Università
della
Svizzera
italiana

**Software
Institute**

M3TRICITY

A 3D evolution-resistant visualization of software systems

Federico Pfahler

Aug 2020

Supervised by
Prof. Dr. Michele Lanza

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

SOFTWARE & DATA ENGINEERING MASTER THESIS

Abstract

Developing software systems is a complex operation. It requires organization, order and knowledge of what the previous choices have been in order to ensure its correct development. Organization guarantees a proper division of labour, the order, on the other hand, makes it easier to access information about the software itself. In conclusion, knowing what has happened in the past allows improvement, extension but also reuse of previously created structures. For companies, maintaining and passing on all these qualities often has a considerable cost, mainly due to the complexity and the size that a software system can reach over its history. Software system visualizations help to reduce the amount of information that needs to be processed, while at the same time it creates a visible representation of the data extracted from the software system itself.

Visualization approaches that leverage a 3D city metaphor have become popular over the years. We have seen multiple variations of this concept, including virtual and augmented reality. However, all the different interpretations fall short when depicting the evolution of a system. The resulting visualization consists of districts and buildings that move around without a logical sense within the city, leading to ambiguous interpretations of the events that just happened.

In this thesis, we propose a novel approach to model and visualize the evolution of software systems *over time*. The model is used to store the relevant information extracted from the evolution of the software system, making it explorable and accessible. On the other hand, the visualization treats the system as an evolving city, where buildings and districts represent classes and packages, which undergo structural changes over time. It renders with fidelity not only the changes but also refactorings in a comprehensive way.

This thesis is dedicated to my mother Patrizia, my step-father Giuseppe and my father Francesco. Three fantastic people, without whom I could never have graduated. I love you with all my heart, because you have always dedicated everything to me and I could not ask for more.

Acknowledgements

The acknowledgments and the people to thank go here, don't forget to include your thesis advisor...

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

If there is something that history has taught us is that we have to make sense of all the information we collect. This operation can be achieved by reorganizing the data in order to add different logic to its content. The newly created order should facilitate the understanding of the data through the use of mind maps. It should also help create and visualize informational links that may not be so clear in the original form. The same concept applies to software systems. They provide a large amount of information with a structure that is not easy to interpret. Over the past decades, scientists have tried to model new structures in order to represent them visually. In fact, software systems provide a wide range of information, such as how they are composed but also how their composition has changed over time, all this with the use of external and complementary development tools such as versioning systems. Those tools exponentially increase the amount of information on software systems, as they log fine-grained details in each step of the software development cycle. Comments, authors and issues are only some of the additional data that can be found within versioning systems. To simplify the access to this kind of data, we can rearrange it into a new structure so that it is more easily accessible and can also be used for visual display of the information extracted from the unstructured data.

A software system is by itself complex. Building a good system can be even more difficult due to changes that might arise over time, such as a change in the requirements or the need for updating the core logic due to obsolete structures. In Software Engineering we are taught how essential structures are and how they will influence the evolution of the final product, creating a situation in which understanding their evolution becomes vital in order to achieve an extendible and reusable structure. Over time, software systems will inevitably drift away from their original architecture, generating dead code that is not used anymore and that is substituted by a newer one. Reengineering aims to improve the general structure of a software system making it more flexible given the new needs. Being able to reengineer software systems implies having knowledge of its history, but to understand its history we need to have a way to explore it. Software visualization is part of this process and consists in the use of designs and graphs together with human interaction in order to facilitate the comprehension of a software system. The core concept consists into creating a visualization that extracts information and displays it in a meaningful way, reducing the amount of time needed to process the data. To create a visualization that helps to understand the evolution of a software system, we need to consider multiple factors. We describe these in the next sections.

1.1 Software evolution

Software systems evolve over their life-cycles. The concept of evolutions comes handy, especially because software systems need to be maintained and updated frequently to remain attractive to their users. This operation allows companies not to lose competitiveness in respect of competing companies and products while also allowing them to attract new customers. The procedure normally consists of adding and modifying existing code in order to reach the desired result concerning the existing product. Such changes can be considered refactoring of the source code. Changing existing code within a software system is a double-edged sword for every company that begins it. Newly features are added and performance might

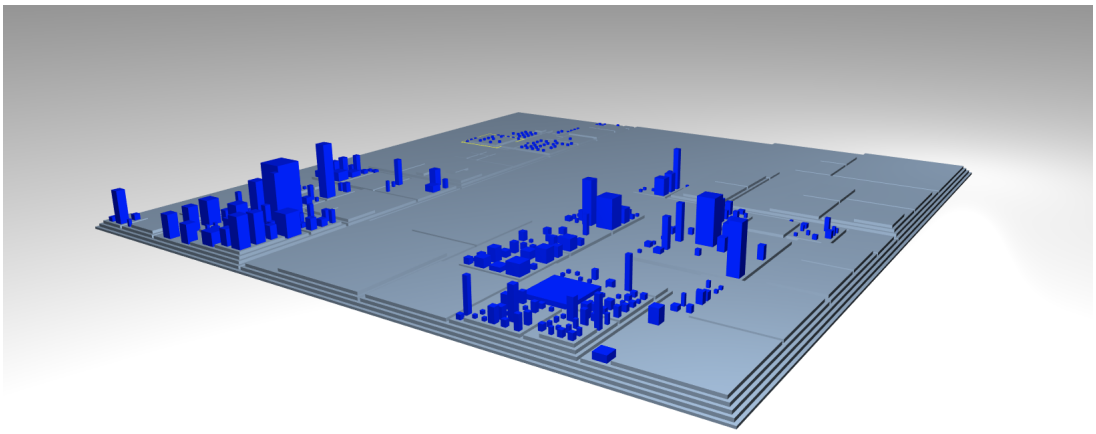


FIGURE 1.1: Screenshot of M3TRICITY

be increased, but new unknowns arise, such as stability of the updated software system when compared to the previous versions. Moreover, Lehman's laws state that as a system evolves over time, more resources are needed in order to maintain a simple and usable structure []. Software systems development implies the need for tracking the changes that happened over its history. This source of information might not only become useful in the case of a breaking change and to understand when changes happened and which components were affected in order to backtrack the sources of possible errors happening at runtime. Versioning systems are the tools that, together with others, give companies the ability to store and access the changes made over time to the source code, ensuring companies the ability to access any version of a software system. In addition, this source of information can be used to visualize the changes that happened over the time.

