Università della Svizzera italiana

Software Institute

M3TRICITY

A 3D Evolution-resistant Visualization of Software Systems

Federico Pfahler

Sep 2020

Supervised by
Prof. Dr. Michele Lanza

Co-Supervised by
Dr. Roberto Minelli
Dr. Csaba Nagy

Software & Data Engineering Master Thesis

Abstract

Visualization approaches that leverage a 3D city metaphor have become popular over the years. There are multiple variations of this concept, including virtual and augmented reality.

Summarizing the information provided by the evolution of software systems comprehensively is beneficial since it provides insights about what happened in the past, allowing improvement, extension and reuse of the previously created structures. Current approaches for visualizing software systems using the city metaphor do not consider their evolution, which results in a visualization where buildings and districts move around in unpredictable ways. Those circumstances can generate uncertainty to the viewer, leading to possible misinterpretations of the information provided by the visualization.

In this thesis, we propose a novel approach to visualize the evolution of software systems using an evolution-resistant layout. The visualization treats the system as an evolving city where buildings and districts represent classes and packages, which undergo structural changes over time. A model that is able to process the evolution of the software system, without losing any information it provides, can be used to generate a layout that resists to changes happening over time.

The final result is a compact and understandable representation in which entities are placed consistently over the evolution of the software system. To present our approach, we developed M3TRICITY, a web application for analyzing and visualizing software systems evolution.

Dedicated to my family and friends who have been so supportive during all these years.

Acknowledgements

My thanks and appreciation to Prof. Dr. Michele Lanza. A fantastic advisor, source of inspiration and knowledge, always available to help when needed. I also appreciated his continuous feedback, always constructive, which led me to grow personally during this thesis's work. But mostly, I have to thank him for letting me work on a subject that has interested me so much.

I am also grateful to my two co-advisors, Dr. Roberto Minelli and Dr. Csaba Nagy. Their help has been essential several times during these months, often dedicating me many hours while trying to teach me new things. Thanks also for the feedback on the thesis, which was always useful and allowed me to improve its quality.

Thanks to the teachers, assistants, and collaborators of the faculty of informatics of the Università della Svizzera Italiana. Not only have you made me fall in love with this subject through your passion and dedication, but you have taught me so many things that I can never be grateful enough.

I must acknowledge as well the many friends and family members that supported me during the writing of this thesis.

Thanks to my father, Francesco, the person who first introduced me to the world of computers. Without you and your passion, I would never have taken this path.

Grazie a Giuseppe, che mi ha aiutato in tutti questi anni. Mi hai visto crescere, standomi vicino e aiutandomi sempre a prendere la scelta giusta.

A special thanks go to my beloved, Chiara, for being such an amazing person. You've always been close to me, making me smile and distracting me when you realized that I needed it most.

A mia mamma, vanno i ringraziamenti più grandi. Sei la persona più importante della mia vita e mi sei sempre stata vicina quando più ne avevo bisogno, credendo sempre in me. Sappi che, senza di te, oggi, non avrei raggiunto questo bellissimo traguardo.

Contents

List of Figures

Chapter 1

Introduction

Analyzing and displaying large software systems can be a difficult task due to the magnitude and complexity they have reached today.

Software systems evolve during their lifecycle, hence extracting and displaying meaningful information in a way that is consistent with their evolution is not a trivial task. According to Lehman's second law of software evolution, as software system evolves, their complexity increases unless work is done to maintain or reduce it [8]. Reducing the complexity of a large software system is a tedious task. It requires in-depth knowledge of the internal software system composition. This type of information is often not easily accessible due to insufficient documentation. Also, knowing how software systems evolve can help plan future decisions by giving insights about what happened in the past.

The complexity of software systems is derived from their intangibility [7]. One method to make them more comprehensible is to use software visualizations. Price *et al.* defines it as the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software [9]. Software visualizations can increase the human understanding of software systems using intuitive, visual representations [10].

To visualize software systems, we can use metaphors. A metaphor is a stable and systematic relationship between two conceptual domains, i.e., source and target [11]. In the past decades, multiple metaphor-based visualizations have been made to remedy the intangibility of software systems. Some examples are cities [12, 13], trees [14] or even solar systems [15].

One widely researched and used metaphor for visualizing software systems is the city metaphor [16, 17, 18, 19, 13, 1] (see Figure 1.1). Cities shares many similarities with software systems. They are complex, human-made entities that follow specific patterns. Like a software system, a city starts with a planning phase in which architects aim to fulfill the requirements while maintaining functionality and design. Following this phase, they are both built incrementally while requiring maintenance over their evolution [7]. This metaphor has been proven to facilitate program comprehension, leading to an increase in terms of task correctness and a decrease in task completion time [20]. However, this type of visualization falls short of expectations when visualizing the evolution of software systems. Buildings and districts are depicted as unique entities among each version, creating situations in which the position of the entities are not consistent, making it difficult to interpret.

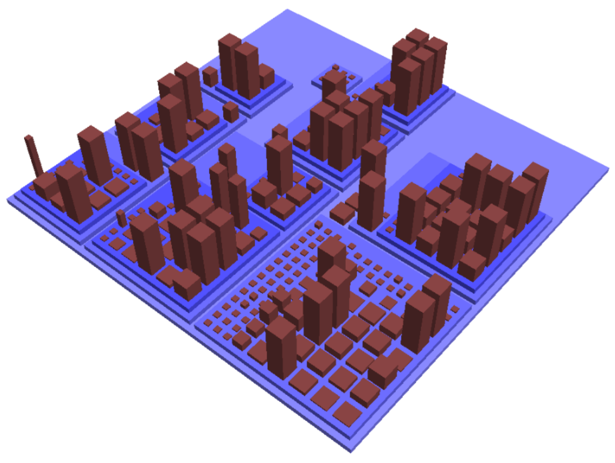


FIGURE 1.1: Software as cities [1].

In this thesis, we propose an approach to overcome this problem. Through the use of the information contained in the versioning system, we have created an evolution-resistant layout that considers the evolution of software systems to position the entities consistently over time. Besides, we have also created a new evolution model for software systems which is used to explore the history of software systems. To validate our layout, we developed M3TRICITY, a tool for visualizing software systems evolution.

1.1 Contributions

The main contributions of our thesis can be summarized as follows:

- We developed a new evolution-resistant layout for software systems. To do so, we created an algorithm that exploits the evolution model.
- We developed a new evolution model for software systems that makes it easier to access and explore their history.
- We developed a web-based application for visualizing software systems using the evolution-resistant layout and the evolution model.

1.2 Document Structure

The document is structured as follow:

- In Chapter 2, we describe the state of the art in the field of software evolution and visualization. We look at evolution models, 2D and 3D visualizations, and some examples of software as cities with particular attention to CODECITY.
- In Chapter 3, we describe M3TRICITY. We discuss the approach, the tool and the current limitations.
- In Chapter 4, we analyze M3TRICITY and three other software systems with our tool. When possible, we give insights about particular situations that have happened during development.
- In Chapter 5, we conclude the thesis with a summary and present possible future directions.

Chapter 2

State of the Art

Over the past decades, much research has been carried out in the fields covered by this thesis. Evolution and software visualization are recognized fields with venues dedicated to them.

2.1 2D Software Visualizations

In the 50s, Haibt created an approach (see Figure 2.1) that was able to facilitate the understanding of a program to developers who were new to it by making use of flow-charts [2]. A couple of years later, Knuth presented another approach that, always under the form of flow-charts, presented the documentation of software programs visually [21]. A decade later, a visualization tool that made use of Nassi-Schneiderman [3] diagrams (see Figure 2.2) was presented by Nassi *et al.*

During the 80s software systems for visualizing software systems started to appear, moving away from the previously used pretty-print style. Knuth's WEB system [22] used as the primary visualization tool a markup language that combined source code and documentation. In this period, software visualization research was mainly focused on program behaviour [1], such as the visualization of sorting algorithms [23]. Muller *et al.* presented a tool that visualized the software's components and their relationship [24, 25].

In the 90s, the field of software visualization gained more interest, with researchers experimenting with different approaches. Eick *et al.* created SeeSoft a tool that provided fine-grained visualizations [26]. The tool is able to visualize multiple information about the evolution of a system. GASE, created by Hold and Pak, was a 2D representation of the architecture over time [27]. In 1999, Lanza proposed a tool to generate 2D visualizations based on the metrics extracted from the source code [28]. The research in 2D visualizations continues today. As an example, Holten *et al.* presented Extravis [29], a tool for visualizing program traces.

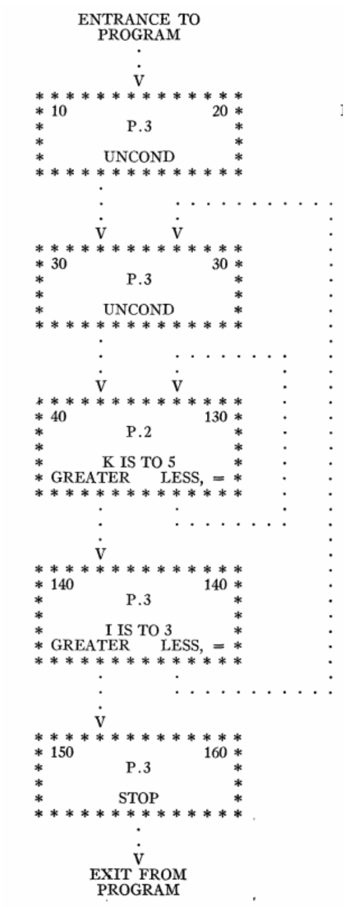


FIGURE 2.1: Flow-chart [2].

The born of UML influenced the research of software visualization. UML, even if it was not born as a software visualization language, it was able to display information about static and dynamic aspects of software systems. In the following years, approaches have been made by making use of different diagrams or their combination. What is interesting is the appearance of 3D visualizations and their metaphors. Having a third dimension increased the space at disposal, therefore increasing information that could be displayed.

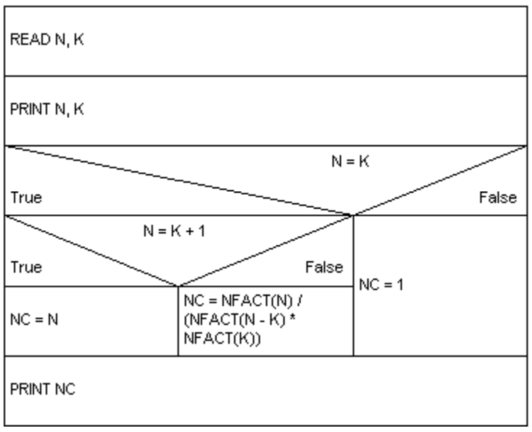
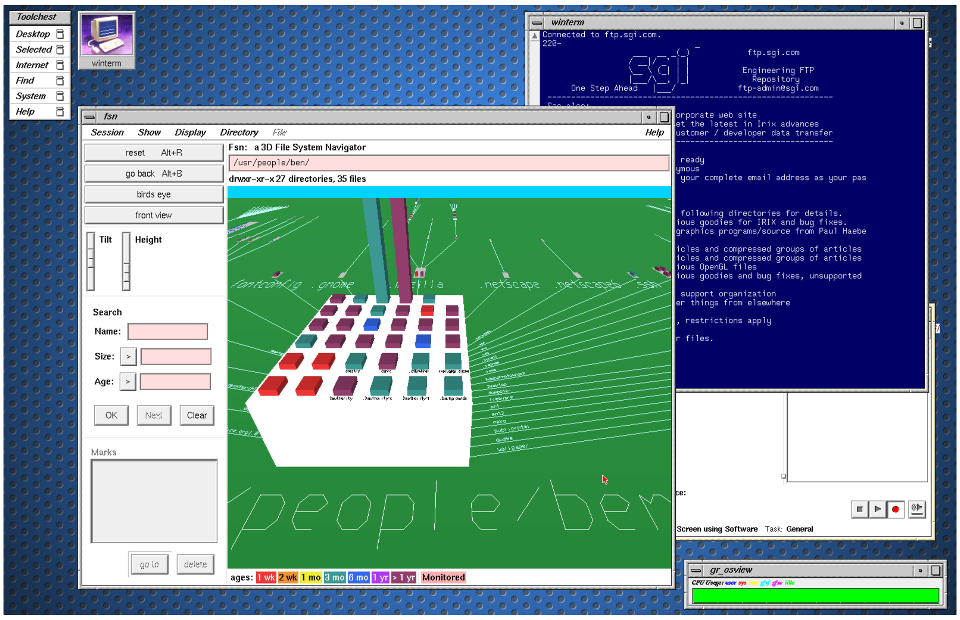


FIGURE 2.2: Nassi-Schneiderman diagram [3]

2.2 3D Software Visualizations

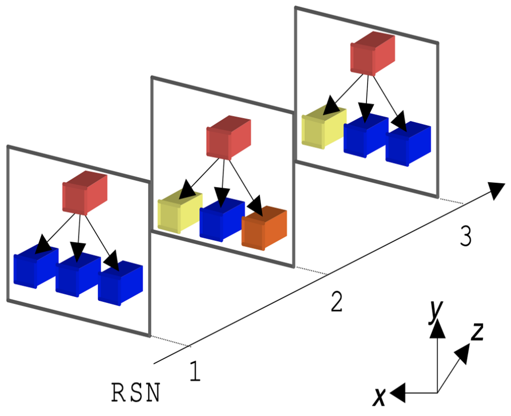
At the beginning of the 90s, 3D software visualization techniques emerged. A third dimension meant having at disposition more space to display information. File System Navigator depicted in Figure 2.3, was developed by Silicon Graphics and was one of the first 3D visualization tools. It was used to visualize the hierarchy of the file system [4]. It gained popularity due to an appearance inside the movie Jurassic Park in 1993.



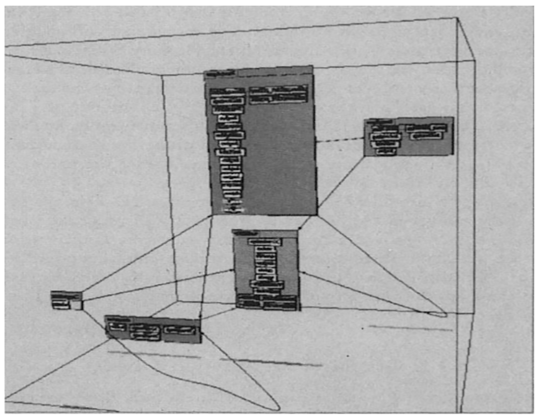
```
Connected to ftp.sgi.com.
220-
                                            ftp.sgi.com

                                          Engineering FTP
                                            Repository
      One Step Ahead                    ftp-admin@sgi.com
------------------------------------------------------------
orporate web site
et the latest in Irix advances
ustomer / developer data transfer
---------------------------------

 ready
ymous
 your complete email address as your pas


 following directories for details.
ious goodies for IRIX and bug fixes.
graphics programs/source from Paul Haebe

icles and compressed groups of articles
icles and compressed groups of articles
ious OpenGL files
ious goodies and bug fixes, unsupported

 support organization
er things from elsewhere

, restrictions apply

r files.
```

FIGURE 2.3: File System Navigator user interface [4].

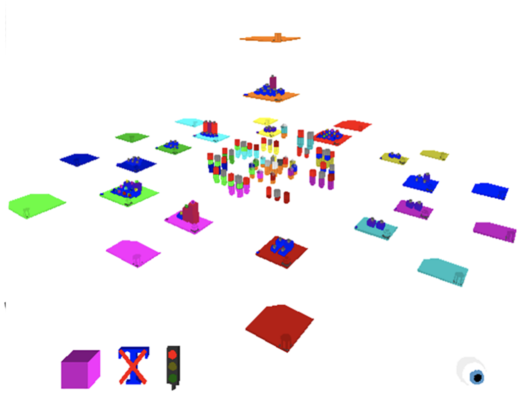
Gall *et al.* created a tool that used both 2D and 3D visualizations (see Figure 2.4A) of software systems [30]. Hardware limitations were quite evident during that decade as visualizing complex information in the 3D space required complex computations. In 1995, Reiss presented PLUM, a 3D customizable engine that displayed a system's information under multiple forms (see Figure 2.4B) [31]. At the same time, Young & Munro started exploring the virtual reality world while visualizing software systems (see Figure 2.4C) [32]. At the beginning of 2000, software visualization gained more attention, with the creation of dedicated venues. Greevy *et al.* proposed TraceCrawler (see Figure 2.4D), a tool that analyzed the interactions between components statically and dynamically [33]. At the time, due to the hardware limitations, one of the challenges was to determine the amount of information to display. This problem was also reflected in software evolution visualizations since the amount of data that needed to be processed was significantly larger, making it difficult to display it all together.



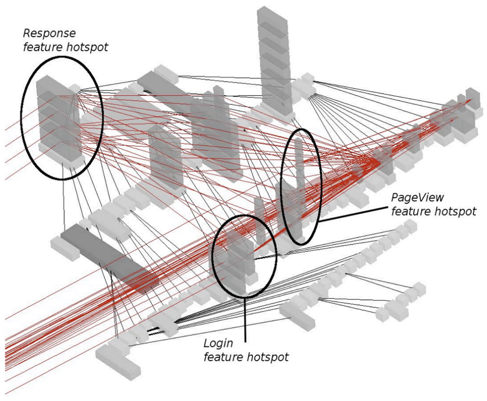
A. Software Release History [30].



B. PLUM [31].



C. FileVis [32].

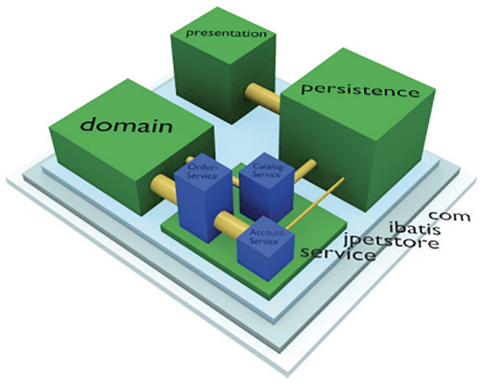


D. TraceCrawler [33].

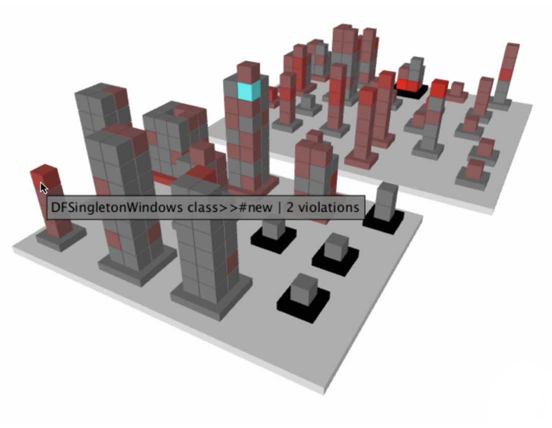
FIGURE 2.4: 3D Visualizations (1995-2006)

At this time, the first web-based visualization techniques appeared. Mesnage and Lanza created White Coats, a tool that visualized software information about systems based on the extracted data derived from revisions in versioning systems [34]. Lungu *et al.* developed the Small Project Observatory, a project that visualized 3D information on the web [35]. 3D, when compared to 2D, permitted the immersion into the data that could be used as a starting exploration point [7]. Knight *et al.* created Software World, where software systems are displayed as cities, with buildings, trees, and streets [16]. Not all the information displayed was relevant, but there was already a mapping between source files, that represented cities, and methods that were representing buildings.

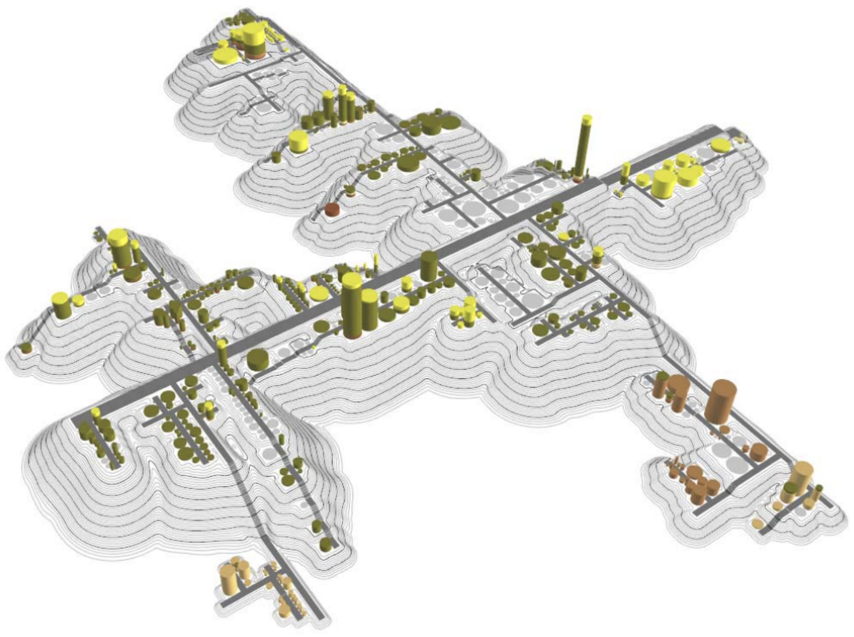
In 2003, Panas *et al.* depicted a software system as a city, with real information about static and dynamic data [36, 37, 38]. Langelier *et al.* presented another landscape metaphor with cities in mind. Verso was the first one that inserted in the 3D plane elements based on specific criteria to have a significant displacement of blocks [39]. In 2007, Wettel *et al.* presented their approach of visualizing software systems as cities. Based on the city metaphor (see Figure 2.10), they aimed to display software metrics in a meaningful way, while keeping the layout of the city consistent with the information and giving viewers a sense of locality in the city [1]. Two years later, they presented another paper in which they used the same tool to analyze and visualize the evolution of the systems [40]. This approach was not resistant to time changes, creating situations where the entire city was moving to another place in the visualization. In 2010, Steinbrückner and Lewerentz, based on the city metaphor, modeled a view that was taking into account also the evolution of software systems (see Figure 2.5C). In this case, time was mapped to the height of the hills on which the class was sitting [13]. As a downside, their approach was not displacing elements on the map in a smart way, creating cities with disconnected districts. By making use of 3D blocks, Fittkau *et al.* presented ExploraViz, a tool for visualizing traces using both 3D and 2D visualizations (see Figure 2.5A) [41]. In 2015, Tymchuk *et al.* proposed ViDI, a tool for quality analysis that made use of 3D visualizations (see Figure 2.5B) [42]. In 2012, Erra and Scaniello proposed CodeTrees, a visualization of the software system under the form of trees [14]. Two years later, always using the tree metaphor, Maruyama *et al.* [43] proposed a 3D visualization that made use of both forests, representing classes, and trees representing information about the class. More recently, Vincur *et al.* presented VR City, a tool that represented a software system in a virtual reality environment [44].



