch. If they had been sorted by considering the branches, then the building would have at the bottom all the commits of one branch and then next the commit of the other branch. The concept may conflict with the fact that the slide city uses time in the legend to slide the commits, which results in breaking the order that the commits had while they were on the building. To avoid these problems, Git graphics use the tree system to represent both the time and the relation between commits, for example, its predecessor and its successor. However when showing in one line, for example on a vertical building, one of the information will be lost.

Another problem with prioritizing the branches over the commit time is the complexity that some branches may have. If the Git workflow is implemented correctly, then the instruction to implement the Git workflow can be used to sort the commits. However, if branches are similar to the example in Figure 5.9, then it becomes very hard to prioritize the branches in a user-friendly way for the user. A separate specific study and analysis is needed to handle these special cases.

List of Figures

3.1	Screenshot of Hoozizat	12
3.2	Information Fragment Model	13
3.3	Major and minor owners	14
3.4	Implicated code	15
3.5	The heat map	16
3.6	Code Ownership River	18
3.7	CHRONOS	18
3.8	Azurite	19
3.9	Chronia	20
3.10	Timeline of Oversight	21
3.11	ArgoUML as a CodeCity	23
3.12	CodeCity	23
3.13	CodeCity history visualization	24
3.14	Evo-Streets	25
3.15	The heat map	26
3.16	The heat map	26
3.17	M3triCity	27
4.1	FileVis visualization	29
4.2	Software World visualization	30
4.3	Software Landscapes visualization	30
4.4	3D City proposal	31
4.5	Vizz3D visualization	31
4.6	VERSO visualization	32
4.7	VERSO extended visualization	32
4.8	CodeCity visualization	33
4.9	CocoViz visualization	33
4.10	MetricView visualization	34
4.11	UML-City visualization	34
4.12	EvoSpaces visualization	34
4.13	Evo-Streets visualization	35
4.14	Evo-Streets wires visualization	35
4.15	VizzAspectJ City & VizzJava City visualizations	35
4.16	VITRAIL City visualization	36

4.17	VITRAIL Streets visualization	36
4.18	SkyscrapAR visualization	37
4.19	SynchroVis visualization	38
4.20	SeeIT 3D visualization	38
4.21	ExplorViz Application visualization	39
4.22	ExploreViz 3D printed model	39
4.23	ExploreViz with heat map overlay	39
4.24	TeamWATCH visualization	40
4.25	CodeMetropolis visualization	41
4.26	Rocat visualization	41
4.27	Origin City visualization	42
4.28	VR City	42
4.29	CityVR visualization	43
4.30	High-Rising Cities visualization	43
4.31	Code Park visualization	43
4.32	Linked Data City visualization	44
4.33	GoCity visualization	45
4.34	IslandViz visualization	45
4.35	PerfVis visualization	46
4.36	Software City in VR visualization	46
4.37	Memory Cities visualization	47
4.38	M3triCity visualization	48
4.39	M3triCity2 visualization	48
4.40	The Layered Software City visualization	49
4.41	DynaCity visualization	49
4.42	BabiaXR-CodeCity visualization	50
4.43	VariCity visualization	50
5.1	Slide city	56
5.2	Abstract buildings	57
5.3	Opened folder	58
5.4	Closed folder	58
5.5	Proposed visualization	59
5.6	Committer color	60
5.7	Code city	61
5.8	Histogram & Box-whisker equalization	62
5.9	Git Branches	63
5.10	Merged commits	64
5.11	File Movements	66
5.12	Slide City sketch	68
5.13	Slide City sketch	68
5.14	Ranked answers for „Who has been working on a feature?“	71
5.15	Ranked answers for „Which features has a developer implemented?“	71

5.16	Ranked answers for „How should the visualization map the metrics?“ . . .	72
5.17	Ranked answers for „How should the visualization be filtered by the author?“	74
5.18	Ranked answers for „How useful is it to exclude files?“	74
5.19	Ranked answers for „How should the blocklisted commits be excluded?“ .	75
5.20	Ranked answers for „Which features are more important?“	75
6.1	Architecture	78
6.2	Commit log	81
6.3	Commit diff	82
6.4	Linear Mapping	84
6.5	Histogram mapping	85
6.6	Logarithmic mapping	86
6.7	Live Configuration	87
6.8	Commit and Files Details	87
6.9	Cylindrical building	90
6.10	Cylindrical building	91
7.1	Visualization purposful	96
7.2	Visualization clarity	96
7.3	Scenario result 1	97
7.4	Scenario result 2	98
7.5	Scenario result 3	99
7.6	Scenario result 4	100
7.7	Scenario result 5	101
7.8	Scenario result 6	102
7.9	Scenario result 7	103
7.10	Scenario result 8	104
9.1	Building with many floors	111

List of Tables

4.1	Systematic mapping study: First table	51
4.2	Systematic mapping study: Second table	52
4.3	Systematic mapping study: Special features occurrence	53
4.4	Systematic mapping study: Special feature per city	54
5.1	Logarithmic mapping results	62

