Sensorial software evolution comprehension

Master's Thesis submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics
Software Development

presented by
Gianlorenzo Occhipinti

under the supervision of
Prof. Michele Lanza
co-supervised by
Prof. Roberto Minelli, Prof. Csaba Nagy

July 2022

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Gianlorenzo Occhipinti
Lugano, Yesterday July 2022

To my beloved

Someone said . . .

Someone

Abstract

The comprehension of software evolution is essential for the understandability and maintainability of systems. However, the sheer quantity and complexity of the information generated during systems development make the comprehension process challenging. We present an approach, based on the concept of synesthesia (the production of a sense impression relating to one sense by stimulation of another sense), which represents the evolutionary process through an interactive visual depiction of the evolving software artifacts complemented by an auditive portrayal of the evolution. The approach is exemplified in SYN, a web application, which enables sensorial software evolution comprehension. We applied SYN on real-life systems and presented several insights and reflections.

Acknowledgements

ACK

Contents

Figures

Tables

Chapter 1

Introduction

In 1971 Dijkstra, made an analogy between computer programming and art. It stated that is not important to learn how to compose software, but instead, it is important to develop its own style and what will be their implications. During the lifetime of a software system, many developers work on it, each one with its own style. This is one of the multiple reasons behind software complexity. Today's software systems are characterized by sheer size and complexity. Software maintenance takes up the most part of a software system's cost. It is hard to quantify the impact of software maintenance on the global cost of the software. However, Researchers, in 1995-96, estimated it to be between 50% and 75% [12] [47], and more recent studies have seen it reaching the peak of 90%. [17] [46]. There are many factors that influence the cost of maintenance Among these, the understanding activity, needed to perform maintenance tasks, takes up to 50-60% of the maintenance time [9].

1.1 Software comprehension

The comprehension of software evolution is essential for the understandability and maintainability of systems. However, the sheer quantity and complexity of the information generated during systems development make the comprehension process challenging.

Lehman and Belady in 1985, were the first to observe that maintaining a software system becomes a more complex activity over time. [33] The term "software evolution" was used for the first time by them in their set of laws. One of the goals of software evolution analysis is to identify potential defects in the system's logic or architecture. Software evolution analysis is often supported by software evolution visualization. Software evolution visualization is software visualization applied to evolutionary information.

1.2 Visualization

Software visualization is a specialization of information visualization with a focus on software [32] Software Visualization techniques are used to support complex software systems analysis. This analysis usually produces a sheer quantity of multivariate data. To support their analysis, several tools have been proposed in the literature [37]. There are cases when software visualization can be used to aid the analysis activity. For example, when programmers need to

comprehend the architecture of a system [41], when researchers analyze version control repositories [20], or to support developers' activity [34].

According to Butler et. al. [6] there are three categories of visualization:

- Descriptive visualization. Widely used for education purposes, the visualization is used to present data to other people.
- Explorative visualziation. Used to discover the nature of the data being analyzed. With this visualization, the user usually does not know what he/she is looking for.
- Analytical visualization. Adapted when we need to find something known in the available data.

All the software visualization approaches vary with respect to two dimensions: the level of abstraction and the visualized data. According to the type of the data, we can classify visualization as:

- Evolutionary visualizations. Used to present information extracted from the history of a system. Mainly used to find the cause of problems in software.
- Static visualizations. Used to present information extracted with static analysis of the software. It provides information about the structure of the system.
- Dynamic visualizations. Used to present information extracted with dynamic instrumentation of the software execution. It provides information about the behavior of the system.

Moreover, the level of abstraction can be classified as:

- Code-level visualization. Used to visualize fine granted sourcecode information, such as the lines of code.
- Design-level visualization. Used to visualize self-contained pieces of code, such as classes in object-oriented systems.
- Architectural-level visualization. Used to visualize the system architecture and the relationships among its components.

In this work, our focus is an explorative visualization, that depicts the evolution of a system. To do that, we created an approach with an evolutionary design-level visualization aimed to facilitate the comprehension of the system's history.

1.3 Sonification

Numerous techniques have been presented in literature that aim to facilitate program comprehension. The main challenge that each visualization technique has is to identify the relevant aspects to be depicted and present them in an effective way. The effectiveness of a software visualization techique could be enhanced by combining it with audio. The term "program auralization" was coined for this reason, it aims to communicate information about program in an auditority way. Several studies was done to measure the advantages given by audio as a communication medium [3].

In our approach, we aim to ...

1.4 Our approach

We present an approach, based on the concept of synesthesia (the production of a sense impression relating to one sense by stimulation of another sense), which represents the evolutionary process through an interactive visual depiction of the evolving software artifacts complemented by an auditive portrayal of the evolution.

1.5 Contribution

We can summarize the main contribution of this work as:

- We proposed an approach to model the history of a git repository.
- We proposed an approach, based on the concept of synesthesia, which represents the evolutionary process through an interactive visual depiction of the evolving software artifacts complemented by an auditive portrayal of the evolution.
- We engineered a tool, SYN, which supports our approach as an interactive web application
- We applied SYN to real-life systems and presented several insights and reflections.

1.6 Structure of the document

This document is organized as follows

- Chapter 1
- Chapter 2
- Chapter 3
- Chapter 4
- Chapter 5

Chapter 2

State of the art

2.1 Software visualization

Software maintenance and evolution are essential parts of the software development lifecycle. Both require that developers deeply understand their system. Mayrhauser and Vans defined *program comprehension* as a process that "*knowledge to acquire new knowledge*" [54]. Generally, programmers possess two types of knowledge: general knowledge and software-specific knowledge. Software comprehension aims to increase this specific knowledge of the system, and it can leverage some software visualization techniques for this purpose. Software visualization supports the understanding of software systems by visually presenting various information about them, e.g., their architecture, source code, or behavior. Stasko et al.[18] conducted a study in 1998 that shows how visualization arguments human memory since it works as external cognitive aid and thus, improves thinking and analysis capabilities.

The earliest software visualization techniques in the literature used 2D diagrams. For example, Haibt, the first to use them in 1959, provided a graphical outline of a program and its behavior with flowcharts [21]. As shown in Figure 2.1, they were 2D diagrams that described the execution of a program. He wrapped each statement in a box, representing the control flow with arrows.

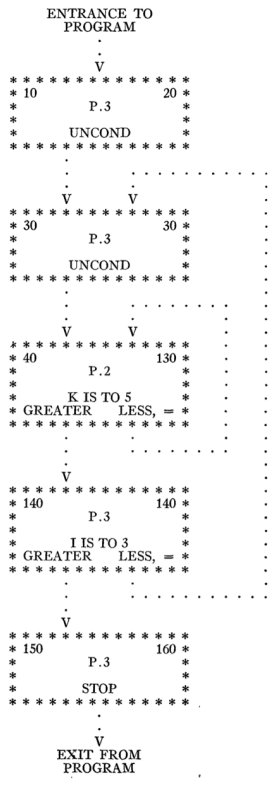

Figure 2.1. Flowchart presented by Haibt in 1959

Ten years later, Knuth also confirmed the effectiveness of flowcharts [29]. He evidenced that programs around that time were affected by a lack of readability. Therefore, he introduced a tool to generate visualizations from the software documentation automatically.

Nassi and Schneiderman[40], in 1973, introduced the Nassi–Shneiderman diagram (NSD), able to represent the structure of a program. The diagram was divided into multiple sub-block, each with a given semantic based on its shape and position.

The 80s registered two main directions of software visualization. The first was the source code presentation. For example, Hueras and Ledgard [22] then Waters [55] developed techniques to format the source code with a prettyprinter. The second direction was the program behavior, used mainly for educational purposes.

One of that period's most prominent visualization systems was Balsa-II [5]. Balsa-II was a visualization system that, through animations, displayed the execution of an algorithm. Programmers were able to customize the view and the control execution of the algorithm, to understand them with a modest amount of effort. The program was domain-independent, and learners could use it with any algorithm.

Around the end of the 80s, Müller et al. [39] released Rigi, a tool used to visualize large programs. It exploited the graph model, augmented with abstraction mechanisms, to represent systems components and relationships.

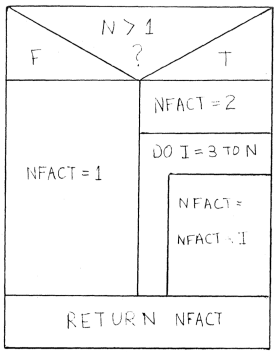

Figure 2.2. NSD of the factorial function.