A. ExploraViz [41].



B. ViDI [42].



C. EVO-STREETS [13].

FIGURE 2.5: 3D Visualizations (2010-2015)

2.3 Evolution Models

The first versioning systems appeared during the 70s. At that time, it became clear that there was the need to be able to revert changes that have been done previously to the source code. The first versioning system was SCCS and introduced some concepts that are still used today, such as the possibility to check out a specific revision or commenting on the changes [45]. SCCS made it possible to understand where, when, and what changes have happened over time. The main problem of this versioning system is that the models representing the history were basic, with only some text-lines, added to log files.

Comparing versions is one of the first approaches to understand the evolution of software systems. Demeyer *et al.* used structural measurements to detect renames at method level over two versions [46]. Xing & Stroulia extended this by comparing different versions simultaneously, going more fine-grained in the detection of changes [47], while Antoniol and Di Penta used word similarities to detect renames, split or merge class [48].

When dealing with the evolution of software systems, different approaches have been developed to make it understandable. In the next sections, we will look at some of them.

2.3.1 Evolution Charts

Evolution charts were used to display information about software systems. Figure 2.6 depicts one example of evolution chart. On the x-axis is represented time, while metrics were represented on the y-axis. On the right side, we can see that to visualize multiple metrics, additional charts are needed. Gall *et al.* also implemented an approach to visualize the evolution of software systems [49]. His approach has also been widely used but has the limitation that only one property can be visualized at a time.

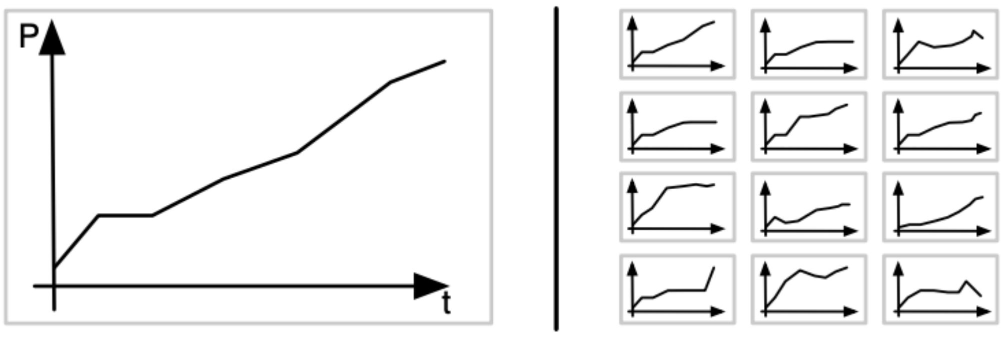


FIGURE 2.6: Example of an evolution chart.

2.3.2 Evolution Matrix

The need for visualizing multiple properties at the same time led to the development of visualization that uses the concept of a matrix. Lanza and Ducasse developed a layout that was able to display multiple metrics over time of a software system (see Figure 2.7)[5]. Every row represents a class while each column represents a version of that class. All entities of the system, represented as a rectangle in the image, can, therefore, represent multiple properties at the same time by using the geometrical properties of the rectangle. At each version, the size of the rectangle can change depending on the changes that happened inside the class.

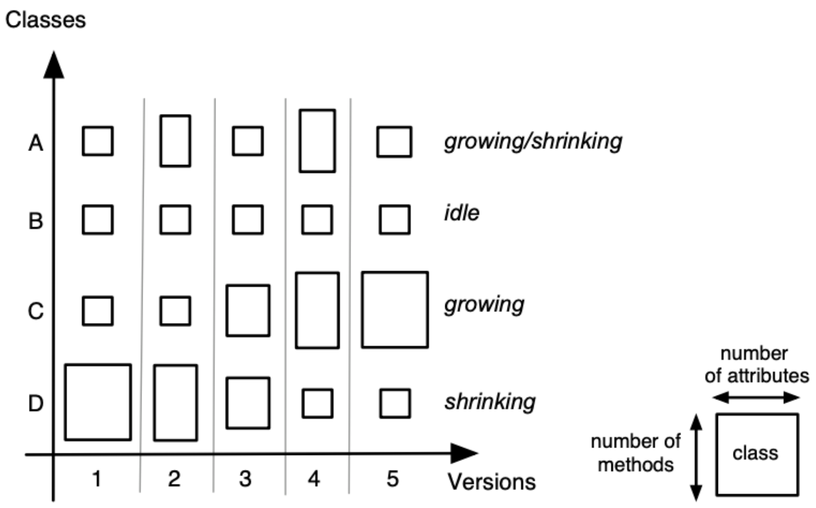


FIGURE 2.7: Evolution matrix in action [5].

2.3.3 Hismo

Hismo [6] is an evolution model that can be easily applied to software systems' evolution due to its internal structure. The core of Hismo is depicted in Figure 2.8 and consists of three main entities:

- *Snapshot* Represents the time in which evolution data is extracted.
- *History* Is the container of a set of Versions. Represents the history of a single entity within the software system.
- *Version* Represents the time and is related to a Snapshot. A Version knows which History belongs to and can exist in only one History.

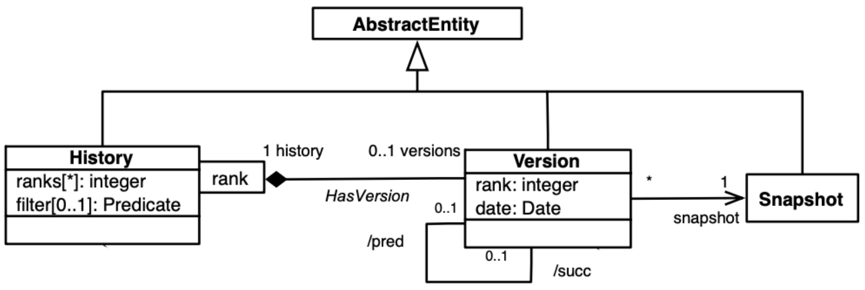


FIGURE 2.8: Hismo core model [6].

In Hismo, the entities are abstract and not tied to any data-models, creating a structure that can be adapted to any entity related system that we want to study. The author of the model itself shows how to apply it to model software systems evolution while using the evolution matrix developed by Lanza *et al.* as a visualization [50].

The model uses the evolution matrix to display the evolution information and has two main concepts: packages and classes. Figure 2.9 depicts Hismo applied to the evolution matrix. In the top part of the image, Hismo is applied to Packages and Classes, while the lower one contains the evolution matrix itself. In Hismo, a row indicates a class history while a column constitutes the version. The whole matrix, therefore, can be visualized as a package history while a complete column constitutes a package version. When dealing with multiple packages, we then need to create multiple matrices, each one containing its own rows and columns.

The model itself is the starting point for the evolution of a software system. In this thesis, we used Hismo, together with the evolution matrix, for developing a new model that is able to store all the information needed for visualizing the evolution of software systems. We discuss this in Section 3.1.

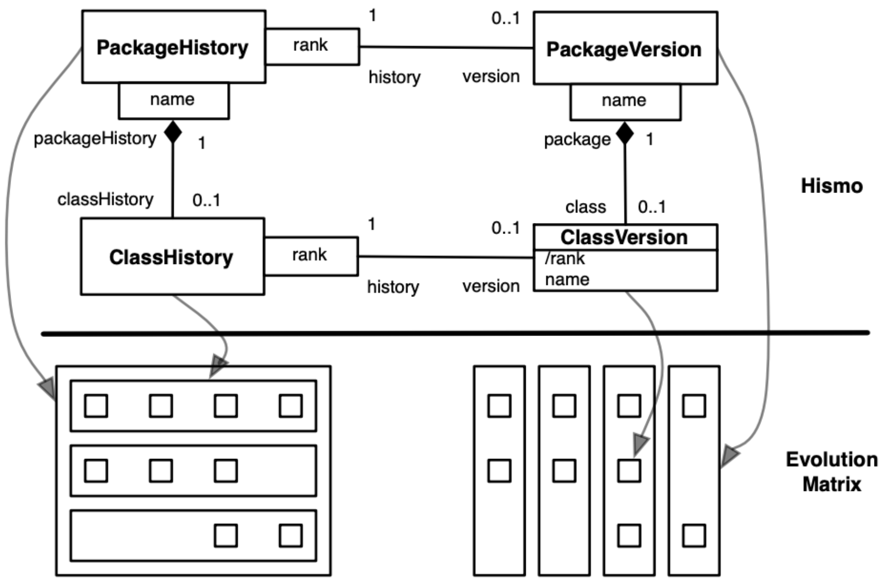


FIGURE 2.9: Hismo mapped to an evolution matrix [6].

2.4 Software Evolution Visualizations

Thanks to free and open-source projects and the popularity of hosted version control sites, the amount of historical data about software systems increased, giving at the same time much larger information about their evolution. Access to this information led to research about the evolution of software systems and their visualization. In 1996, Holt and Pak created the first tool, called GASE, a tool to visualize information about the evolution of software systems [51]. GASE displayed information about the architecture of the system in multiple versions. In 2001, Lanza proposed the concept of *evolution matrix* [52]. This matrix kept track of the evolution of single classes during the software system lifespan and defined the evolution pattern of each file.

In 2004, Fischer *et al.* visualized the evolution of the features of large software systems [53]. One year later, based on the work done by Lanza, Gîrba *et al.* presented Hismo, a tool for modeling software evolution [6]. The meta-model of Hismo is based on the concept of "package history" and "class history" composed by the information of the previous versions. D'Ambros and Lanza in 2006 visualized the evolution of software bugs and the coupling between classes over time [54]. Later, D'Ambros *et al.* proposed "Evolution Radar," a method to visualize coupling between objects [55]. An approach that displayed metric information was then presented by Gonzalez *et al.* [56] They discussed a 2D visualization of software systems that was able to compare multiple metrics at once all in the same view. Alcocer *et al.* then presented the Performance evaluation matrix, which aimed to display software systems' performance during their evolution [57]. Novais *et al.* proposed a proactive and interactive visualization strategy for software evolution [58]. Carol *et al.* proposed another visualization of software evolution called Evo-Clock, an alternative way of displaying a large amount of historical information by making a trade-off between reduced-accuracy for a large amount of historical data and high accuracy of single components [59]. Ivapp *et al.* presented EvoCells, a tool that visualizes changes using treemaps [60]. In addition to the classic 2D visualization, the work done by Sondag *et al.* is interesting to mention, because even if it was not related to the visualization of software systems, they implemented a stable visualization that makes use of treemaps [61]. In their publication, the authors propose alternative ways to construct treemaps for hierarchical data, which algorithm tries to modify the structure as little as possible when its internal data changes. EVO-STREETS, on the other hand, can be seen in Figure 2.5C [13], in this case, the authors decided to map evolution on the height of the hills, where a higher hill represents a more recently modified class. Another application of this metaphor was found within CodeMetropolis [62].

2.4.1 CODECITY

CODECITY is a language-independent 3D visualization tool for large software systems. The authors have selected the city metaphor due to the many similarities it has with software systems. With this metaphor, properties of the software systems such as source code metrics and package locations are mapped to properties of the city. By analyzing extensively multiple projects, the authors have demonstrated the versatility of the city metaphor visualization for reverse engineering, increasing the program comprehension by reducing software systems' intrinsic complexity [7, 20]. CODECITY can be visualized in Figure 2.10, in which Java JDK namespace is analyzed. Districts represent packages, each one containing multiple source files. The building's shape is defined by the metrics extracted by the lines of codes, such as number of methods (NOM) for the height and number of attributes (NAM) for the width and depth.

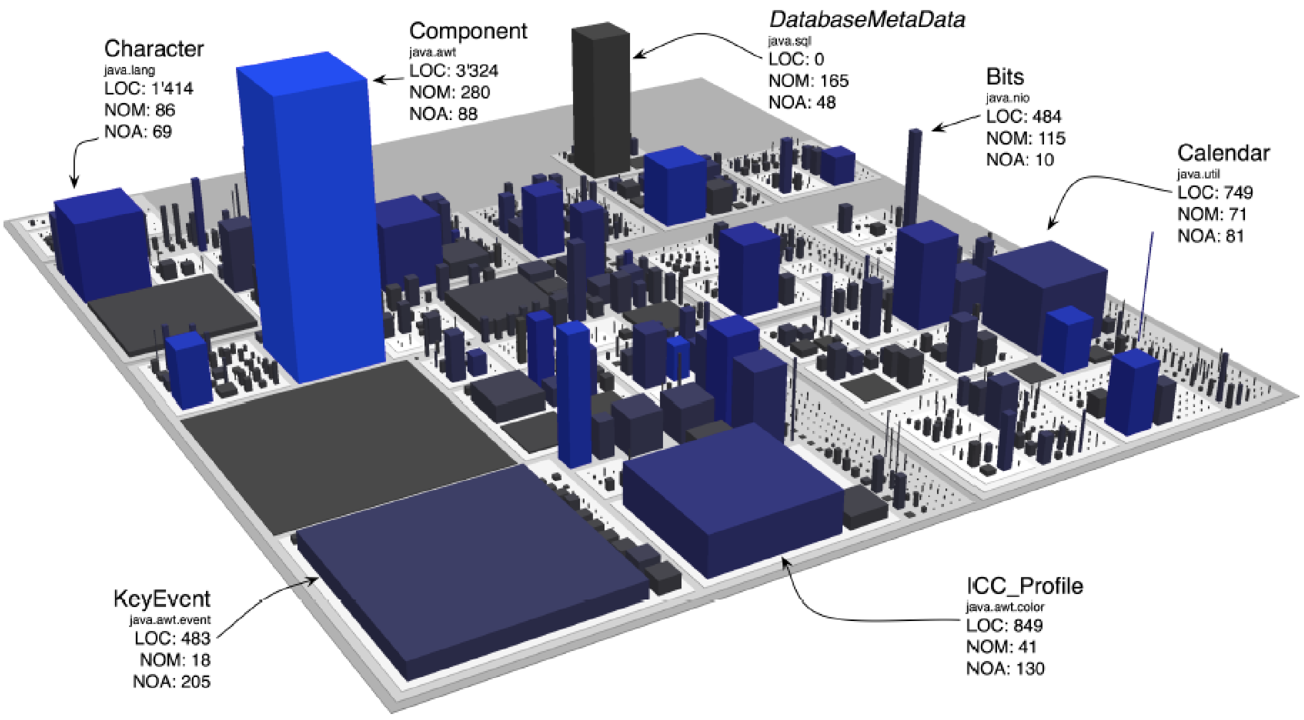


FIGURE 2.10: CODECITY on JDK's java namespace [7].

2.5 Summary

The previously mentioned techniques mostly focus on giving information about software systems, but their visualizations did not consider the evolution of the software system itself. Regarding software as cities, CODECITY can display the evolution of a system, but the visualization is not time resistant, with building positions changing over time. EVO-STREETS has the same problem. Even if the history is mapped on hills, the visualization of the evolution changes the position of streets and blocks, adapting them based on the new information.

Compared to these techniques, we propose a novel approach to display software system information consistently over time. The city's initial layout will be computed by taking into consideration the history of the software system so that building positions are consistent over the complete evolution visualization.

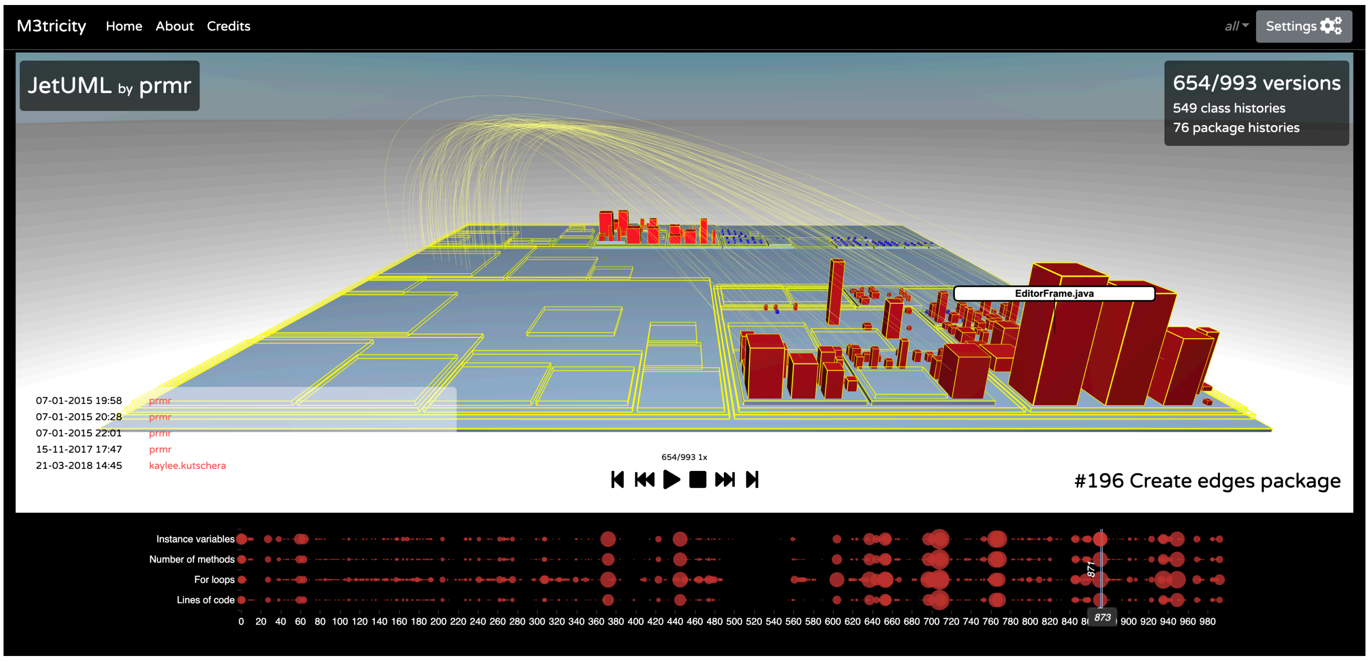
Chapter 3

M3TRICITY

In this chapter, we present the approach and the tool we developed to visualize software systems using the city metaphor and the evolution-resistant layout. In fact, despite its popularity, the city-metaphor falls short when depicting the evolution of software systems, which results in buildings and districts moving around in unpredictable ways throughout the revisions of a system. Part of the problem is that current approaches for visualizing software systems do not use the information produced by the versioning systems, creating views in which each revision potentially represents a new city, and therefore a new layout. In generating a layout resistant to the evolution of the system, we want to reduce the dynamism of the city by decreasing the number of movements inside it. At the same time, we want to represent the evolution as accurately as possible. All this to make the visualization itself more understandable to the viewer.

For this reason, we present a novel approach to visualize software systems as evolving cities that treats evolution as a first-class concept. It renders with fidelity not only changes but also refactorings in a comprehensive way. To do so, we developed custom ways to traverse time. We implemented our approach in a publicly accessible web-based platform named M3TRICITY depicted in Figure 3.20.

In Section 3.1, we discuss the approach used to model and visualize the history of software systems. In Section 3.2, we discuss the tool that implements our approach, by looking at its architecture, at how history was modeled but also by looking at features and user interface. Finally, in Section 3.3 we discuss the limitations that have been discovered, together with some extra work that should be explored to achieve a better structure for the history.



07-01-2015 19:58 prmr
07-01-2015 20:28 prmr
07-01-2015 22:01 prmr
15-11-2017 17:47 prmr
21-03-2018 14:45 kaylee.kutschera

FIGURE 3.1: Evolution of a city with M3TRICITY.

3.1 Approach

Software systems comprehending is difficult, mostly because of their size and complexity [7]. When dealing with large systems, operations such as code refactorings can be used to decrease their complexity. Software systems visualization is an approach that can be used to comprehend their functioning. Concerning their evolution, current approaches represent this information inconsistently, generally by displaying the evolution under the form of multiple single snapshots of every file in the system, where information between two snapshots is missing. Visualizations in 2D, as opposed to 3D, do not give the possibility to explore the environment due to the intrinsic limitation the flat world implies. At the same time, they also limit the amount of information that can be displayed at once. 3D visualizations, on the other hand, let the viewer immerse in environments with natural interaction. At the same time, they give much more possibilities to develop new features. However, 3D visualizations also have problems such as occlusion, perspective foreshortening and difficulty navigating [63]. Current solutions in software visualizations provide little interaction; therefore, the user can not explore the view. Moreover, interaction with the objects is absent or often difficult to obtain, typically requiring a multi-step procedure to acquire and extract information. 3D visualizations also let objects move freely within the scene, giving more in-depth information about what is happening.

By using 3D scenes, we can visualize software systems using the city metaphor. Cities and software systems have many things in common, making it possible to easily map the two domains. For example, cities are constructed incrementally, just like software systems are being developed over time. Similarly, they are composed according to a hierarchical structure, i.e., buildings inside a district vs. files inside a directory or classes in packages. Using this simple mapping, we can construct a city using the software system's information.

Our approach provides an evolution-resistant layout, that places elements within the view using the information extracted from the history of the software. Using this solution permit to place elements in the view so that their initial placement is consistent over each revision. Particular attention must be given to cases where a file has been renamed not to lose any kind of information about its evolution.

Combining the resistant layout with the evolution model is essential to give more knowledge about software systems. In fact, the resistant-layout depends on the information that the software system's evolution model provides. In the next sections, we present our approach to model the evolution and visualization of software systems using these two concepts.

3.1.1 M3TRICITY Evolution Model

To model the evolution of software systems, we developed a new evolution model that can be seen in Figure 3.2. The ideas of our model are based on Hismo, an evolution model for software systems developed by Tudor Gîrba.

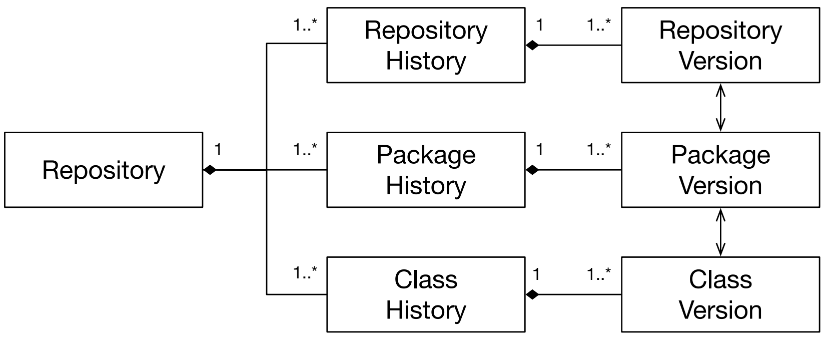


FIGURE 3.2: Evolution model of M3TRICITY.

The need for creating a new evolution model comes from the fact that Hismo was developed on the Subversion (SVN) versioning system.[1] In SVN, a revision represents the state of the whole system, i.e., a commit with changes of only a few files will increase the revision of the whole system. In Git,[2] only the modified files would get the version field updated. This notable difference influences our model too. The author created Hismo based on three main entities which architecture can be seen in Figure 2.8:

- Snapshot. Represents the time, it contains multiple versions of the software system.
- History. It is a set of versions. History represents a specific entity's evolution that might be, for example, a class or a package.
- Version. It is part of a History and represents a specific revision over the evolution of that entity.

Our model consists of Histories and Versions, but the concept of Snapshot is dropped. The model has been developed with the idea that a Repository has its own story, consisting of multiple versions. In this way, we wanted to make more consistent the model proposed by Gîrba *et al.* for software systems. With the new model, everything is represented as a tree of histories referencing versions, where on top, we have the RepositoryHistory node. By making these changes, we made the model more suitable for describing software systems.

