# 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 [15]. It stated that it is not essential to learn how to compose software but instead, it is necessary to develop its own style and implications. During the lifetime of a software system, many developers work on it, each one in their manner. This is one of the multiple reasons behind software complexity. Today's systems are characterized by sheer size and complexity. Software maintenance takes up the most of a system's cost. It is hard to quantify the impact of software maintenance on the global cost of the software. However, Researchers estimated it to be between 50% and 90% [12] [48][18] [47]. Many factors influence the cost of maintenance; among these, there is the understanding activity needed to perform maintenance tasks [9].

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

Lehman and Belady, in 1985, were the first to observe that maintaining a software system becomes a more complex activity over time. [34] The term "software evolution" was used for the first time in their set of laws. One of the goals of its analysis is to identify potential defects in the system's logic or architecture. Visualization techniques often support software evolution analysis.

Software visualization is a specialization of information visualization with a focus on software [33]. In literature, a lot of visualization techniques have been presented to support a complex software system's analysis. Usually, a massive quantity of multivariate evolutionary data needs to be depicted. Several tools have been proposed in the literature to do that [38].

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

In this work, our focus is an explorative visualization that depicts the evolution of a system. We present an approach with an evolutionary design-level visualization to facilitate the comprehension of the system's history. Our technique models and mines efficiently large git repositories. We also offer a visualization strategy 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.

## 1.1 Contribution

We can summarize the main contribution of this work as follows:

- We proposed an approach to model the history of a large git repository.
- We proposed an approach to mine large git repositories.
- We proposed an approach based on 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.

## 1.2 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*" [55]. 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.[19] 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.

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 [42], when researchers analyze version control repositories [21], or to support developers' activity [35].

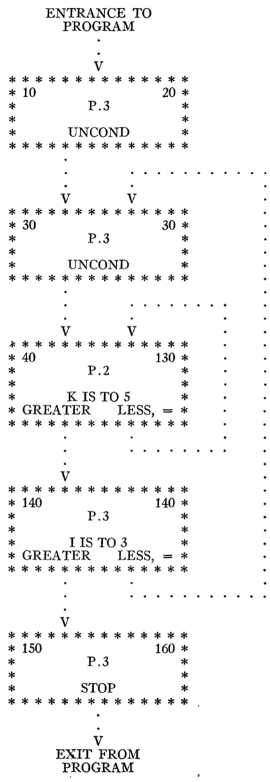


Figure 2.1. Flowchart presented by Haibt in 1959

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.

Dijkstra1971a 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.

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 [22]. 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.

Ten years later, Knuth also confirmed the effectiveness of flowcharts [30]. 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[41], 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 [23] then Waters [56] 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. [40] released Rigi, a tool used to visualize large programs. It exploited the graph model, augmented with abstraction mechanisms, to represent systems components and relationships.

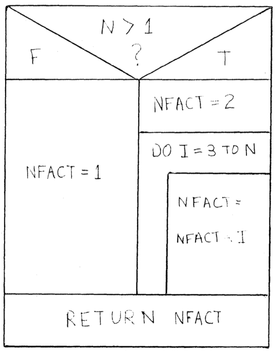


Figure 2.2. NSD of the factorial function.

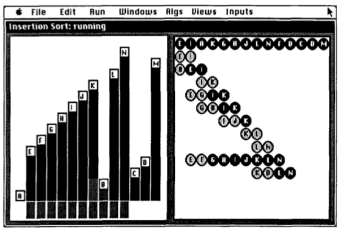


Figure 2.3. Balsa-II

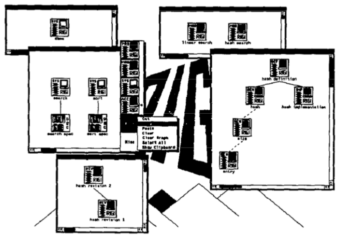

Figure 2.4. Rigi

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 [16]. 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.

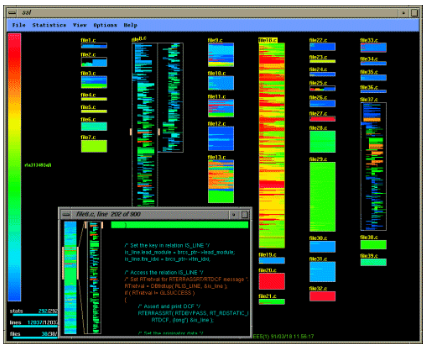

Figure 2.5. Seesoft

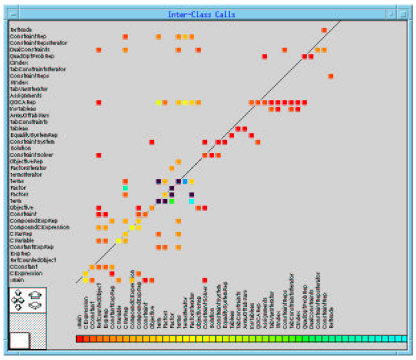


Figure 2.6. Jinsight

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 Timewell 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

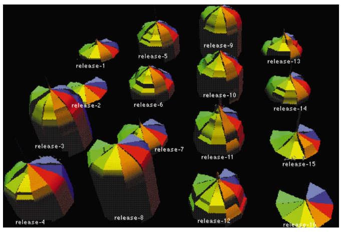

Figure 2.8. 3D wheel

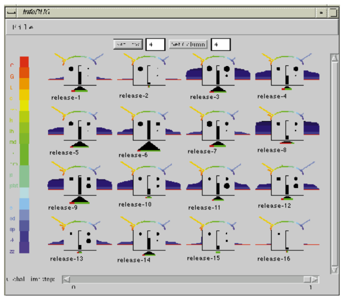

Figure 2.9. Infobug

Also in 1998, Young and Munro [58] explored representations of software for program comprehension in VR.