1.2 Software visualization

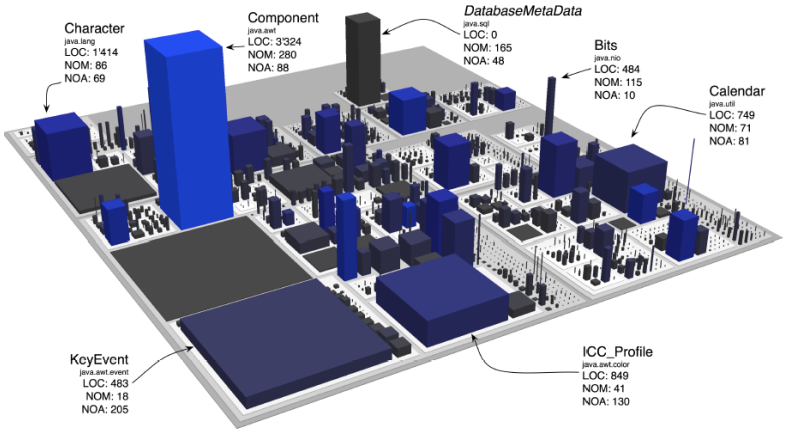
Software systems are made of different text files that are then compiled into code that machines can run and execute. One of the problems of the source code is creating a visualization that correctly represents all the information contained in it even to people that have no knowledge about programming. Therefore, there is a need to make the software system tangible to the developers and the managers who want to understand the system but have no time or knowledge to go through all the source files for every revision. Software visualization is the tool needed to make the system tangible by creating an intuitive visualization. Regarding the software visualization, starting from the 50's multiple approaches have been developed to give insights about the logic and construction of a software system. Due to the size and complexity that software systems have reached in the past years, visualizing them has become always more important for multiple purposes such as maintenance and re-engineering. One of the approaches used in this field was using metaphors in order to visualize software systems. As before, multiple metaphors have been created. In this thesis, we will explore the city-metaphor to visualize software systems.

1.3 Software as cities

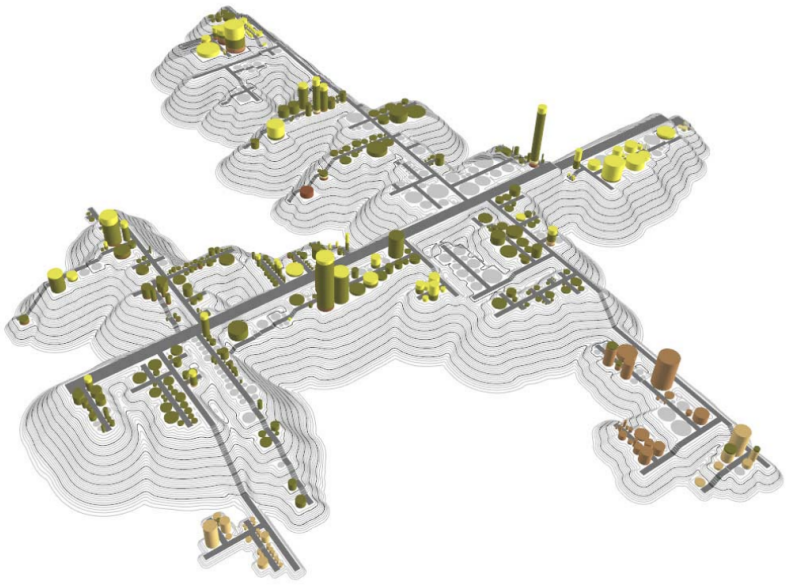
Software as cities is a metaphor-based approach for visualizing software systems that have become always more widely used over the past decades. This metaphor best represents software systems due to the number of similarities that software systems have with cities. A metaphor consists of mapping one domain into another one, creating a stable but straightforward structure that aims at creating alternative forms of

information that are usually more familiar to the users. In software visualization, multiple metaphors have been created, such as cities, and trees or waterfalls.

Software as cities uses the benefits of metaphors to increase the ability to extract information from a system simplifying its understanding. The motivating idea of this metaphor is that the structure of a system is similar to the formation of a city. If we think of cities, we know that each city has multiple districts, where each district belongs to one specific area of the city. At the same time, each building is placed within a district that lies inside a city, becoming part of it and, therefore, of the city. Similarly, software systems are made of folders, that might contain other folders but also source code files. Each file, if placed inside a folder, becomes part of it. In both situations, structures are developed by humans during the planning phase, a phase that needs to be designed for a long time to have the most flexible and functional structure possible.



(A) Codecity [55]



(B) EVO-STREETS [50]

FIGURE 1.2: Software as cities

1.4 An evolution resistant visualization

In this thesis, we demonstrate that it is possible to create an evolution resistant visualization. Current approaches to visualize software systems as cities do not take into consideration the evolution of the system, creating cities that evolve in a not intuitive manner, with situations of ambiguous interpretations. To do so, we decided to implement a layout that uses information provided by the versioning system to place elements within the view. The final result, is a layout that do not change over the time since the initial placement do not change over time.

1.5 Contributions

To prove our thesis, we first started by looking at multiple evolution models, concentrating on Hismo, an evolution model that has been developed by Tudor Girba[22]. Starting from this model, we adapted it while considering that we will be using Git versioning systems, together with the fact that packages and classes histories can be merged due to their internal structure. After developing the model, we started by looking at multiple software systems and we analyzed their history without the notion of visualization. The second phase consisted of creating a flat layout in which each class would have had a predefined space on the backend. We started from a single version of a software system and tried different solutions. Finally, we started developing a layout that uses the previously created evolution model to place the object based on the history data extracted from the system itself to create a resistant visualization.

To apply the previously mentioned approach, we developed M3TRICITY, a software resistant visualization based on the city metaphor and evolution model. We used Java and Javascript as main programming languages, together with srcML to analyze and extract the metrics from the source code.

Based on this, the main contributions are:

- we developed a new evolution model that considers the possibility of merging histories.
- we developed an evolution resistant layout that uses information that can be extracted from the evolution model.
- we developed a city-based visualization for visualizing software systems as cities using web-based technologies.
- we developed a web-based application called M3TRICITY that is able to analyze Java repositories and display them as of cities.
- we evaluated M3TRICITY through analyzing the evolution of real-life software systems

1.6 Document Structure

The document is structured as follow:

- In Chapter 2, we describe the state of the art in the field of software visualization and evolution. We will look at evolution models, 2D and 3D visualizations and some examples of software as cities, we particular attention to CodeCity.
- In Chapter 3, we describe M3TRICITY. We will look at the approach, at the tool itself and the current limitations.
- In Chapter 4, we will look at some case studies. We will analyze some software systems in order to understand how they evolved while explaining particular situations that happened over their evolution.