The repository entity is used to access all the histories extracted from the software system. The RepositoryHistory follows the same logic of PackageHistories and ClassHistories, where each has a series of versions. In our case, a RepositoryVersion is defined as a series of changes during a commit.

The newly derived mode consists of one core entity, Repository, and three sub-entities:

- RepositoryHistory. A repository history is a single entity that represents a repository evolution. It has some basic properties, such as the repository's name and owner, together with a list of RepositoryVersions. Using this as an entry point for any repository history, we can traverse its evolution over time.
- PackageHistory. Similarly to a RepositoryHistory, a PackageHistory contains a list of PackageVersions. By traversing this node, one can reconstruct all the changes that happened to a specific package. A package is identified by the path representing it.
- ClassHistory. It is the leaf of all histories. It represents a single file history and therefore is used to construct the evolution of a class.

The structure defined above is the base for traversing histories that are extracted from the versioning system. Thus, histories represent unique objects that can evolve, such as classes, packages, and the repository. The structure, as described above, does not give information about how versions are defined. Versions are not always the same, and each RepositoryVersion, PackageVersion and ClassVersion entity needs to be slightly different to have a complete representation of their evolution. In M3TRICITY they have been defined in the following way:

- RepositoryVersion. It is part of a single RepositoryHistory and has a unique number representing its version. It points to a collection of PackageVersions.
- PackageVersion. It contains a list of ClassVersions. In this way for each PackageVersion we can navigate to the classes contained in that package at one specific version.
- ClassVersion. Is the actual class or file contained in the software system and is the base entity for defining packages and repositories versions entities.

With this structure, when we traverse the list of RepositoryVersions found within a RepositoryHistory we can find all the changes that happened over time.

[1]See `https://subversion.apache.org/`
[2]See `https://git-scm.com/`

Building Histories

Defining History entities is not straightforward in all cases. ClassHistory, PackageHistory and Repository-History all need to be uniquely modeled due to their internal structure. The simplest model of history is the class. The information that we can extract for the Git only regards the changes at the class level, this all histories will be based on class changes. Git provides the following information about file changes:

- Add. A file is added to the system.
- Delete. A file has been removed from the system.
- Modify. File content is modified.
- Copy. A file has been copied to a location (no information whether the location or the filename has been changed).
- Rename. A file path has changed (whether the location or the filename has been changed).

From this, we can extract additional information for each file, such as its parent folder (defined by the complete path except the filename) and also the filename itself. Given a set of modified files, for each operation, we can determine how the change impacted the history of an entity.

When building histories, everything starts with An *add* operation. When a file is added to the system, it will be inserted in our model as a new ClassHistory. Regarding PackageHistories, a file addition does not always represent a new PackageHistory. A PackageHistory might be already existing if a file within the same package has been added previously. In this case, the new file will be simply added at the latest PackageVersion of that PackageHistory. A *delete* represents an end in a ClassHistory. However, regarding PackageHistories, it only represents an end if all files contained inside the current PackageVersion have been renamed or deleted. If not, then the package cannot be considered deleted. *Modify* is the most straightforward operation, in which the new version is simply added in any case to the history that has been previously constructed when the first *add* operation has happened. Regarding PackageHistories, if all files contained in a specific version have been renamed, it represents a continuation of the history. In contrast, if only a few have been renamed, it is a history split. On the other hand, *copy* also represents a history split since a file starts to appear in multiple locations.

Figure 3.3 shows an example of how a ClassHistory, represented as a row in the evolution matrix, is created. The first operation that needs to be detected inside the versioning system is an *add*. With this operation, we know that a new file has been added to the system and therefore, a new ClassHistory needs to be created (purple). At version two (r2), the file is modified, while at version 3 (r3), the file is copied. A copy is a split within the linear evolution of the history of the file. Those cases are rare and are the only ones in which the row of an evolution matrix is split into two (pink). However, conceptually the history is still a linear succession of ClassVersions. Version 4 consists of a *rename* operation for the pink entity, but in the matrix, this does not affect the evolution. The change in the name of the file does not imply a change in the history. A rename is only used to keep track of the entity's actual name and might be used for understanding the location (package) of the file. Concluding, the *delete* operation consists of an end to the evolution of that history when no copies have been created. If copies are found, it merely represents a deletion of a child.

Detecting Changes: Merging Package Histories and Versions

To better understand changes within the development (for example, to handle renamed or merged packages), in addition to Git we also use tools that compute a similarity score between commits.

The simplest history model does not consider that package histories and class histories can be merged, a crucial consideration for having a correct representation of the evolution of a software system. A file's

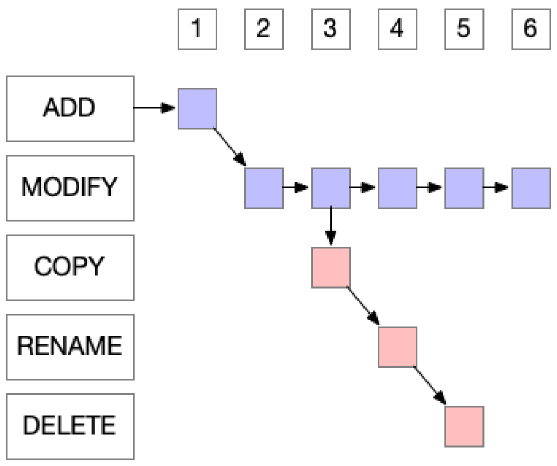


FIGURE 3.3: Example of ClassHistory in M3TRICITY with a copy operation.

history can be seen as a linked list, where a file gets inserted, modified up until no other elements are added. Rename happens less frequently than modify, addition, or deletion; still, they represent a crucial part of the system's history. When a renamed is detected, we can see this as a simple addition to an already existing history.

In our model, for simplicity, during the first pass every time we detect a new file path, we create a new history. This operation results in creating unique histories, each one identified only by its complete path. At the same time, we track every change that happens at name level, like renames, and we maintain a map representing the evolution of the name itself. As soon as we finish creating histories, we can merge them based on the information extracted from the versioning system and store in our map. Concluding, we can merge histories, creating a linear succession of events that represent the complete evolution of a file. Figure 3.4 show this operation in practice. On the left, we see the evolution matrix without information about renames. History *h2* and *h3* are not linked and they represent two different files. On the right, we can see what happens when information about renames is added. History *h2* is merged with *h3*, and *h3* being created after *h2*, it ceases to exist.

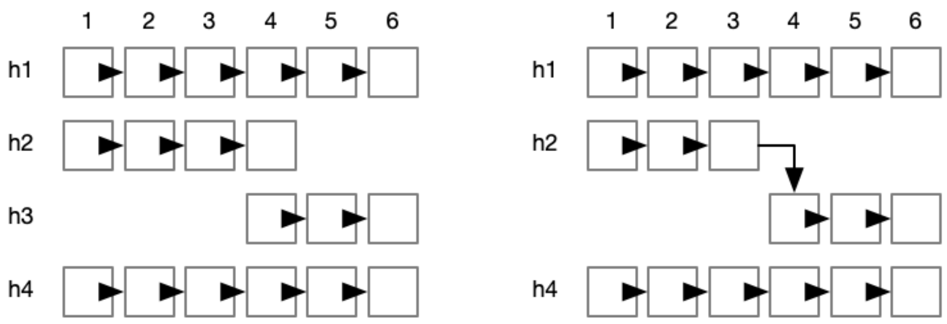


FIGURE 3.4: Example of merged ClassHistories: two separated histories are merged at version 4 due to the presence of a *rename*.

PackageHistory entities are much more challenging to define and represent within the evolution matrix. Their creation happens when one or more files are added within a specific path that not existed before in the history. After the appearance of the first source file in a folder, we need to keep track of every new addition to the same location, together with all modifications and copies. To detect *PackageHistory* deletions, we need to find a PackageVersion in which only *delete* and *rename* operations are found (rename at package level and not a filename level). Moreover, new packages might be added to the path itself, therefore creating sub-packages. The newly created history will not contain only a single reference to a file as it happens to a ClassVersion, but each PackageVersion will reference multiple ClassVersion entities. However, we need to consider that renames might end up in different locations and in different packages. As an example of history split, package A contains files A1, A2, A3. During a project refactoring, A1 is deleted, and A2 is moved into a new package B then A3 into a new package C. In this case, we can state that the history of A has been split into two more histories, B and C. This lead to another problem with the concept of history. History is unique for a set of files, where each history represents the evolution over time of one or more entities, but connecting the history of B and C with the history of A is conceptually wrong since packages B and C are new and created after the rename operation.

A PackageHistory is defined as a series of PackageVersions, each containing ClassVersion. However, nothing guarantees that in the history of a ClassVersion, it will always belong to a single PackageHistory. ClassHistories can contain renames representing a move from a package to another. For this reason and the sake of the simplicity of the model, we create a new History for every new package or folder that is found. Figure 3.5 shows the multiple possibilities that happen when dealing with PackageHistory entities, as we can see the amount of complexity that this type of entity can achieve is high due to all possibilities that can happen. At version four (r4) *h2* is renamed to h3, meaning that all files contained at version r3 of h2 have been moved to a new location. Still, the same files that have been previously moved suffer the same fate another time at version six (r6). Differently from before, the files of the h3 package are divided into two different locations. This situation leads to some complexity in defining what a package's history is. In our model, the PackageHistories can be linked together by looking at the internal movements of the files, but this creates situations in which distinguishing one history from another can be difficult.

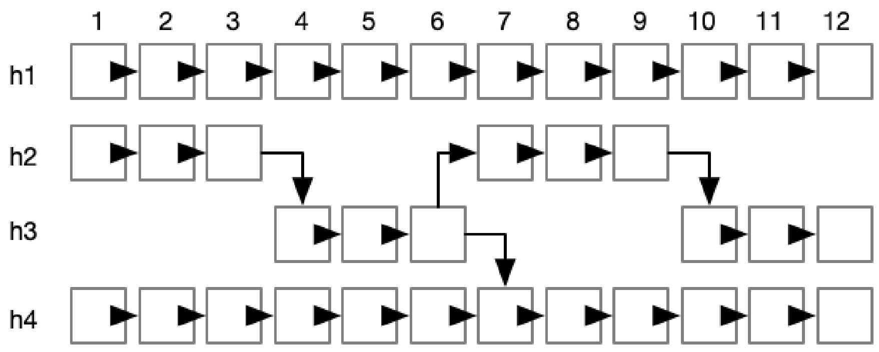


FIGURE 3.5: Example of merged PackageHistories.

Histories are represented as unnamed entities as we cannot define their name due to the Git operations cited above. We cannot merely say History A represents the "Activity.java" file, but either we can say that a specific history represents a file that was named "Activity.java" for a particular time during its evolution.

Histories as Unnamed Objects

A significant difference in our model compared to Hismo and the evolution matrix is that histories are represented as unnamed entities. That is, it is impossible to define a history simply by a filename or a path. Paths and filenames are only represented inside versions and not inside histories because over the evolution of a software system, files and packages can be renamed and, therefore, it would be wrong to limit a history to a specific path. This concept is fundamental and makes it possible to model dynamic histories. Without the concept of unnamed objects, we could not connect two histories such as it happens with a *rename* or *copy* operation, making it too tied to the filename or path that is representing.

Figure 3.6 shows an example of how unnamed entities work. Inside the box, we can see an example of a PackageHistory. The history itself can not represent a single path or package due to the multiple possibilities that might happen to its name. The image shows that when the first file is added to the path

`/test/ui/tools`. As soon as a *copy* operation occurs, the history itself consists of two different names, `/test/ui/tools` and `/test/ui/utils`, generating a history that is represented with two different names. If a *rename* operation also happens, the history will be represented by three different paths.

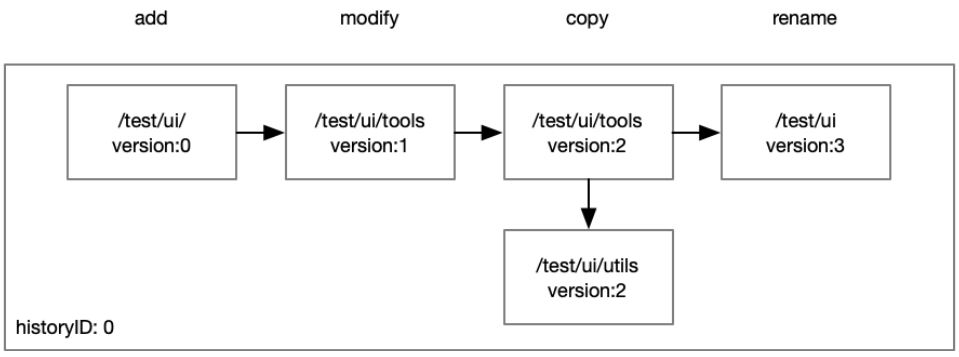


FIGURE 3.6: Example of a PackageHistory with multiple filenames.

3.1.2 Visualization

M3TRICITY is a web application for visualizing software metrics evolution. It uses the city metaphor, a way to visualize software systems, that has been widely explored in the past years. We base our visualization on CODECITY [7]. Due to its limitations when visualizing the evolution of software systems, we developed a history-resistant layout that leverages the evolution matrix to place objects within the view to avoid the viewer having wrong interpretations of the evolution. To understand the approach, we start by describing CODECITY, a visualization for software systems that uses the city metaphor. After that, we will give the notion of districts and buildings. In the end, we will discuss the history resistant layout, the core section of the new visualization and of this thesis.

Software systems as cities

To understand our visualization, we start by describing CODECITY, a visualization that uses the city metaphor. A city has many things in common with software systems. First of all, both can be seen as a hierarchy of two types of entities:

- Districts and folders: both can be seen as a container for other elements. In a city, a district contains multiple buildings, while in a software system, a folder can contain multiple files. Districts and folders can contain recursively other entities of the same type.
- Buildings and files: are the smallest elements within cities and software systems. Buildings define the landscape of a city, like source files define the main characteristics of a system. Properties of a source file can be mapped to a building, for example, by letting the height of a building be proportional to a specific metric.

In CODECITY, this similarity has been used to develop a visualization for software systems in which cities and software systems are linearly mapped to transform a folder containing multiple files into a district. In contrast, file metrics are used to shape buildings. The authors did not stop there, but also searched for proof of utility for this visualization, by validating through controlled experiment that visualizing software systems using the city metaphor was, in fact, useful to understand the system itself [7] .

The key idea of CODECITY comes from the fact that we need to learn from software systems, and the internal complexity of a software system can also be found in cities [64] . By creating a visual mapping,

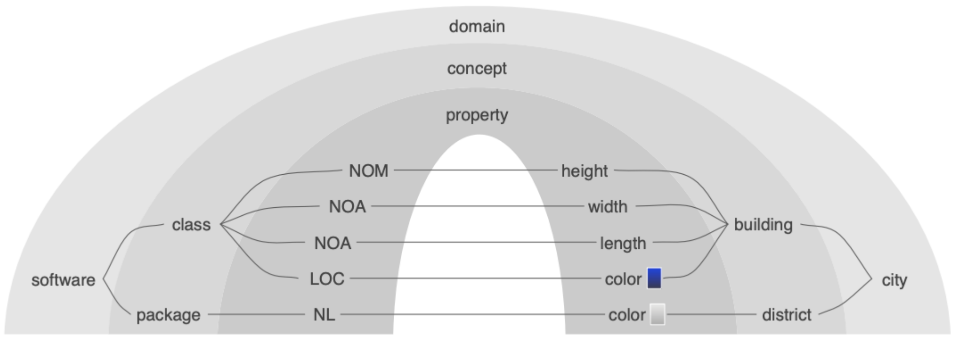


FIGURE 3.7: Mapping from software system to cities in CODECITY [7].

the amount of information that needs to be processed is reduced. Moreover, the similarities among software systems and a city let the user explore it as it would do in real life. To create a meaningful mapping, CODECITY approach used a static mapping between the entities that compose the city to those that compose software systems, such as districts and buildings. Figure 3.7 depicts the mapping used by the authors. We have the two main entities at the domain level, *software systems* and *city*. At concept level we have *class* and *package* for software systems, while for cities we have *building* and *district*. At the lowest level, we have the properties. As an example, we have *number of methods (NOM)* for a class and *height* for a building. In addition to mapping numeric values to shape properties, colors are also used for highlighting.

M3TRICITY differently from the approach used in CODECITY, let the user select the metrics to be visualized, modifying the shape of the city.

The layout of the city is constructed based on the Rectangle Packing Layout for a collection of elements. The idea behind this algorithm is that the structure representing the city is a tree, where each node contains other packages or classes. The algorithm works recursively. It starts from the leaves and goes up in the hierarchy in a post-traversal order laying out all the elements. Figure 3.8 shows the result of the algorithm applied to ArgoUML.

One problem of CODECITY is that it did not use the evolution data to display a city where buildings and districts do not move every revision. In fact, over the evolution, objects such as buildings and districts were moved around, possibly leading to wrong interpretation of the visualization. For this reason, we developed a history-resistant layout.

Bin-packed history resistant layout

The history-resistant layout is used to consistently place elements within the view over the evolution of the software system. The main difference is that it is using the evolution matrix to discover the final size of each component of the software system to find the best placement within the view while minimizing the total area occupied by the city.

The construction of the layout starts with analyzing the history of each class (row in the matrix) found within the software system evolution. This information is represented directly through ClassHistory entities that are stored within all Repository models. By iterating over all versions of class histories, we understand how the class evolved over time (Figure 2.7). In this case, since we want to place the object in a specific position, we will be looking for the largest evolution version among all. By looking for that specific version, we find the maximum size that the class will reach over time. This information can be used to define a placeholder in the layout for that class, which is the maximum width and depth that the building will reach over all revisions. Software systems do not only contain classes but also packages. To maintain the resistant layout, additional steps need to be done to determine the final size of the packages.

To define the layout of a package in the view, we need to iterate over all PackageVersions of it. For each version, we then look at all the ClassVersion that the version itself contains. While doing this operation,

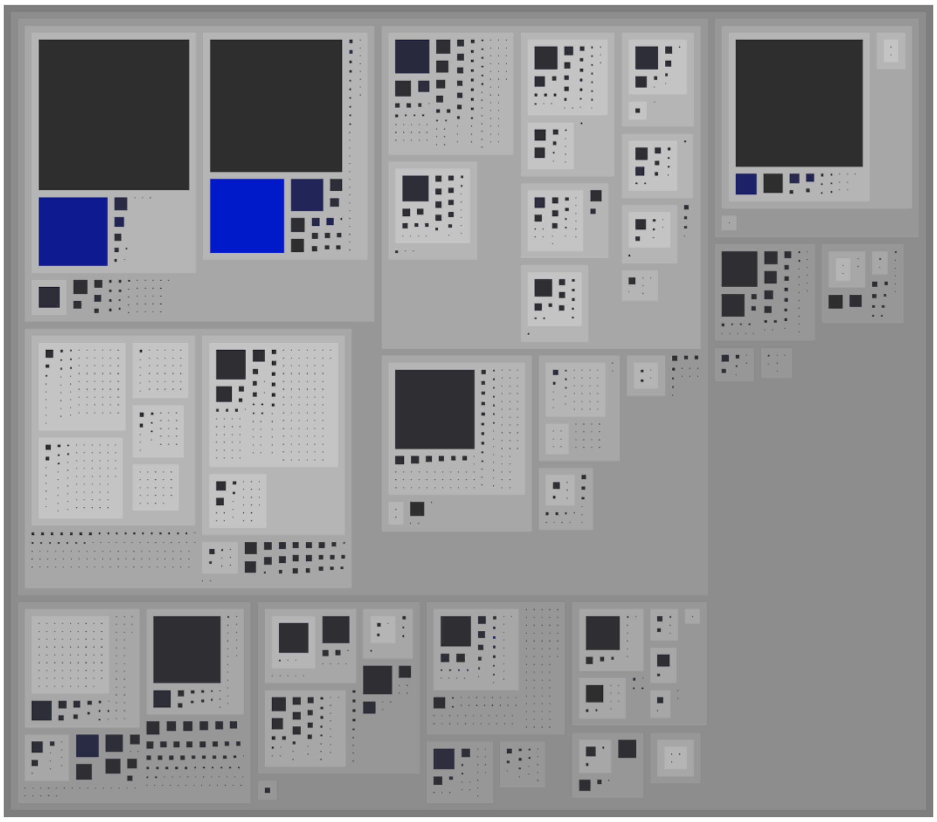


FIGURE 3.8: CODECITY layout algorithm of ArgoUML.

we need to extract and keep track of each maximum ClassVersion contained in each package. In the end, by adding all the information collected about the ClassVersion, we know the size of all packages and classes within the package itself. This operation requires the creation of an entity called PhantomVersion, which represent packages in the recursion. This entity has much less information, and it is used only as a placeholder for packages. It can be seen as a class in the system, but in reality, it contains other classes. PhantomVersion is used to simplify the creation of the layout because, for each nesting level, we only deal with entities as they were classes and not as they were nested packages. At the end of this first step of the algorithm, we have a list of versions for each package, each one of them representing the maximum evolution of that class. This information is then passed to a second algorithm that assigns each class and packages to a specific position inside the layout. In this thesis, we use a simple implementation of a bin-packing algorithm. The algorithm's output is a list of History entities each one with a specific position within the layout.

Figure 3.9 shows the history-resistant layout (left side) as opposed to a standard bin-packing layout (right side), like the one used by CODECITY, where the goal was to optimize space.

In the figure, each row is a revision of a system. We use blue to denote classes and light blue to denote packages. Consider **C**, for example, which is not present in revision **R1**. In the bin-packing layout, **C** appears in **R2**, and it is placed according to some heuristic (e.g., to optimize space). In **R3**, however, it is moved. Using our history-resistant layout, the final space and position of **C** is kept free in **R1**, then occupied with the first instance of **C** in **R2** and assumes its final larger shape in **R3**. The "jumps" we mentioned before are visible in the bin-packing layout on the right side, as its goal is to use space as efficiently as possible. Therefore, the package containing classes **A** and **B** in **R1** is only as big as needed and grows in the successive versions. With the history-resistant layout, that package uses the space it will ultimately need, already starting from **R1**.