Finally, in 1999, Jacobson et al. [24] 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 [32] 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.

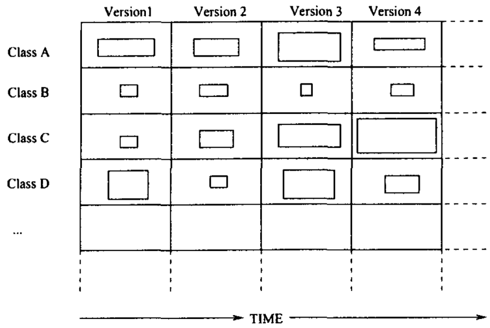


Figure 2.10. A schematic display of the Evolution Matrix

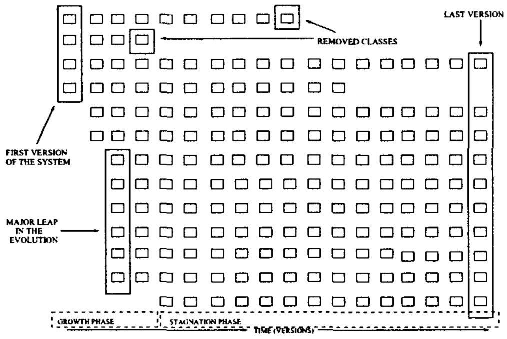


Figure 2.11. Some characteristics of the Evolution Matrix

Taylor and Munro [52], 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. [43] 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 [44], a visualization approach and tool to explore evolutionary data through structural and temporal views.

Langelier et al. [31] investigated the interpretation of a city metaphor [29] 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.

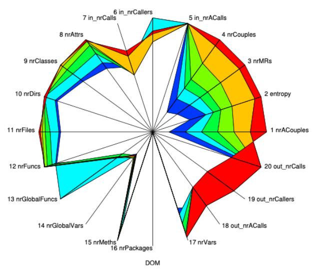

Figure 2.12. RelVis

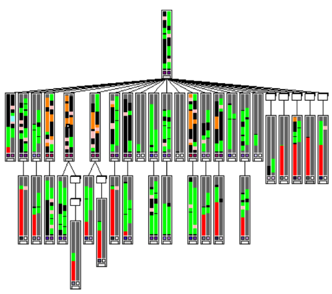

Figure 2.13. Tree of Discrete Time Figures

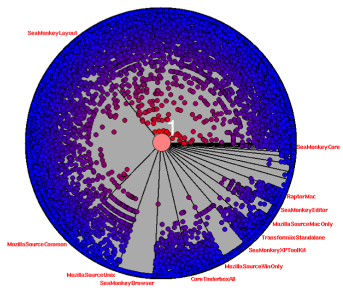

Figure 2.14. Evolution Radar

Steinbrückner and Lewerentz [51] 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.

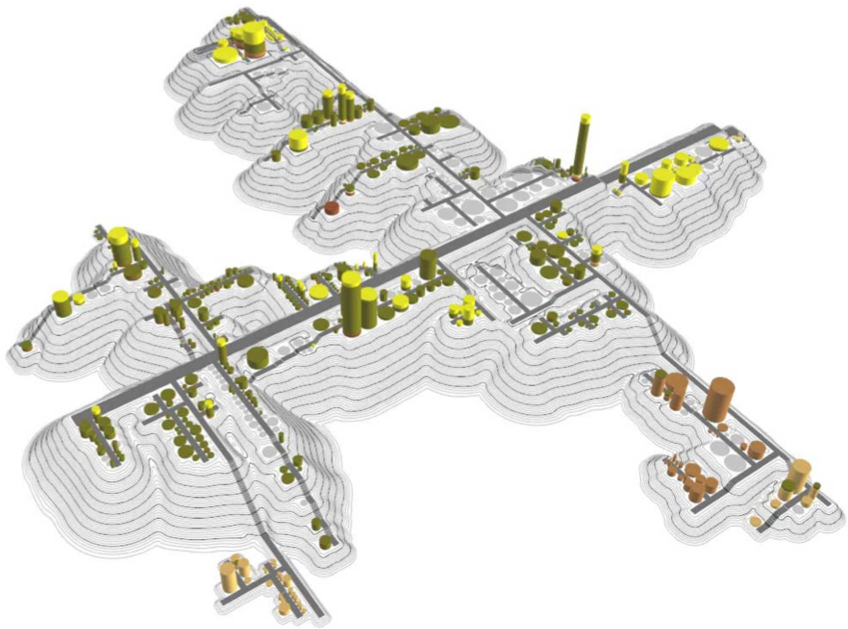

Figure 2.15. Evo-Streets

Wettel revised the city metaphor to represent metrics meaningfully [57]. 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.