Bibliography

- [1] R. Purushothaman and D. E. Perry, “Toward understanding the rhetoric of small source code changes,” *IEEE Transactions on Software Engineering*, vol. 31, pp. 511–526, jun 2005.
- [2] T. Fritz and G. C. Murphy, “Using information fragments to answer the questions developers ask,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, vol. 1, (New York, New York, USA), p. 175, ACM Press, 2010.
- [3] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse, “How developers drive software evolution,” *International Workshop on Principles of Software Evolution (IWPSE)*, vol. 2005, pp. 113–122, 2005.
- [4] C. Liu, X. Ye, and E. Ye, “Source code revision history visualization tools: Do they work and what would it take to put them to work?,” *IEEE Access*, vol. 2, pp. 404–426, 2014.
- [5] R. Wettel and M. Lanza, “Visual exploration of large-scale system evolution,” *Proceedings - Working Conference on Reverse Engineering, WCRE*, pp. 219–228, 2008.
- [6] F. Steinbrückner and C. Lewerentz, “Representing Development History in Software Cities,” *Proceedings of the ACM Conference on Computer and Communications Security*, 2010.
- [7] F. Pfahler, R. Minelli, C. Nagy, and M. Lanza, “Visualizing Evolving Software Cities,” *Proceedings - 8th IEEE Working Conference on Software Visualization, VISSOFT 2020*, pp. 22–26, sep 2020.
- [8] C. Knight and M. Munro, “Comprehension with[in] virtual environment visualisations,” *Proceedings - 7th International Workshop on Program Comprehension, IWPC 1999*, pp. 4–11, 1999.
- [9] C. Knight and M. Munro, “Virtual but visible software,” *Proceedings of the International Conference on Information Visualisation*, vol. 2000-July, pp. 198–205, 2000.

- [10] R. Wetzel and M. Lanza, "Program comprehension through software habitability," *IEEE International Conference on Program Comprehension*, pp. 231–240, 2007.
- [11] T. Panas, R. Berrigan, and J. Grundy, "A 3D metaphor for software production visualization," *Proceedings of the International Conference on Information Visualisation*, vol. 2003-Janua, pp. 314–319, 2003.
- [12] S. Ardigo, C. Nagy, R. Minelli, and M. Lanza, "Visualizing Data in Software Cities," *Proceedings - 2021 Working Conference on Software Visualization, VISSOFT 2021*, pp. 145–149, 2021.
- [13] D. Moreno-Lumbreras, R. Minelli, A. Villaverde, J. M. Gonzalez-Barahona, and M. Lanza, "CodeCity: On-Screen or in Virtual Reality?," *Proceedings - 2021 Working Conference on Software Visualization, VISSOFT 2021*, pp. 12–22, 2021.
- [14] J. Mortara, P. Collet, and A. M. Dery-Pinna, "Visualization of Object-Oriented Variability Implementations as Cities," *Proceedings - 2021 Working Conference on Software Visualization, VISSOFT 2021*, pp. 76–87, 2021.
- [15] A. Krause, M. Hansen, and W. Hasselbring, "Live Visualization of Dynamic Software Cities with Heat Map Overlays," *Proceedings - 2021 Working Conference on Software Visualization, VISSOFT 2021*, pp. 125–129, 2021.
- [16] V. Dashuber and M. Philippsen, "Trace Visualization within the Software City Metaphor: A Controlled Experiment on Program Comprehension," *Proceedings - 2021 Working Conference on Software Visualization, VISSOFT 2021*, pp. 55–64, 2021.
- [17] K. Petersen, S. Vakkalanka, and L. Kuzniarz, "Guidelines for conducting systematic mapping studies in software engineering: An update," 2015.
- [18] K. R. Felizardo, E. Mendes, M. Kalinowski, É. Ferreira Souza, and N. L. Vijaykumar, "Using Forward Snowballing to update Systematic Reviews in Software Engineering," *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2016.
- [19] S. Jalali and C. Wohlin, "Systematic Literature Studies: Database Searches vs. Backward Snowballing," *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement - ESEM '12*, 2012.
- [20] P. Young and M. Munro, "Visualising software in virtual reality," in *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242)*, pp. 19–26, IEEE Comput. Soc, 1998.
- [21] B. Sharif, G. Jetty, J. Aponte, and E. Parra, "An empirical study assessing the effect of SeeIT 3D on comprehension," *2013 1st IEEE Working Conference on Software Visualization - Proceedings of VISSOFT 2013*, 2013.

- [22] H. Kallio, A. M. Pietilä, M. Johnson, and M. Kangasniemi, "Systematic methodological review: developing a framework for a qualitative semi-structured interview guide," *Journal of Advanced Nursing*, vol. 72, pp. 2954–2965, dec 2016.
- [23] J. Dahmann, D. Gregorio, and P. Modigliani, "Systems engineering processes for agile software development," *SysCon 2013 - 7th Annual IEEE International Systems Conference, Proceedings*, pp. 351–355, 2013.
- [24] M. Weninger, L. Makor, and H. Mossenbock, "Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor," *Proceedings - 8th IEEE Working Conference on Software Visualization, VISSOFT 2020*, pp. 110–121, sep 2020.
- [25] A. Begel, K. Y. Phang, and T. Zimmermann, "Codebook: Discovering and Exploiting Relationships in Software Repositories," *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, vol. 1, pp. 125–134, may 2010.
- [26] A. Begel and T. Zimmermann, "Analyze this! 145 questions for data scientists in software engineering," *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pp. 12–23, 2014.
- [27] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code! Examining the effects of ownership on software quality," *SIGSOFT/FSE 2011 - Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 4–14, 2011.
- [28] M. Cataldo, P. A. Wagstrom, J. D. Herbsleb, and K. M. Carley, "Identification of coordination requirements: Implications for the Design of collaboration and awareness tools," *Proceedings of the ACM Conference on Computer Supported Cooperative Work, CSCW*, pp. 353–362, 2006.
- [29] R. M. Henderson and K. B. Clark, "Architectural Innovation: The Reconfiguration of Existing Product Technologies and the Failure of Established Firms," *Administrative Science Quarterly*, vol. 35, p. 9, mar 1990.
- [30] M. Foucault, J.-R. Falleri, and X. Blanc, "Code Ownership in Open-Source Software," *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering - EASE '14*, 2014.
- [31] M. Foucault, C. Teyton, D. Lo, X. Blanc, and J. R. Falleri, "On the usefulness of ownership metrics in open-source software projects," *Information and Software Technology*, vol. 64, pp. 102–112, aug 2015.
- [32] F. Rahman and P. Devanbu, "Ownership, experience and defects: A fine-grained study of authorship," *Proceedings - International Conference on Software Engineering*, no. 11, pp. 491–500, 2011.