- In Chapter 5, we conclude the thesis with a summary and present possible directions for future work.
-

Chapter 2

State of the art

Many researchers studied various software visualization techniques which are also important to understand the motivation of our thesis. Therefore, in this section we overview the state of the art in this field.

2.1 2D software visualizations

The concept of *visualizing software* emerged in the '50s. The first approaches used to represent software under the form of diagrams (see fig. 2.1). Haibt created a tool (see fig. 2.1(a)) that was able to facilitate the understanding of a program to developers who were new to it by making use of flow-charts [25]. A couple of years later, Knuth presented a tool that, always under the form of flow-charts, generated and presented the documentation of software programs visually [29]. Still, those approaches were based on flow-chart diagrams. A decade later, a visualization tool that made use of Nassi-Schneiderman diagrams (see fig. 2.1(b)) was described by Nassi *et al.* [40].

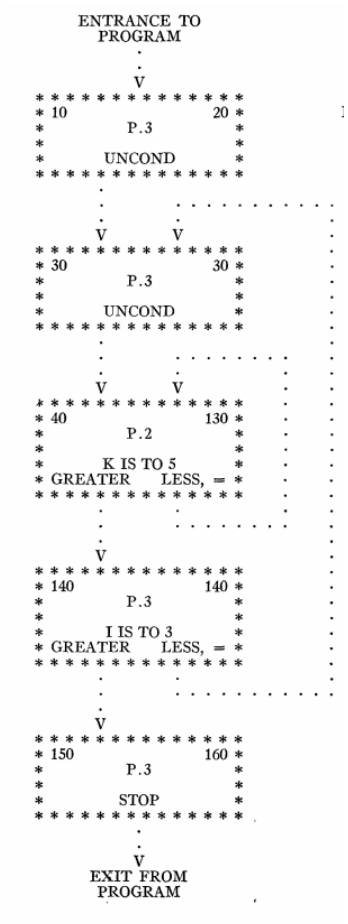
During the '80s graphical user interfaces for visualizing software systems started to appear, moving away from the previously used pretty-print style. Knuth's WEB system [30] used as the main visualization tool a markup language that combined source code and documentation. In this period, software visualization research was mainly focused on program behaviour [56], such as the visualization of sorting algorithms [9]. Muller *et al.* presented a tool that visualized the software's components and the relationship among them [38, 39].

In the '90s, the field of software visualization gained more interest, generating multiple different results. This decade was also the first where 3D visualizations appeared. Eick *et al.* created SeeSoft that provided fine-grained visualizations [15]. The tool is able to visualizse multiple information about the evolution of a system. GASE, created by Hold and Pak, was a 2D representation of the architecture over time. In 1999, Lanza's thesis proposed a tool that constructed 2D trees based on the metrics extracted from the source code [34]. As an example of more recent 2D visualisations, Holten *et al.* presented Extravis [27], a tool for visualising program traces.

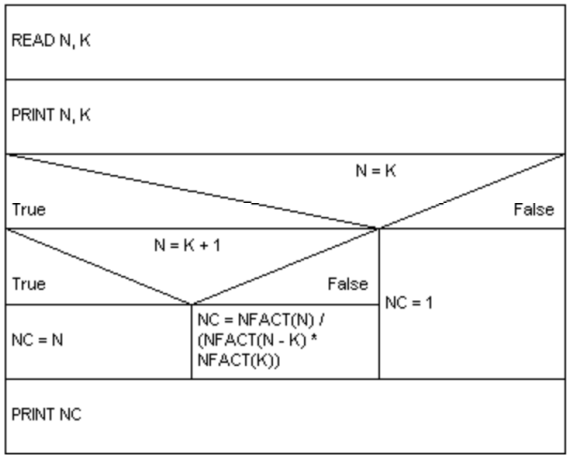
The born of UML influenced the fma research of software visualization. Even if it wasn't born as a software visualization language, it was able to display information about static and dynamic aspects of software systems. In the following years, approaches have been made by making use of different diagrams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

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, that can be see in Figure 2.2, 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[51]. It gained fame due to an appearance inside the movie Jurassic Park in 1993.



(A) Flow-chart [25]



(B) Nassi-Schneiderman Diagram [40]