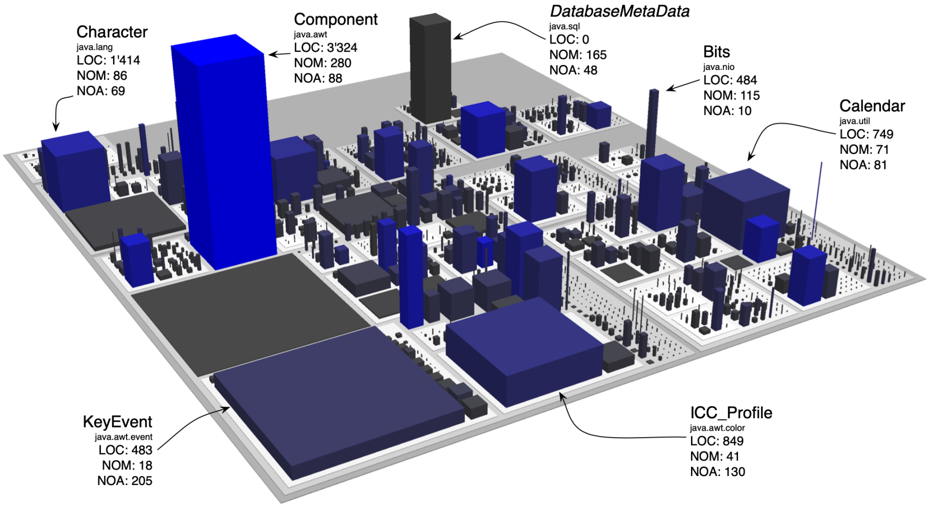


Figure 2.16. CodeCity

Ens et al. [17] 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. [27] 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. [46] 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. [39] 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.

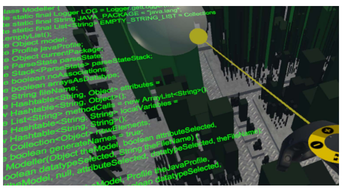

Figure 2.17. CityVR

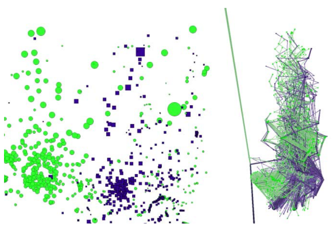

Figure 2.18. ChronoTwigger

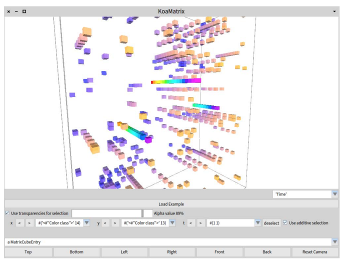

Figure 2.19. CuboidMatrix

Khaloo et al. [28] 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].

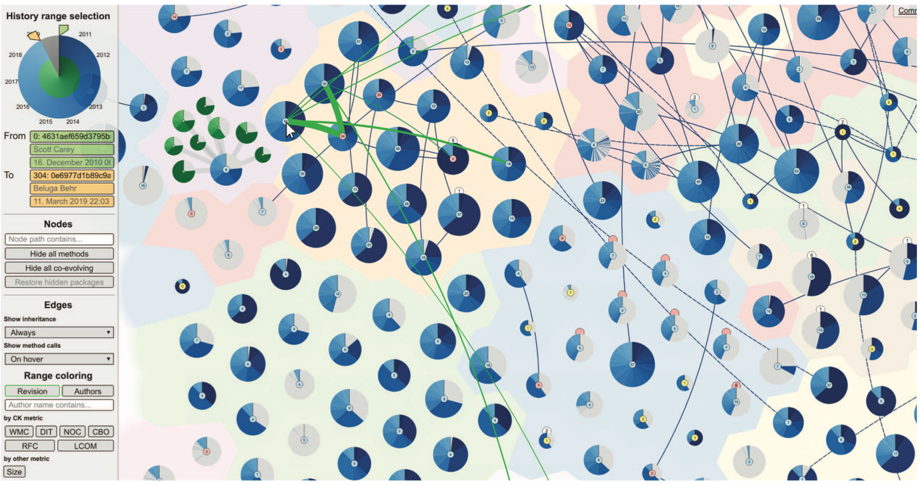


Figure 2.20. Evo-Clock

## 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.

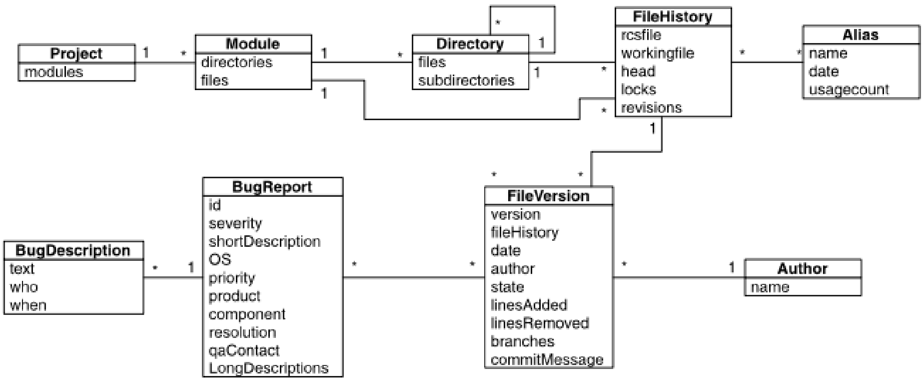


Figure 2.21. RHDB

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. [26] 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 [50]. 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 [45]. 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.