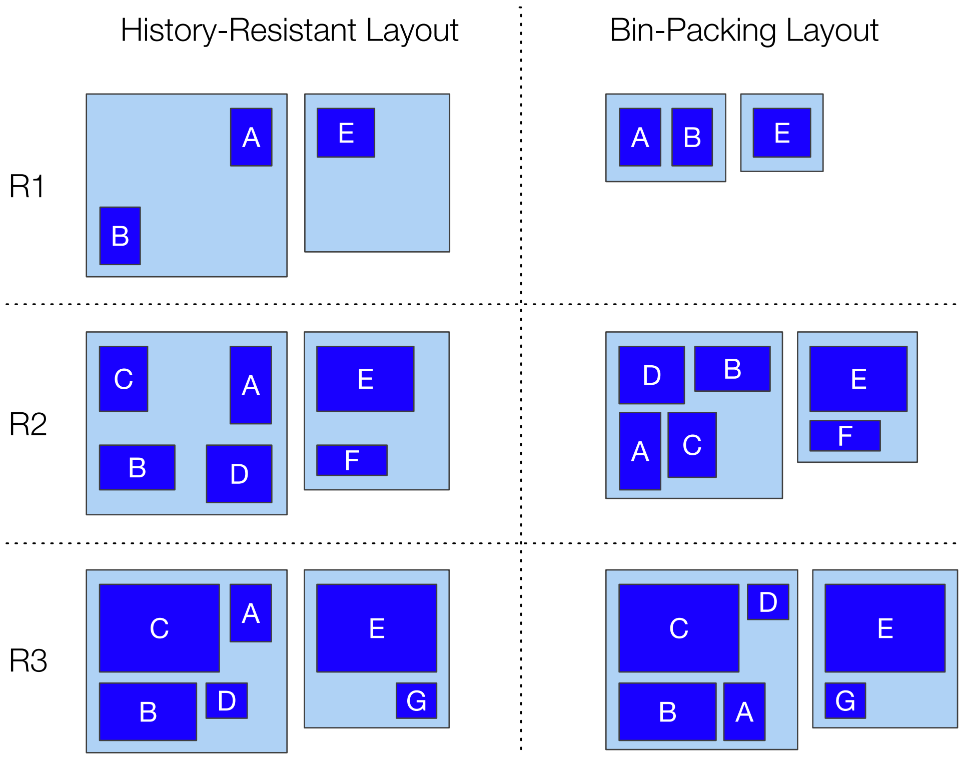


FIGURE 3.9: History-Resistant Layout vs. Bin-Packing.

The overall effect of our history-resistant layout is, therefore, a more robust and less jumpy visualization from revision to revision, making it easier to follow the evolutionary process, as we illustrate in Section 4.

3.2 Tool

In this section, we discuss the architecture of M3TRICITY and the design choices that have been made throughout its development. In Section 3.2.1, we describe the architecture of the tool by looking at the backend and frontend. Finally, in Section 3.2.2 we present the user interface of the tool.

3.2.1 Architecture of M3TRICITY

The architecture of M3TRICITY consists of a backend and a frontend. The application itself is accessible through a web interface and can be used to analyze and visualize Java projects. Figure 3.10 shows a high-level diagram of the application.

The backend is written in Java and built using Spring Boot[3] that is Spring's convention-over-configuration [65] solution for creating stand-alone, production-grade Spring-based applications. It consists of two core modules, History-builder and View. The History-builder is responsible for cloning the repository locally and initiating the analysis of the history. This module is the only one that requires access to the outside world as it needs to contact Git. On the other side, the View is responsible for preparing the data for users who want to visualize the evolution of a software system. This module, given the evolution-matrix of the system, generates a layout that is resistant to the evolution. In the end, the backend will send the required data to the frontend. In addition to the two core modules, there is also a third one that is responsible for providing public endpoints.

[3]See `https://spring.io/projects/spring-boot`

The frontend application is developed in Vue.js[4] an open-source JavaScript framework for building user interfaces and single-page applications. The frontend communicates with the backend through REST endpoints together with WebSockets. The frontend also has settings that can be personalized and a chart that shows where changes have been made over time.

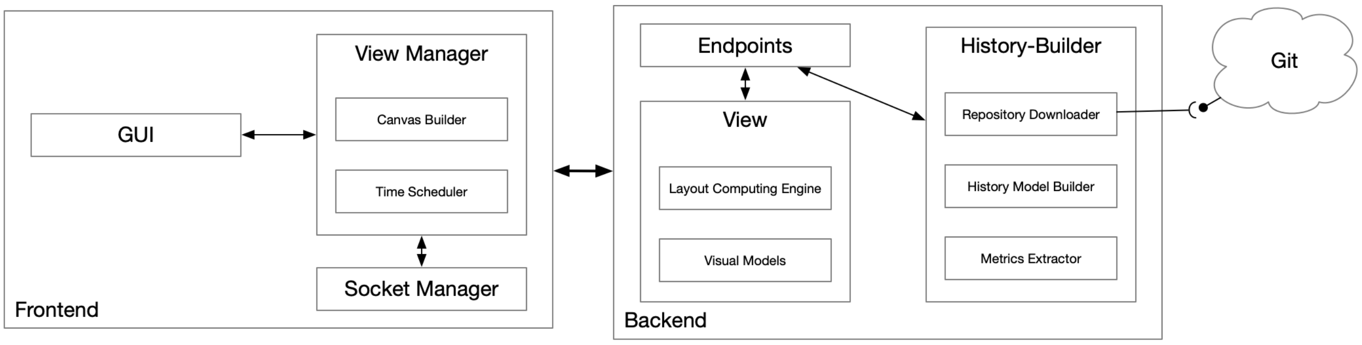


FIGURE 3.10: Architecture of M3TRICITY.

Backend

The backend service is implemented using Java.[5] We also used Spring Boot, a Spring-based tool for developing web applications efficiently. The backend is developed using two main technologies for communication: Rest and WebSockets.

REST API This type of communication is used in M3TRICITY for retrieving information based on the users' interactions. In our application, two main HTTP methods can be used: GET and POST, where the first one is used to access data and the second one to insert it. The endpoints can be divided into three different sections.

Home: are the ones used to initiate a new repository analysis and to retrieve the analyzed repositories, together with their status:

⇒ `POST /api/analyze` is used to start a new analysis and only requires an URL to a Github repository.

⇒ `GET /api/home/listing/repositories` returns a list of repositories that have been analyzed or are in the process of being analyzed.

General: when a repository is selected for visualization, interaction with the canvas acts on the following endpoints.

⇒ `GET /api/repository/{owner}/{name}` uses the information provided by the two parameters to retrieve some basic information about the repository that will be then used for constructing the city. The type of information is mainly numeric and textual. The first one is used to give visual information about the evolution while the textual one is used for configuration purposes.

⇒ `GET /api/repository/{owner}/{name}/{commitHash}` is used to retrieve information about a specific commit in a repository. Each commit contains some basic information such as time, author and a message.

⇒ `GET /api/repository/{repoOwner}/{repoName}/commits/{page}` this endpoint list all the commits of a repository. It uses paging as the amount of commits can be large.

[4]See https://vuejs.org/
[5]See https://go.java/

⇒ `GET /api/repository/{repoOwner}/{repoName}/{version}/terminal` this endpoint, given a version, return information about authors of the current, previous and future commits.

⇒ `GET /api/class/history/{owner}/{name}/{id}` endpoint is responsible for retrieving information about class objects, that might be any type of file existing in a specific repository. It requires an Id, the repository owner and a repository name. The fields are used to access the in memory-database through an interface. After retrieving the object, it is converted to a DTO object through a mapper that only contains the necessary data.

Evolution timeline: is the element responsible for visualizing where changes have been made over the software system's evolution. The endpoints return a list of numeric values for each version and each metric. Higher values mean more changes involving that metric.

⇒ `GET /api/repository/{owner}/{name}/timeline/series` it returns a series of values for each metric. The values are normalized between 0 and 1 and represent the amount of changes for each version.

⇒ `GET /api/repository/{owner}/{name}/{version}/class/timeline/series` it returns a series of values for a specific class, representing the changes over time of that class.

⇒ `GET /api/repository/{owner}/{name}/{version}/package/timeline/series` it returns a series of values for a specific package, representing the changes over time of that package.

⇒ `GET /api/repository/{owner}/{name}/timeline/series/{mode}` it returns a series of values but grouped by day, week, month or year.

⇒ `GET /api/repository/owner/name/version/class/timeline/series/mode` similarly as before, it return a series of values but grouped by day, week, month or year for a specific class.

⇒ `GET /api/repository/{owner}/{name}/{version}/package/timeline/series/mode` it return a series of values but grouped by day, week, month or year for a specific package.

Communication using WebSockets is done through the following endpoints:

⇒ `PUBLISH /api/history/evomatrix/versioned/start` is responsible for initializing the layout of the city.

⇒ `PUBLISH /api/history/evomatrix/versioned/update` is responsible for updating the view with the information of a new version.

History-builder Module The history-builder module is responsible for analyzing software system histories and constructing their evolution. It is composed of three different sub-modules and two core entities using them. Figure 3.11 shows the composition of this module.

M3ConstructStarter is the service that is called directly from the API endpoint `/api/analyze` and is responsible for initiating the download, setting up the structures that will store the histories together with their linkers. This component constructs the RepositoryHistory and save it in the memory. *CommitWalker*, together with *DiffResolver*, are used to iterate over the commits using JGit.[6] DiffResolver makes the diff between the two commits while also using the RenameDetector, which is responsible for detecting renames and copies within two commits. This solution, although it is not perfectly precise, aims to solve some problems with Git internal solution that not always detect these two cases safely. This component is also responsible for generating all the version entities. Versions are derived directly from the changes of a commit; therefore, every change in a commit represents a new version.

In the constructor module, we also have three other sub-modules: *downloader* that is responsible for creating a local copy of the remote repository, *metric* that uses srcML[7] to extract and compute the source

[6]See `https://www.eclipse.org/jgit/`
[7]See `https://www.srcml.org`

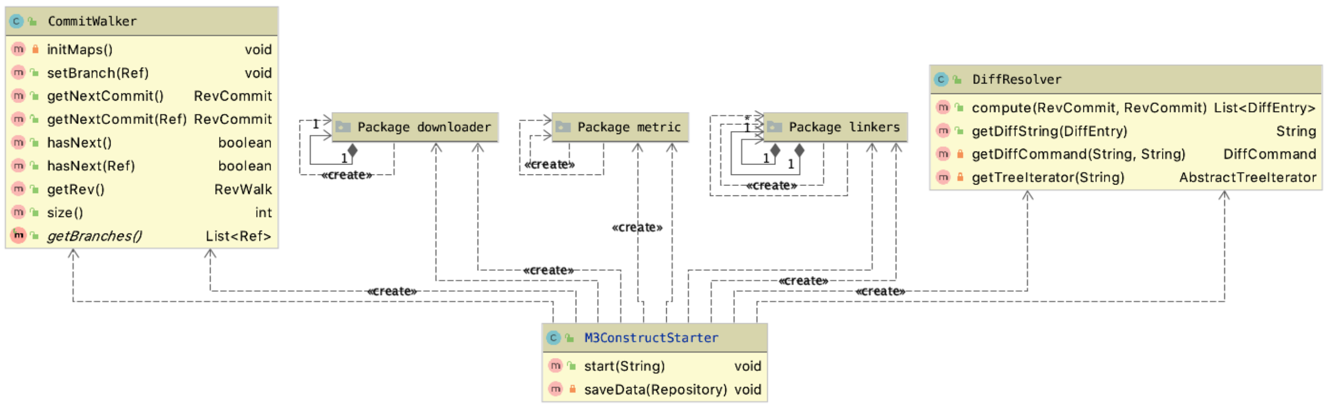

CommitWalker
initMaps() void
setBranch(Ref) void
getNextCommit() RevCommit
getNextCommit(Ref) RevCommit
hasNext() boolean
hasNext(Ref) boolean
getRev() RevWalk
size() int
getBranches() List<Ref>
Package downloader
Package metric
Package linkers
«create»
DiffResolver
compute(RevCommit, RevCommit) List<DiffEntry>
getDiffString(DiffEntry) String
getDiffCommand(String, String) DiffCommand
getTreeIterator(String) AbstractTreeIterator
M3ConstructStarter
start(String) void
saveData(Repository) void

FIGURE 3.11: Class Diagram of the *History-builder* module.

code metrics, and *linker*. Linking histories is a crucial part of these modules. Figure 3.12 shows the internal structure of a linker together with their methods.

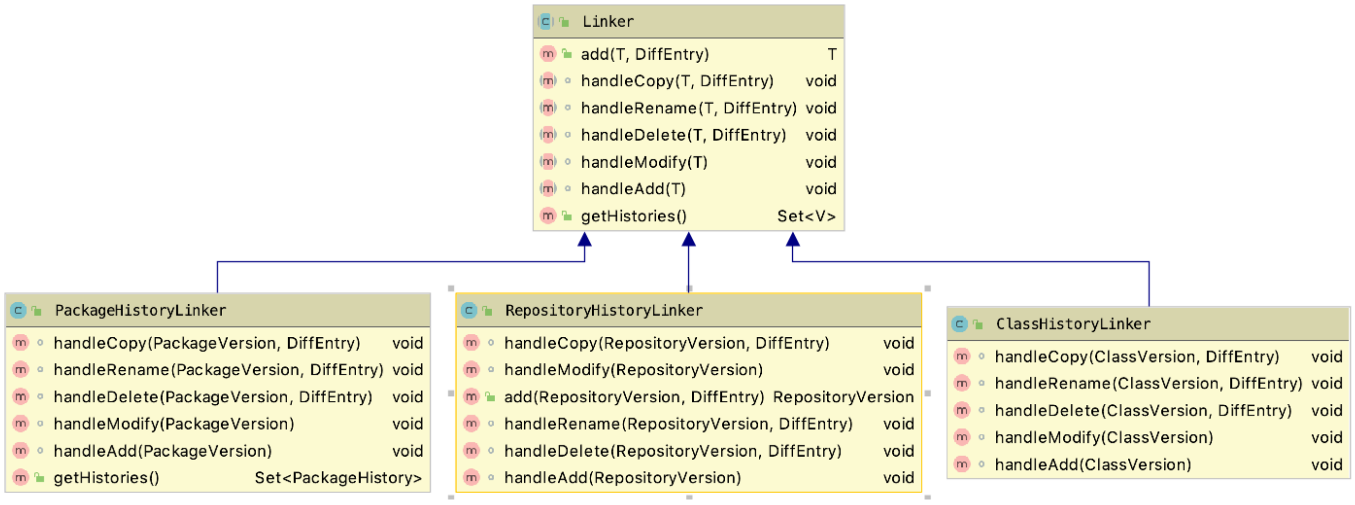

Linker
add(T, DiffEntry) T
handleCopy(T, DiffEntry) void
handleRename(T, DiffEntry) void
handleDelete(T, DiffEntry) void
handleModify(T) void
handleAdd(T) void
getHistories() Set<V>
PackageHistoryLinker
handleCopy(PackageVersion, DiffEntry) void
handleRename(PackageVersion, DiffEntry) void
handleDelete(PackageVersion, DiffEntry) void
handleModify(PackageVersion) void
handleAdd(PackageVersion) void
getHistories() Set<PackageHistory>
RepositoryHistoryLinker
handleCopy(RepositoryVersion, DiffEntry) void
handleModify(RepositoryVersion) void
add(RepositoryVersion, DiffEntry) RepositoryVersion
handleRename(RepositoryVersion, DiffEntry) void
handleDelete(RepositoryVersion, DiffEntry) void
handleAdd(RepositoryVersion) void
ClassHistoryLinker
handleCopy(ClassVersion, DiffEntry) void
handleRename(ClassVersion, DiffEntry) void
handleDelete(ClassVersion, DiffEntry) void
handleModify(ClassVersion) void
handleAdd(ClassVersion) void

FIGURE 3.12: Class Diagram of the *Linkers* module.

Linker module is made up of components that are responsible for the various types of History entities. In this case, we have RepositoryHistoryLinker, PackageHistoryLinker and ClassHistoryLinker all extending the abstract class Linker. It exposes methods: `add(T,DiffEntry)` and `getHistories()`. Internally, when a new VersionEntity is created and added to a linker, it will be added and connected based on the information contained in the file's path that generated the version. When `add(T,DiffEntry)` is called, it will look for the type of Git operation that the version represents and then call the appropriate method.

Figure 3.13 shows the sequence diagram for the creation of the history of a repository. When the controller receives a new analysis request, it contacts the ConstructStarter that initializes a repository. The acknowledgement is sent back to the user as soon as the repository is downloaded. From that point on, the CommitWalker iterates over the commits and asks the linker to compute the history between the various versions. At the end of the loop, the histories are retrieved and stored.

View Module The view module is the second core component of the backend. It is responsible for the preparation and visualization of the city. It communicates with the clients through the use of WebSockets. It is composed of two sub-modules, and one core component, the View.

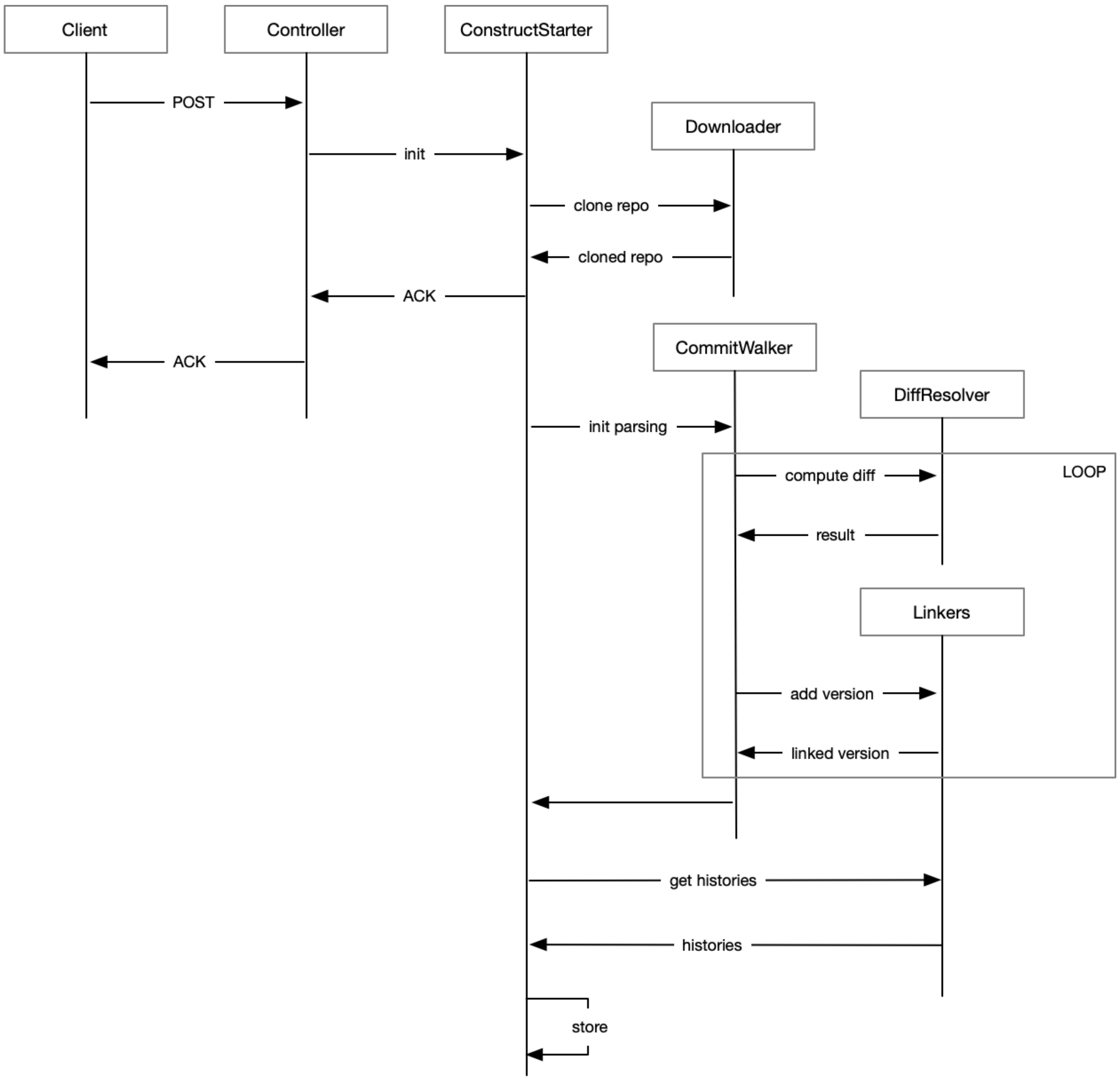


FIGURE 3.13: Sequence diagram for the creation of histories.

View is an abstract class that contains all the logic for visualizing the software system. During the development cycle of M3TRICITY, multiple iterations of different layouts have been developed:

- EMView. With this view, all classes are placed with a flat layout, i.e., there is no package concept. It can be used to visualize the evolution of classes. The main idea of this view is that the software system can be visualized as a matrix, where each class has a slot assigned and will evolve inside it.
- EMViewLight. Extends EMView, and uses the same logic except for the positioning of the elements. In this case, the matrix layout has better performance when compared to the previous one.
- EMVBinPackedView. Extends EMView with the concept of a package that contains other classes or packages. With this view, we also started to model the visualization while making it resistant.
- EMVBinPackedViewDate. Extends EMVBinPackedView and has added bucketing for the evolution data. Bucketing can be by day, week, month and year, reducing the amount of information processed inside the visualization.

The View uses the Packing module to compute the evolution-resistant layout for clients visualization. It exposes a service that can be used to call the algorithm used to compute the layout. The main idea behind this is that each class occupies a space that is derived from properties of their evolution. The other module, *glyphs*, contains the models of the cuboids that will be displayed on the frontend. Their creation is the result of a mapping between the layout algorithm results and the information needed in the frontend.

Figure 3.14 shows the sequence diagram for computing the layout of a software system. The client subscribes to the two topics (start and update) and communicates which repository it wants to visualize. The View then retrieves all the evolution history and calls the packing service with the information that has been collected and modeled. The result of the packing algorithm assigns a location to every package and class of the system. When the `/update` is ready, it will provide information about the successive version of the visualization.

Resistant Layout: From Model to Objects In Section 3.1.2 we discussed our approach for modeling histories. The model implemented in metricity is the result of multiple iterations in which the relationships between the entities have been modified to best represent the evolution of a software system. For this reason, in this paragraph, we discuss more in detail the data contained within each of the entities of the model. Figure 3.15 show relationships of the evolution-model developed within metricity.

Everything starts with the concept of Repository. The Repository is the entity from which everything is accessible. The object contains a list of Commits, a RepositoryHistory, a list of PackageHistory and ClassHistory entities. In addition to these entities, the Repository has a particular object called MXNode. This object is used to describe the relationship between PackageHistories. PackageHistories have an internal hierarchy to describe the package structure in the source code. By traversing this unique object, we can understand the nesting level of every package.

History entities constitute the evolution of single versions that are found within the software system. M3TRICITY, features a single RepositoryHistory, and multiple PackageHistories and ClassHistories, each one representing a package or a class. Due to the evolution, packages and classes can be merged. This information is extracted from the versioning system, by looking for rename operations. It is important to mention that Histories do not have a name, but either an ID that defines them, as discussed in Section 3.1.1. Class *A* will not be contained inside a ClassHistory that's named in the same way. This choice is the result of the possibility that the class itself in the future might change the name into another one, therefore naming the history as the class would have been a wrong choice. As a result, the linking of history to a name is done only at the version level. PackageHistories, on the other hand, also contain a list of sub-packages that are the result of the hierarchy of MXNode. Figure 3.16 depicts the entities' hierarchy inside our tool.