FIGURE 2.1: First 2D visualizations

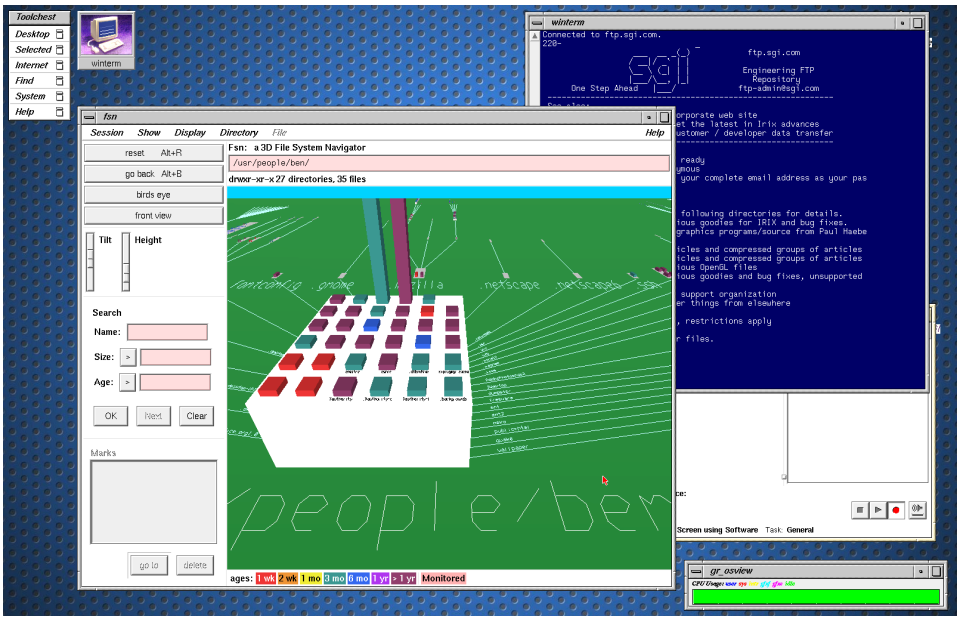


FIGURE 2.2: File System Navigator UII

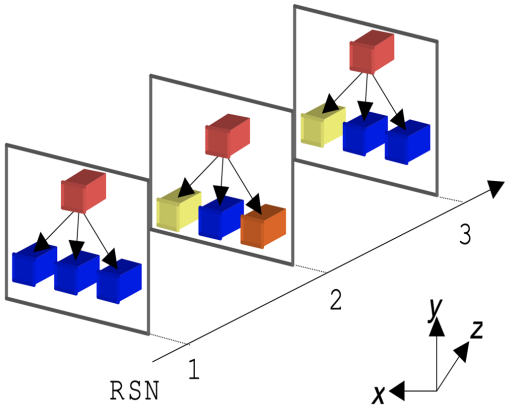
Gall *et al.* created a tool that used both 2D and 3D visualizations (see fig. 2.3(a)) of software systems [21]. Hardware limitations were quite evident during that decade as visualizing complex information on 3D planes also required complex computations. In 1995, Reiss presented PLUM, a 3D customizable engine that displayed a system's information under multiple forms (see fig. 2.3(b)) [45]. At the same time, Young & Munro started exploring the virtual reality world while visualizing software systems (see fig. 2.3(c)) [58]. At the beginning of 2000 software visualization got more attention, with the creation of dedicated venues. Greevy *et al.* proposed TraceCrawler (see fig. 2.3(d)), a tool that analysed the interactions between components statically and dynamically [24]. Up to now, multiple and different techniques have been developed to display various types of information about software systems. Still, one of the challenges was to find the granularity of information to visualize due to the hardware limitations. This problem was also reflected in software evolution visualization since the amount of data that needed to be processed was significantly larger.

At this time 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 [37]. Lungu *et al.* developed the Small Project Observatory, another project that visualized 3D information on the web [35]. The metaphor-based visualization was also already becoming popular. 3D, when compared to 2D, permitted the immersion into the data that could be used as a starting exploration point [55]. Knight *et al.* created Software World, where software systems are displayed as cities, with buildings, trees and streets [28]. 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.

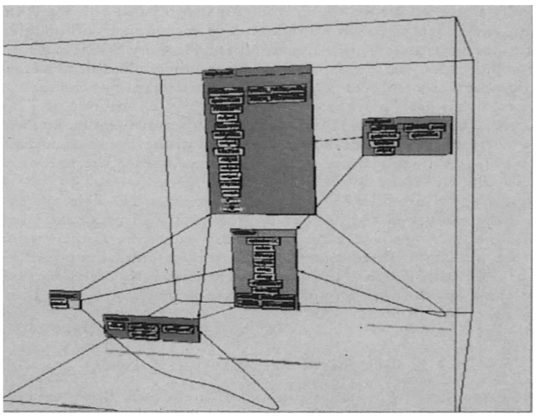
In 2003, Panas *et al.* was the first one who depicted a software system as a city, with real information about static and dynamic data [42, 44, 43]. Langelier *et al.* presented another landscape metaphor with cities in mind. The tool's name was Verso and was mapping metrics to blocks. This tool was also the first one that inserted in the 3D plane elements based on specific criteria to have a significant displacement of blocks [31]. In 2007, Wettel *et al.* presented their approach of visualizing software systems as cities. Based on the city metaphor (see fig. 1.2(a)), their aim was 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 [56]. Two years later, they presented another paper in which they used the same tool to analyze and visualize the evolution of the systems [54]. This approach was not resistant to time changes, creating situations in which the entire city was moving to another place in the plan during the time. 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 fig. 1.2(b)). In this case, time was mapped to the height of the hills on which the class was sitting [50]. As a downside, their approach was not displacing elements on the map in a smart way, creating cities with disconnected neighbourhoods. By making use of 3D blocks, Fittkau *et al.* presented ExploraViz, a tool for visualizing traces using both 3D and 2D visualizations (see fig. 2.3(e)) [18]. In 2015, Tymchuk *et al.* proposed ViDI, a tool for quality analysis that made use of 3D visualizations (see fig. 2.3(f)) [52]. In 2012, Erra and Scaniello proposed CodeTrees, a visualization of the software system under the form of trees [16]. Two years later, always using the tree metaphor, Maruyama *et al.* [36] 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 [53].

2.3 Evolution models

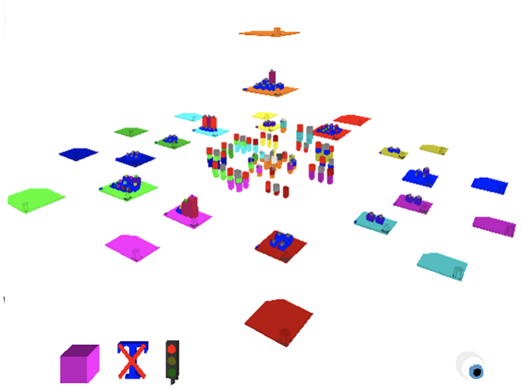
The first versioning systems appeared during the 1970s. At this time, it became clear that there was the need of being able to go revert changes that have been done previously to source code. The first versioning system was SCCS and introduced some concepts that are still used today [46]. SCCS made it 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.



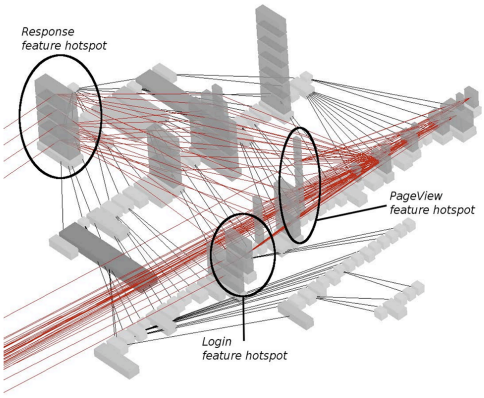
(A) Software Release History [21]



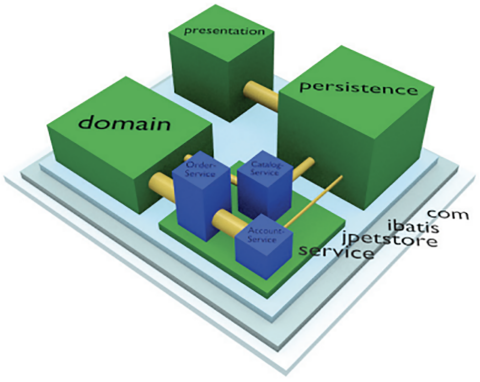
(B) PLUM [45]