- [33] K. Beck, “Embracing change with extreme programming,” *Computer*, vol. 32, pp. 70–77, oct 1999.
- [34] K. H. Judy and I. Krumins-Beens, “Great scrums need great product owners: Unbounded collaboration and collective product ownership,” *Proceedings of the Annual Hawaii International Conference on System Sciences*, 2008.
- [35] J. Sutherland, G. Schoonheim, E. Rustenburg, and M. Rijk, “Fully distributed scrum: The secret sauce for hyperproductive offshored development teams,” *Proceedings - Agile 2008 Conference*, pp. 339–344, 2008.
- [36] V. Augustine, J. Hudepohl, P. Marcinczak, and W. Snipes, “Deploying Software Team Analytics in a Multinational Organization,” *IEEE Software*, vol. 35, pp. 72–76, jan 2017.
- [37] M. Orrú and M. Marchesi, “A case study on the relationship between code ownership and refactoring activities in a Java software system,” *Proceedings - 7th International Workshop on Emerging Trends in Software Metrics, WETSoM 2016*, pp. 43–49, may 2016.
- [38] P. Thongtanunam and C. Tantithamthavorn, “Code Ownership: The Principles, Differences, and Their Associations with Software Quality,”
- [39] M. H. D. D. Moura, H. A. D. D. Nascimento, and T. C. Rosa, “Extracting new metrics from version control system for the comparison of software developers,” *Proceedings - 28th Brazilian Symposium on Software Engineering, SBES 2014*, pp. 41–50, 10 2014.
- [40] S. G. Eick, J. L. Steffen, and E. E. Sumner, “Seesoft—A Tool for Visualizing Line Oriented Software Statistics,” *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 957–968, 1992.
- [41] J. P. S. Alcocer, F. Beck, and A. Bergel, “Performance evolution matrix: Visualizing performance variations along software versions,” *Proceedings - 7th IEEE Working Conference on Software Visualization, VISSOFT 2019*, pp. 1–11, sep 2019.
- [42] A. Hanjalić, “ClonEvol: Visualizing software evolution with code clones,” *2013 1st IEEE Working Conference on Software Visualization - Proceedings of VISSOFT 2013*, 2013.
- [43] C. V. Alexandru, S. Proksch, P. Behnamghader, and H. C. Gall, “Evo-clocks: Software evolution at a glance,” *Proceedings - 7th IEEE Working Conference on Software Visualization, VISSOFT 2019*, pp. 12–22, sep 2019.
- [44] A. Telea and D. Auber, “Code Flows: Visualizing Structural Evolution of Source Code,” *Computer Graphics Forum*, vol. 27, pp. 831–838, may 2008.

- [45] M. Wittenhagen, C. Cherek, and J. Borchers, "Chronicler: Interactive exploration of source code history," *Conference on Human Factors in Computing Systems - Proceedings*, pp. 3522–3532, may 2016.
- [46] B. Ens, D. Rea, R. Shpaner, H. Hemmati, J. E. Young, and P. Irani, "ChronoTwigger: A visual analytics tool for understanding source and test co-evolution," *Proceedings - 2nd IEEE Working Conference on Software Visualization, VISSOFT 2014*, pp. 117–126, dec 2014.
- [47] C. Klammer, G. Buchgeher, and A. Kern, "A retrospective of production and test code co-evolution in an industrial project," *2018 IEEE 2nd International Workshop on Validation, Analysis and Evolution of Software Tests, VST 2018 - Proceedings*, vol. 2018-March, pp. 16–20, mar 2018.
- [48] J. Wu, C. W. Spitzer, A. E. Hassan, and R. C. Holt, "Evolution spectrographs: Visualizing punctuated change in software evolution," *International Workshop on Principles of Software Evolution (IWPSE)*, pp. 57–66, 2004.
- [49] L. Voinea, A. Telea, and J. J. Van Wijk, "CVSscan: Visualization of code evolution," *Proceedings SoftVis '05 - ACM Symposium on Software Visualization*, pp. 47–56, 2005.
- [50] M. Ogawa and K. L. Ma, "Code-swarm: A design study in organic software visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, pp. 1097–1104, nov 2009.
- [51] S. Rufiange and G. Melancon, "AniMatrix: A matrix-based visualization of software evolution," *Proceedings - 2nd IEEE Working Conference on Software Visualization, VISSOFT 2014*, pp. 137–146, dec 2014.
- [52] J. Grabner, R. Decker, T. Artner, M. Bernhart, and T. Grechenig, "Combining and Visualizing Time-Oriented Data from the Software Engineering Toolset," in *Proceedings - 6th IEEE Working Conference on Software Visualization, VISSOFT 2018*, pp. 76–86, Institute of Electrical and Electronics Engineers Inc., nov 2018.
- [53] F. Servant and J. A. Jones, "Chronos: Visualizing slices of source-code history," *2013 1st IEEE Working Conference on Software Visualization - Proceedings of VISSOFT 2013*, 2013.
- [54] Y. S. Yoon, B. A. Myers, and S. Koo, "Visualization of fine-grained code change history," *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*, pp. 119–126, 2013.
- [55] A. Kuhn and M. Stocker, "CodeTimeline: Storytelling with versioning data," *Proceedings - International Conference on Software Engineering*, pp. 1333–1336, 2012.