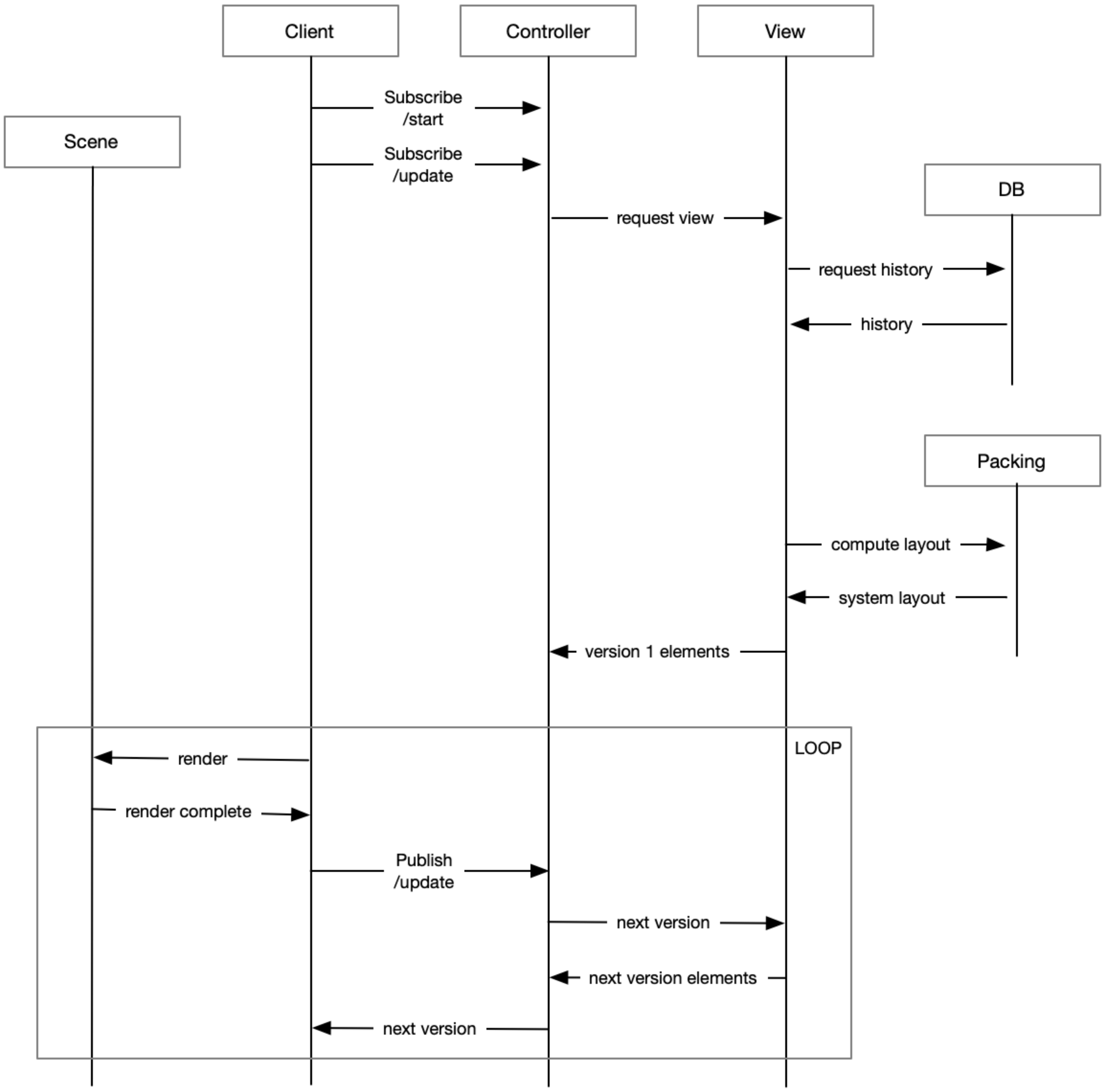


FIGURE 3.14: Sequence diagram for the visualization of software systems evolution.

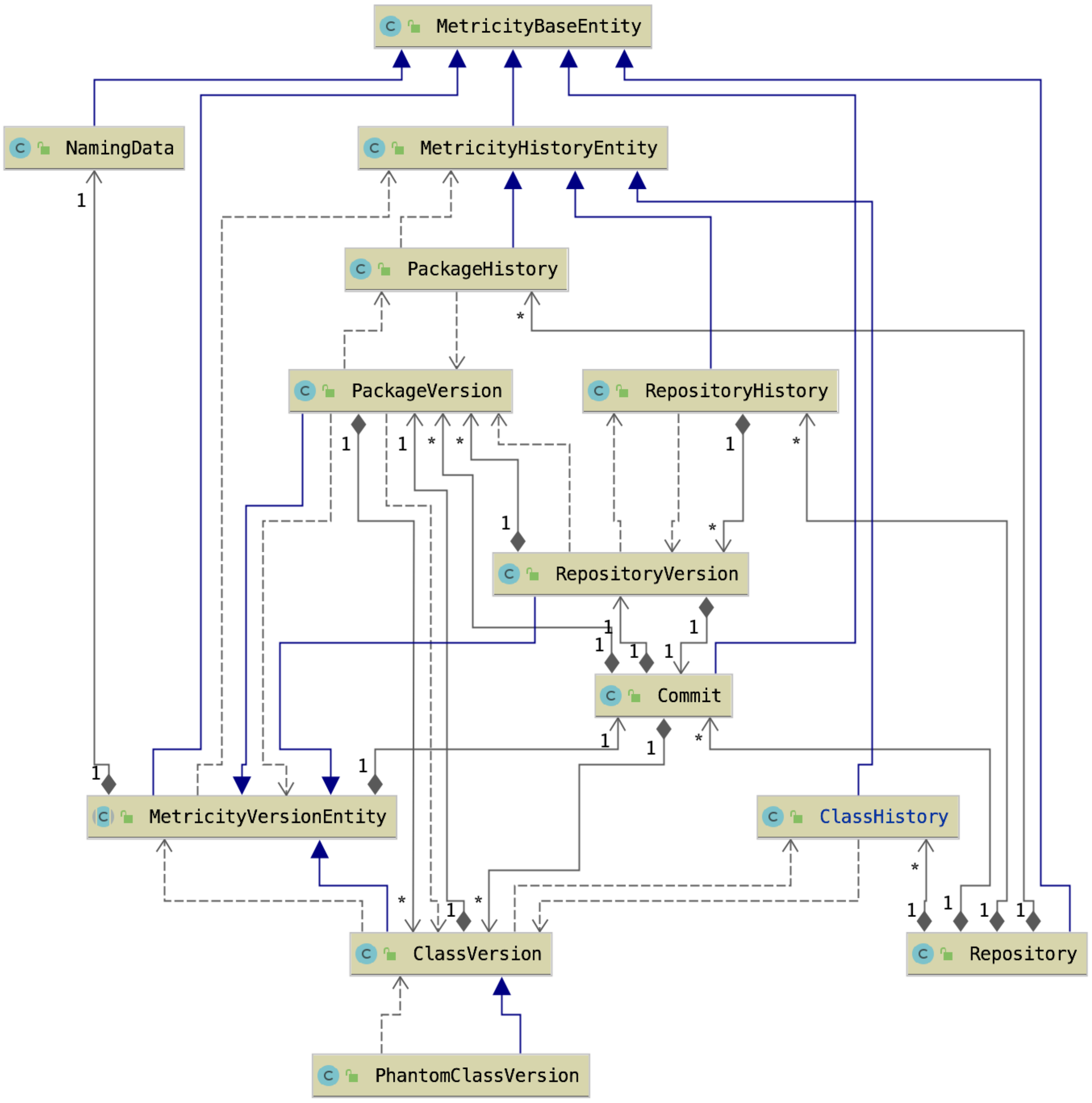


FIGURE 3.15: Evolution model as represented in M3TRICITY.

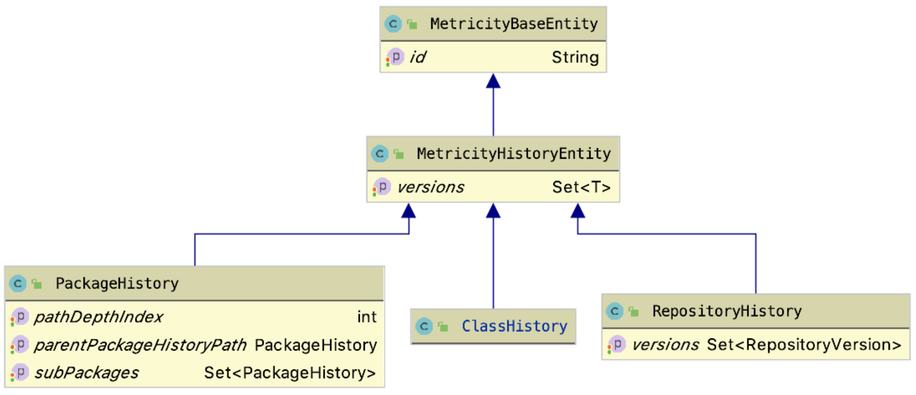


FIGURE 3.16: Class diagram of *History* entities.

Version entities represent the leaves of the evolution. A version is a single point in the history of the software system i.e., snapshot of the system. MetricityVersionEntity is the base object that is extended by all other versions. Inside it, we find the NamingData object that contains the naming information given to the entity at that point in time. To understand to which history it belongs to, we have an inverse relationship with the HistoryEntity. A parent-child relation depicts a simple pointer between two consecutive versions, while the set of MetricData contains information about the metrics at that specific version. PackageVersion and ClassVersion entities have two special fields named *versions* and *modifiedVersions*. The first one contains all the versions within that specific version, while the second one contains only the modified ones. We separated these cases because packages can contain multiple classes making expensive to iterate on the *versions* field. On the other hand, commits typically contain only a few changes and iterating over them is much faster. Figure 3.17 shows the final structure of the version entities.

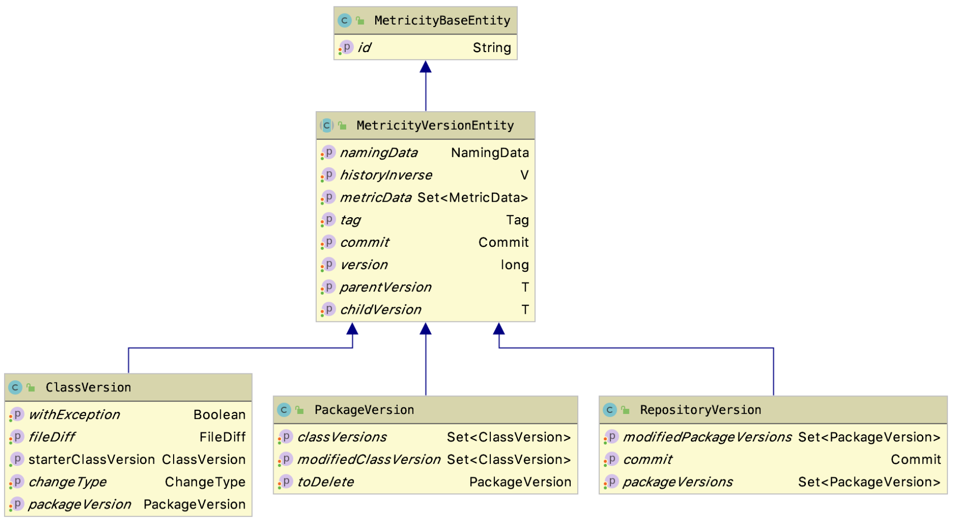


FIGURE 3.17: Class diagram of *Version* entities.

Frontend

The frontend is developed using Typescript,[8] and uses Vue.js, an open-source JavaScript framework for building user interfaces. Regarding the styling framework, we decided to use Bootstrap,[9] an open-source CSS framework directed at responsive, mobile-first frontend web development. Regarding the visualization of the software systems, we used Babylon.js,[10] is a real-time 3D engine using a JavaScript library for displaying 3D graphics in a web browser.

In Figure 3.18, we summarize the interactions needed to visualize the evolution of a software system on the canvas. Everything starts when the user selects a repository. At this point, the client subscribes itself to specific endpoints through the WebSocket protocol. After having received the information about the first version of the system, it creates an initial layout. Every object gets assigned an identifier representing the same identifier that is generated in the backend. This will be later used during the interaction with the canvas to retrieve additional information. When the objects are constructed and displayed on the canvas, the application checks whether the rendering of objects completed before requiring an update with the next version. From this point on, the logic is the same until no more versions are found. After the first version, animations are added to give a sense of growing/shrinking city. The animations are computationally heavy and require resources from the user's machine. For this reason, updates are managed by a scheduler which guarantees that the animation has been completed correctly, to ensure a correct view of the evolution.

When developing the frontend, we needed to fulfill the following requirements:

- *Easily extensible to new languages*. M3TRICITY is designed to support multiple programming languages. By extending the MetricExtractor, developers can implement their logic to extract metrics from files and add new metrics.
- *Portable*. M3TRICITY is accessible from anywhere through the web. The tool was developed to be used on multiple platforms, including mobile, smoothly.
- *Resistant layout*. The placement of the elements within the view is resistant to changes. Their placement is computed in advance while taking into consideration the evolution of the element itself. This placement guarantees a better representation of the evolution of the system.
- *Navigable*. The evolution of any software system can be navigated through specific interactions. Thus, we consider the possibility of exploring only a selected part of the history that is in the interest of the user. For this, users can jump in time or increase the speed of the evolution.
- *Customizable*. Users can personalize the metrics used in the visualization. In addition, for example, the user can use specific functions to highlight different areas of interest within the city.
- *Interactive*. M3TRICITY was designed to ease the access to versions information. User can access it simply by using simple operations such as hovering and clicking (e.g., highlighting of the evolution-timeline and details about the selected class).

3.2.2 User Interface

In this section, we present an overview of the user interface of M3TRICITY. To do so, we go through the web pages of the application and discuss the main features available on each page.

[8]See `https://www.typescriptlang.org/`
[9]See `https://getbootstrap.com/`
[10]See `https://www.babylonjs.com/`

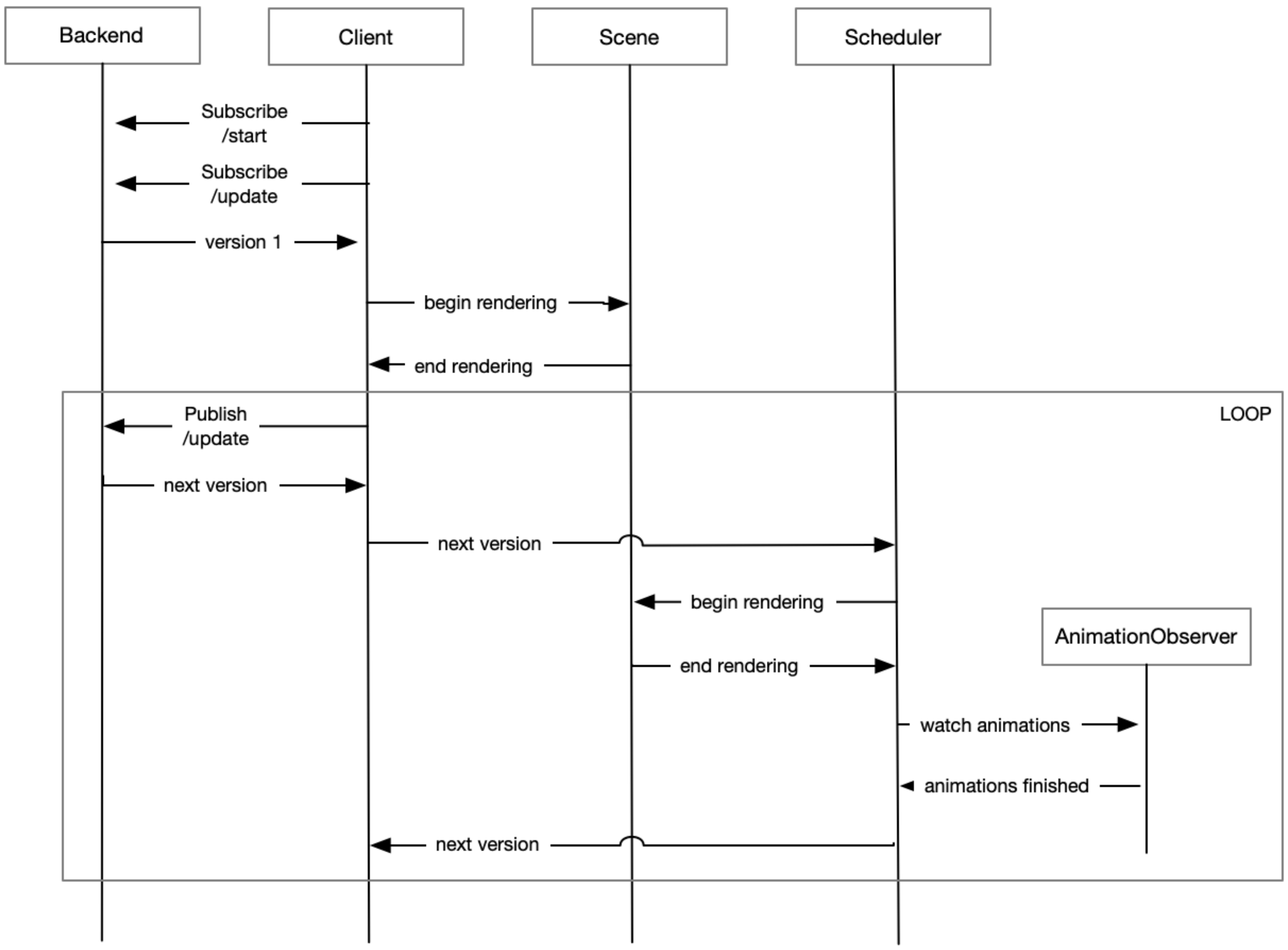
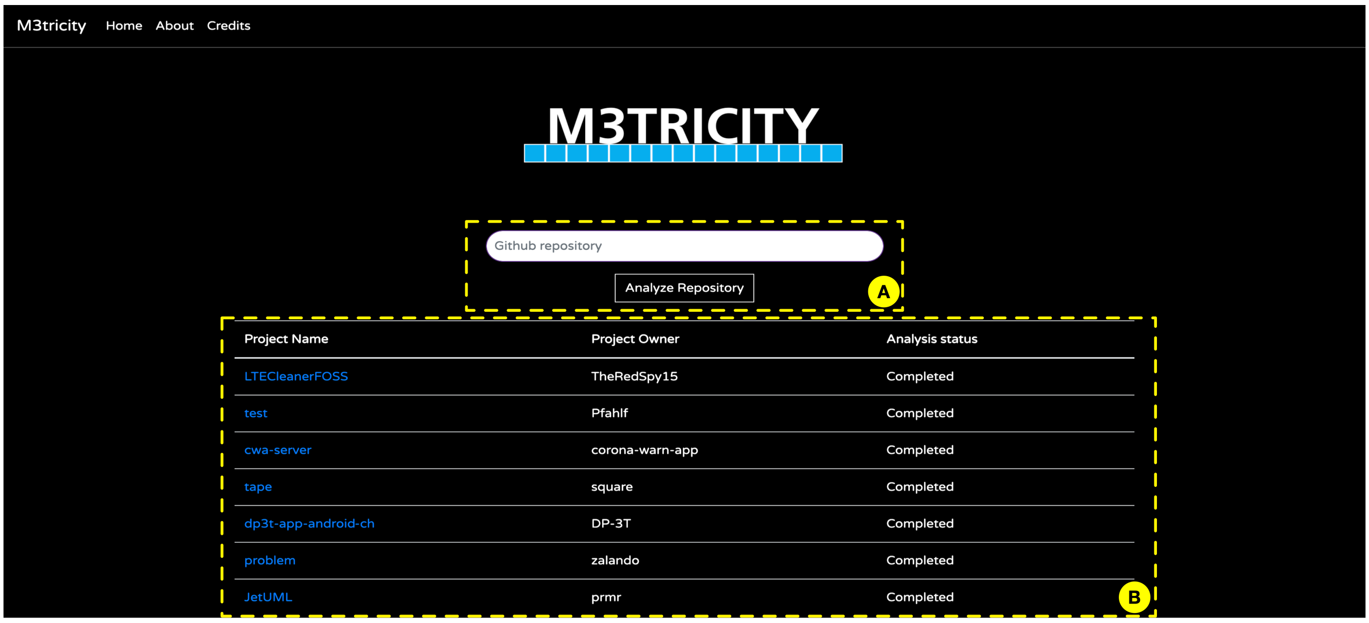


FIGURE 3.18: Sequence diagram for the visualization of software systems evolution in the frontend.

Home Page

The home page of M3TRICITY consists of two main sections: an input field and a repositories list. Figure 3.19A shows the input field, in which a user can insert a GitHub url. As soon as the analysis is started, the repository will appear in the list that is highlighted in Figure 3.19B. The list also contains the already analyzed repositories that can be navigated by clicking on the project name.



Project Name	Project Owner	Analysis status
LTECleanerFOSS	TheRedSpy15	Completed
test	Pfahlf	Completed
cwa-server	corona-warn-app	Completed
tape	square	Completed
dp3t-app-android-ch	DP-3T	Completed
problem	zalando	Completed
JetUML	prmr	Completed

FIGURE 3.19: M3TRICITY home page.

City View

Figure 3.20 shows the main view of M3TRICITY. To access it, the user needs to select one of the analyzed repositories from the home page. If the user wants to analyze other repositories, he needs to submit a new request. From the top bar (see Figure 3.20A), users can learn more about M3TRICITY or reach the project homepage where they can open the visualization on a pre-existing project or start the analysis of a new GitHub repository. M3TRICITY shows the name and the author of the project being analyzed (see Figure 3.20B) and displays the timestamps and the authors of the five last commits (see Figure 3.20C).