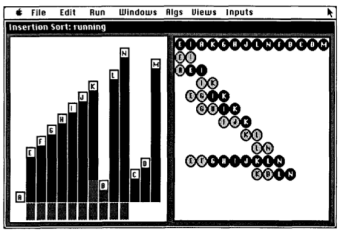

Figure 2.3. Balsa-II

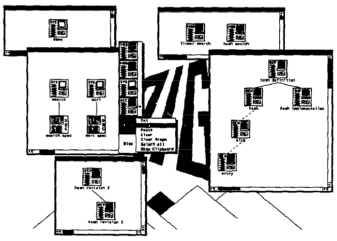

Figure 2.4. Rigi

The 1990s recorded more interest in the field of software visualization. In 1992, Erik et al. introduced a new technique to visualize line-oriented statistics [15]. It was embodied in Seesoft, a software visualization system to analyze and visualize up to 50,000 lines of code simultaneously. On their visualization, each line was mapped to a thin row. Each row was associated with a color that described a statistic of interest.

One year later, De Pauw et al. [13] introduced Jinsight, a tool able to provide animated views of object-oriented systems' behavior.

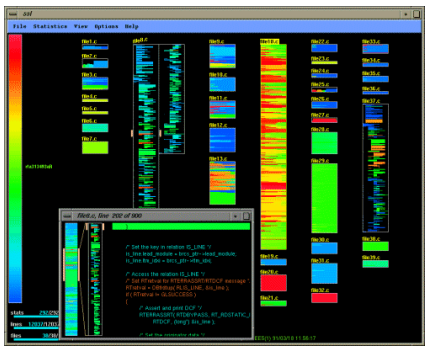

Figure 2.5. Seesoft

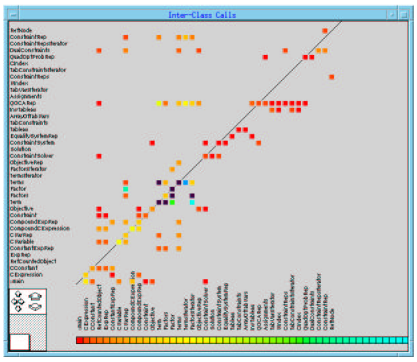

Figure 2.6. Jinsight

That period was favorable also for experimenting with novel research directions for visualization, such as 3D visualization and Virtual Reality.

In 1998, Chuah and Erick [7] proposed three different techniques to visualize project data. They exploited the concept of glyphs, a graphical object that represents data through visual parameters. The first technique was the Timewhell glyph, used to visualize time-oriented information (number of lines of code, number of errors, number of added lines). The second technique was the 3D wheel glyph; it encoded the same attributes of the time wheel, and additionally, it used the height to encode time. Infobug glyph was the last technique, where each glyph was composed of four parts, each representing essential data of the system, such as time, code size, and the number of added, deleted, or modified code lines.

Figure 2.7. Timewhell

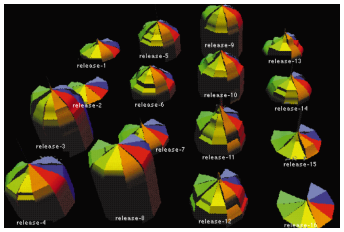

Figure 2.8. 3D wheel

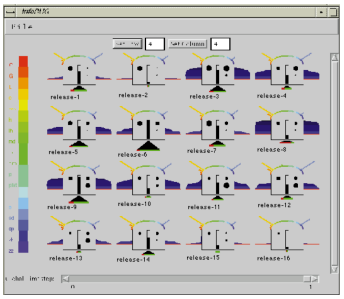

Figure 2.9. Infobug

Also in 1998, Young and Munro [57] explored representations of software for program comprehension in VR.

Finally, in 1999, Jacobson et al. [23] introduced what we now know as de facto the standard language to visualize the design of a system: UML.

At the beginning of the 21st century, thanks to the spread of version control systems and the open-source movement, visualizing a software system's evolution became a more feasible activity thanks to publicly accessible system information. As a result, many researchers focused their work on software evolution visualization.

Lanza [31] introduced the concept of the Evolution Matrix. It was a way to visualize the evolution of software without dealing with a large amount of complex data. Furthermore, this approach was agnostic to any particular programming language. The Evolution Matrix aimed to display the evolution of classes in object-oriented software systems. Each column represented a version of the software; each row represented a different version of the same class. Cells were filled with boxes whose size depended on evolutionary measurements. The shape of the matrix could also be used to infer various evolutionary patterns.

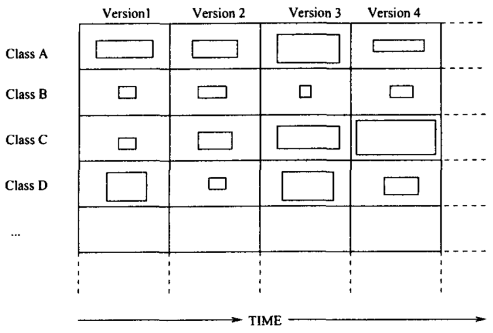

Figure 2.10. A schematic display of the Evolution Matrix

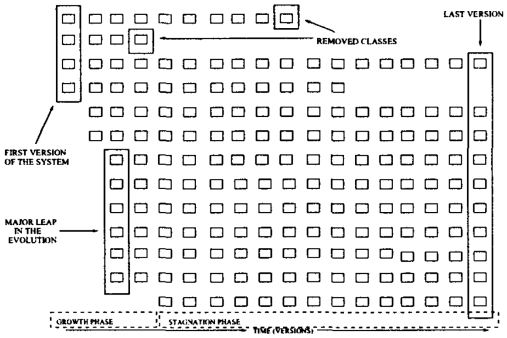

Figure 2.11. Some characteristics of the Evolution Matrix

Taylor and Munro [51], demonstrated that it was possible to use the data contained in a version control repository to visualize the evolution of a system. They developed Revision Tower, a tool that showed change information at the file level. Pinzger et al. [42] visualized the evolution of a software system through Kivat diagrams. RelVis, their tool, was able to depict a multivariate visualization of the evolution of a system.

During the same year, Ratzinger et al. presented EvoLens [43], a visualization approach and tool to explore evolutionary data through structural and temporal views.

Langelier et al. [30] investigated the interpretation of a city metaphor [28] to add a new level of knowledge to the visual analysis.

D'Ambros and Lanza [10] introduced the concept of Discrete-Time Figure concept. It was a visualization technique that embedded historical and structural data in a simple figure. Their approach depicted relationships between the histories of a system and bugs. They also presented the Evolution Radar [11], a novel approach to visualize module-level and file-level logical coupling information.

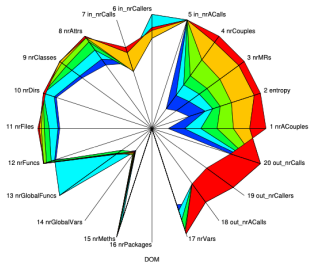

Figure 2.12. RelVis

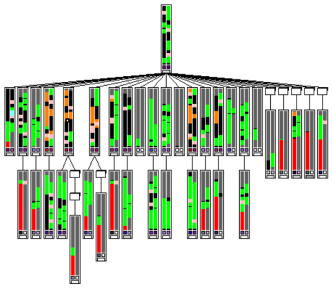

Figure 2.13. Tree of Discrete Time Figures

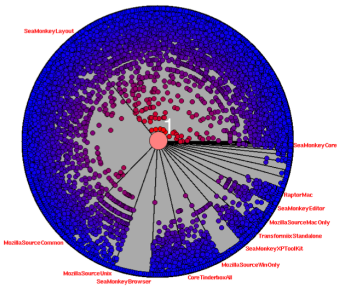

Figure 2.14. Evolution Radar

Steinbrückner and Lewerentz [50] described a three-staged visualization approach to visualize large software systems. Thir visualization was supported by a tool called Evo-Streets. Each stage of their approach was responsible for representing a different aspect of the system with the city metaphor.

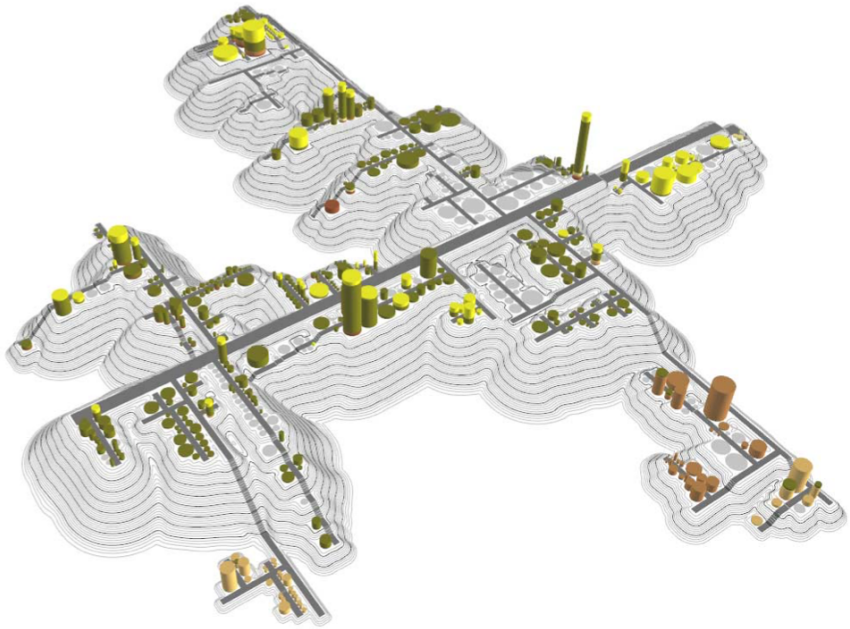

Figure 2.15. Evo-Streets

Wettel revised the city metaphor to represent metrics meaningfully [56]. In his thesis, he represented packages as districts and classes as buildings. The metaphor was used for various purposes, e.g., reverse engineering, program comprehension, software evolution, or software quality analysis. He claimed that the city metaphor brought visual and layout limitations; for example, not all visualization techniques fit well. Under those circumstances, he preferred simplicity over the accuracy, so he obtained a simple visual language that facilitated data comprehension. His approach was implemented as a software visualization tool called CodeCity.