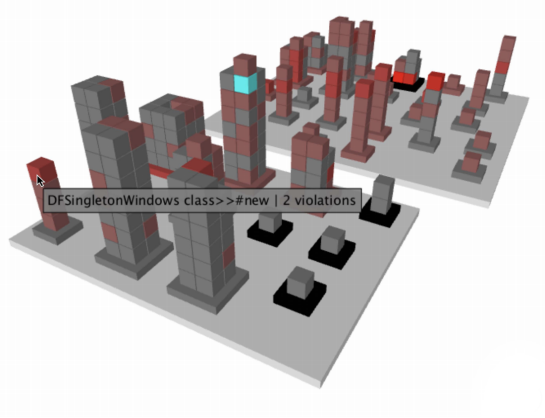
(C) FileVis [58]



(D) TraceCrawler [24]



(E) ExploraViz [18]



(F) ViDI [52]

FIGURE 2.3: 3D visualisations

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 [14]. Xing & Stroulia extended this by comparing different versions at the same time, going more fine-grained in the detection of changes [57], while Antonial and Di Penta used word similarities always to detect renames, split or merge class [8].

2.3.1 Evolution Charts

Evolution charts were used to display information about software systems. They consisted of charts where on the x-axis time was represented while metrics were represented on the y-axis Figure 2.4. Gall *et al.* also implemented an approach to visualize the evolution of software systems [20]. His approach has also been widely used but has the limitation that only one property can be visualized at a time.

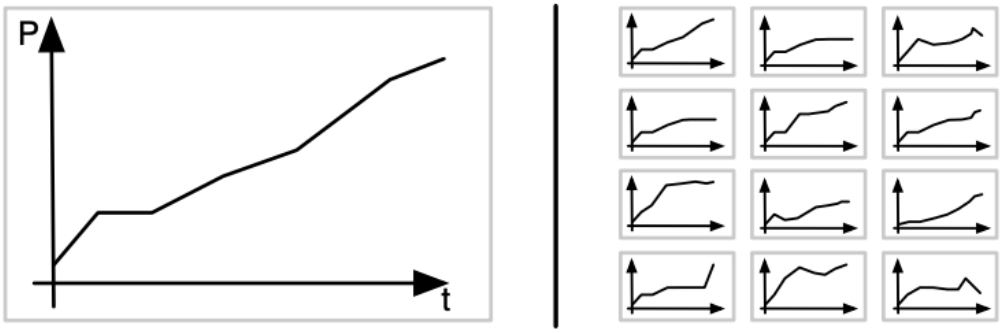


FIGURE 2.4: Example of an evolution chart. On the x-axis we have the representation of time while on the y-axis we have the property that we are measuring. On the right, we can see other measurements.

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 2.5 [33]. Every row represents a class while each column represents a version of that class. The rectangle itself can, therefore, represent multiple properties at the same time by making use of the width/height of the rectangle itself. At each version, the size of the rectangle can change depending on the evolution of the class. By using this visualization, we can extract patterns of the classes. The concepts presented in their work to represent software system evolution will be widely used during the development of this thesis due to the simplicity.

2.3.3 Hismo

Hismo [22] evolution model can be considered the first building block of this thesis. Hismo core consists of three main entities (Figure 2.7):

- *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.

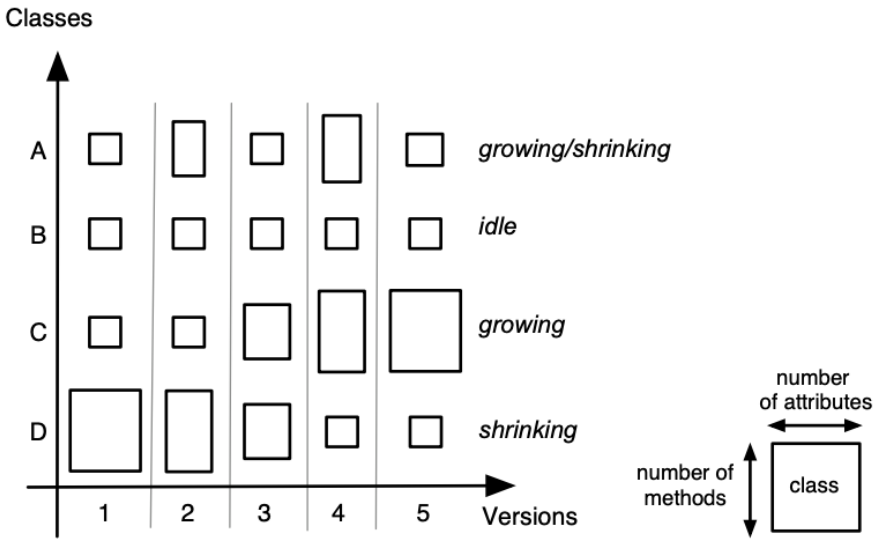


FIGURE 2.5: Evolution matrix in action

In Hismo, the entities are abstract and not tied to any data-models, creating a structure that can be adapted to any entity related to the system that we want to study.

The model can also be mapped to an evolution matrix. It uses the evolution matrix to store information about the evolution and has the concept of packages and classes Figure 2.6. In the first part it is applied to Packages and Classes, while the lower one contains the evolution matrix itself. A row represents the evolution of an entity, while a column represent the version of that entity. In Hismo, a row mean a class history while, a column not only constitute the version but also the complete package version. The whole matrix therefore can be visualized as a package history. This type of abstract model can be modified and applied to multiple fields. In the thesis itself, the author shows how to apply it directly to the evolution of software systems, by using the evolution matrix developed by Lanza *et al.* as a visualization for the model itself.

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 was able to store all the information needed for visualizing the evolution of software systems. We will discuss this in Section 3.1.

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 increased giving at the same time much larger information about their evolution. In 1996, Holt and Pak created the first tool, called GASE, that aimed to visualize information about the evolution of software systems [26]. GASE displayed information about the architecture of the system in multiple versions. Lanza, in 2001, proposed the concept of *evolution matrix*[32]. This "matrix" was able to keep track of the evolution of single classes during the software lifespan and to define the evolution pattern of each file.