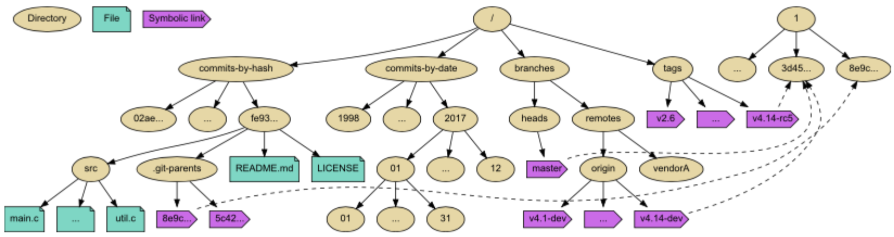


Figure 2.22. RepoFS

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. [49] 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 [25]] developed Sonnet, audio-enhanced monitoring and debugging tool. Alty and Vickers [54] 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 [53] evidenced the necessity of a multi-threaded 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.

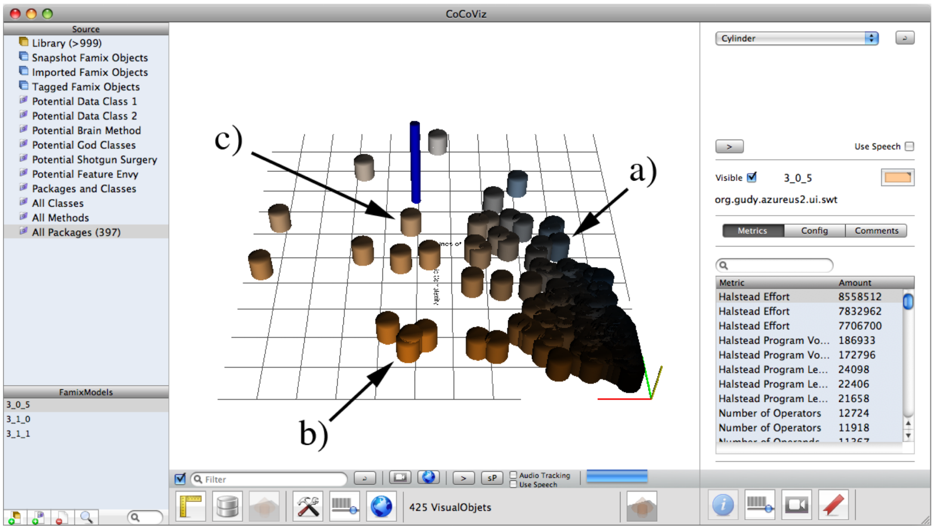


Figure 2.23. CocoViz

McIntosh et al. [37] 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 [36] 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

Comprehending the evolution of a software system is a complex activity, mainly because of the sheer amount of data and its complexity. The term "software evolution" was coined for the first time by Lehman in 1985 in a set of laws. [34] He stated that the complexity of a system is destinated to increase over time, as the system always needs to be adapted to its evolutionary environments. To be managed, software systems need to be comprehended by developers, and this activity can be simplified with software visualization.

Over the last ten years, developers have stored their codebase on git repositories. For this reason, we focused our attention on systems versioned with this protocol.

Git is a versioning control system that tracks all the changes made to every system file. Internally git holds all the information that we need to reconstruct the history of a repository.

In this chapter, we present the approach that we developed to visualize a software system using a visual and auditive depiction of the evolution of a system. To fulfill this purpose, we have chosen to leverage the phenomenon of Synesthesia, the production of a sense impression relating to one sense by stimulation of another sense. Moreover, we also present how we reconstruct and model the history of a repository.

Our approach is composed of three parts: in the first part, we model the evolution of a system; in the second part, we present two ways to visualize the system and finally, in the last part, we show how music can be used to communicate evolutionary information.

## 3.1 Evolution Model

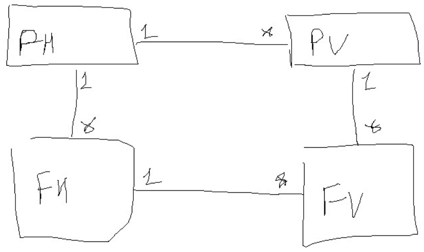


Figure 3.1. Evolutionary Model

To model the evolution of software systems, we developed the model, showed in figure 3.1, based on Hismo, a solution presented by Tudor Girba [20].

The need to develop a novel evolutionary model is because Hismo was designed to work with another versioning system: Subversion (SVN). There are several differences between SVN and git. In terms of design, the most important is how they keep track of changes. SVN works with the concept of "snapshot" while git works with "commits." In SVN, when a file has been changed, a new revision of the whole system is created, and then the number of revisions is incremented. In contrast, with git, only the modified files would get committed, and thus we don't have every time a new snapshot. Therefore, since we have to work with git and not with SVN, we took Hismo as the starting point of our model and adapted it to the git protocol. Initially, the model was based on:

- 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 History.
- History. An entity that holds a set of Versions.