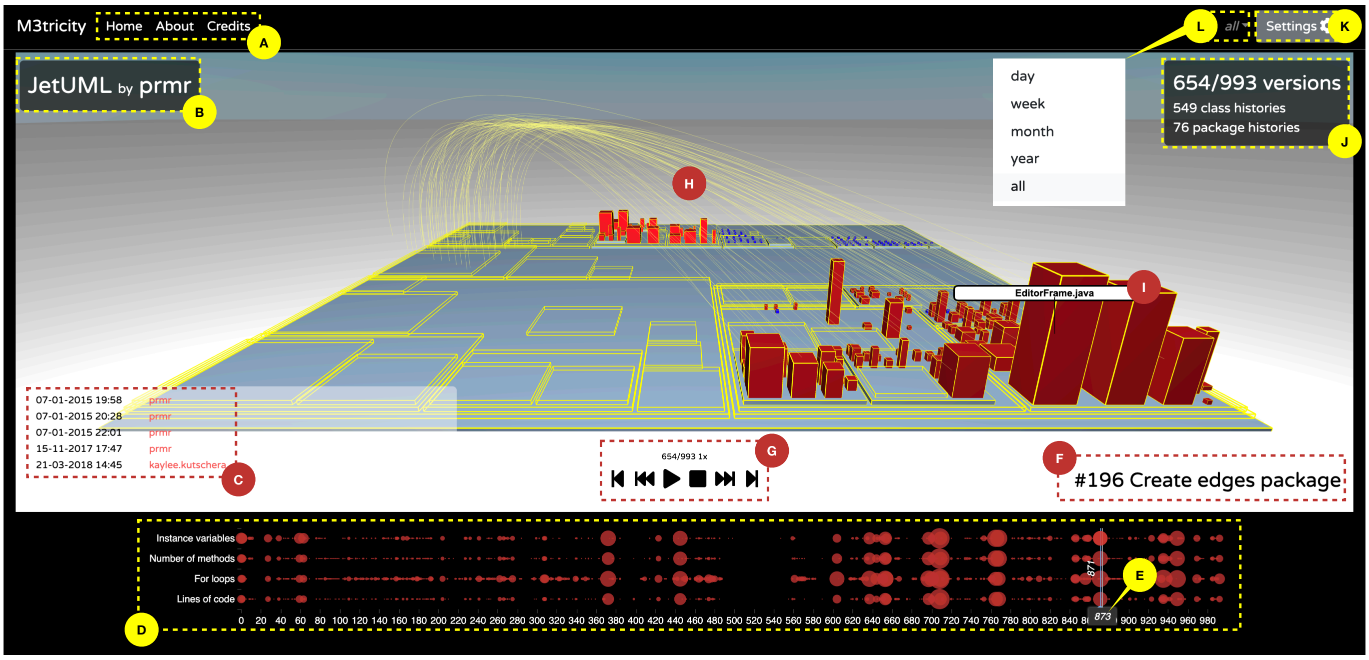
At the bottom, a timeline (Figure 3.20D) summarizes the evolution of the project. It indicates when metrics have been changed, *i.e.,* the number of fields, methods, for-loops, and lines of code, computed for each revision. We use a timeline visualization to depict this information: On the x-axis, there are the project versions crawled from the repository, while on the y-axis, there are the four categories of changes. The size of each dot represents the value for each change category in a specific version normalized across the complete history of the system. A larger dot means, therefore, more changes during the version. A cursor, highlighted in Figure 3.20E, keeps track of the snapshot being depicted in the canvas (Figure 3.20H). By clicking on the timeline, users can jump to the visualization of the selected version.

On the bottom right, M3TRICITY displays the commit message of the current snapshot (Figure 3.20F). The control panel (Figure 3.20G) lets users move through time by playing, pausing, forwarding, or rewinding the visualization. The visualization canvas (Figure 3.20H) occupies the central part of the screen.

By hovering on elements in the visualization, M3TRICITY shows a tooltip with additional information (Figure 3.20I), while by clicking over any of the elements, additional information is displayed like the complete filename but also its metrics and version.

On the top right corner of the screen, M3TRICITY summarizes information about the history of the system at hand (Figure 3.20J), lets the user choose between different ways to traverse time (Figure 3.20L), and change the settings (Figure 3.20K).

In the snapshot depicted in Figure 3.20H, JETUML is undergoing a structural refactoring [66] where all classes in a package `com.horstmann.violet` are moved to package `ca.mcgill.cs.stg.jetuml`. The visualization uses 3D edge bundling to highlight structural refactorings depicting arcs from the original position to the new one. In addition, the packages and classes that are modified are highlighted during each revision.



07-01-2015 19:58 prmr
07-01-2015 20:28 prmr
07-01-2015 22:01 prmr
15-11-2017 17:47 prmr
21-03-2018 14:45 kaylee.kutschera

FIGURE 3.20: Visualizing a snapshot of JETUML with M3TRICITY with the main UI elements highlighted.

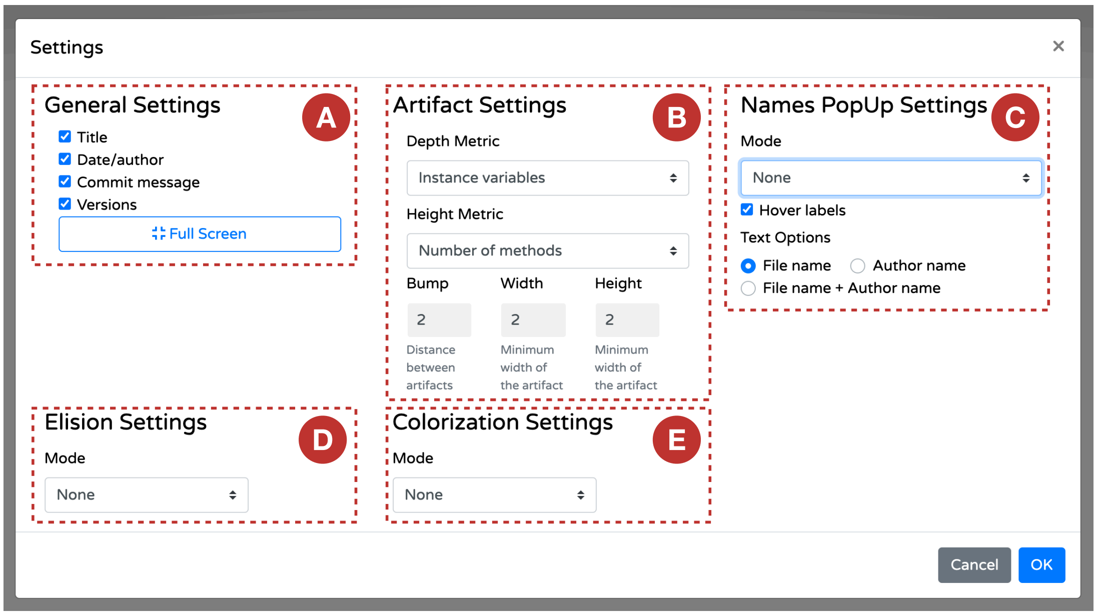
Settings Panel

Figure 3.21 shows the settings panel of M3TRICITY where users can toggle the visualization of different elements (e.g., title, date, commit messages, pop-ups) and change the metrics used in the visualization (e.g., depth and height of the cuboids).

General Settings (Figure 3.21A) enable the user to fine-tune the visualization. It is possible to hide some elements on the canvas or switch to full-screen mode. *Artifacts Settings* (Figure 3.21B) allow choosing the metrics used in the visualization and defining some parameters, *i.e.*, the minimum width and height, the gap between the objects (*i.e.*, bump). *Names PopUp Settings* (Figure 3.21C) lets users toggle automatic tooltips to show the names of the elements (*i.e.*, classes and packages) that have been modified in the current snapshot. Figure 3.21D is used for hiding objects matching a regular expression. *Colorization* (Figure 3.21E) lets users assign specific colors to objects, based on the tags extracted from the name of the file, a regular expression, or by manually selecting the elements on the canvas.

3.3 Limitations

This section discuss the limitations of M3TRICITY. When possible, we will also propose possible solutions to these problems.


Settings ×

General Settings A
- ✓ Title
- ✓ Date/author
- ✓ Commit message
- ✓ Versions

Full Screen

Artifact Settings B

Depth Metric: Instance variables

Height Metric: Number of methods

Bump	Width	Height
2	2	2
Distance between artifacts	Minimum width of the artifact	Minimum width of the artifact

Names PopUp Settings C

Mode: None

✓ Hover labels

Text Options: [✓ File name] Author name File name + Author name

Elision Settings D

Mode: None

Colorization Settings E

Mode: None

Cancel OK

FIGURE 3.21: The settings of M3TRICITY.

Improving the history building. Constructing the evolution model of a software system requires to check out every commit in increasing order. Performance on this phase of the analysis is sub-optimal, with times that increase as the number of change in each commit increases. Git versioning systems hierarchy is based in a tree-like structure, where each commit is a node. To increase the performance, one possible solution for the future is not to check out every commit but either using the information contained in the log, such as the lines modified and what has actually changed. By doing so, we will avoid to check out every commit. Another solution consists in keeping the current structure while trying to parallelize the extraction of the commits. This second solution requires to rethink how to connect histories since results might not be ordered anymore.

More resistant layout. The layout of M3TRICITY could be more resistant if we are able to deal with all possible cases that might happen with packages. In fact, our visualization approach assigns an area in the layout for every package, without taking into consideration the possibility that packages might be renamed or split. However, the correct approach would have been to try to display also those cases, while reusing the space that the package before the rename/split was using. The problem with this approach is that time must be spent to understand all possible situation that might happen during the evolution of software systems. In addition to this, even the simplest solution would require to rethought the visualization, together with the packing algorithm as our implementation do not support it.

Space reuse. To achieve a more compact layout, the free space that in future revisions of the software system is not used anymore, should be reused by newly appearing packages and classes to avoid to have dead areas within the city. To do this operation, we need to track the evolution when we compute the resistant-layout. We can then reassign previously used space to new elements. Figure 3.22 depicts how the layout will change when the space is reused. On the top row we can see the current implementation of the history-resitant layout. In revision 2 (R2), package B gets deleted. Still, within the layout that space will be reserved for the complete evolution of the software system, even if not needed anymore. At the bottom, the space reuse let package C occupy the space that was reserved by B in the previous version. With this operation, the layout becomes more compact as can be seen in version 3 (R3), where D is placed closer to the other packages.

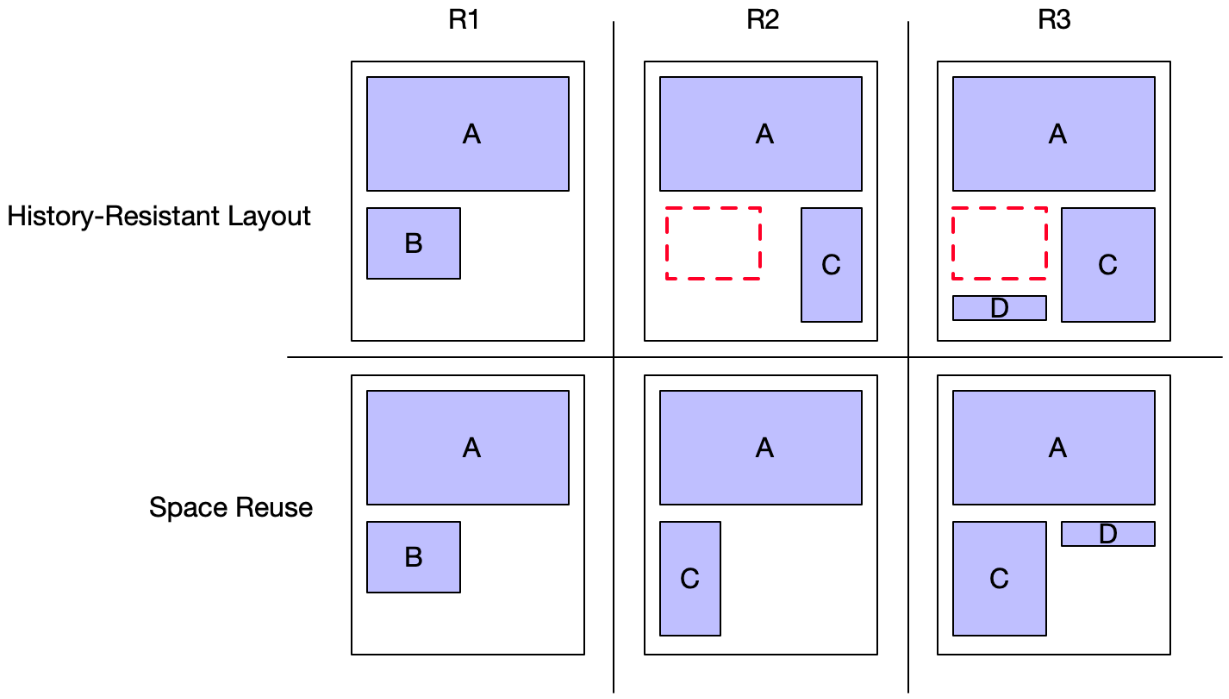


FIGURE 3.22: Example of space reuse vs History-resistant layout.

Rename and copy detection. Git entities can be added, renamed, copied and deleted. In two of these four cases, Git does not seem to do a good job: rename and copy detection. JGit, the tool used for accessing Git by Java, has an external library that tries to increase the chances of detecting these two crucial cases. It uses math to understand if two files are the same or not. Still, this is far from optimal because some files might be undetected. The library itself has parameters that can be changed to try to increase the match score. In the future, we will try to increase the detection of those cases, going down to method level to have a more in-depth knowledge of the changed that happened over time.

Metric Extraction For each commit, M3TRICITY it generates an XML for each source file, containing information about its composition using tags. This double pass is slow as the tool not only needs to wait that the commits are checked-out, but also that srcML finishes the parsing of each file and writes the output to another file. These steps require unnecessary IO operations that could be avoided. In the end, M3TRICITY compute the metrics based on the information found in the XML. As a solution to this problem, our tool should avoid using external tools for this pass and work directly within M3TRICITY. For this, a parser for multiple languages should be integrated and used to compute metrics.

Database integration. One of the first simplifications made to our tool, was to drop the remote database in favour of an in-memory one. The major problem is due to the performance of the remote database when retrieving the histories. In fact, to generate the resistant-layout, all histories for a repository need to be retrieved, discussed in the next paragraph.

Bin-packing at construction level. The logic of the application consists of two main operations:

- History building: deals with constructing the evolution of the software system
- Visualization: deals with computing how the elements should be visualized and simplify the objects sent to the frontend.

The View component is also responsible for computing the layout of the software system. The main problem with this approach is that during the first request of the visualization, the component needs to retrieve the whole history of the repository to compute the layout. By moving this operation in the history-building phase, the amount of time spent would be decreased since no requests would be made to the database while asking for all entities. In addition, the position of each class and package will be already defined at any version, not requiring any computation when trying to access it.

Redefine PackageHistories. PackageHistories have a big problem, as they represent the history of a series of entities. When we think of a history of a class or package, we imagine it as a single entity with a start and an end. PackageHistory entities can have a much more complicated structure, in which a package is split into multiple new ones, or in which it is merged into an existing one.

Dealing with this case of scenarios is not straightforward. In our case, we decided to generate a new PackageHistory for every package, while in reality histories might be related to other histories depending on their content. This approach simplified on one side the visualization of the evolution but created a situation in which information extracted from the versioning system was dropped to preserve a linear and straightforward structure.

In future, we will model these entities differently. PackageHistories might connect to other histories since what defines them are the actual classes contained, and they might belong to multiple packages during their evolution. A better representation for this would be a graph, in which history is not unique but can accept that at a given point in time gets connected to other histories, creating a hierarchy of histories.

Chapter 4

M3TRICITY in Practice

This section shows the capabilities of our evolution-resistant layout and M3TRICITY in general. We use M3TRICITY to analyze the evolution of the tool itself, followed by JetUML, the test project on which we relied during the development of the tool. Finally, we conclude by viewing the development of two other projects. The characteristics of these systems in terms of the number of Java classes and lines of code (LOC, excluding comments and empty lines) are described in Table 4.1.

Name	First Analyzed Commit	Last Analyzed Commit	Classes	LOC
M3tricity	Apr 10, 2020	Aug 18, 2020	141	6,587
JetUML	Jan 4, 2015	Jul 28, 2020	247	23,639
cwa-server	Apr 26, 2020	Aug 25, 2020	194	10,293
jib	Jul 23, 2017	Aug 27, 2020	512	46,815

TABLE 4.1: Main characteristics of the analyzed software systems.

4.1 M3TRICITY

In this section, we analyze the evolution of M3TRICITY using M3TRICITY. With this approach, we want to verify if the work done by us in the past six months is reflected in our visualization. Our tool has been developed in Java and Typescript and consists of 141 Java files and 195 commits on the Master branch.

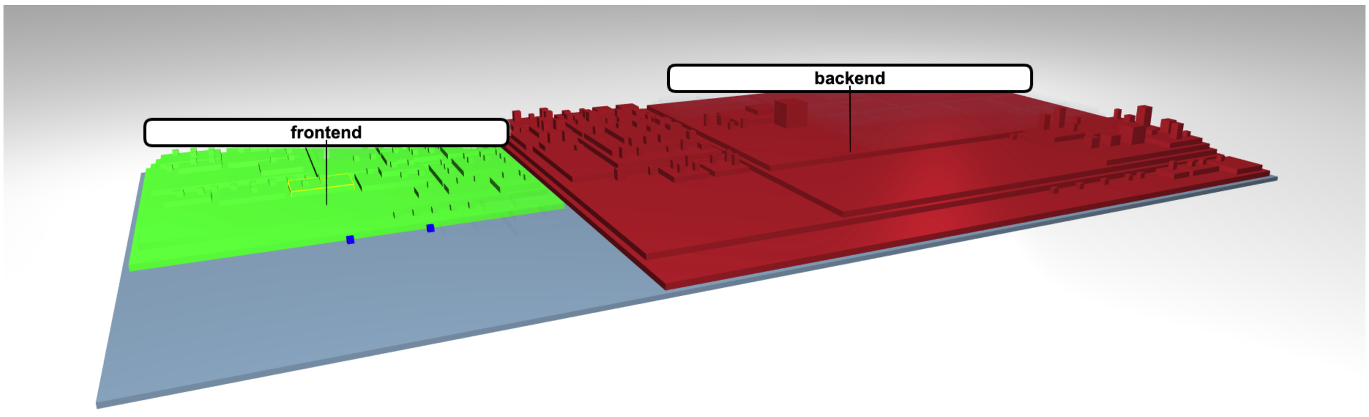


FIGURE 4.1: Separation of frontend and backend in the view using highlighting.

Figure 4.3A shows the first implementation of the backend. On the other hand, the frontend is not visible since we decided to prioritize the creation of the evolution model over the actual visualization. The separation between frontend and backend inside our visualization can also be seen in Figure 4.1. In the picture, the backend has been highlighted in red and the frontend in green.

Figure 4.3B depicts the moment when the frontend has been added to the Master branch. Inside the picture, we can see that all elements have the same size. This effect is the result of our tool not being able to extract metrics information for Typescript and Javascript languages (see Figure 4.2). M3TRICITY still assigns a minimum size to every element in the source code and the folder, therefore creating a complete representation of the software system.

In Figure 4.3C, version 128 represents the moment in which we started developing the evolution-resistant layout using the bin-packing algorithm. During this phase, we tried multiple different approaches while trying to achieve a pleasant visualization that respects the software system's evolution. All the different approaches for the bin-packing algorithm and resistant layout can be visualized in the bottom-left part of the city.

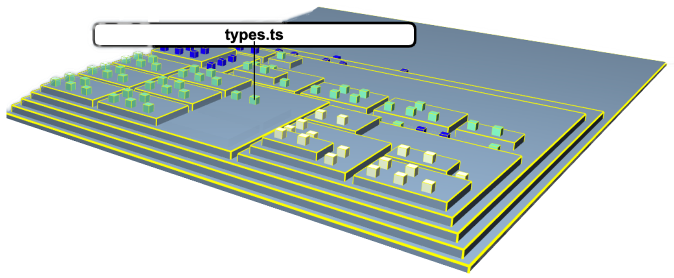
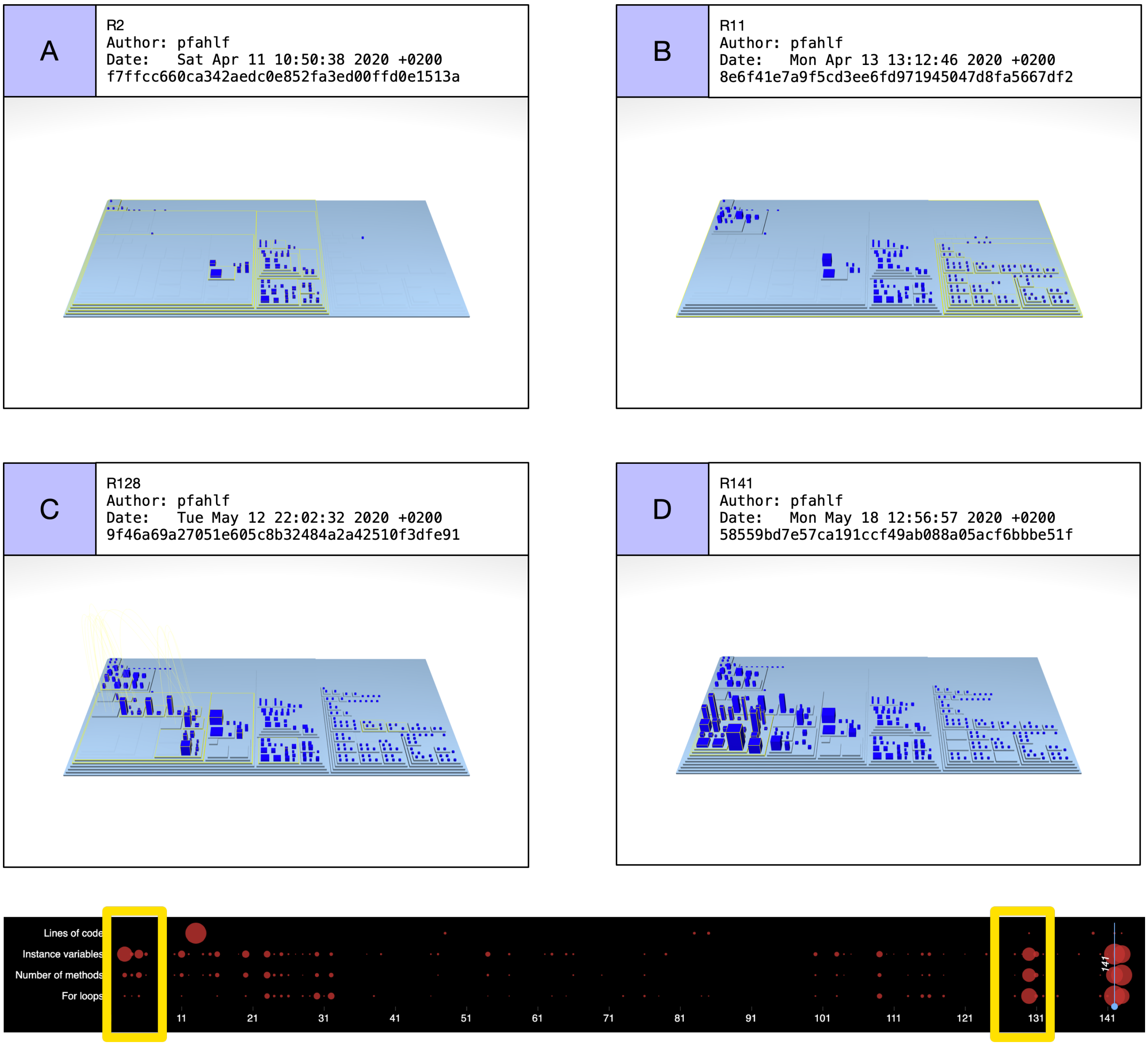


FIGURE 4.2: M3TRICITY frontend district.

One of the latest versions of our tool can be visualized in Figure 4.3D. In the picture, we can see that the evolution of the system regarding the packing algorithm, in the bottom-left district, has grown over time, resulting from multiple buildings, representing classes, added to the city.

Nevertheless, the tool was able to visualize the evolution of M3TRICITY, in a way that, being familiar with the development, we could identify all the major phases in the lifespan of M3TRICITY.