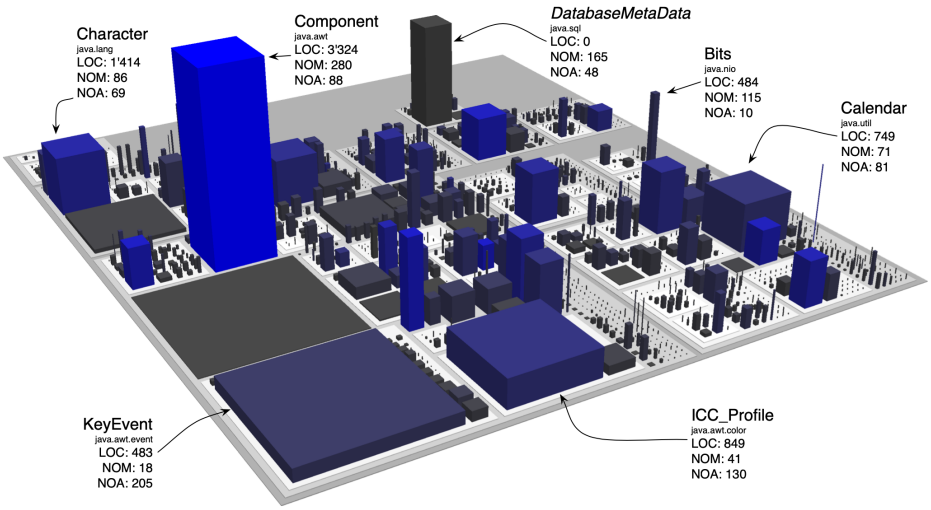

Figure 2.16. CodeCity

Ens et al. [16] applied visual analytics methods to software repositories. His approach helped users comprehend co-evolution information by visualizing how source and test files were developed together.

Kapec et al. [26] proposed a graph analysis approach with augmented reality. They made a prototype of a tool that provided a graph-based visualization of software, and then they studied some interaction methods to control it with augmented reality.

Schneider et al. [45] presented a tool, CuboidMatrix, that employed a space-time cube metaphor to visualize a software system. A space-time cube is a well-known 3D representation of an evolving dynamic graph.

Merino et al. [38] aimed to augment software visualization with gamification. They introduced CityVR, a tool that displays a software system through the city metaphor with a 3D environment. Working with virtual reality, they scaled the city visualization to the physically available space in the room. Therefore, developers needed to walk to navigate the system.

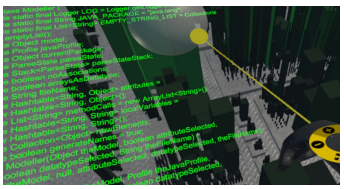

Figure 2.17. CityVR

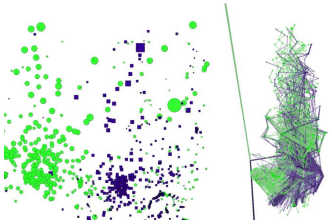

Figure 2.18. ChronoTwigger

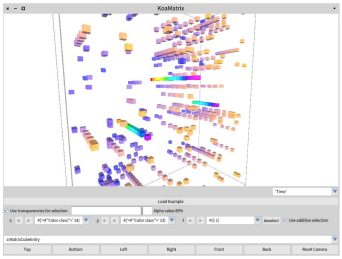

Figure 2.19. CuboidMatrix

Khaloo et al. [27] revised the idea of gamification with a 3D park-like environment. They

mapped each class in the codebase with a facility. The wall structure depended on the class's constituent parts, e.g., methods and signatures.

Finally, we mention Alexandru et al., who proposed a method to visualize software structure and evolution, with reduced accuracy and a fine-grained highlighting of changes in individual components [2].

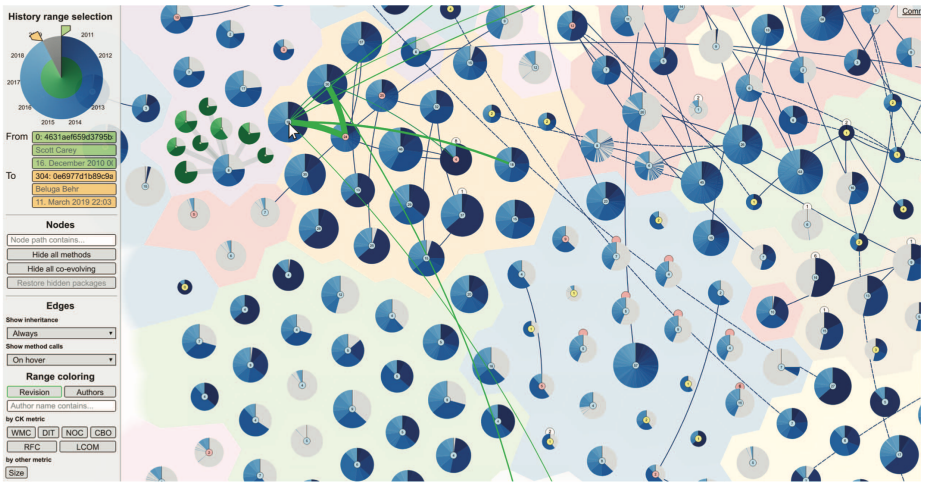

Figure 2.20. Evo-Clock

2.2 Analysis of software evolution

Software repositories contain historical data about the evolution of a software system. Thanks to the spread of the git protocol, and consequently of GitHub, Mining Software Repositories (MSR) has become a popular research field.

D'Ambros et al. in [1] presented several analysis and visualization techniques to understand software evolution. They developed an approach based on a Release History Database (RHDB). It is a database that stores historical information about source code and bugs. The strength of RHDB was the association between historical versions of flies and bugs. Having this information stored on a database, they were able to run some evolution analysis to obtain information such as how many developers worked on a file to fix a bug or how the effort was to fix it.

Finally, they concluded by evidencing two main challenges in MSR:

- Technical challenge: repositories contain a sheer amount of data, posing scalability problems.
- Conceptual challenge: how to do something meaningful with the collected data. Most of the approaches present in literature to visualize software evolution have unanswered questions about the effectiveness of the comprehension.

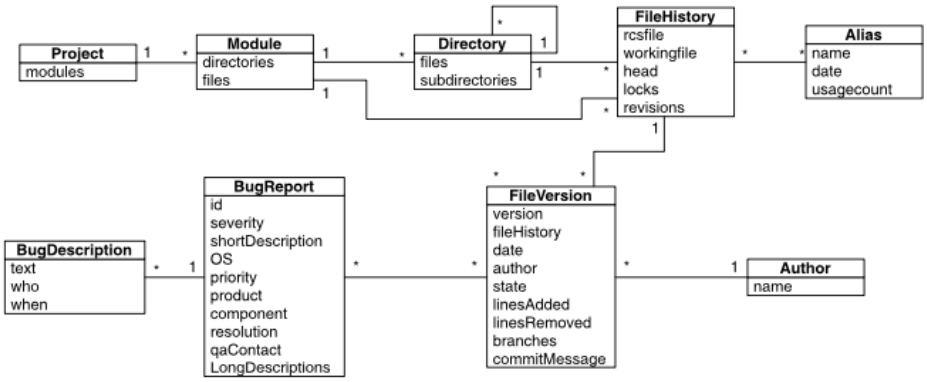

Figure 2.21. RHDB

In 2022, the number of GitHub repositories lays around 200 million. Even if it seems a promising data source, Kalliamvakou et al. raised some issues with its mining. [25] For example, they evidenced that a repository does not always match a project. A reason for this can be found in the fact that most repositories had had very few commits before becoming inactive. Over 70 percent of the GitHub projects were personal when they did their research, and some weren't used for software development. Finally, the last perils they raised were related to GitHub features that software developers do not properly use. They considered only projects with a good balance between the number of commits, the number of pull requests, and the number of contributors to find actively developed repositories.