Git does not have the concept of Snapshot; we replaced it with a new component called FileVersion. A file version is essentially the version of a file at a particular time. It has the same fashion as a Snapshot. Still, instead of being related to every version of the system, it is related only to the Versions where the file was effectively updated. Moreover, we made a distinction between File entities and Project entities. So, we mapped the concept of History to FileHistory and the idea of Version to ProjectVersion. The entity responsible for holding both of them is called ProjectHistory. To summarize, these are the four main concepts of our model:

- **ProjectHistory**: it represents the history of a repository. It is the holder of two sets: a set of FileHistories and a set of ProjectVersions.
- **FileHistory**: it represents a file of the repository. We consider each file as an entity of the system. Even if the entity's name or location is changed, our mode will treat it as the

same. So, our model is resilient to the renaming and moving activities. Each FileHistory holds a set of FileVersion, each one representing a different version of the entity at another 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 at a particular point in time. It is responsible for holding all evolutionary information of an entity since it represents 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 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 repository timeline. Usually, developers exploit branches to develop features on them and then merge the developed code with the existing codebase. They need to create a "merge commit" to do that. Each time we create a new git commit, we deploy a new system version that records all the changes made to the commits' tracked files. Internally, in git, all the commits are stored as nodes of a commit-tree tree. The root node represents the repository's first commit since it has no parents. All the other nodes instead represent the commits made during the whole lifecycle of the repository. Each commit usually has only one parent, representing the previous commit made before that one. One case where a commit might have more than one parent: merges commits.

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

Git can recognize the following actions made on a file:

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

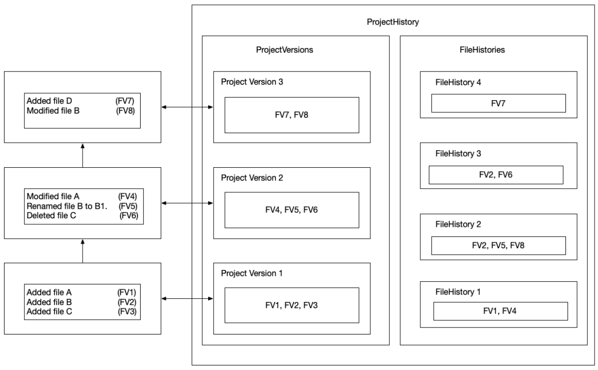


Figure 3.2. Rebuilding history example

- **MOVE**. A file was moved from one location to another, so the file path changed. This action detected whether 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 after the changes, and many more. We used the commit's information to track all the paths of an entity. We can update the entity path when it was renamed or moved to follow it during 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 kinds of actions: MODIFY, RENAME, and MOVE. An entity might be deleted in some cases, so the last FileVersion held by a FileHistory will represent a DELETE action.

Figure 3.2 shows an example 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 channels and create a new ProjectVersion for each list entry. With this operation, we can discover if a file has been added to the system because, in 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 change was mapped to a FileVersion (FV) and consequently added to the respective FileHistory and ProjectVersion. We did the same thing for ProjectVersion 2 and 3.

## Partial historical representation

One of the goals that we had when we developed this approach was the possibility of analyzing a 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 systems, such as LibreOffice, Elastisearch, and Linux. They all have more than 500,000 commits in each, and thus, we cannot aim to reconstruct their histories with a single analyzer; it will take too much time. To prove that, just consider the worse case: Linux. When this thesis was redacted, the repository of Linux

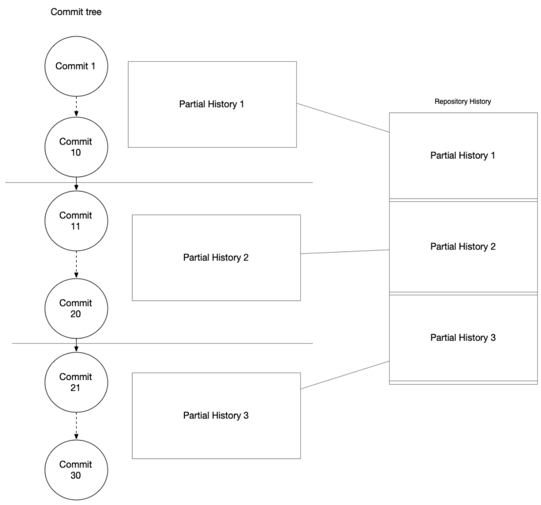


Figure 3.3. Partial history example

had 1,090,563 commits. To move from one commit to another, we assume that git needs one second. 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 vast amount of time that the analyzer needs to extract metrics from every single 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 repository's history into multiple chunks, each represented by a partial record. Then when all the analyses 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 piece. This analysis can be done parallelly since these chunks are independent of each other. In 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. 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 represents 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.
The result of a single analysis and 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 partial history, we need to preserve the characteristics of our model. In particular, if FileHistory is already present in our history, we don't have to duplicate it, but instead, we need to update it.

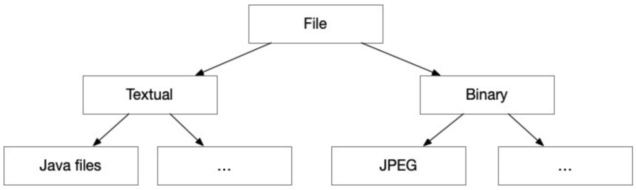


Figure 3.4. Taxonomy

| | |
|---|---|
| File | SIZE |
| Textual | LOC, LinesAdded, LinesRemoved |
| Binary | |
| Java | SLOC |

Table 3.1. aaaa

### Evolutionary metrics

Every version of the system holds a set of files. As we said, each file is represented by a FileVersion, which is part of a FileHistory. To understand all the differences between FileVersions of a FileHistory, we decided to collect metrics to represent the state of a file in a version.

Since we aim to have a language-agnostic approach, we have selected only language-agnostic metrics. We have defined a taxonomy to classify and categorize all the files present in a system. Each category is then mapped to a set of metrics. Moreover, metrics can be inherited by parent categories.

We mapped in the following way.