A

```
R2
Author: pfahlf
Date:   Sat Apr 11 10:50:38 2020 +0200
f7ffcc660ca342aedc0e852fa3ed00ffd0e1513a
```

B

```
R11
Author: pfahlf
Date:   Mon Apr 13 13:12:46 2020 +0200
8e6f41e7a9f5cd3ee6fd971945047d8fa5667df2
```

C

```
R128
Author: pfahlf
Date:   Tue May 12 22:02:32 2020 +0200
9f46a69a27051e605c8b32484a2a42510f3dfe91
```

D

```
R141
Author: pfahlf
Date:   Mon May 18 12:56:57 2020 +0200
58559bd7e57ca191ccf49ab088a05acf6bbbe51f
```

FIGURE 4.3: M3TRICITY evolution.

4.2 JetUML

JETUML[1] is a desktop application for fast UML diagramming. Written mostly in Java, the history of the system starts in early 2015 with the first commits of M.P. Robillard. Up to today (August 31, 2020) has 1,846 commits and of 247 Java files.

Violet, Violetta, and JetUML. Figure 4.4 shows six key snapshots of JETUML visualized with M3TRICITY.

Figure 4.4A depicts the first available version [67] of the project. Most of the canvas is empty but our history-resistant layout already pre-allocated the space needed to contain the whole evolution of JETUML. On the left part of the visualization, there is a densely populated district: By hovering on the visualization, we learn that this is the complete source code of the VIOLET[2] project contained in a package called `com.horstmann.violet`.

Figure 4.4B shows the 28th [68] revision of the system. Its commit message says *"#8 moved to dedicated package."* In this revision, half of the classes contained in the package `com.horstmann.violet` are moved into a newly created package named `ca.mcgill.cs.stg.violetta.graph` giving birth to the VIOLETTA project.

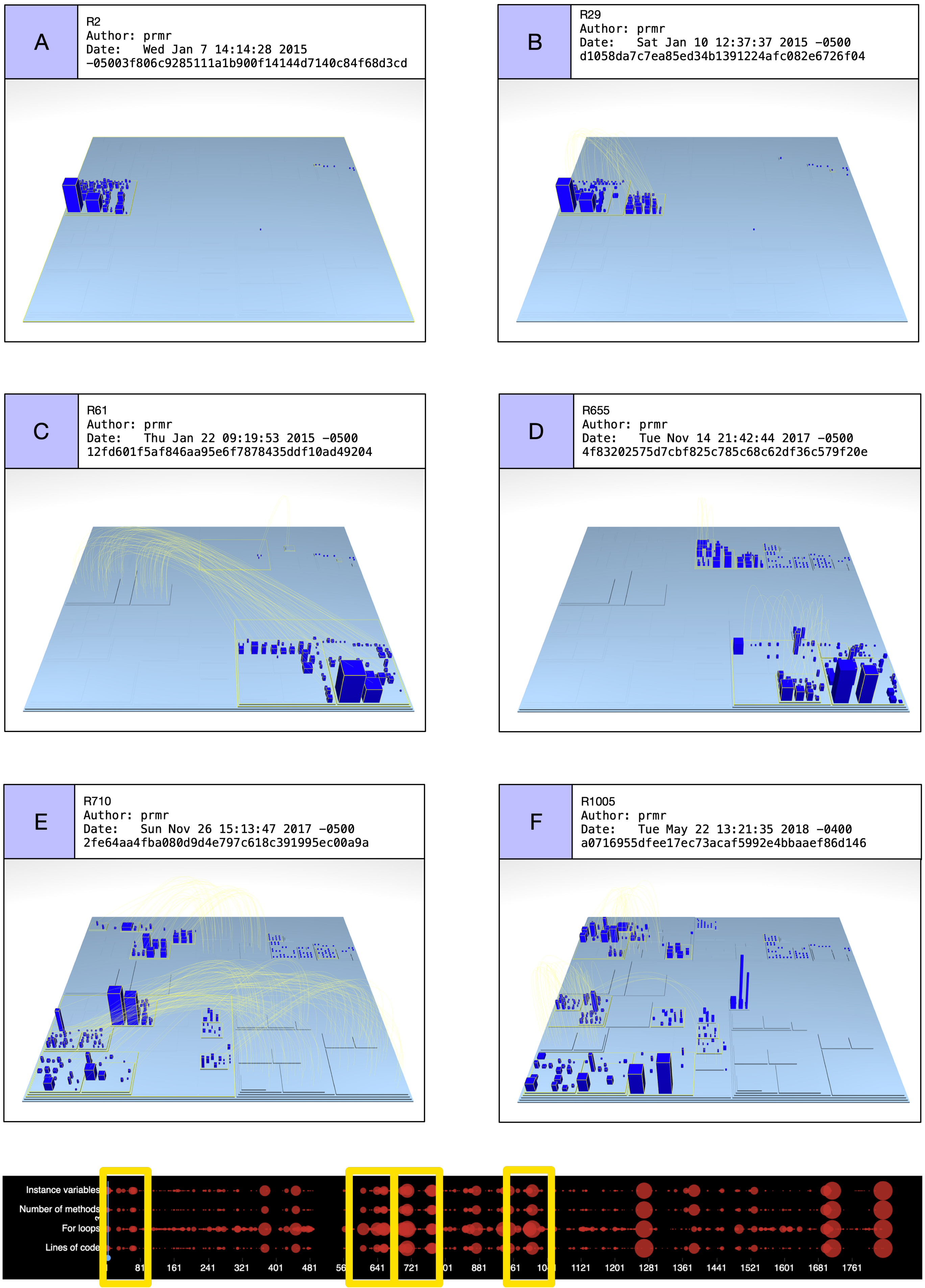
In the snapshot depicted in Figure 4.4C [69], all the classes of VIOLET and VIOLETTA are moved into a new package called `ca.mcgill.cs.stg.jetuml`, giving officially birth to the JETUML project. From this snapshot up to version 654 [70] a few changes happen inside the main package.

Figure 4.4D is the result of a move refactoring where all the classes dedicated to edges and nodes are moved into dedicated packages. A major change happens at version 710 [71], depicted in Figure 4.4E: packages were renamed by removing the acronym "`stg`" from their names (e.g., `ca.mcgill.cs.stg.jetuml` became `ca.mcgill.cs.jetuml`), leading to a new placement of the classes. While this looks like a simple renaming, which would not be displayed as a relocation, in fact it is a restructuring and M3TRICITY displays it as an explicit movement of classes. The last structural change we depict in Figure 4.4F is at revision 1005 [72] when numerous classes are affected by a package renaming where the word *"graph"* was substituted by *"diagram."*

After spotting these interesting stages in the development of JetUML, we were interested in the opinion of its developers and asked its creator to visualize it, who commented:

> " The perspective I see is consistent with my recollection of the evolution of the system. The visualization is helpful to identify some of the events that impacted the maturity of the system. Examples include the emergence of the test suite and its gradual increase in importance and the refactoring of God classes. This information is not directly available from the release notes, because in JetUML releases tend to be aligned with user-facing changes more than code restructuring. "
>
> *Martin P. Robillard – creator of JetUML*

[1]See https://github.com/prmr/JetUML
[2]See https://horstmann.com/violet



```
R2
Author: prmr
Date:   Wed Jan 7 14:14:28 2015
-05003f806c9285111a1b900f14144d7140c84f68d3cd
```

```
R29
Author: prmr
Date:   Sat Jan 10 12:37:37 2015 -0500
d1058da7c7ea85ed34b1391224afc082e6726f04
```

```
R61
Author: prmr
Date:   Thu Jan 22 09:19:53 2015 -0500
12fd601f5af846aa95e6f7878435ddf10ad49204
```

```
R655
Author: prmr
Date:   Tue Nov 14 21:42:44 2017 -0500
4f83202575d7cbf825c785c68c62df36c579f20e
```

```
R710
Author: prmr
Date:   Sun Nov 26 15:13:47 2017 -0500
2fe64aa4fba080d9d4e797c618c391995ec00a9a
```

```
R1005
Author: prmr
Date:   Tue May 22 13:21:35 2018 -0400
a0716955dfee17ec73acaf5992e4bbaaef86d146
```

FIGURE 4.4: JetUML evolution.

4.3 cwa-server

CORONA-WARN-APP SERVER[3] is Germany's official backend application for coronavirus contact tracing. Developed by SAP SE in a short amount of time (2 months), it consists of 419 commits and has 193 Java files.

Figure 4.6A shows the project setup [73]. As before, space is pre-allocated by taking into consideration the whole evolution of CWA-SERVER. Since the beginning of the development, the layout consisted of three main areas: on the bottom left `tools`, on the top left `services` and bottom-right `common`. Accessing this information can be done by hovering over the canvas (see Figure 4.5).

Figure 4.6B shows the 8th [74] revision of the system, in which a large building appears. It is a class responsible for the risk score. Figure 4.6C [75], shows the real start of the application development. The newly created district contains the first version of the API while also added a mock-server.

Figure 4.6D depicts the moment in which all modules containing `svc` are renamed [76]. Around the same time, the developers renamed the application from `ena` to `cwa`. Another major change happens at version 40 [77], depicted in Figure 4.6E: the commit message says *"Update distribution mock server #64"*, in which the latest version of the API is made public. As we can see, it does not consist anymore of a copy-paste as happened in Figure 4.6D, but either completely new files [78]. This information is derived from the fact that no arcs appear in the visualization. Figure 4.6F depicts one of the final revisions of the software system [78].

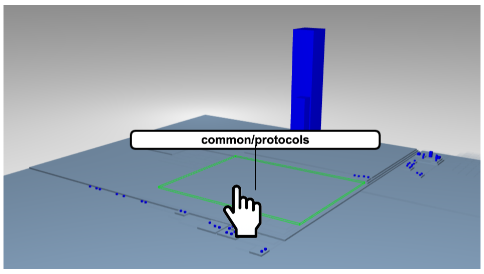
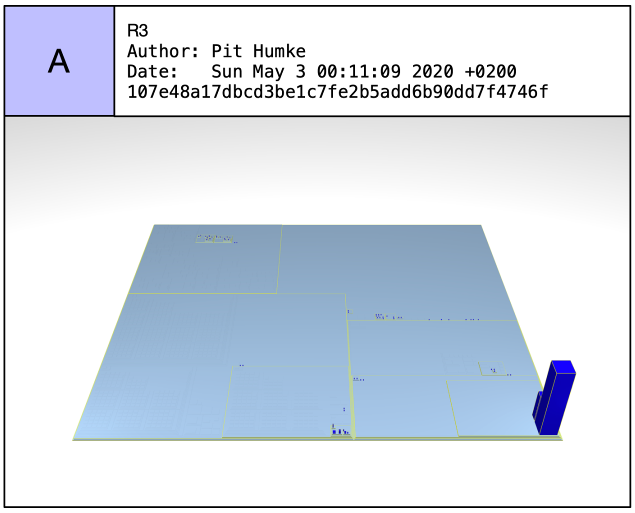
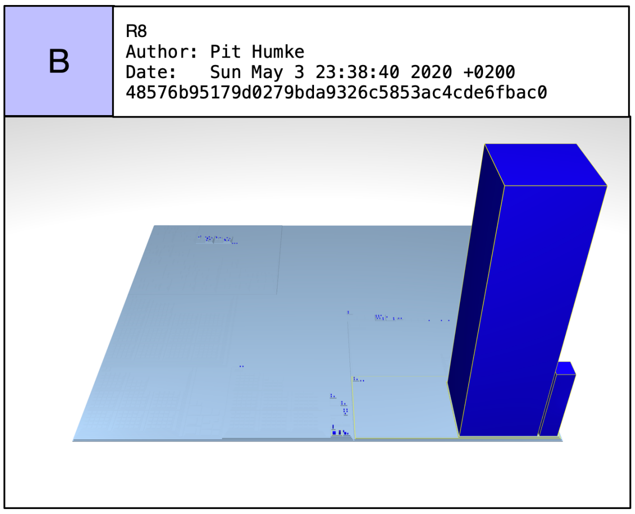
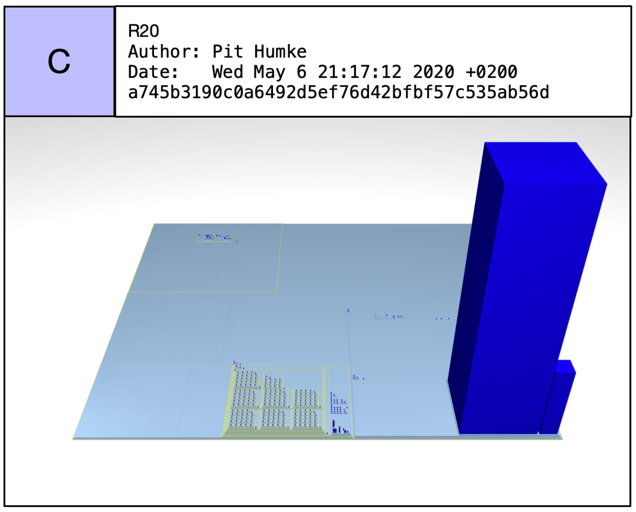


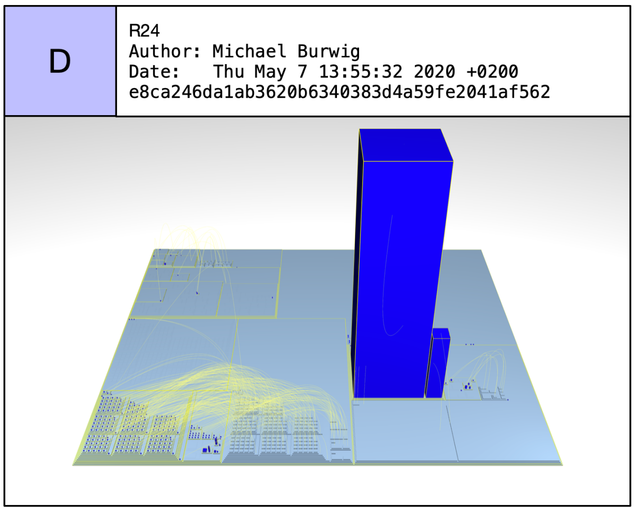
FIGURE 4.5: M3TRICITY with hover tooltip.

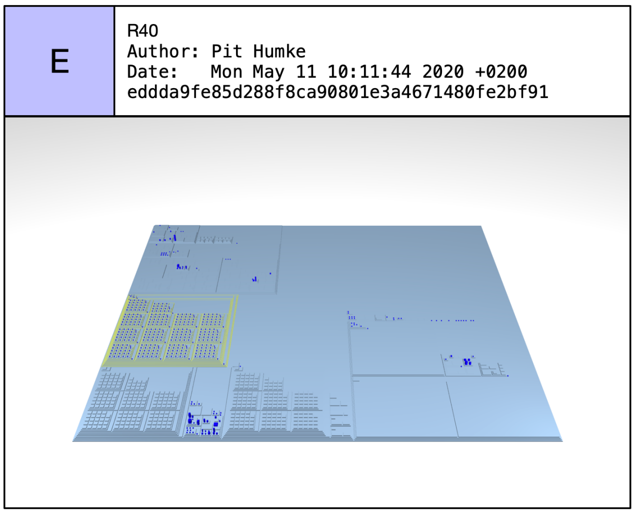
[3]See https://github.com/corona-warn-app/cwa-server

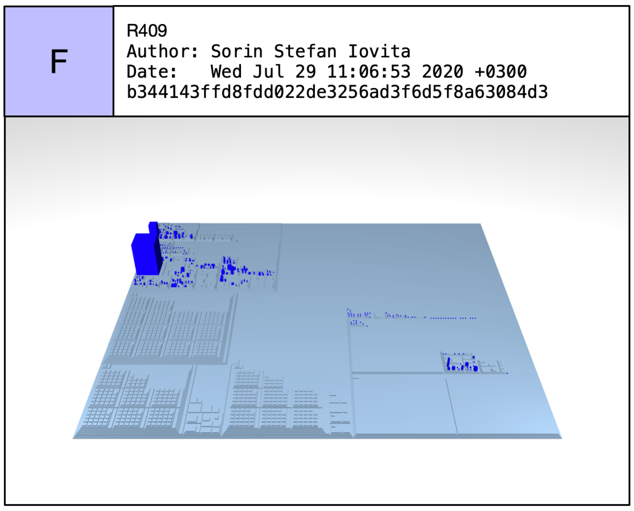

A
R3
Author: Pit Humke
Date: Sun May 3 00:11:09 2020 +0200
107e48a17dbcd3be1c7fe2b5add6b90dd7f4746f


B
R8
Author: Pit Humke
Date: Sun May 3 23:38:40 2020 +0200
48576b95179d0279bda9326c5853ac4cde6fbac0


C
R20
Author: Pit Humke
Date: Wed May 6 21:17:12 2020 +0200
a745b3190c0a6492d5ef76d42bfbf57c535ab56d


D
R24
Author: Michael Burwig
Date: Thu May 7 13:55:32 2020 +0200
e8ca246da1ab3620b6340383d4a59fe2041af562


E
R40
Author: Pit Humke
Date: Mon May 11 10:11:44 2020 +0200
eddda9fe85d288f8ca90801e3a4671480fe2bf91


F
R409
Author: Sorin Stefan Iovita
Date: Wed Jul 29 11:06:53 2020 +0300
b344143ffd8fdd022de3256ad3f6d5f8a63084d3

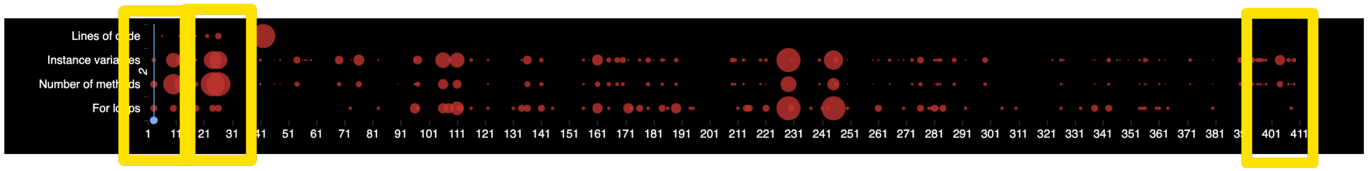


FIGURE 4.6: Cwa server evolution.

4.4 jib

JIB[4] is an open-source project by GoogleContainerTools. In its latest version, consists of 512 Java files and 1,552 commits. The project itself aims to create a library for building optimized Docker images without knowledge about Docker best-practices.

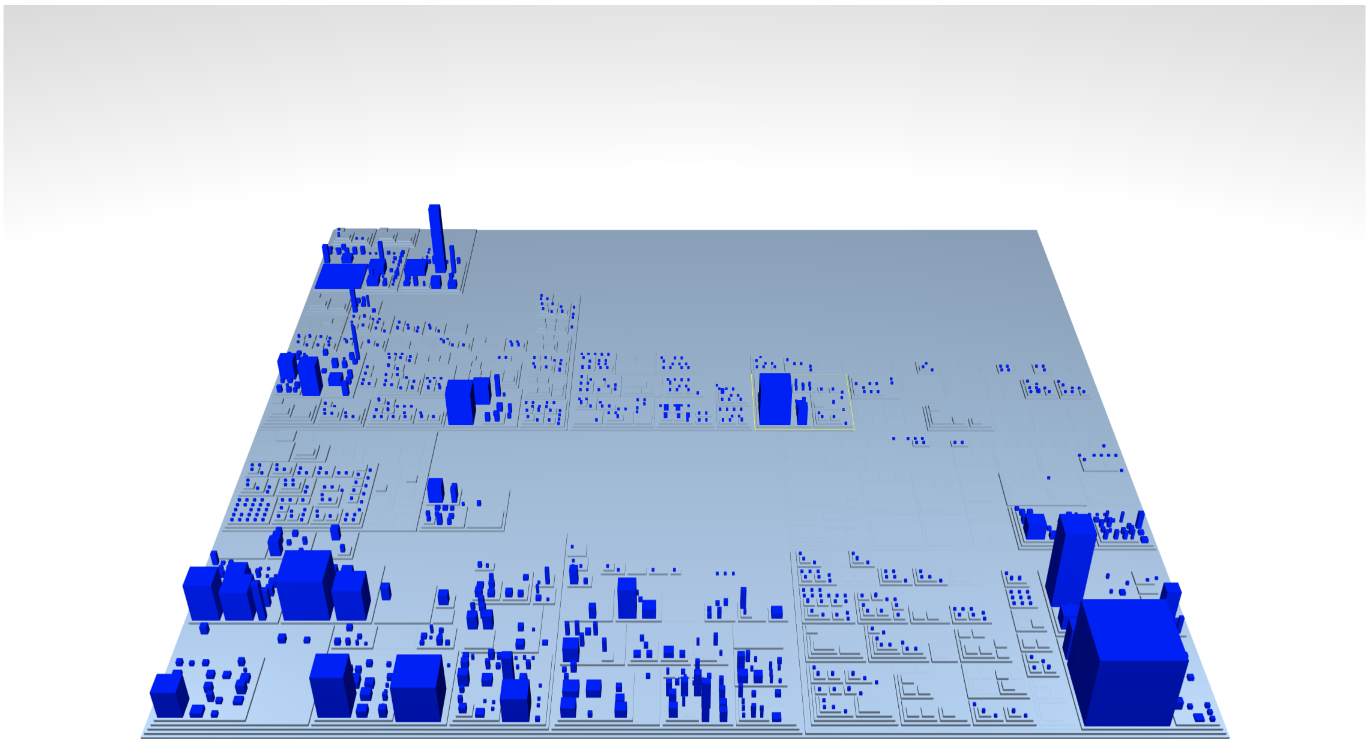
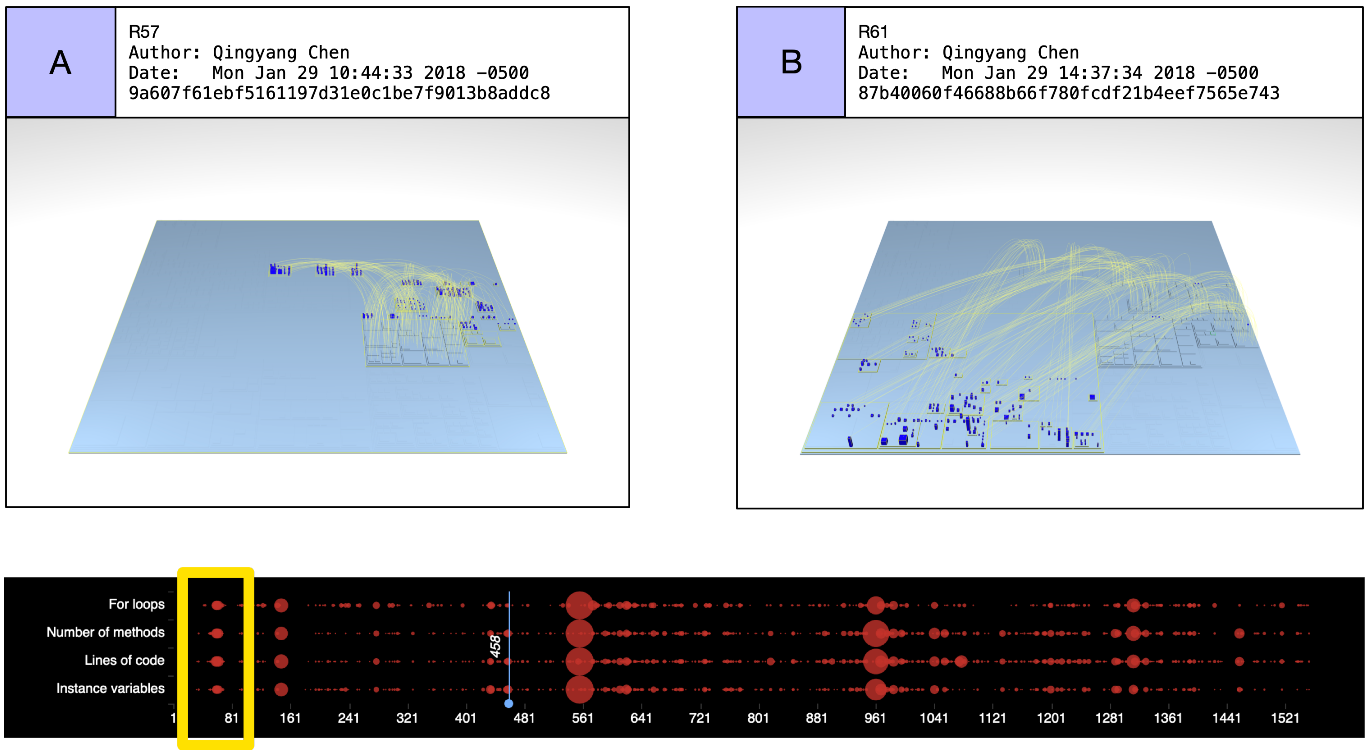


FIGURE 4.7: Jib latest revision.