Spadini, Aniche and Bacchelli [49]. They developed a Python framework called PyDriller, enabling users to mine software repositories. Their tool can be used to extract information about the evolution of a software system from any git repository.

We also mention the work done by Salis and Spinellis [44]. They introduced RepoFS, a tool that allows navigating a git repository as a file system. Their approach sees commits, branches, and tags as a separate directory tree. Figure 2.22 shows an example of a repository data structure.

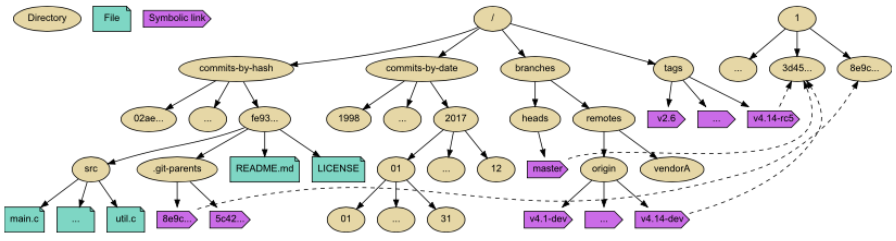

Figure 2.22. RepoFS

Clem and Thomson [8], members of the semantic code team at GitHub, built a static analyzer of repositories to implement symbolic code navigation. That feature was released on

GitHub some years ago and let developers click on a name identifier to navigate to the definition of that entity. They were looking for a solution that would not bring them scalability problems. Moreover, they built the symbolic navigation feature around some ideas like:

- Zero configuration needed by the owner of a repository
- Incrementality of the process. There was no need to process the entire repository for every commit made by a developer. Instead, they analyzed only the files that had changed.
- Language agnosticism of the static analysis.

Working on that feature, they recognized the difficulty of scaling a static analysis like that regarding human behavior. Nevertheless, their idea was to have an agnostic static analyzer, but they could not reach this goal, and they were forced to implement it for just nine programming languages.

2.3 Data sonification

External auditory representations of programs (known as "program auralisation") is a research field that is getting even more interest in the recent years.

Sonnenwald et al. made one of the first attempts. [48] They tried to enhance the comprehension of complex applications by playing music and special sound effects. This approach was supported by a tool called InfoSound It was mainly adopted to understand the program's behavior.

Many other researchers followed this first technique. To cite some of them, DiGiano and Baecker [14] made LogoMedia, a tool to associate non-speech audio with program events while the code is being developed. Jameson [24]] developed Sonnet, audio-enhanced monitoring and debugging tool. Alty and Vickers [53] had a similar idea. Using a structured musical framework, they could map the execution behavior of a program to locate and diagnose software errors.

Despite the usefulness of these tools, they adopted an essential kind of mapping, and thus they had a limited musical representation. Vickerts [52] evidenced the necessity of a multithreaded environment to enhance the comprehension given by the musical representation. He proposed adopting an orchestral model of families of timbres to enable programmers to distinguish between different activities of different threads.

The size and the complexity of systems can represent a problem for the effectiveness of a visual representation of a software system. Having a large number of visual information, observers might find it difficult to focus only on the relevant aspects. Boccuzzo and Gall [4] supported software visualization with sonification. They used audio melodies to improve navigation and comprehension of their tool, called CocoViz. Their ambient audio software exploration approach exploited audio to describe the position of an entity in the space intuitively. Thanks to the adoption of surround sound techniques, the observers perceived the origin of an audio source so it could adjust their navigation in the visualization. Each kind of entity played a different sound based on mapping criteria.

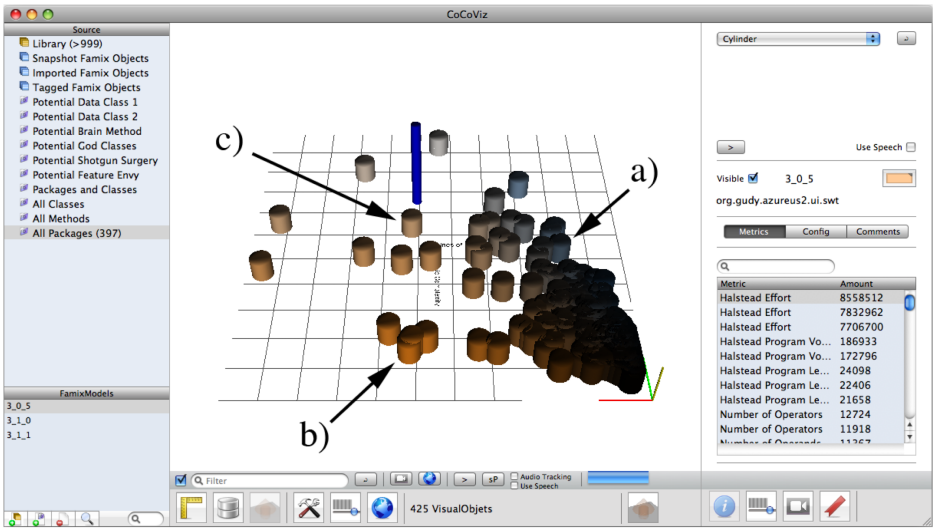

Figure 2.23. CocoViz

McIntosh et al. [36] explored the use of a parameter-based sonification to produce a musical interpretation of the evolution of a software system. Their technique mapped musical rests to an inactive period of development and consonance and dissonance to interesting phenomena (like co-changing of components).

Finally, Mancino and Scanniello [35] presented an approach to transforming source code metrics into a musical score that can be both visualized and played.

2.4 Conclusion

We have seen many different techniques and tools focused on visualizing the source code of software systems, their evolution, or some metrics. Our approach won't consider the source code visualization. Instead, it is focused on the evolution of a software system and how some metrics run over its history. Moreover, in contrast to what some tools did, we are not focused on the evolution code bugs.

The codebase of a system is composed of a group of files. In our approach, each file represents a system entity that mutates over time. It is not based on a previously identified metaphor, such as CodeCity or CityVR with the city metaphor. We created a new layout where the position of each entity is defined by its discovery time.

At present, git has become de-facto the standard tool for version control. Having this in mind, we aim to find a suitable model to represent the histories of mined git repositories. Therefore, we created a model inspired by the EvolutionMatrix, but with some adjustments to make it work with git.

As the GitHub team did, we propose a scalable approach that works with large repositories. It differs from what they did because we are not focused on a semantic analysis of the source

code; instead, we need to extract some metrics. Moreover, in contrast to what they did, our technique is purely language-agnostic.

Finally, we augmented the effectiveness of our approach with an external auditory representation. Conversely to what they did in the first approaches, we used a multithreaded environment to play the musical notes. Whereas CocoViz used audio melodies to support the navigation of space created for the visualization, we mapped sounds to the magnitudes of changes in a given moment.

Chapter 3

Approach

In this chapter we present the approach that we developed to visualize software system using a visual and auditive depiction of the evolution of a system. To do that, we have chosen to leverage on the phenomenom of the Synesthesia, the production of a sense impression relating to one sense by stimulation of another sense. .....

Our approach consists of two parts: In the first part we modelled the evolution of the software artifacts, and we enginereed a tool that implement it. In the second part, we used the concept of synesthesia (the production of a sense impression relating to one sense by stimulation of another sense) to enhance the effectiveness of the visualization.

Comprehending the evolution of a software system is difficult, mostly for the sheer amount of data and its complexity. The term "software evolution" was conied for the first time by Lehman in 1985 in a set of laws. [33] He stated that the complexity of a system is destinated to increase overtime, as the system always needs to be adapted to its evolutionary environments. To be managed, software systems needs to be compprehended by developers, and this activity can be simplified with software visualization.

..

To visualize the evoluton of a system, there are many aspects that we can consider.

In our approach we focus on systems that resides on git repositories. We made this choiche because git is the most common repository management system and it also track all the changes made to every file of the system. So, every git repository holds the whole history of a software system.

In our approach, we model the evolution of repository files, therefore we model its history. To do that, we considered all the information that can be extracted from a git repository: files and commits.

The git protocol is responsible for tracking the changes made to the system. To do that, every time we made a commit, it stores only a list of files that have been modified. Every commit is represented by a tree of hashes, each one representing a file.

Git has the possibility to inspect every commit of a repository by using the command `git checkout`. In this way, we can navigate through the history of a repository, to track all the files,

their changes and their metrics.

A commit operation contains also other meta-information such as the author, the date and the message.