- [56] W. Aigner, S. Miksch, H. Schumann, and C. Tominski, *Visualization of Time-Oriented Data*. 2011.
- [57] C. Ware and G. Franck, "Viewing a graph in a virtual reality display is three times as good as a 2D diagram," *IEEE Symposium on Visual Languages, Proceedings*, pp. 182–183, 1994.
- [58] C. Ware, D. Hui, and G. Franck, "Visualizing Object Oriented Software in Three Dimension s,"
- [59] S. K. Card, J. D. Mackinlay, and B. Shneiderman, "Readings in information visualization: using vision to think," *Technical Communication Quarterly*, vol. 9, pp. 347–351, 2000.
- [60] G. S. Hubona, G. W. Shirah, and D. G. Fout, "3D object recognition with motion," *Conference on Human Factors in Computing Systems - Proceedings*, vol. 22-27-Marc, pp. 345–346, mar 1997.
- [61] R. Wettel and M. Lanza, "Visualizing Software Systems as Cities," *VISSOFT 2007 - Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pp. 92–99, 2007.
- [62] A. Marcus, L. Feng, and J. I. Maletic, "3D representations for software visualization," p. 27, 2003.
- [63] R. Wettel and M. Lanza, "Code city: 3D visualization of large-scale software," *Proceedings - International Conference on Software Engineering*, pp. 921–922, 2008.
- [64] M. Gao and C. Liu, "TeamWATCH demonstration: A web-based 3D software source code visualization for education," *1st International Code Hunt Workshop on Educational Software Engineering, CHESE 2015 - Proceedings*, pp. 12–15, jul 2015.
- [65] M. Balzer, A. Noack, O. Deussen, and C. Lewerentz, "Software Landscapes : Visualizing the Structure of Large Software Systems," *Joint EUROGRAPHICS-IEEE TCVG Symposium on Visualization*, 2004.
- [66] T. Panas, R. Lincke, and W. Löwe, "Online-Configuration of Software Visualizations with Vizz3D," *Proceedings of the ACM 2005 Symposium on Software Visualization, St. Louis, Missouri, USA, May 14-15, 2005*, 2005.
- [67] T. Panas, T. Epperly, D. Quinlan, A. Sæbjørnsen, and R. Vuduc, "Communicating software architecture using a unified single-view visualization," *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems, ICECCS*, pp. 217–226, 2007.
- [68] G. Langelier, H. Sahraoui, and P. Poulin, "Visualization-based analysis of quality for large-scale software systems," *20th IEEE/ACM International Conference on Automated Software Engineering, ASE 2005*, pp. 214–223, 2005.

- [69] G. Langelier and K. Dhambri, “Visual analysis of Azureus using VERSO,” *VIS-SOFT 2007 - Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pp. 163–164, 2007.
- [70] G. Langelier, H. Sahraoui, and P. Poulin, “Exploring the evolution of software quality with animated visualization,” *Proceedings - 2008 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2008*, pp. 13–20, 2008.
- [71] O. Benomar, H. Sahraoui, and P. Poulin, “Visualizing software dynamicities with heat maps,” *2013 1st IEEE Working Conference on Software Visualization - Proceedings of VISSOFT 2013*, 2013.
- [72] R. Wettel and M. Lanza, “Visually localizing design problems with disharmony maps,” *SOFTVIS 2008 - Proceedings of the 4th ACM Symposium on Software Visualization*, pp. 155–164, 2008.
- [73] S. Boccuzzo and H. Gall, “CocoViz: Towards cognitive software visualizations,” *VISSOFT 2007 - Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pp. 72–79, 2007.
- [74] S. Boccuzzo and H. C. Gall, “Software visualization with audio supported cognitive glyphs,” *IEEE International Conference on Software Maintenance, ICSM*, pp. 366–375, 2008.
- [75] C. F. Lange, M. A. Wijns, and M. R. Chaudron, “A visualization framework for task-oriented modeling using UML,” *Proceedings of the Annual Hawaii International Conference on System Sciences*, 2007.
- [76] S. Alam and P. Dugerdil, “EvoSpaces visualization tool: Exploring software architecture in 3D,” *Proceedings - Working Conference on Reverse Engineering, WCRE*, pp. 269–270, 2007.
- [77] M. Steinbeck, R. Koschke, and M. O. Rudel, “How EvoStreets Are Observed in Three-Dimensional and Virtual Reality Environments,” *SANER 2020 - Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution, and Reengineering*, pp. 332–343, feb 2020.
- [78] S. Benträd and D. Meslati, “2D and 3D visualization of AspectJ programs,” *Proceedings of the 10th International Symposium on Programming and Systems, ISPS’ 2011*, pp. 183–190, 2011.
- [79] P. Caserta, O. Zendra, and D. Bodenes, “3D hierarchical edge bundles to visualize relations in a software city metaphor,” *Proceedings of VISSOFT 2011 - 6th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 2011.
- [80] R. R. G. e. Souza, T. Mendes, B. c. da Silva, and M. Mendonça, “SkyscrapAR: An Augmented Reality Visualization for Software Evolution,” 2012.