In 2004, Fischer *et al.* visualized the evolution of the features of large software systems [17]. One year later, based on the work done by Lanza, Girba *et al.* presented Hismo, a tool for modeling software evolution [22]. 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 visualised the evolution of software bugs and the coupling between classes over time [12]. Later, D'Ambros *et al.* proposed "Evolution Radar", a method to visualize coupling between objects [13]. An approach that displayed

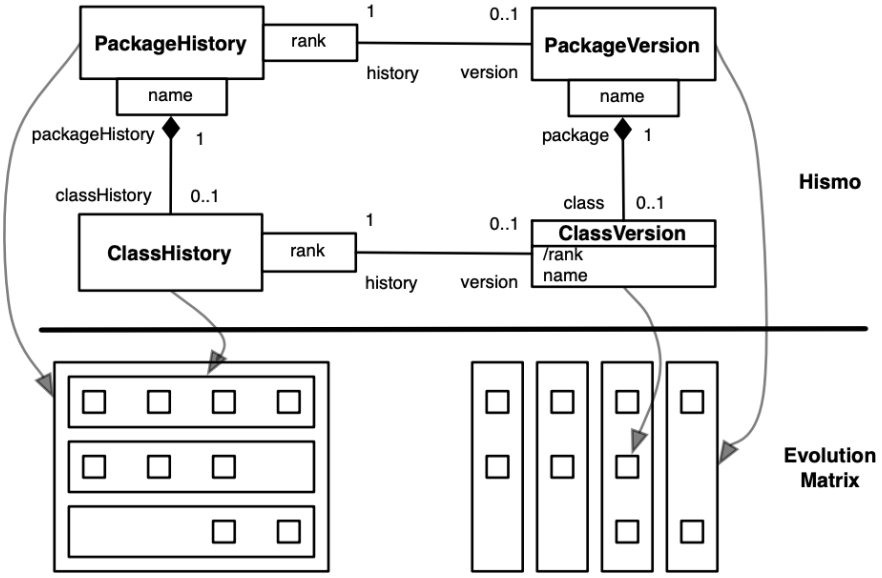


FIGURE 2.6: Hismo mapped to an evolution matrix

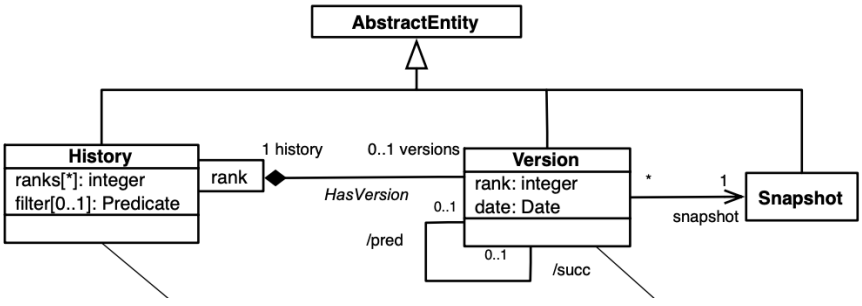


FIGURE 2.7: Hismo core model

metric information was then presented by Gonzalez *et al.* They discussed a 2D visualization of software systems that was able to compare multiple metrics at once all in the same view [23]. Alcocer *et al.* then presented Performance evaluation matrix, that aimed to display the performance of software systems during their evolution [47]. Novais *et al.* proposed a proactive and interactive visualization strategy for software evolution [41]. 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 [7]. Ivapp *et al.* presented EvoCells, a tool that visualizes changes using treemaps [48]. In addition to the classic 2D visualization, the work done by Sondag *et al.* is interesting to mention, because even if it was not really related to the visualization of software systems, they implemented a stable visualization that makes use of treemaps [49]. In their publication, the authors propose alternatives ways to construct treemaps for hierarchical data which algorithm try to modify the structure as little as possible when its internal data changes. EVO-STREETS, on the other hand, can be see in Figure 1.2(b) [50], in this case, the authors decided to map evolution on the height of the hills, where where a higher hill represents a more recently modified class. Other application of this metaphor were found within CodeMetropolis too [10].

2.4.1 CODECITY

The core concepts of CODECITY are widely used for the visualization of this thesis. CODECITY is a language-independent 3D visualization tool for large software systems. CODECITY offers the possibility to explore the software as it was a real city by offering a sense of locality. A building represents a class while districts represent the packages or folders containing the files. Figure 2.8 shows the user interface of CODECITY, while Figure 1.2(a) shows an overview of a city analyzed with CODECITY. The image shows the many similarities CODECITY has in common with M3TRICITY. Still, when visualizing the evolution of software systems we can see that their movements do not have a real logic.

I need to find an image that shows last phrase, f1-csaba

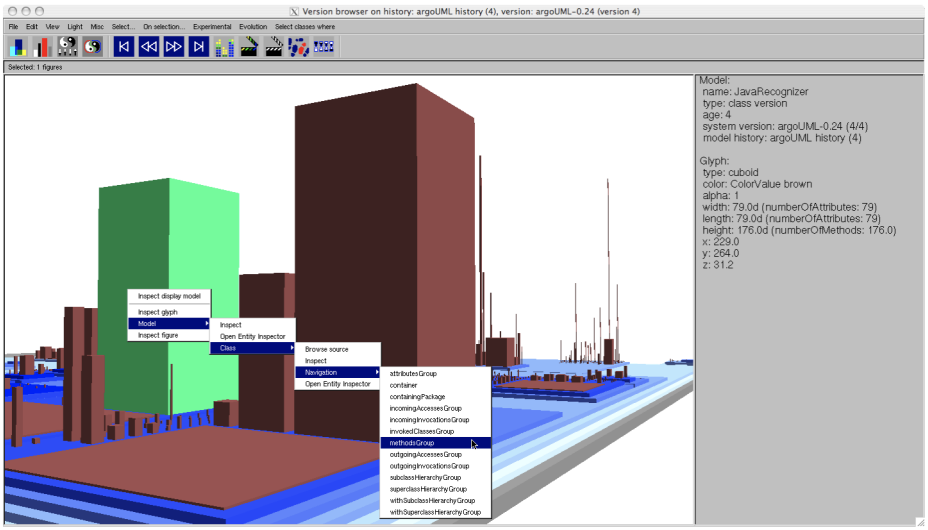


FIGURE 2.8: CODECITY in action

2.5 Summary

The previously mentioned techniques mostly focus on how and what to present of a software system, but their presentation did not handle the evolution of the system properly. Regarding software as cities, CODECITY is able to display the evolution of a system, but the visualization is not time resistant, with blocks placed changing positions 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, our approach displays information consistently over time. The initial layout of the city will be computed by taking into consideration the history of the software system so that blocks positions will be consistent over the complete evolution visualization.