3.1 Evolution Model

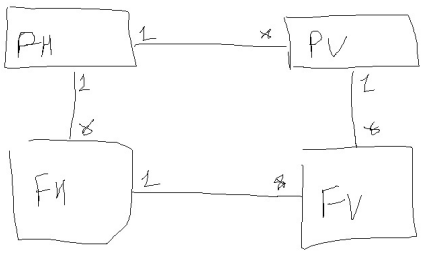

Figure 3.1. Evolutionary Model

To model the evoltion of software systems, we delveloped a new model, showed in figure 3.1, based on Hismo, a solution presented by Tudor Girba [19].

The needed to develop a new evolutionary comes from the fact that Hismo was developed to work with another versioning system: Subversion (SVN). There are several difference between SVN and git. The most important, in terms of design, is on how they keep track of changes. In fact, SVN works with the concept of snapshot. Every time a developer changes some files, committing theese changes, it will increase the revision of the whole system. In constrast, git, only the modified files would get the version field updated. Therefore, we need to adapt our model to work well with git because the Hismo model is based on three main entities:

- Snapshot. A representation of the entity whose evolution is studied.
- Version. An entity that adds the notion of time to a Snapshot by relating it to the History.
- History. An entity that holds of a set of Versions.

Since in git we don't have the concept of Snapshot, we needed to replace it with file versions. A file version is essentially the version of a file in a particular point in time. It is associated with a project version, that set the time and with a file history, that set the file that has been updated. To summarize, the main entities of our model are:

- **ProjectHistory**: it represents the repository of a system. It is an holder of two sets: FileHistories and ProjectVersions.

- **FileHistory**: it represents a file in the whole history of the repository. If a file has been renamed or moved, the entity representing that file will remain the same. So, it is resilient to renaming and moving activities. It holds a set of FileVersions, that represents the version of the entity in a particular point in time.
- **ProjectVersion**: it represents a commit or a version of the system. For each changed file inside a commit, the respective ProjectVersion contains a FileVersion representing that change. A ProjectVersion also holds contextual information about the commit such as the timestamp, the hash of the commit and the message.
- **FileVersion**: it represents the version of an entity in a particular point in time. It is responsible to hold all evolutionary information of an entity, since it represent an evolutionary step of that entity.

Historical information retrival

To build the history of a repository, we need to extract the historical information from git.
To understand better how we approached to it, we have to explain how git internally represents the repository history. Git works with the concept of branches, each branch can be seen as a different timeline of the repository. Usually, developers exploit branches to develop features on it and then merge the developed code with the existing codebase. To do that, they need to create a commit called "merge commit". Each time we create a new git commit, we are deploying a new version of the system, that records all the changes made to the commits's tracked files. Internally, in git, all the commits are stored as nodes of a tree called commit-tree. The root node represents first commit of the repository, since it has not any parents. All the other nodes instead, represent the commits made during the whole lifecycle of the repository. Each commit usually have only one parent, that represents the previous commit made before that one. There is one case where a commit might have more than one parent: merges commits.

As a convention, each repository should has a branch that should contain the stable, production-ready code. Usually this branch is named "main" or "master". In our approach, we aim to analyze the timeline of this branch. To do that, we start from the first commit present in the repository history, and then we traverse the whole commit tree. However, during this proccess, we don't consider merge commits, since they incorporate commits already made, and thus they would be considered twice. Once we have extracted all the valid commits, that resides on the main branches, we need to take from them all the representative information that we need for a project version.

Git, is able to recognize the following actions made on a file:

- **ADD**. A file is to the repository.
- **DELETE**. A file is has been removed from the repository.
- **MODIFY**. A file is has been modified.
- **RENAME**. A file name has been changed. Wheater the filepath has been changed, the parent directory path must remain the same.

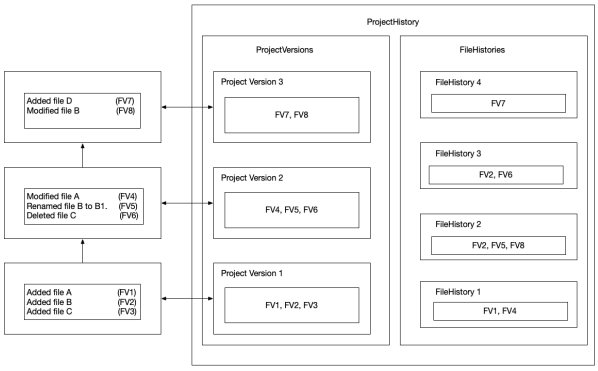

Figure 3.2. Rebuilding history example

- **MOVE**. A file was moved from one location to another, so the file path has been changed. This action is detected wheter the filename remained the same.

From a commit we could also extract additional information such as the name of the file being modified, the parent directory, the number of lines added and removed, the path of the file before and affter the changes and many more. We used the commit's information to track all the paths of an entity. In this way, we can update the entity path when it was renamed or moved to track it during all its lifecycle.

When we reconstruct the history of a repository, each FileHistory starts with a FileVersion representing an ADD action. Then, in the middle of the FileVersion set we can find only three kind of actions: MODIFY, RENAME and MOVE. In some cases an entity might be deleted, so the last FileVersion holded by a FileHistory will represent a DELETE action.

Figure 3.2 shows an exaple of how history is rebuilt. First we create a ProjectHistory that holds a set of ProjectVersions and a set of FileHistories. After that, we start to traverse the repository's commit tree. As we can notice, for each commit we create a new ProjectVersion. It represents a new version of the system in our model. Therefore, we inspect the commit's changelist and we create a new ProjectVersion for each entry of the list. With this operation, we can discover if a file has been added to the system, because on that case the change should represent an ADD operation and thus we can create a new FileHistory. At version 1, three new files were added to the repository (A, B, C) and, as we can see, three new FileHistories were created. Each chage was mapped to a FileVersion (FV) and consequently added to the respective FileHistory and to the respective ProjectVersion. We did the same thing for the ProjectVersion 2 and 3.

Partial historical representation

One of the goal that be had when we developed this approach, was the possibility to analyze large repository in an acceptable amount of time. In other words, our approach needs to be scalable. GitHub host the code of some notorious open source sytems, such as LibraOffice, Elastisearch and Linux. All of them have more than 500,000 commints in each and thus, we cannot aim to reconstruct their histories with a singe analyzer, it will take too much time. To

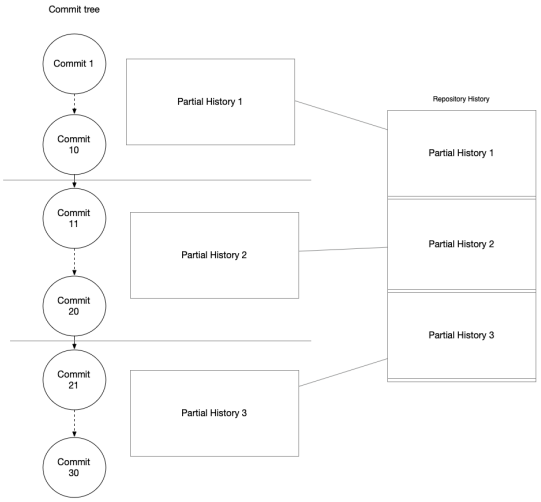

Figure 3.3. Partial history example

prove that, just consider the worse case: Linux. When this thesis was redacted, the repository of Linux has 1,090,563 commits. To move from one commit to another, we can assume that git needs one seconds. As a result, just to navigate through the whole history of Linux, we would need 11 days. Moreover, in this simple estimation we are omitting the huge amount of time that the analyzer needs to extract metrics form every sigle file on each version.

We present a scalable approach, based on the concept of a "partial history". A partial history is an entity that holds information about a specific range of time of the ProjectHistory. So, it can be seen as a subset of a ProjetHistory. We can split the whole repository's history into multiple chunks, each one represented by a partial history, and then when all the analysis are completed, we can merge them to reconstruct the whole story of the repository.

Figure 3.3 shows an example of PartialHistory representation. We split the commit-tree into multiple chunks, and then we can run an analysis on each chunck. This analysis can be done parallelaly since these chunks are independent from each other. At the end, the final history will be represented by a merge of all the PartialHistories.

Nonetheless we can build PartialHistories in parallel, we cannot do the same for the final History. In fact, the final merge needs to be done sequentially. The sequence needs to follow the order of the commit tree. In figure 3.3, for example PartialHistory1 represent the history from commit 1 to commit 10, PartialHistory 2 represents the history from commit 11 to commit 20, and finally PartialHistory 3 represents the history from commit 21 to commit 30. Therefore, the commit order is respected if we merge them in this order: 1, 2, 3.
Obviously, the result of a single analysis and the result of a parallel analysis must be identical. To ensure that, we need to pay attention to the merge operations of our analysis. When we merge the history of a repository with a partialhistory, we need to preserve the characteristics of our model. In particual, if FileHistory is already present in our history, we don't have to duplicate it, but instead, we need to update it.