So, for each file, we compute the metric SIZE, and then, based on the type of the file, if it is a textual file, we also calculate the Lines Of Code (LOC) and the number of lines added and removed. In addition, if a file is also a Java file, we compute the Source Lines of Code (SLOC).

Our approach can be extended in multiple ways. For example, we can define some object-oriented metrics defined only for a particular kind of file, or also, we can introduce a new category (e.g., Object Oriented or Functional).

## 3.2 Visualization

The approach that we have defined can be applied to different contexts. We can represent a ProjectHistory with two kinds of visualization: a 2D visualization, that uses a matrix and works better with small systems, and it is easier to be implemented, and a 3D environment that can exploit human perception as a vector of information.

### 2D representation

This visualization is based on the EvolutionMatrix approach defined by Lanza [32].

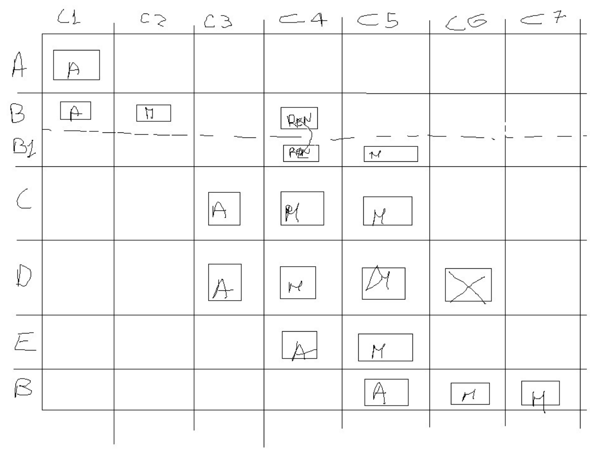


Figure 3.5. Evolution matrix of a repository

We said that a ProjectHistory is a holder of ProjectVersions and FileHistories. A ProjectVersion represents a commit or a version of the system, whereas a FileHistory represents the history of a file. The connection between these two entities is a FileVersion that describes the state of a file in a system's version.

We can represent a ProjectHistory through a matrix with the following properties:

- Each column of the matrix represents ProjectVersion, so a version of a commit of the repository.
- Each row of the matrix represents a FileHistory, so the history of a file.
- Each cell of the matrix represents a ProjectVersion, so the state of a file is in a specific version.

Since we built our model on the top of git, we don't track snapshots of a system but only change. As a consequence, the difference between our matrix and the one defined by Lanza [32] is that we can have holes in the row of an entity. A FileHistory was not modified in a ProjectVersion; this event will be represented by an empty cell. This concept was not present in the Evolution Matrix of Lanza because its model worked with SVN, and thus, it worked with the idea of incremental snapshots.

Figure 3.5 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 C and D were added in the third revision. The fourth revision recorded a rename of B to B1. It's essential to notice that B and B1 represent the same entity; therefore, the same FileHistory represents them.

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

- **by rows**, if we are interested in the history of a particular entity of our system. For example, the FileHistory represented by the first third row in figure 3.5 represents the history

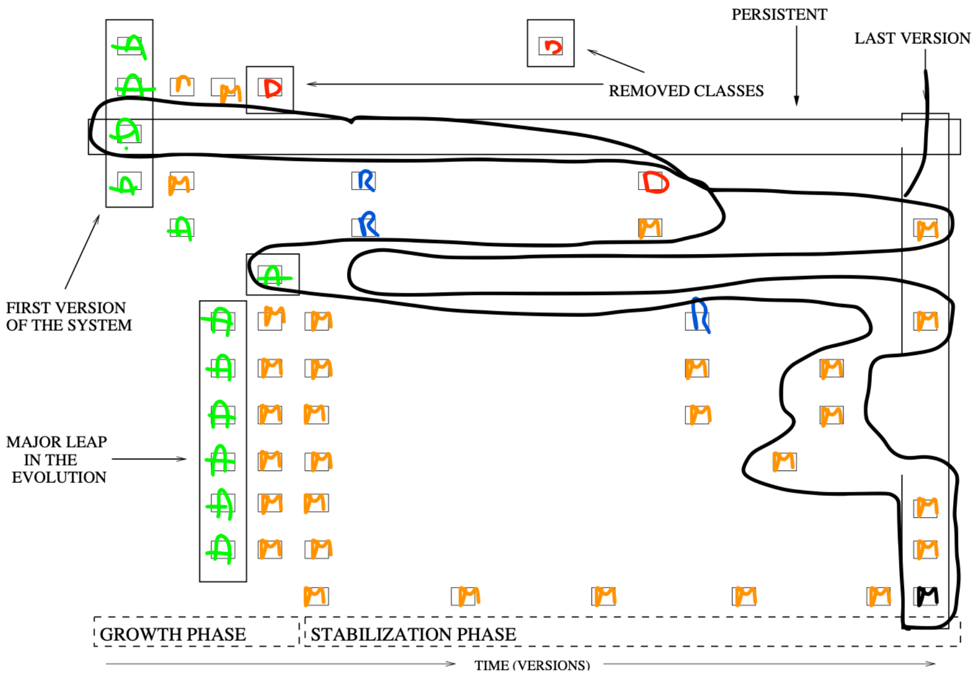


Figure 3.6. Evolution matrix of a repository

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 3.5 is also an excellent example to understand why we cannot rely on the name of the file to identify the entity. We can notice that file B, represented by the second FileHistory, was added on the first version and then renamed on the fourth from B to B1. Then, a new file called B was added in the fifth version. Nonetheless, the name of the files are the same; they must represent two different entities. Even if file B were added in version four, we would have had the same result.