Being developed by a large company, we selected this project due to its size. As compared with the previous projects, in Figure 4.7 we can see that the sizes of the buildings are more balanced during all revisions of the project despite the large amount of classes.

For this project, we selected two evolution changes during the first iterations of developments of this project. Figure 4.6A shows revision 57 [79]. The comment says *"Deletes minikube-related content. #1"* and even if no comments are accessible for this merge requests, we can see that the internal structure of the system changes by splitting it into multiple smaller ones. Another major change happens later at revision 61 [80], in this case, the content of the whole system gets moved from package `core` to the package `jib-core`.

[4]See `https://github.com/GoogleContainerTools/jib`



A

R57
Author: Qingyang Chen
Date: Mon Jan 29 10:44:33 2018 -0500
9a607f61ebf5161197d31e0c1be7f9013b8addc8

B

R61
Author: Qingyang Chen
Date: Mon Jan 29 14:37:34 2018 -0500
87b40060f46688b66f780fcdf21b4eef7565e743

FIGURE 4.8: Jib evolution.

4.5 Analysis Summary

To illustrate the validity of our approach for visualizing software system evolution, we analyzed four different software systems.

The first case consisted of M3TRICITY itself, the tool that computes and generates the evolution-resistant layout used for displaying software systems as cities. We used this case to verify that the visualization resulting from the analysis was represented at best compared to what we had developed in the previous months.

The second case consisted in analyzing the evolution of JetUML, a desktop application for interactively creating and editing diagrams in the Unified Modeling Language. In this case, the analysis was more in-depth than the others. Using our visualization, we looked for the changes that happened over its evolution, discovering exciting insights such as the various name changes that happened. In addition, we contacted the author asking him if what he was seeing was respecting the development of the system.

In the end, we analyzed two other projects, cwa-server and jib. The first is the official coronavirus tracking application in Germany and was selected to understand how an application that needs to be developed very quickly evolves. The result was a quickly changing city. Jib, on the other hand, showed us how large software systems evolve.

Chapter 5

Conclusions and Future Work

In this chapter, we summarize our work, discuss key lessons learned throughout the development of M3TRICITY and present directions for future work. Section 5.1 recaps of what has been done. Section 5.2 reflects on what has been learned during the development of M3TRICITY, giving some insights into what could have been done differently. Concluding, in Section 5.3 discuss possible changes and extensions for the future.

5.1 Summary

The main goal of this thesis is to visualize software systems evolution using the city metaphor and an evolution-resistant layout. We wanted to show that creating a resistant layout would, therefore, make the visualization more understandable to the user who studies the evolution of a system.

We believe that visualizing software systems evolution is an excellent way to discover and understand how a system was created since the beginning of its development. With M3TRICITY, a web application that we developed for this thesis, we implemented a layout that leverages the evolution model to create a history-resistant layout. In addition to this, we rethought Hismo evolution model, to make it more accessible and explorable, by simplifying its internal structure. We examined the best way to store the evolution of software systems, given the information that the versioning system was providing us. In the end, we found out that visualizing software systems in a resistant way involves many individual cases, such as class and package renames but also package split. Nevertheless, in the end, our solution is capable of handling many of these cases and visualizing the development of a system in an evolution-resistant way.

When we reached a stable development of the evolution model, the history-builder and layout algorithm, we decided it was time to add functionalities to the visualization.

We used M3TRICITY to see how real software systems developed over time. We discovered that none of the applications we decided to analyze was stale. All of them developed uniquely, with some common visual patterns. For example, large refactorings happened but also many renames.

5.2 Reflections

In this section, we to recap what we learned and what could be done differently. We summarize these in four categories: reflections on the evolution model, reflections about extracting metrics efficiently and reflections on web-based visualizations.

5.2.1 Reflections on the Evolution Model

We saw that evolution models had been developed in the past, with the idea of creating a generic model that fits all possible cases. In this thesis, we decided to focus on the model proposed by Tudor Gîrba [6]. We tried to simplify the model to make it more consistent with the structure itself proposed by the author.

The absence of a RepositoryHistory and RepositoryVersion reduced the logic built in the evolution model itself, making it less understandable.

The model proposed a concept of Snapshot to understand where and when something was stored. However, at the underlying levels, the hierarchy was well defined, while at Snapshot level, it was not. With the use of RepositoryHistory and RepositoryVersion, we had added a responsible layer to understand where and when something was seen, without however changing the hierarchy of the underlying levels. Moreover, the original model was developed for Subversion and not for Git, with different concepts for a version control system. Adapting it to Git required keeping ftrack of every change in the name of a source file over the complete evolution of a software system.

Our model is not perfect. Still, it has reached a point in which is able to store the information that regards software systems evolutions. Even if PackageHistories will need to be investigated more in-depth, the model is able to display consistently the evolution of classes, packages and even repositories.

5.2.2 Reflections about Extracting Metrics Efficiently

From the beginning of the implementation, one of the main problems was to figure out the extraction of metrics from a software system.

Contrary to what we expected, there were not many possibilities to choose from. Most of the libraries were almost always language-dependent, creating the need to develop specific handlers for each language. In the end, we decided to use `srcML`, an infrastructure for the exploration, analysis, and manipulation of source code that converts the source code into XML. At this point, we were able to compute the metrics based on the output of the tool.

A generic library capable of parsing and extracting metrics for each language could simplify the work of analyzing software systems. This library might also be used by developers who need to analyze and extract additional information. It is also true that languages change quickly and that maintaining such a library could be a tedious job, but with benefits.

5.2.3 Reflections on Web-GL Visualizations and Babylon.js

One of the core requirement was to create a tool that was accessible from anywhere by anyone. For this reason, we decided to develop a web application that uses a 3D library called Babylon.js to display software system as cities.

A large software system often consists of many classes, many packages and many refactoring. This can lead to scenes with thousands of objects and multiple animations. Resources on web browsers are much more limited than native software, for this reason in all libraries is suggested to limit the number of active meshes within the view. Alternative solutions to visualize large amount of objects exist, such as *solid particle systems* or *mesh instances*, but do not always fit the need of the developer. Solid particle systems consist of objects that can have different materials and different shapes, but at rendering time they are transformed into a single mesh, removing the possibility to interact with any of those elements. On the other hand, mesh instances do have the possibility to add control over the single instance, while sharing some properties with the root mesh, reducing the calls for drawing the meshes. Still, instances will occupy a larger amount of memory when compared to solid particle systems. In addition, instances always share the material with the root mesh, making it impossible to change it.

Functionalities to convert particles into meshes when close to the camera or hovered over them are absent from the library. Operation for this type of problem should be added to add interactivity to the scene while maintaining an overall good performance.

5.3 Future Work

In this section, we discuss some future directions for M3TRICITY.

Rebuilding frontend. The logic behind the frontend is quite complicated and should be refactored to implement different solutions while aiming to increase the number of objects that can be rendered. Knowledge of the web-gl framework has increased over time, and current implementation does not make use of it. Interaction with the backend and time-scheduling as it is now is over complicated.

It would be interesting to try multiple approaches depending on the user's devices. In fact, we could try to use different solutions depending on the specification of the laptop or phone. In addition to this, dropping some features in favour of alternative solutions such as mesh reuse and partial scene rendering should be tried.

Moving elements. Elements within the canvas are not moved but rather disposed and rendered into a new location during a rename operation. Having the possibility to see the objects moving into the new location could be intriguing. The same logic could apply during packages split, copying and renames.

Learning from history. By analyzing multiple software systems, we could try to learn if common patterns exist within their evolution. The evolution model provides additional information that we are not using, and that can be used for different purposes not related to the city metaphor. For example, determine how classes that have similar names develop over time in different systems, or if given a package name multiple systems have a specific class in it that can ease the future development of the system.

Exporting RepositoryHistory A requirement of M3TRICITY was to export the evolution history of the software system under CSV format. By doing so, could export the information provided by the versioning system to be used by different applications or visualization that is not based on the city metaphor.

5.4 Final Words

During the past four months, we developed M3TRICITY, an application that leverages the 3D city metaphor with an evolution-resistant layout. We implemented it as a web application available at http://metricity.si.usi.ch/.

We believe that what we learned should be used to create a better evolution model together with a more resistant layout.

The ideas and important lessons learned within this thesis about visualizing software systems in an evolution-resistant layout can serve as a starting point and become a useful reference for future studies.

Bibliography

[1] R. Wettel and M. Lanza, "Visualizing software systems as cities," in *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 2007, pp. 92–99.

[2] L. M. Haibt, "A program to draw multilevel flow charts," in *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*, ser. IRE-AIEE-ACM '59 (Western). New York, NY, USA: ACM, 1959, pp. 131–137.

[3] I. Nassi and B. Shneiderman, "Flowchart techniques for structured programming," *SIGPLAN Not.*, vol. 8, no. 8, pp. 12–26, Aug. 1973.

[4] J. Tesler and S. Strasnick, "Fsn: The 3d file system navigator," *Silicon Graphics, Inc., Mountain View, CA*, 1992.

[5] M. Lanza and S. Ducasse, "Understanding software evolution using a combination of software visualization and software metrics," in *In Proceedings of LMO 2002 (Langages et Modèles à Objets*. Citeseer, 2002.

[6] T. A. Girba, "Modeling history to understand software evolution," Ph.D. dissertation, University of Bern, 2005.

[7] R. Wettel, "Software systems as cities," Ph.D. dissertation, Università della Svizzera italiana, 2010.

[8] M. M. Lehman and L. A. Belady, *Program evolution: processes of software change*. Academic Press Professional, Inc., 1985.

[9] B. A. Price, R. M. Baecker, and I. S. Small, "A principled taxonomy of software visualization," *Journal of Visual Languages & Computing*, vol. 4, no. 3, pp. 211–266, 1993.

[10] C. Ware, "Information visualization: perception for design. 2000," *New York: Morgan Kauffman*, 2004.

[11] G. Lakoff and M. Johnson, "Metaphor we live by," *Chicago/London*, 1980.

[12] R. Wettel and M. Lanza, "Codecity: 3d visualization of large-scale software," in *Companion of the 30th International Conference on Software Engineering*, ser. ICSE Companion '08. Association for Computing Machinery, 2008, p. 921–922.

[13] F. Steinbrückner and C. Lewerentz, "Representing development history in software cities," in *Proceedings of the 5th International Symposium on Software Visualization*, ser. SOFTVIS '10. ACM, 2010, pp. 193–202.

[14] U. Erra and G. Scanniello, "Towards the visualization of software systems as 3d forests: The codetrees environment," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. ACM, 2012, pp. 981–988.

[15] H. Graham, H. Y. Yang, and R. Berrigan, "A solar system metaphor for 3d visualisation of object oriented software metrics," in *Proceedings of the 2004 Australasian symposium on Information Visualisation-Volume 35*, 2004, pp. 53–59.

[16] C. Knight and M. Munro, "Virtual but visible software," in *2000 IEEE Conference on Information Visualization. An International Conference on Computer Visualization and Graphics*. IEEE, 2000, pp. 198–205.

[17] T. Panas, R. Berrigan, and J. Grundy, "A 3d metaphor for software production visualization," 08 2003, pp. 314 – 319.

[18] A. Marcus, L. Feng, and J. Maletic, "3d representations for software visualization." 01 2003, pp. 27–36, 207.

[19] M. Balzer, A. Noack, O. Deussen, and C. Lewerentz, "Software landscapes: Visualizing the structure of large software systems," in *IEEE TCVG*, 2004.

[20] R. Wettel, M. Lanza, and R. Robbes, "Software systems as cities: A controlled experiment," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 551–560.

[21] D. E. Knuth, "Computer-drawn flowcharts," *Commun. ACM*, vol. 6, no. 9, pp. 555–563, Sep. 1963.

[22] ——, "Literate programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, 1984.

[23] R. Baecker, "Sorting out sorting: A case study of software visualization for teaching computer science," *Software visualization: Programming as a multimedia experience*, vol. 1, pp. 369–381, 1998.

[24] H. A. Müller and K. Klashinsky, "Rigi-a system for programming-in-the-large," in *Proceedings of the 10th international conference on Software engineering*. IEEE Computer Society Press, 1988, pp. 80–86.

[25] H. A. Müller and K. Klashinsky, "Rigi-a system for programming-in-the-large," in *Proceedings of the 10th International Conference on Software Engineering*, ser. ICSE '88. Los Alamitos, CA, USA: IEEE Computer Society Press, 1988, pp. 80–86.

[26] S. C. Eick, J. L. Steffen, and E. E. Sumner, "Seesoft-a tool for visualizing line oriented software statistics," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 957–968, 1992.

[27] R. Holt and J. Y. Pak, "Gase: visualizing software evolution-in-the-large," in *Proceedings of WCRE'96: 4rd Working Conference on Reverse Engineering*. IEEE, 1996, pp. 163–167.

[28] M. Lanza, S. Ducasse, and S. Demeyer, "Combining metrics and graphs for object oriented reverse engineering," Ph.D. dissertation, Universität Bern, 12 1999.

[29] D. Holten, B. Cornelissen, and J. J. van Wijk, "Trace visualization using hierarchical edge bundles and massive sequence views," in *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 2007, pp. 47–54.

[30] H. Gall, M. Jazayeri, and C. Riva, "Visualizing software release histories: the use of color and third dimension," in *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change'*, Aug 1999, pp. 99–108.

[31] S. P. Reiss, "An engine for the 3D visualization of program information," *J. Vis. Lang. Comput.*, vol. 6, no. 3, pp. 299–323, Sep. 1995.

[32] P. Young and M. Munro, "Visualising software in virtual reality," *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98*, pp. 19–26, 1998.

[33] O. Greevy, M. Lanza, and C. Wysseier, "Visualizing live software systems in 3d," in *Proceedings of the 2006 ACM Symposium on Software Visualization*, ser. SoftVis '06. New York, NY, USA: ACM, 2006, pp. 47–56.

[34] C. Mesnage and M. Lanza, “White coats: Web-visualization of evolving software in 3d,” in *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, Sep. 2005, pp. 1–6.

[35] M. Lungu, M. Lanza, T. Gîrba, and R. Robbes, “The small project observatory: Visualizing software ecosystems,” *Science of Computer Programming*, vol. 75, no. 4, pp. 264 – 275, 2010, experimental Software and Toolkits (EST 3): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT 2008).

[36] T. Panas, R. Berrigan, and J. Grundy, “A 3d metaphor for software production visualization,” in *Proceedings on Seventh International Conference on Information Visualization, 2003. IV 2003.*, July 2003, pp. 314–319.

[37] T. Panas, R. Lincke, and W. Löwe, “Online-configuration of software visualizations with vizz3d,” in *Proceedings of the 2005 ACM Symposium on Software Visualization*, ser. SoftVis ’05. ACM, 2005, pp. 173–182.

[38] T. Panas, T. Epperly, D. Quinlan, A. Saebjornsen, and R. Vuduc, “Communicating software architecture using a unified single-view visualization,” in *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, July 2007, pp. 217–228.

[39] G. Langelier, H. Sahraoui, and P. Poulin, “Visualization-based analysis of quality for large-scale software systems,” in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’05. ACM, 2005, pp. 214–223.

[40] R. Wettel, “Visual exploration of large-scale evolving software,” in *2009 31st International Conference on Software Engineering-Companion Volume*. IEEE, 2009, pp. 391–394.

[41] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring, “Live trace visualization for comprehending large software landscapes: The explorviz approach,” in *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 2013, pp. 1–4.

[42] Y. Tymchuk, A. Mocci, and M. Lanza, “Vidi: The visual design inspector,” in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE ’15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 653–656.

[43] K. Maruyama, T. Omori, and S. Hayashi, “A visualization tool recording historical data of program comprehension tasks,” in *Proceedings of the 22Nd International Conference on Program Comprehension*, ser. ICPC 2014. ACM, 2014, pp. 207–211.

[44] J. Vincur, P. Navrat, and I. Polasek, “VR City: Software analysis in virtual reality environment,” in *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, July 2017, pp. 509–516.

[45] M. J. Rochkind, “The source code control system,” *IEEE transactions on Software Engineering*, no. 4, pp. 364–370, 1975.

[46] S. Demeyer, S. Ducasse, and O. Nierstrasz, “Finding refactorings via change metrics,” *ACM SIGPLAN Notices*, vol. 35, no. 10, pp. 166–177, 2000.

[47] Z. Xing and E. Stroulia, “Understanding phases and styles of object-oriented systems’ evolution,” in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 2004, pp. 242–251.

[48] G. Antoniol, M. Di Penta, and E. Merlo, “An automatic approach to identify class evolution discontinuities,” in *Proceedings. 7th International Workshop on Principles of Software Evolution, 2004.* IEEE, 2004, pp. 31–40.

[49] H. Gall, M. Jazayeri, R. R. Klosch, and G. Trausmuth, "Software evolution observations based on product release history," in *1997 Proceedings International Conference on Software Maintenance*. IEEE, 1997, pp. 160–166.

[50] M. Lanza, "The evolution matrix: Recovering software evolution using software visualization techniques," *International Workshop on Principles of Software Evolution (IWPSE)*, 09 2001.

[51] R. Holt and J. Y. Pak, "Gase: visualizing software evolution-in-the-large," in *Proceedings of WCRE '96: 4rd Working Conference on Reverse Engineering*, Nov 1996, pp. 163–167.

[52] M. Lanza, "The evolution matrix: Recovering software evolution using software visualization techniques," in *Proceedings of the 4th International Workshop on Principles of Software Evolution*, ser. IWPSE '01. New York, NY, USA: ACM, 2001, pp. 37–42.

[53] M. Fischer and H. Gall, "Visualizing feature evolution of large-scale software based on problem and modification report data," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 16, no. 6, pp. 385–403, 2004.

[54] M. D'Ambros and M. Lanza, "Reverse engineering with logical coupling," in *Proceedings of the 13th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 189–198.

[55] M. D'Ambros, M. Lanza, and M. Lungu, "Visualizing co-change information with the evolution radar," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 720–735, Sep. 2009.

[56] A. Gonzalez, R. Theron, A. Telea, and F. J. Garcia, "Combined visualization of structural and metric information for software evolution analysis," in *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, ser. IWPSE-Evol '09. New York, NY, USA: ACM, 2009, pp. 25–30.

[57] J. P. Sandoval Alcocer, F. Beck, and A. Bergel, "Performance evolution matrix: Visualizing performance variations along software versions," in *2019 Working Conference on Software Visualization (VISSOFT)*, Sep. 2019, pp. 1–11.

[58] R. Novais, C. Nunes, C. Lima, E. Cirilo, F. Dantas, A. Garcia, and M. Mendonça, "On the proactive and interactive visualization for feature evolution comprehension: An industrial investigation," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 1044–1053.

[59] C. V. Alexandru, S. Proksch, P. Behnamghader, and H. C. Gall, "Evo-clocks: Software evolution at a glance," in *Proceedings of VISSOFT 2019 (Working Conference on Software Visualization)*, 2019, pp. 12–22.

[60] W. Scheibel., C. Weyand., and J. Döllner., "Evocells - a treemap layout algorithm for evolving tree data," in *Proceedings of the 13th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 2: IVAPP,*, INSTICC. SciTePress, 2018, pp. 273–280.

[61] M. Sondag, B. Speckmann, and K. Verbeek, "Stable treemaps via local moves," *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, no. 1, pp. 729–738, 2017.

[62] G. Balogh and A. Beszedes, "Codemetropolis-code visualisation in minecraft," in *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2013, pp. 136–141.

[63] T. Munzner, *Visualization analysis and design*. CRC press, 2014.

[64] R. Wettel and M. Lanza, "Program comprehension through software habitability," in *15th IEEE International Conference on Program Comprehension (ICPC'07)*. IEEE, 2007, pp. 231–240.

[65] N. Chen, "Convention over configuration," 2006.

[66] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[67] JetUML [`3f806c9`]. https://github.com/prmr/JetUML/commit/3f806c9.

[68] JetUML [`d1058da`]. https://github.com/prmr/JetUML/commit/d1058da.

[69] JetUML [`12fd601`]. https://github.com/prmr/JetUML/commit/12fd601.

[70] JetUML [`4f83202`]. https://github.com/prmr/JetUML/commit/4f83202.

[71] JetUML [`2fe64aa`]. https://github.com/prmr/JetUML/commit/2fe64aa.

[72] JetUML [`a071695`]. https://github.com/prmr/JetUML/commit/a071695.

[73] cwa [`107e48a`]. https://github.com/corona-warn-app/cwa-server/commit/107e48a.

[74] cwa [`48576b9`]. https://github.com/corona-warn-app/cwa-server/commit/48576b9.

[75] cwa [`a745b31`]. https://github.com/corona-warn-app/cwa-server/commit/a745b31.

[76] cwa [`e8ca246`]. https://github.com/corona-warn-app/cwa-server/commit/e8ca246.

[77] cwa [`eddda9f`]. https://github.com/corona-warn-app/cwa-server/commit/eddda9f.

[78] cwa [`b344143`]. https://github.com/corona-warn-app/cwa-server/commit/b344143.

[79] jib [`9a607f6`]. https://github.com/GoogleContainerTools/jib/commit/9a607f6.

[80] jib [`87b4006`]. https://github.com/GoogleContainerTools/jib/commit/87b4006.