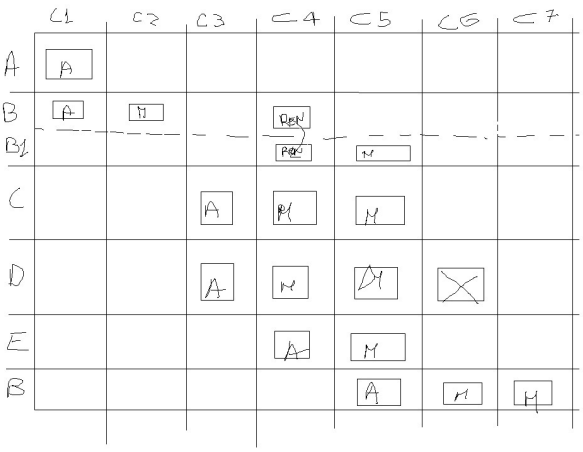

Figure 3.4. Evolution matrix of a repository

3.2 Visualization

2D representation

This model, could be also representd by a 2D matrix.

From our point of view, a file can be seen as an holder of file versions. A file version represents a file in a particular point of time. The difference between a file version a

As a consequence, our new model, depicts every File in the system, as an holder of a set of FileVersions. The File entity, named to be consistent with its scope as FileHistory, is stored on another entity called ProjectHistory, that follows the same logic. In fact a ProjectHistory is no less than a holder of two sets: one for the FileHistories and one for the Versions.

- Each column of the matrix represent a commit, a version of the software.
- Each row of the matrix represent a file, named FileHistory.
- Each cell of the matrix represent the different versions of a file. Empty cells represent a file that has not been modified.
- A file can have multiple names (tollerant to renaming and moving operations).
- Files are sorted by addition time, on the top we will have all the files that were added in the first commit of the repository.

For now on, we will use the following notation:

Therefore, a FileHisory represents the history of an entity of our system. The name of the file is not a concern for us, until a file is not deleted it will always represent the same entity.

And we will use them in our model to identify the action associated to a FileVersion. Figure 3.4 present a schematic evolution matrix of a repository with seven versions. As we can see, in the first ProjectVersion, there were added two files, A and B. In the second revision B was modified, and in the third revision C and D were added. The fourth revision recorded a rename

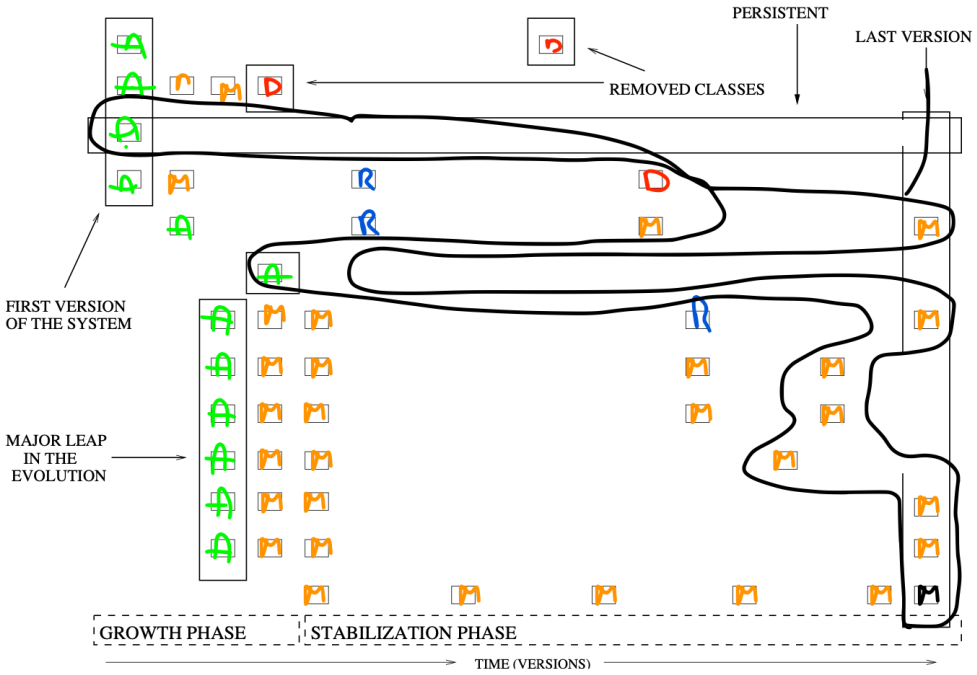

Figure 3.5. Evolution matrix of a repository

of B to B1. It's important to notice that B and B1 represent the same entity, therefore they are represented by the same FileHistory.

Based on our aim, we can read this matrix as follows:

- **by rows**, if we are intrested on the history of a particular entity of our system. For examle, the FileHistory represented by the first third row in figure 3.4, represents the history of the file D. The file D was added in the third ProjectVersion (so the third commit), modified in the fourth and fifth ProjectVersion, and then deleted in the sixth ProjectVersion. The figure 3.4 is also a good exaple to understand why we cannot rely on the name of the file to identify the entity. We can notice that the file B represented by the second FileHistory, was added on the first version and then renamed on the fourth from B to B1. Then, in the fifth version, a new file called B was added. Nonetheless the name of the files are the same, they must represent two different entity. We would have had the same result, even if the file B was been added in the version four.

- **by columns**, if we are intrested on which entities were updated on each ProjectVersion. For example, on the first ProjectVersion we have added the first and the second entity. On the fourth ProjectVersion we have renamed the second entity, we have modified both the third and the fourth, and finally we have added the fifth entity.

Figure 3.5 shows an exaple of how to recover evolution information from the matrix. As we have seen, each version doesn not represent a snapshot of the system. Instead it represents only the differentce in term of changes made to the previous version. Given that, to recover a

snapshot of a specific version, we need to consider the last changes made before that specific version. Under those circumstances, for each FileHistory, we need to go back in time until we find the leftmost change. Of course, if the leftmost change was a delete, we have to ignore the releated FileHistory. In contrast, if we have to display the evolution of a snapshot, we need to consider only the changes made after that snapshot. So, each time we need to display a ProjectVersion, we have to take all its FileVersions and merge them with the current state of the snapshot.

3D representation

Software systems are hard to understand due to the complexity and the sheer size of the data to be analyzed. Our approach aims to make the analysis of a system easier for engineers, through the exploitation of the human senses. This is the reason why we have chosen to leverage on the phenomenom of the Synesthesia. The phenomenon of the synesthesia occurs when a stimulation of a sense or a cognitive pathway leads to the involountarly stimuation of another sense or a cognitive pathway. We experience synesthesia when two or more things are perceived as the same. For exaple, synesthetic people might associate the red color with the letter D or the green color with the letter A. There are many forms of synesthesia, each one representing a different type of perception, such as visual forms, auditory, tactile, etc.

In our approach we use the following visual aspects to trigger involountarly associations:

- **Color**: we use the color of the entity to describe the last action made on that entity.
- **Shape**: we use the shape of the entity to describe the type of the entity. For exaple, a java file could be represented by a cube whereas a binary file could be represented by a sphere.
- **Height**: we use the height of the entity to describe the value of a representative metric, used to compare entities.

Entities are displayed with an outward spiral layout to emphatize their order on the evolution matrix.

There are several ways to traverse the history of a repository. The visualization needs to start from the first moment and then go forward until the end. The question is, how we should go forward in time? We came up with two strategies:

- We can display n version at time, so we are traversing the history as it was written. A limitation of this approach is that we lose the concept of time. We cannot have any idea about how much time was passed between two commit, thus we cannot distinguish active development phases from unactive development phases.
- We can group version by their timestamp. So, all the commit made in the sime time period, will be displayed at the same time. This strategy works very well if we need to comphrend how the system evolved and at which speed in time.

We conchretized our strategies within the concept of **moment**. A moment is a group of version that will be displayed at the same time.

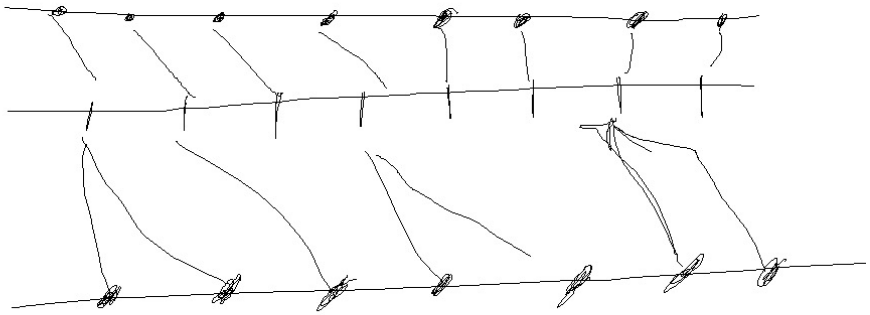

Figure 3.6. Example of two different strategies to identity moments. On the first strategy, we mapped one commit with one moment, so the total number of moments will be equals to the total number of commit. Alternatively, on the second strategy, we created a moment every day. As a result, we have some moments with many commits, and some without anyone. With this strategy, the number of moments will be the same as the number of days that have passed between the first and the last commit.