- **by columns**, if we are interested in 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 3.6 shows an example of how to recover evolution information from the matrix. As we have seen, each version does not represent a snapshot of the system. Instead, it represents only the difference in changes made to the previous version. To recover a snapshot of a specific version, we need to consider the last changes made before that particular 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 difference was deleted, we have to ignore the related 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 3D representation aims to make the analysis of a system easier for engineers by exploiting the human senses. This is why we have chosen to leverage the phenomenon of Synesthesia. The phenomenon of synesthesia occurs when stimulation of a sense or a cognitive pathway leads to the involuntary stimulation of another reason or a cognitive path. We experience synesthesia when two or more things are perceived as the same. For example, 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 involuntary associations:

- **Color**: we use the color to describe actions made on an entity (ADD, MODIFY, RENAME, MOVE, DELETE). Ideally, when an entity is deleted, it should be removed from the visualization. This decision is up to the user.

- **Shape**: we use the shape of the entity to describe the category of the entity. We have defined a taxonomy to categorize entities, and these categories are also mapped. For example, a java file could be represented by a cube, whereas a sphere could represent a binary file.

- **Height**: we use the height of the entity to describe the value of a metric. The decision of which metric is up to the user.

### Layout

We developed our visualization approach to visualize the evolution of a system incrementally. We cannot track hierarchical relationships with a language-agnostic approach, such as classes belonging to a package. So, we have adopted a simple representation that takes care of the creation order of entities. To layout FileHiistories, we use an outward spiral layout, where the center of the spiral is the first entity added to the system.

### Time traversal

Histories are definer over real human time. Some repositories have been developed for more than ten years. Consequently, there are several ways to traverse the history of an entity. We can consider the human time or just view the commits made.

The visualization needs to start from the first moment and go forward until the end. In our approach, we define two strategies to traverse the history of a repository:

- Strategy 1 We can display n version at the 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 the two commits, thus we cannot distinguish active development phases from unactive development phases.

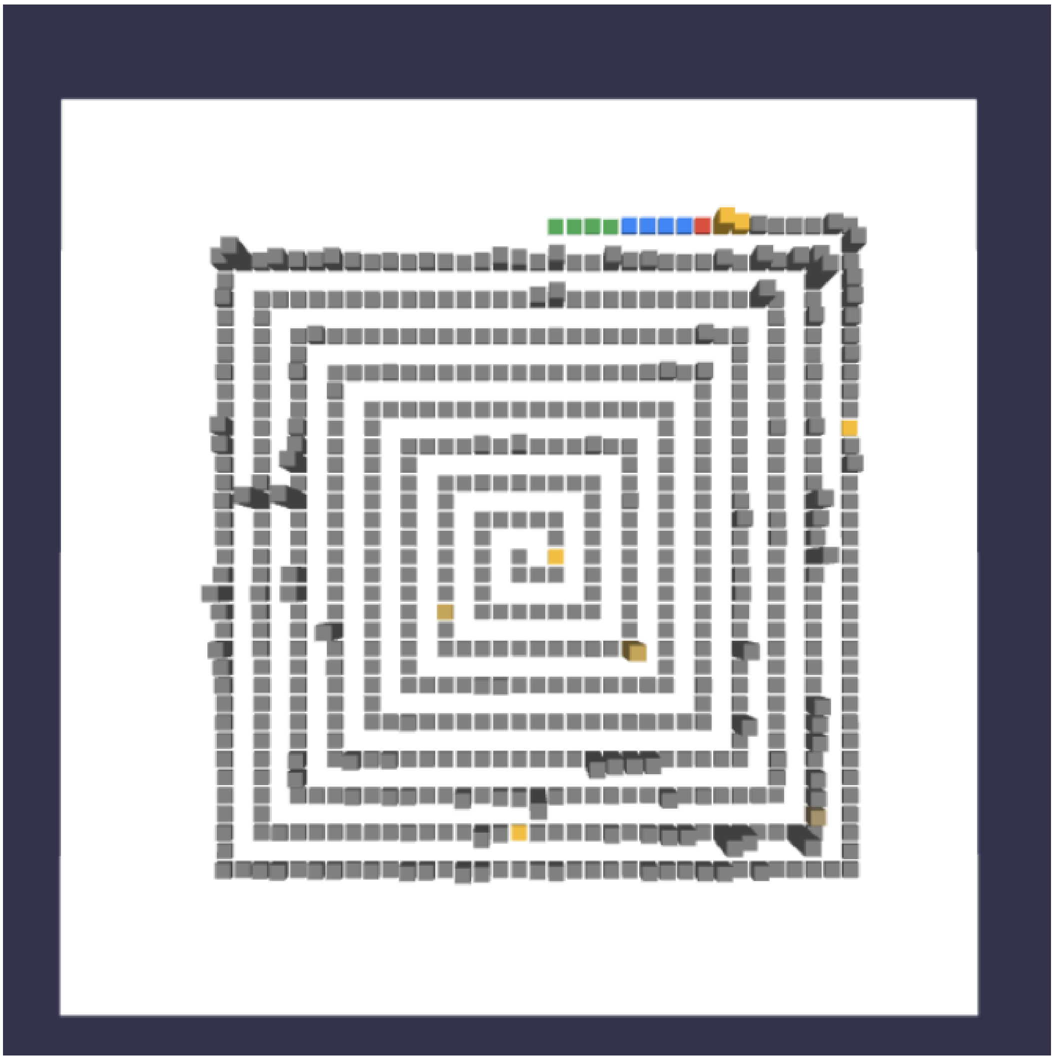

Figure 3.7. Spial layout exaple

- Strategy 2 We can group versions by timestamp. So, all the commits made in the same period will be displayed simultaneously. This strategy works very well if we need to comprehend how the system evolved and at which speed in time.