- [81] J. Waller, C. Wulf, F. Fittkau, P. Döhring, and W. Hasselbring, “SynchroVis: 3D visualization of monitoring traces in the city metaphor for analyzing concurrency,” *2013 1st IEEE Working Conference on Software Visualization - Proceedings of VISSOFT 2013*, 2013.
- [82] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring, “Live trace visualization for comprehending large software landscapes: The ExplorViz approach,” *2013 1st IEEE Working Conference on Software Visualization - Proceedings of VISSOFT 2013*, 2013.
- [83] F. Fittkau, A. Krause, and W. Hasselbring, “Software landscape and application visualization for system comprehension with ExplorViz,” *Information and Software Technology*, vol. 87, pp. 259–277, jul 2017.
- [84] F. Fittkau, S. Roth, and W. Hasselbring, “ExplorViz: Visual runtime behavior analysis of enterprise application landscapes,” 2015.
- [85] F. Fittkau, A. Krause, and W. Hasselbring, “Exploring software cities in virtual reality,” *2015 IEEE 3rd Working Conference on Software Visualization, VISSOFT 2015 - Proceedings*, pp. 130–134, nov 2015.
- [86] F. Fittkau, E. Koppenhagen, and W. Hasselbring, “Research perspective on supporting software engineering via physical 3D models,” *2015 IEEE 3rd Working Conference on Software Visualization, VISSOFT 2015 - Proceedings*, pp. 125–129, nov 2015.
- [87] G. Balogh, A. Szabolics, and A. Beszedes, “CodeMetropolis: Eclipse over the city of source code,” *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation, SCAM 2015 - Proceedings*, pp. 271–276, nov 2015.
- [88] T. Ichinose, K. Uemura, D. Tanaka, H. Hata, H. Iida, and K. Matsumoto, “RO-CAT on KATARIBE: Code Visualization for Communities,” *Proceedings - 4th International Conference on Applied Computing and Information Technology, 3rd International Conference on Computational Science/Intelligence and Applied Informatics, 1st International Conference on Big Data, Cloud Computing, Data Science*, pp. 158–163, 2016.
- [89] R. Ishizue, H. Washizaki, Y. Fukazawa, S. Inoue, Y. Hanai, M. Kanazawa, and K. Namba, “Metrics Visualization Technique Based on the Origins and Function Layers for OSS-Based Development,” *Proceedings - 2016 IEEE Working Conference on Software Visualization, VISSOFT 2016*, pp. 71–75, dec 2016.
- [90] J. Vincur, P. Navrat, and I. Polasek, “VR City: Software Analysis in Virtual Reality Environment,” *Proceedings - 2017 IEEE International Conference on Software Quality, Reliability and Security Companion, QRS-C 2017*, pp. 509–516, aug 2017.

- [91] L. Merino, M. Ghafari, C. Anslow, and O. Nierstrasz, "CityVR: Gameful software visualization," *Proceedings - 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017*, pp. 633–637, nov 2017.
- [92] L. Merino, J. Fuchs, M. Blumenschein, C. Anslow, M. Ghafari, O. Nierstrasz, M. Behrisch, and D. A. Keim, "On the Impact of the Medium in the Effectiveness of 3D Software Visualizations," *Proceedings - 2017 IEEE Working Conference on Software Visualization, VISSOFT 2017*, vol. 2017-October, pp. 11–21, oct 2017.
- [93] K. Ogami, R. G. Kula, H. Hata, T. Ishio, and K. Matsumoto, "Using High-Rising Cities to Visualize Performance in Real-Time," *Proceedings - 2017 IEEE Working Conference on Software Visualization, VISSOFT 2017*, vol. 2017-October, pp. 33–42, oct 2017.
- [94] P. Khaloo, M. Maghoumi, E. Taranta, D. Bettner, and J. Laviola, "Code Park: A New 3D Code Visualization Tool," *Proceedings - 2017 IEEE Working Conference on Software Visualization, VISSOFT 2017*, vol. 2017-October, pp. 43–53, oct 2017.
- [95] K. Andries De Graaf and A. Khalili, "Visualizing Linked Data as Habitable Cities," *Third International Workshop on Visualization and Interaction for Ontologies and Linked Data co-located with the 16th International Semantic Web Conference*, 2017.
- [96] R. Brito, A. Brito, G. Brito, and M. T. Valente, "GoCity: Code City for Go," *SANER 2019 - Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering*, pp. 649–653, mar 2019.
- [97] A. Schreiber, L. Nafeie, A. Baranowski, P. Seipel, and M. Misiak, "Visualization of Software Architectures in Virtual Reality and Augmented Reality," *IEEE Aerospace Conference Proceedings*, vol. 2019-March, mar 2019.
- [98] L. Merino, M. Hess, A. Bergel, O. Nierstrasz, and D. Weiskopf, "PerfVis: Pervasive Visualization in Immersive Augmented Reality for Performance Awareness," *ICPE 2019 - Companion of the 2019 ACM/SPEC International Conference on Performance Engineering*, vol. 4, 2019.
- [99] F. Jung, V. Dashuber, and M. Philippsen, "Towards Collaborative and Dynamic Software Visualization in VR," *Proceedings of the 15th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2020) - Volume 3: IVAPP, pages 149-156*, 2020.
- [100] V. Dashuber, M. Philippsen, and J. Weigend, "A Layered Software City for Dependency Visualization," *Proceedings of the 16th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 3: IVAPP*, vol. Proceeding, 2021.
- [101] W. Harrison, K. Magel, R. Kluczny, and A. DeKock, "Applying Software Complexity Metrics to Program Maintenance," *Computer*, vol. 15, no. 9, pp. 65–79, 1982.