Figure 3.6 highlights the difference between the two strategies mentioned above.

Color

The color of the entity should recall the last action made on that entity. To achieve this purpose, we used the color association described in figure 3.7. Nonetheless, each person has its own perception of colors, thus, we can not assume that this color will work in the same way for all the pepoles. To remedy this issue, users can customize the color palette as they wish.

We decided to put another information on the color of the entity: the **aging**. We define the aging of an entity as the number of moments since the last modification of that entity happened. To do that, we mapped the age of an entity with the darkness of its color. As a result, older entities will be displayed with a darker color. In this way, users can immediately recognize the last action and the amount of time passed since the entity was modified.

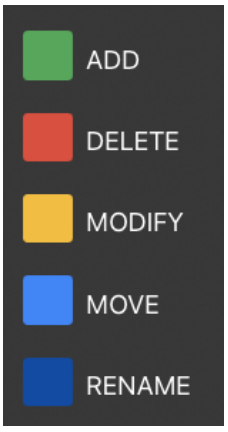

Figure 3.7. Color association

Shape

We have chosen to play with the shape of the entity to distinguish
them based on their type. Our visualization layour works with an
ourward spiral, that always adds a new entity at the tail. If a commit involves lots of entities, it would be useful to know immediately, which kind of entities are been modified. For this reason, we mapped the shape of the entity to the file type of the entity itself. Usually, in a repository, there are many file types, so we cannot aim to have a exhaustive set of shapes, to represent each file type individually. Therefore, we have defined a taxonomy

Height

MAPPERS

The height of an entity should represent the value of a metric.

3.3 Auditive model

Chapter 4

Implementation

This chapter details sensorial SYN, a tool that implements the software evolution comprehension approach defined in chaper X.

4.1 Platform overview

SYN is a platform tools that allows developers to have a visual and auditive deption of an evolving system. This section aims to describe the tools and modules that are part of SYN.

4.1.1 SYN CLI

SYN CLI is a command line interface that allows developers to interact with SYN. It gives to developers full control over the system. For example, with the command `syn project create` it is possible add a project and then, analyze it with the command `syn analyze`.

* LIST OF AVAIL COMMANDS - APPENDIX? *

4.1.2 SYN Analyzer

SYN works with evolutionary metrics that represents the history of a system. To this aim, we developed SYN Analyzer, a Java tool on top of jGit. Having a languange-agnostic implementation, it can analyze every git repository written in any programming language.

Four steps compose the analysis process:

1. The repository is cloned.
2. The source code of the HEAD revision is obtained.
3. All the files are analyzed and the metrics are obtained.
4. If the revision has a parent, the step 2 is repeated with the parent revision.

As a result, starting from the HEAD revition it will go back in time until the first revision. All the collected metric will be stored in a object, called analysis result, that can be serialized in a JavaScript Object Notation (JSON) object. We chose JSON because is a lightweight, easy to use and human readable format. It is capable to analyze large repositories, as it uses a join algorithm to merge these analysis results if they are computed in parallel.

4.1.3 SYN Server

SYN Server is responsable for providing the elaborated information, given by the analysis results, in an intermediate languange between the front-end (SYN Debugger) and the back-end. We chose to spin up a GraphQL web server, that uses JSON as exchange language between the front-end and the back-end. In this way, the front-end can ask exactly for the information it needs, and the back-end can send it back once they are computed.

The computation made by the back-end, is responsable to create the view that will be shown in the front-end. To do that, the server process a *view specification, that must be given by the front-end, and then provides to the front-end JSON objects representing only the object that must be depicted. Although the information provided by the server are limitated to the view itself, it also provides debugging information if requested.*

4.1.4 SYN Debugger

SYN Debugger is a web application that allows developers to interact with SYN. It is written with React.js, a popular JavaScript framework. The aim of this application is to have a visual depition of the view generated by the server, plus some additional information. For example, it allows you to ckick on a entity and see the information that is related to it. The visualization is based on Babylon.js, a popular 3D library. SYN Debugger provides different kind of customizations to the view, such as the shape and the colors of the entities. All theese customizations are sent to the back-end server, through a view specification file.

The main purpose of this application is to debug the view and explore all the possible visualization combinations of a system.

4.1.5 SYN Sonic

....

Bibliography

[1] Software evolution. *Springer, Berlin, 2008. ISBN 9783540764397.*

[2] *Carol V. Alexandru, Sebastian Proksch, Pooyan Behnamghader, and Harald C. Gall. Evoclocks: Software evolution at a glance. In* 2019 Working Conference on Software Visualization (VISSOFT)*, pages 12–22, 2019. doi: 10.1109/VISSOFT.2019.00010.*

[3] *James L Alty. Computer-human communication? In* People and Computers X: Proceedings of the HCI'95 Conference*, volume 10, page 409. Cambridge University Press, 1995.*

[4] *Sandro Boccuzzo and Harald C. Gall. Cocoviz with ambient audio software exploration. In* 2009 IEEE 31st International Conference on Software Engineering*, pages 571–574, 2009. doi: 10.1109/ICSE.2009.5070558.*

[5] *Marc H. Brown. Exploring algorithms using balsa-ii.* Computer*, 21(5):14–36, may 1988. ISSN 0018-9162. doi: 10.1109/2.56. URL* `https://doi.org/10.1109/2.56`*.*

[6] *David M. Butler, James C. Almond, R. Daniel Bergeron, Ken W. Brodlie, and Robert B. Haber. Visualization reference models. In* Proceedings of the 4th Conference on Visualization '93*, VIS '93, page 337–342, USA, 1993. IEEE Computer Society. ISBN 0818639407.*

[7] *M.C. Chuah and S.G. Eick. Information rich glyphs for software management data.* IEEE Computer Graphics and Applications*, 18(4):24–29, 1998. doi: 10.1109/38.689658.*

[8] *Timothy Clem and Patrick Thomson. Static analysis at github: An experience report.* Queue*, 19(4):42–67, aug 2021. ISSN 1542-7730. doi: 10.1145/3487019.3487022. URL* `https://doi.org/10.1145/3487019.3487022`*.*

[9] *T. A. Corbi. Program understanding: Challenge for the 1990s.* IBM Systems Journal*, 28(2):294–306, 1989. doi: 10.1147/sj.282.0294.*

[10] *M. D'Ambros and M. Lanza. Software bugs and evolution: a visual approach to uncover their relationship. In* Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 10 pp.–238, 2006. doi: 10.1109/CSMR.2006.51.*

[11] *Marco D'Ambros, Michele Lanza, and Mircea Lungu. The evolution radar: Visualizing integrated logical coupling information. In* Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, page 26–32, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933972. doi: 10.1145/1137983.1137992. URL* `https://doi.org/10.1145/1137983.1137992`*.*

[12] *Alan M. Davis.* 201 Principles of Software Development. *McGraw-Hill, Inc., USA, 1995. ISBN 0070158401.*

[13] *Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. Visualizing the behavior of object-oriented systems. In* Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '93, page 326–337, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0897915879. doi: 10.1145/165854.165919. URL https://doi.org/10.1145/165854.165919.*

[14] *Christopher J. DiGiano, Ronald M. Baecker, and Russell N. Owen. Logomedia: A sound-enhanced programming environment for monitoring program behavior. In* Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, CHI '93, page 301–302, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0897915755. doi: 10.1145/169059.169229. URL https://doi.org/10.1145/169059.169229.*

[15] *Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner. Seesoft-a tool for visualizing line oriented software statistics.* IEEE Trans. Softw. Eng.*, 18(11):957–968, nov 1992. ISSN 0098-5589. doi: 10.1109/32.177365. URL https://doi.org/10.1109/32.177365.*

[16] *Barrett Ens, Daniel Rea, Roiy Shpaner, Hadi Hemmati, James E. Young, and Pourang Irani. Chronotwigger: A visual analytics tool for understanding source and test co-evolution. In* 2014 Second IEEE Working Conference on Software Visualization*, pages 117–126, 2014. doi: 10.1109/VISSOFT.2014.28.*

[17] *L. Erlikh. Leveraging legacy system dollars for e-business.* IT Professional*, 2(3):17–23, 2000. doi: 10.1109/6294.846201.*

[18] *Jean-Daniel Fekete, Jarke van Wijk, John Stasko, and Chris North.* The Value of Information Visualization*, volume 4950, pages 1–18. 07 2008. ISBN 978-3-540-70955-8. doi: 10.1007/978-3-540-70956-5_1.*

[19] *Tudor Gîrba. Modeling history to understand software evolution. 2005.*

[20] *Gillian J Greene, Marvin Esterhuizen, and Bernd Fischer. Visualizing and exploring software version control repositories using interactive tag clouds over formal concept lattices.* Information and Software Technology*, 87:223–241, 2017.*

[21] *Lois M. Haibt. A program to draw multilevel flow charts. In* Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*, IRE-AIEE-ACM '59 (Western), page 131–137, New York, NY, USA, 1959. Association for Computing Machinery. ISBN 9781450378659. doi: 10.1145/1457838.1457861. URL https://doi.org/10.1145/1457838.1457861.*

[22] *Jon Hueras and Henry Ledgard. An automatic formatting program for pascal.* SIGPLAN Not.*, 12(7):82–84, jul 1977. ISSN 0362-1340. doi: 10.1145/954639.954645. URL https://doi.org/10.1145/954639.954645.*

[23] *Ivar Jacobson, Grady Booch, and James Rumbaugh.* The Unified Software Development Process*. Addison-Wesley Longman Publishing Co., Inc., USA, 1999. ISBN 0201571692.*

[24] David H Jameson. Sonnet: Audio-enhanced monitoring and debugging. In SANTA FE INSTITUTE STUDIES IN THE SCIENCES OF COMPLEXITY-PROCEEDINGS VOLUME-, volume 18, pages 253–253. ADDISON-WESLEY PUBLISHING CO, 1994.