Chapter 3

Metricity

In this chapter we present the tool we developed in order visualize software systems. The goal of the thesis is to shows that the layout of a software system visualized using the city metaphor can be created in advance by using the information provided by its versioning system. In the past, numerous variation of visualizations based on the city-metaphor have been developed. Despite its popularity, the city-metaphor falls short when depicting the evolution of software systems, which results in buildings a districts moving around in unpredictable ways throughout the revisions of a system.

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.

In Section **??** we present the goal of this thesis. In Section 3.1, we discuss the approaches used to model and visualize the history of software systems. In Section 3.2 we discuss the actual tool, by looking at its architecture, at how history was modeled practically but also features and user interface. Concluding, in Section 3.3 we discuss the limitations that have been discovered, together with some extra work that should be explored in order to achieve a better structure for the history.

3.1 Approach

Software systems comprehension is difficult due to their size and complexity. Companies frequently have to deal with large software systems. When dealing with such large systems, they might encounter different problems, such as the introduction of new developers to the existing system and code refactoring in order to increase certain qualities. Current approaches use visualization to provide information about software systems in a non-consistent way, generally by displaying the evolution under the form of multiple single snapshots, where information between two snapshots is not well structured resulting in particular representations. Visualizations 2D as opposed to 3D, do not give the possibility to explore the environment due to the intrinsic limitation the a 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 explore the world as if one can immerse herself inside a new world. At the same time, they give much more possibilities to develop new features. The current solutions in this field are usually static, therefore 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 map the two domains easily. 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. By using this simple mapping, we can construct a city using the information provided by the software system. In theory, the mappings can be even more in-depth, with, for example,

methods representing the floors of a building or roads representing the internal relationship between two different classes.

An evolution-resistant layout, on the other hand, lets the user know where each element is placed with respect to its evolution. With this approach, elements within the city are not be placed randomly for each revision but are placed on a specific spot that is consistent over time. In this case, attention must be given to cases in which a file has been renamed or copied. This solution also helps reduce the amount of space that is required by the visualization itself. In fact, our approach implements an algorithm that tries to pack in a minimal space all the elements.

Combining the resistant layout with the evolution model is the key operation to give more knowledge about software systems. A resistant layout cannot exist without the evolution model, and the evolution model might not make sense if not visualized. In the next sections, we present our approach to model the evolution and visualization of software systems using these two concepts.

3.1.1 Evolution Model

The evolution model follows the logic of Hismo, an evolution model for software systems developed by Tudor Girba[22]. Recall that this model has three main concepts:

- Snapshot. Represents the time, is a container of multiple versions of the software system
- History. It is a set of versions. History represents the evolution of a specific entity, 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.

Hismo is a good starting point since its model is already applied to software systems, as can be seen in Figure 2.6. It was developed based on the Subversion (SVN) versioning system. 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, only the modified files would get the version field updated. This notable difference influences our model too.

Our model also 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 Girba *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 moved away from the abstract model proposed in Hismo while at the same time, we made it more suitable for describing software systems.

Derived evolution model

Hismo model was based on SVN, while today's standard is Git, a more recent and capable versioning system. Given this, some changes needed to be made in order to port Hismo to the new versioning system. The derived model has the underlying logic in common with the original, but differs in the final structure and in how information is stored. By dropping the concept of snapshot in favor of the RepositoryHistory, we reduced the level of complexity, so now we only have to deal with histories and versions. In addition to the RepositoryHistory we also have the concept of the Repository itself. The Repository is where everything gets connected: it can be seen as the superset of all histories. The entity is responsible for accessing any data that is contained about the repository. The RepositoryHistory, contrarily, is the substitute of Hismo's Snapshot. In this case, the RepositoryHistory follow the same logic of PackageHistories and ClassHistories, where each one has a series of version. In our case, the 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 name and owner of the repository together with a list of RepositoryVersions. By using this as an entry point for the history of any repository, 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, consisting in the smallest granularity. It represents a single file history and therefore can be 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 over the time, such as classes, packages but also the repository. Still, the structure, as described above, does not give information about how Version are defined. Versions, are not always the same, and each RepositoryVersion, PackageVersion and ClassVersion entity needs to be slightly different. For this thesis, versions have been constructed in order to explore the content over time and have been defined in the following way:

- RepositoryVersion. It is part of a single RepositoryHistory as has a unique number representing its version. It points to a series of PackageVersions.
- PackageVersion. It contains a list of ClassVersions, in this way for each package version we can navigate to the classes contained in that package.
- ClassVersion. Is the actual class contained in the file and is used to construct the information also about packages. Information about packages and repository are all extracted by the changes that occurred on files.

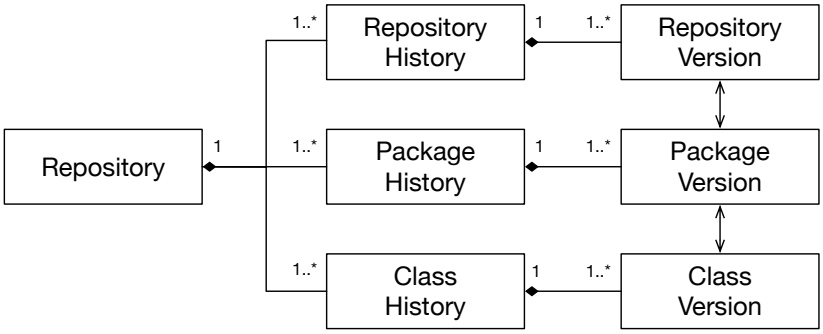


FIGURE 3.1: The Evolution Model of M3TRICITY

Figure 3.1 shows the final relations among entities in the evolution model. History consists of a series of versions, where a version constitute changes made in one or more objects within the system. Starting from the Repository, we can access all its histories. The RepositoryHistory is a succession of RepositoryVersions. Similarly, PackageHistory and ClassHistory have their own set of versions. ClassVersion represents the smallest entity. In fact it only represent a single file within the system, while PackageVersion represents a sets of files. Similarly, a RepositoryVersion contains a set of PackageVersions. By traversing the RepositoryHistory, we can extract all the changes that happened over time.

Defining Histories

Defining History entities is not straightforward in all cases. 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 (or both)).
- Rename. A file path has changed (whether the location or the filename has been changed (or both)).