- [102] R. Wettel, M. Lanza, and R. Robbes, “Software Systems as Cities: A Controlled Experiment,” *Proceedings - International Conference on Software Engineering*, pp. 551–560, 2011.

Appendix

Semi-Structured Expert Interview——Questions

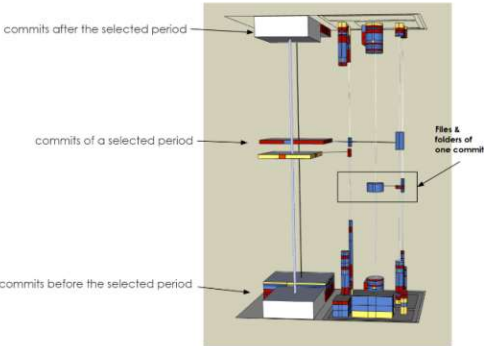
Visualizing Historical Ownership with Code City Metaphor

Slide City is a software evolution visualization based on the code city metaphor. It shows all the commits implemented or merged into a GIT branch, with a focus on ownership. **Ownership** is calculated with lines of code (LoC) per file or folder. The **highest code owner** of a file/folder has the largest ratio of implemented LoC against the total LoC in the file/folder.

The buildings in code city may be directories (cuboid shape) or files (cylindric shape). The user can click to open or close the cuboid buildings like folders. The cuboid shape building will be changed to a rectangle district and show its internal files and folders as buildings.

A building is constructed with the list of commits that are related to the folder or file. The lower the floors are set into the building the older the commits that they represent are. Each floor represents a commit related to that file or folder colored by the highest code owner at the commit moment. The surrounding building color represents the highest code owner.

The slide feature pulls the commits up from the city. Therefore the floors of the buildings go up one by one. This simulates a time travel in the city where it is possible to see when the commits were done and how the city looked with or without them.



commits after the selected period

commits of a selected period

commits before the selected period

Files & folders of one commit

enrikndou@gmail.com (not shared) [Switch account](#)

Slide City

Section 2: Demographics

Age *

☐ [18 < 30) - From 18 to 29

☐ [30-40) - From 30 to 39

☐ [40-50) - From 40 to 50

☐ >50 - More than 50

Gender

☐ Female

☐ Male

☐ Other:

How many years have you been working with GIT? *

☐ 0 No experience

☐ [0-2) Less than 2 years

☐ [2-5) From 2 to 5 years

☐ [5-10) From 5 to 10 years

☐ More than 10 years

Did you ever have a leading role in a software engineering project? If yes, please *
shortly describe the role.

Your answer

Section 3: Information needs

The following questions are related to the metrics you would use to answer ownership questions. The first is to find the contributors for a particular feature, and the second is to find the features that a developer has contributed.

Terms Description:

The file level calculates and shows the ownership for each file individually.

The folder level (or package level) can be selected by the user to show an overview of its internal files' ownership.

Which metrics do you find useful to answer the question, "Who has been working on a feature?" *

Please rank the options from 1 (most important) to 6 (least important).

Terms Description:

The file level calculates and shows the ownership for each file individually.

The folder level (or package level) can be selected by the user to show an overview of its internal files' ownership.

(Optional) If you propose another alternative, please describe it in the next question.

	1	2	3	4	5	6
Present Ownership at file level	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Historical Ownership at file level	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Present Ownership at folder level	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Historical Ownership at folder level	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Last commiter	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Other	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

(Optional) Please describe further alternatives for: Which metrics do you find useful to answer the question, "Who has been working on a feature?"

Your answer

Which metrics do you find useful to answer the question, "Which features has a developer implemented?" *

Please rank the options from 1 (most important) to 5 (least important).

Terms Description:

The file level calculates and shows the ownership for each file individually.

The folder level (or package level) can be selected by the user to show an overview of its internal files' ownership.

(Optional) If you propose another alternative, please describe it in the next question.

	1	2	3	4	5
Present Ownership at file level	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Historical Ownership at file level	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Present Ownership at folder level	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Historical Ownership at folder level	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Other	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

(Optional) Please describe further alternatives for: Which metrics do you find useful to answer the question, "Which features has a developer implemented?"

Your answer

Section 4: Slide City Features

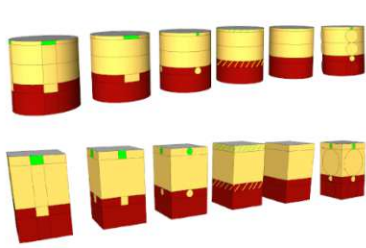
How should the committer color be shown along with the highest code owner color? *

Each floor represents a commit on a file or folder. The file or folder ownership may change with every new commit. In the following example, the second commit is from the yellow author; however, the highest owner in that commit remained the red author. Similar is the 5th commit: The green author commits, but the highest owner is yellow.

The main color (the one surrounding the floor) represents the highest code owner at that time. When a new author commits but does not take ownership, it will result in a building floor with a committer different than the highest code owner.

Please rank the options from 1 (most important) to 7 (least important).

(Optional) If you propose another alternative, please describe it in the next question.



	1	2	3	4	5	6	7
Rectangle, for each floor	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Rectangle, only for floors where the highest code owner is different than the committer	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Circle, only for floors where the highest code owner is different than the committer	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Stripes	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Configurable. The user can switch the colors for the whole city.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Circle, for each floor	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Other	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

(Optional) Please describe further alternatives for: How should the committer color be shown along with the highest code owner color?

Your answer

How to map colors to multiple authors? *

Please rank the options from 1 (most important) to 5 (least important).