In both our strategies, we might need to aggregate more versions into a single entity: a **moment**. We define a moment as a group of versions that will be displayed simultaneously.

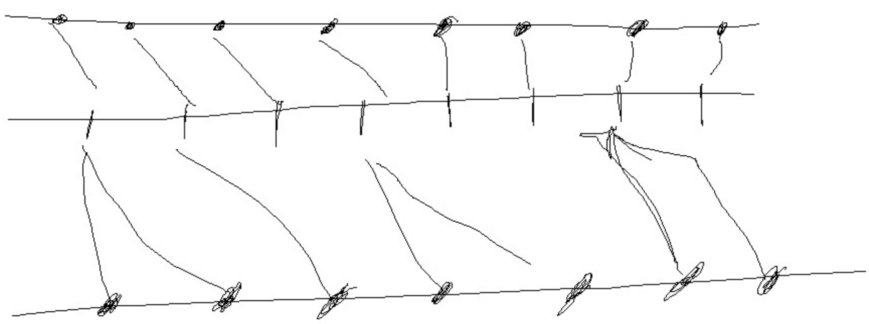

Figure 3.8. Example of two different strategies to identity moments. In the first strategy, we mapped one commit with one moment, so the number of moments will equal the number of commits. Alternatively, in the second strategy, we created a moment every day. As a result, we have moments with many commits and some without anyone. With this strategy, the number of moments will be the same as the number of days between the first and the last commit.

Figure 3.8 highlights the difference between the two strategies mentioned above.

### Color

The entity's color should recall the last action made on that entity. To achieve this purpose, we used the color association described in figure 3.9. Nonetheless, each person has their perception of colors. Thus, we can not assume that this color will work similarly for all people. To remedy this issue, users can customize the color palette.

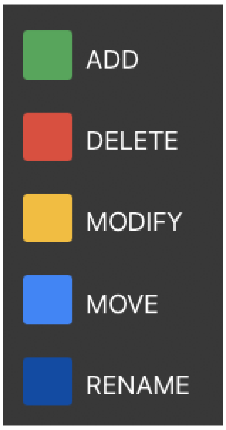


Figure 3.9. Color association

We decided to put another piece of 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. We mapped the age of an entity with the darkness of its color to do that. As a result, older entities will be displayed with a darker color. This way, users can immediately recognize the last action and the time passed since the entity was modified.

### Shape

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

### Height

As we said, the height of an entity must represent the value of a chosen metric. Entities without the selected metric would not be represented in our approach.

To extract the value of the height from a metric, we used to map it with a **mapper**. A mapper is defined as a function that returns its height given an entity and a metric.

This mapping function might be defined in several ways, such as with a linear process whose maximum value is predefined and its minimum value is 0. The choice of which strategy should be used is up to the user.

## 3.3 Auditive model

We supported our visualization approach with an auditive portrayal of ProjectVersions being displayed. The goal is to transmit information such as the number of version changes, number of additions, and number of deletions, through audio notes.

We investigated several approaches to map evolutionary metrics to sound; as a result, our approach uses the following audio concepts:

- BPM: Beats Per Minute or tempo sets the rhythm of a song.
- Note's pitch: the quality that makes it possible to judge sounds as "higher" and "lower" in a sense associated with musical melodies.
- Synthesizer: a producer of audio signals.

As we said, our visualization works over the concept of moments. We mapped each moment to a sound that is generated procedurally. Each sound is composed of three different synthesizers. We used them to distinguish between the three pieces of information that we want to provide: number of version changes, number of additions, and number of deletions. Moreover, the note reproduced by each synthesizer depends on the magnitude of the metric. For example, if we compose the sound of a moment that holds lots of additions, the note played by the synthesizer that represents additions will be high. In contrast, a moment with few additions will be mapped to a note with a low pitch. Finally, the tempo depends on the number of changes in our approach. So a moment with lots of changes will sound more fierce than a moment with few changes.

# 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. Evo-clocks: 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] *Edsger Wybe Dijkstra.* Ewd316: A short introduction to the art of programming. *Technische Hogeschool, 1971.*

[16] *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`*.*

[17] *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.*

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

[19] *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.*

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

[21] *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.*

[22] *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`*.*

[23] *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`*.*

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

[25] *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.*

[26] *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`.

[27] *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.*

[28] *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.*

[29] *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.*

[30] *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`.

[31] *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`.

[32] *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`.

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

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

[35] *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.*

[36] *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.*

[37] *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.*

[38] *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`.

[39] *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.*

[40] *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.*

[41] *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`.

[42] *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.*

[43] *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`.

[44] *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.*

[45] *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`.

[46] *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.*

[47] *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.*

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

[49] *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.*

[50] *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.*

[51] *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.*

[52] *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.*

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

[54] *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.*

[55] *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.*

[56] *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.*

[57] *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.*

[58] *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.*