[25] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining github. In Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, page 92–101, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328630. doi: 10.1145/2597073.2597074. URL https://doi.org/10.1145/2597073.2597074.

[26] P. Kapec, G. Brndiarová, M. Gloger, and J. Marák. Visual analysis of software systems in virtual and augmented reality. In 2015 IEEE 19th International Conference on Intelligent Engineering Systems (INES), pages 307–312, 2015. doi: 10.1109/INES.2015.7329727.

[27] Pooya Khaloo, Mehran Maghoumi, Eugene Taranta, David Bettner, and Joseph Laviola. Code park: A new 3d code visualization tool. In 2017 IEEE Working Conference on Software Visualization (VISSOFT), pages 43–53, 2017. doi: 10.1109/VISSOFT.2017.10.

[28] C. Knight and M. Munro. Virtual but visible software. In 2000 IEEE Conference on Information Visualization. An International Conference on Computer Visualization and Graphics, pages 198–205, 2000. doi: 10.1109/IV.2000.859756.

[29] Donald E. Knuth. Computer-drawn flowcharts. Commun. ACM, 6(9):555–563, sep 1963. ISSN 0001-0782. doi: 10.1145/367593.367620. URL https://doi.org/10.1145/367593.367620.

[30] Guillaume Langelier, Houari Sahraoui, and Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE '05, page 214–223, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1581139934. doi: 10.1145/1101908.1101941. URL https://doi.org/10.1145/1101908.1101941.

[31] Michele Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In Proceedings of the 4th International Workshop on Principles of Software Evolution, IWPSE '01, page 37–42, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 1581135084. doi: 10.1145/602461.602467. URL https://doi.org/10.1145/602461.602467.

[32] Michele Lanza. Object-oriented reverse engineering —- coarse-grained, fine-grained, and evolutionary software visualization. 2003.

[33] M. M. Lehman and L. A. Belady. Program Evolution: Processes of Software Change. Academic Press Professional, Inc., USA, 1985. ISBN 0124424406.

[34] Roberto Erick Lopez-Herrejon, Sheny Illescas, and Alexander Egyed. A systematic mapping study of information visualization for software product line engineering. Journal of software: evolution and process, 30(2):e1912, 2018.

[35] Andrea Mancino and Giuseppe Scanniello. Software musification. In 2017 21st International Conference Information Visualisation (IV), pages 127–132, 2017. doi: 10.1109/iV.2017.28.

[36] *Shane McIntosh, Katie Legere, and Ahmed E. Hassan. Orchestrating change: An artistic representation of software evolution. In* 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 348–352, 2014. doi: 10.1109/CSMR-WCRE.2014.6747192.*

[37] *L. Merino, M. Ghafari, C. Anslow, and O. Nierstrasz. A systematic literature review of software visualization evaluation.* Journal of Systems and Software*, 144:165–180, 2018. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2018.06.027. URL* `https://www.sciencedirect.com/science/article/pii/S0164121218301237`*.*

[38] *Leonel Merino, Mohammad Ghafari, Craig Anslow, and Oscar Nierstrasz. Cityvr: Gameful software visualization. In* 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 633–637, 2017. doi: 10.1109/ICSME.2017.70.*

[39] *H. A. Müller and K. Klashinsky. Rigi-a system for programming-in-the-large. In* Proceedings of the 10th International Conference on Software Engineering*, ICSE '88, page 80–86, Washington, DC, USA, 1988. IEEE Computer Society Press. ISBN 0897912586.*

[40] *I. Nassi and B. Shneiderman. Flowchart techniques for structured programming.* SIGPLAN Not.*, 8(8):12–26, aug 1973. ISSN 0362-1340. doi: 10.1145/953349.953350. URL* `https://doi.org/10.1145/953349.953350`*.*

[41] *Thomas Panas, Thomas Epperly, Daniel Quinlan, Andreas Saebjornsen, and Richard Vuduc. Communicating software architecture using a unified single-view visualization. In* 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 217–228. IEEE, 2007.*

[42] *Martin Pinzger, Harald Gall, Michael Fischer, and Michele Lanza. Visualizing multiple evolution metrics. In* Proceedings of the 2005 ACM Symposium on Software Visualization*, SoftVis '05, page 67–75, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930736. doi: 10.1145/1056018.1056027. URL* `https://doi.org/10.1145/1056018.1056027`*.*

[43] *J. Ratzinger, M. Fischer, and H. Gall. Evolens: lens-view visualizations of evolution data. In* Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, pages 103–112, 2005. doi: 10.1109/IWPSE.2005.16.*

[44] *Vitalis Salis and Diomidis Spinellis. Repofs: File system view of git repositories.* SoftwareX*, 9:288–292, 2019. ISSN 2352-7110. doi: https://doi.org/10.1016/j.softx.2019.03.007. URL* `https://www.sciencedirect.com/science/article/pii/S2352711018300712`*.*

[45] *Teseo Schneider, Yuriy Tymchuk, Ronie Salgado, and Alexandre Bergel. Cuboidmatrix: Exploring dynamic structural connections in software components using space-time cube. In* 2016 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 116–125, 2016. doi: 10.1109/VISSOFT.2016.17.*

[46] *Robert C. seacord, Daniel Plakosh, and Grace A. Lewis.* Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co., Inc., USA, 2003. ISBN 0321118847.*

[47] *Ian Sommerville.* Software Engineering (5th Ed.)*. Addison Wesley Longman Publishing Co., Inc., USA, 1995. ISBN 0201427656.*

[48] D.H. Sonnenwald, B. Gopinath, G.O. Haberman, W.M. Keese, and J.S. Myers. Infosound: an audio aid to program comprehension. In Twenty-Third Annual Hawaii International Conference on System Sciences, volume 2, pages 541–546 vol.2, 1990. doi: 10.1109/HICSS.1990.205229.

[49] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. Pydriller: Python framework for mining software repositories. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, page 908–911, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355735. doi: 10.1145/3236024.3264598. URL https://doi.org/10.1145/3236024.3264598.

[50] Frank Steinbrückner and Claus Lewerentz. Representing development history in software cities. In Proceedings of the 5th International Symposium on Software Visualization, SOFTVIS '10, page 193–202, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300285. doi: 10.1145/1879211.1879239. URL https://doi.org/10.1145/1879211.1879239.

[51] Christopher M. B. Taylor and Malcolm Munro. Revision towers. In Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT '02, page 43, USA, 2002. IEEE Computer Society. ISBN 0769516629.

[52] Paul Vickers. External auditory representations of programs: Past, present, and future an aesthetic perspective. 01 2004.

[53] Paul Vickers and James L. Alty. Siren songs and swan songs debugging with music. Commun. ACM, 46(7):86–93, jul 2003. ISSN 0001-0782. doi: 10.1145/792704.792734. URL https://doi.org/10.1145/792704.792734.

[54] A. Von Mayrhauser and A.M. Vans. Program comprehension during software maintenance and evolution. Computer, 28(8):44–55, 1995. doi: 10.1109/2.402076.

[55] Richard C. Waters. User format control in a lisp prettyprinter. ACM Trans. Program. Lang. Syst., 5(4):513–531, oct 1983. ISSN 0164-0925. doi: 10.1145/69575.357225. URL https://doi.org/10.1145/69575.357225.

[56] Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities: a controlled experiment. In 2011 33rd International Conference on Software Engineering (ICSE), pages 551–560, 2011. doi: 10.1145/1985793.1985868.

[57] P. Young and M. Munro. Visualising software in virtual reality. In Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242), pages 19–26, 1998. doi: 10.1109/WPC.1998.693276.