(Optional) If you propose another alternative, please describe it in the next question.

	1	2	3	4	5
Assign all authors random colors	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Let the user assign the colors	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Assign to minor authors only shades of a particular color (e.g., red) and assign the rest of the colors to the primary authors.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Assign to authors only dark colors. The user can then manually switch authors of interest to a bright color	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Other	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

(Optional) Please describe further alternatives for: How to map colors to multiple authors?

Your answer

What should the height of a floor represent? *

Please rank the options from 1 (most important) to 3 (least important).

(Optional) If you propose another alternative, please describe it in the next question.

	1	2	3
Added lines. (Commits which only remove lines will not be shown.)	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Added and removed lines. (Moved code may be displayed twice)	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Other	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

(Optional) Please describe further alternatives for: What should the height of the building represent?

Your answer

How should the visualization map the metrics? *

Mapping parameters linearly into buildings may result in some very large buildings while others are very short. Another anomaly may happen when the visualization shows a few very large (e.g., with 50 000) and a few very small files; then the 99% of the files that are in the range of 100 to 200 lines of code may risk looking equal if they are mapped into a range from 0 to 50 000. Some techniques to handle mapping the metrics to the visualization are listed below:

Terms Description:

Linear mapping: Maps attributes 1 to 1 from GIT-data to slide city.

Box-whisker plot: Maps data into four quartiles and shows data between a range as equal. (See Box-whisker plot image).

Histogram: Similar to Box-whisker plot, but instead of 4 quartiles, there can be n range separation.

Threshold: Different from the histogram, the selected ranges are not split independently from the current data set. Helpful to compare different code cities. However, inside one code city, data distribution will be skewed to only small or large buildings.

Logarithmic mapping: As larger the numbers become, the less their effect is seen in the visualization.

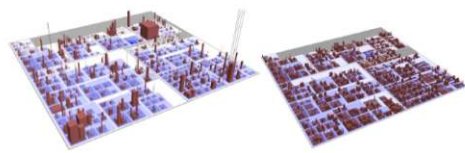
For example, some values with their logs:

1 -> 0 (This value can be replaced with a default minimum, e.g., 0.001)
10 -> 1 100 -> 2 1000 -> 3
20 -> 1,301 110 -> 2,041 1010 -> 3,004
50 -> 1,699 500 -> 2,699 5000 -> 3,699

The example below is from Wettel et al. CodeCity left (linear mapping) and right (box-whisker plot).

Please rank the options from 1 (most important) to 7 (least important).

(Optional) If you propose another alternative, please describe it in the next question.



	1	2	3	4	5	6	7
Linear	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Box-whisker plot	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Histogram	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Threshold	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Logarithmic	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Other	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Histogram & Box-whisker equalization (Aigner et al. Visualization of Time-Oriented Data)

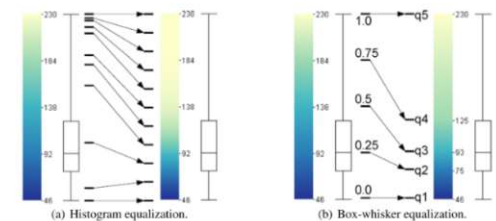


Fig. 4.15: Equalization schemas for adapting a color scale to the data distribution, which is depicted as box-whisker plots.

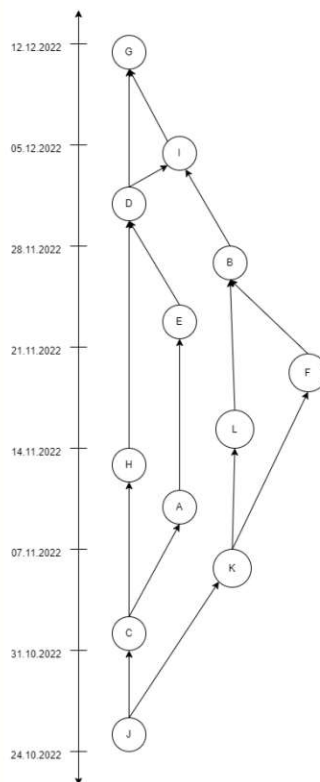
(Optional) Please describe further alternatives for: How should the visualization map the metrics?

How should the commits be sorted? *

The example assumes all commits have changes related to one file or folder that will be rendered as one building.

Please rank the options from 1 (most important) to 3 (least important).

(Optional) If you propose another alternative, please describe it in the next question.



	1	2	3
Sort by date: J, C, K, A, H, F, E, B, D, I, G	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Sort branches from earliest merge-commit first: J, K, L, F, B, C, A, E, H, D, I, G	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Other	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

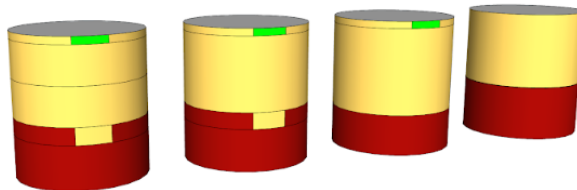
(Optional) Please describe further alternatives for: How should the commits be sorted?

Your answer

How should consecutive commits be shown? *

Please rank the options from 1 to 5.

(Optional) If you propose another alternative, please describe it in the next question.



	1	2	3	4	5
Don't merge	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Merge floors from the same committer which also have the same ownership	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Merge all floors from the same committer	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Merge floors that have same ownership. (Ignore committer)	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Other	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

(Optional) Please describe further alternatives for: How should consecutive commits be shown?

Your answer

How should merge commits be shown? *

When two or more branches are merged, the merge commit has different changes to each parent branch.

In some cases, it is necessary to resolve conflicts manually. These changes will be different for each parent branch.

Please rank the options from 1 (most important) to 3 (least important).

(Optional) If you propose another alternative, please describe it in the next question.

	1	2	3
Don't show merge commits	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Render only changes that are different to every parent (i.e., resolved conflicts)	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Other	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

(Optional) Please describe further alternatives for: How should merge commits be shown?

Section 5: Building Construction and Deconstruction

This section contains questions related to the folder and file construction.

During project implementation, the structure of folders and files may have changed several times, i.e., files and folders can be deleted or moved to other folders. Additionally, developers may remove folders and split their content into current or new folders. The files themselves can be split, removed, or added again later.

The questions of this section will be related to the following example with three commits:

Commit 1: Red author creates **Folder A** and **Folder B**, and creates **File C** to **Folder B**.

Commit 2: Yellow author commits changes to **Folder A**, moves **File C** to **Folder A**, and deletes **Folder B**.

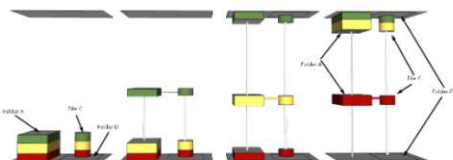
Commit 3: Green author commits changes to **Folder A**, creates **Folder D**, and moves **File C** to **Folder D**.

Terms explanation:

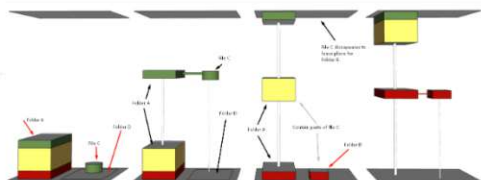
Intact file history: Every change related to a file is shown in one building. The latest folder where the file was moved includes all the files' history.

Split file history: Each file modification is shown inside the folder where it was changed. The latest folder where the file was moved into includes only the changes of the file that happened in that folder.

Slide City with "Intact File history"



Slide City with "Split file history"



How should the Slide City visualize the city base initially? *

On the left "Intact File history"; on the right "Split file history".

Please rank the options from 1 to 3.

(Optional) If you propose another alternative, please describe it in the next question.

	Show "Intact file history"	Show "Split file history"	Other
Rank 1	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Rank 2	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Rank 3	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

(Optional) Please describe further alternatives for: How should Slide City visualize the city base initially?

Your answer

Which visualization represents better the sliding of floors? *

When the city is slid up, the floors move up by the selected commit sort.

Please rank the options from 1 (most important) to 3 (least important).

(Optional) If you propose another alternative, please describe it in the next question.

	1	2	3
Slide City with "Intact File history"	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Slide City with "Split file history"	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Other	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

(Optional) Please describe further alternatives for: Which visualization represents better the use case?

Your answer

Section 6: Filter Features

Should the generality level affect folders opened or closed manually by the user * or only those not touched?

The generality level feature helps the users choose a particular level or folder depth to render them as districts while the folders outside that level will be constructed as rectangular buildings. Usually, it can be cumbersome to close or open many folders one by one into a specific level of generality.

The visualization can keep track of the building that were touched by the user.

Please rank the options from 1 (most important) to 4 (least important).

(Optional) If you propose another alternative, please describe it in the next question.

	1	2	3	4
Always change the generality level for all folders and files	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Change the generality level only for those files and folder that were not touched by the user	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Change the generality level or all folders except the last one the user changed	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Other	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

(Optional) Please describe further alternatives for: Should the generality level affect folders opened or closed manually by the user or only those not touched?

Your answer

How should the visualization be filtered by the author? *

Please rank the options from 1 (most important) to 4 (least important).

(Optional) If you propose another alternative, please describe it in the next question.

	1	2	3	4
Show the whole city but color only the commits related to the author.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Show only the buildings related to the author.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Show only the floor buildings related to the author	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Other	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

(Optional) Please describe further alternatives for: How should the visualization be filtered by the author?

How useful is it to exclude files? *

Files like package-lock.json or other generated ones may often have thousands of lines of changes in one commit. This can bring a lot of noise to the visualization.

Please rank the options from 1 (most important) to 4 (least important).

(Optional) If you propose another alternative, please describe it in the next question.

	1	2	3	4
Exclude files before construction (improves visualization construction performance)	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Exclude after construction	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Do not exclude files	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Other	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

(Optional) Please describe further alternatives for: How useful is it to exclude files?

Your answer

How useful is to blocklist commits? *

Some commits are known to have generated code that the visualization can ignore.

Your answer

How should the blocklisted commits be excluded? *

Please note that there are three visualization attributes that the lines from a particular commit influence the city: (1) The size of the floor (that represents the commit), (2) the ownership calculation of other floors, (3) the surface of the building (the represents the current file/folder size).

- ☐ Don't show the floor representing the commit
- ☐ Don't include its lines in any floor's ownership calculation
- ☐ Exclude the lines from blocklisted commits from the buildings surface
- ☐ Other: _____

Which features are more important? *

Please rank the options from 1 (most important) to 4 (least important).

	1	2	3	4
Exclude commits	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Exclude files	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Change generality level for all folders	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Show ownership and commiter	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

FLOSS
LD-City
MVO
OC
QSMP
RO
SO
TAR
vp-s
XP
LoC

Ree/libre and open-source software
Linked Data City
Most valued owner
Origin City
Qlik Sense Metrics Portal
relational ownership
subjective ownership
technical action research
variation points
Extreme Programming
Lines of Code

Acronyms