Given this base information, we can extract 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. An *add* consists of a new ClassHistory. Regarding PackageHistories, and *add* do 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 it 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. When dealing with renames and ClassHistory, nothing special happens. We use the information provided by the versioning system about the old name and we continue that history. Regarding PackageHistories, if all files contained in a specific version have been renamed, it represents a continuation of the history. In contrast, only a few have been renamed, it is a history split. Copy, on the other hand, it also represents a history split since a file starts to appears in multiple locations.

Figure 3.2 shows an example of row creation of a ClassHistory. 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 color). 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 (history) is split into two (pink color). 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 affected the evolution. The change in name of the file does not implies a change in the history. A rename is only used to keep track of the actual name of the entity and might be used for understanding the location (package) of the file. Concluding, the *delete* operation consists in a end to the evolution of that History when no copies have been created. If copies are found, it merely represent a deletion of a children.

Detecting changes: merging package histories and versions

In order to detect changes within the development of a software system, we can use the software versioning system. The current trend is to use Git[1], a distributed version control system that tracks changes in the source code. Still, the system itself has some difficulties in detecting renames and copies, but we will discuss this in the next sectiono.

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

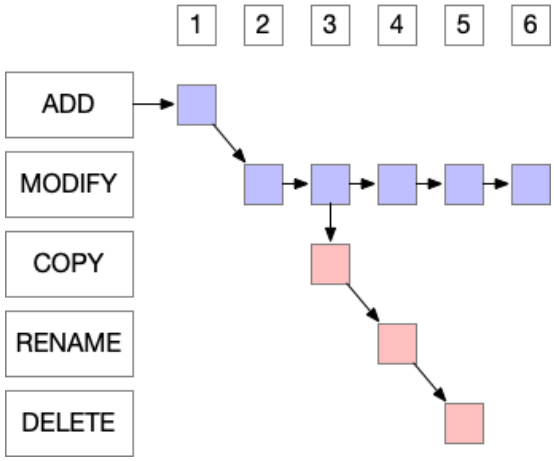


FIGURE 3.2: Example of row creation in M3TRICITY when a copy operation happens

The simplest history model does not take into consideration that package histories and class histories can be merged, a crucial part of having a correct representation of the evolution of a software system. The history of a file can be seen as a linked list, where a file gets inserted, modified up until no other elements are added. In fact, in most of the cases, a file is either added, modified or deleted and it is really difficult to find situations in which files are copied or renamed. Copy and rename operations happen during refactoring operations that are not the standard during software development. A rename also results in merely adding a new element to the list due to the internal composition of the model.

In our model, every time we detect a new file path, we create a new history in which information is added. If we imagine our model, then every linked list is a row within the evolution matrix. When a file is renamed, in the model it becomes a merging of rows, in which two histories, that seems completely separated, are merged into a single one at a specific moment of the evolution. Such a representation that is linear with respect to time as can be seen in Figure 3.3 .

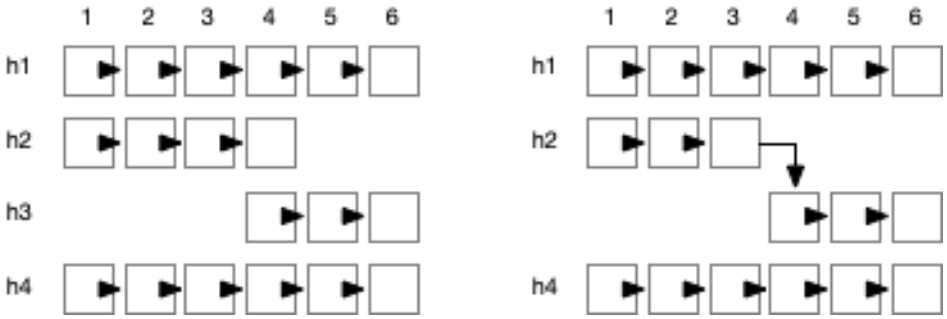


FIGURE 3.3: Example of merged class histories within an evolution matrix. Two separated histories are merged at version 4 due to the presence of a RENAME.

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 did not exist before in the history. From this point on, we need to keep track of every new addition to the same path, together with all modifications and copies. An end to its history is 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 therefore 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 into different locations and in different packages. As an example, suppose we have package A that contains

files A1, A2, A3. During a code refactoring, A1 is eliminated, 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 that B and C are new and created after the rename operation. For this reason and for the sake of the simplicity of the model, we decided to create a new History for every new package that is created, while linking only the files version between the last one in the old package and the new one. Figure 3.4 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, but a version six (r6) a refactoring operation moves the content of h3 into h4 and back to h2. With our solution, we will not deal with such cases, but for a correct representation of the history, we should consider that PackageHistories might be much more complex than ClassHistories.

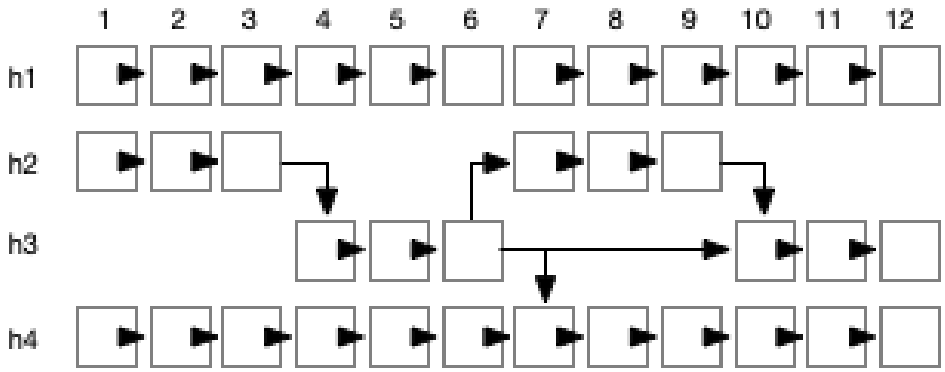


FIGURE 3.4: Example of merged package histories within an evolution matrix

Histories in the model 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 certain time during its evolution.

Histories as unnamed objects

A major difference of our model compared to other ones is that histories are represented as unnamed entities. That is, it is impossible to define a history with a filename or a path, but we can suppose that a history will represent some object within the history of the software system. 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. Physical file names must be considered the tool to build the story, but not the story itself.

Figure 3.5 shows an example of how unnamed entities work. Inside the box, we can see *h0* representing 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` but also `/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.

Export image as PDF, f1-csaba

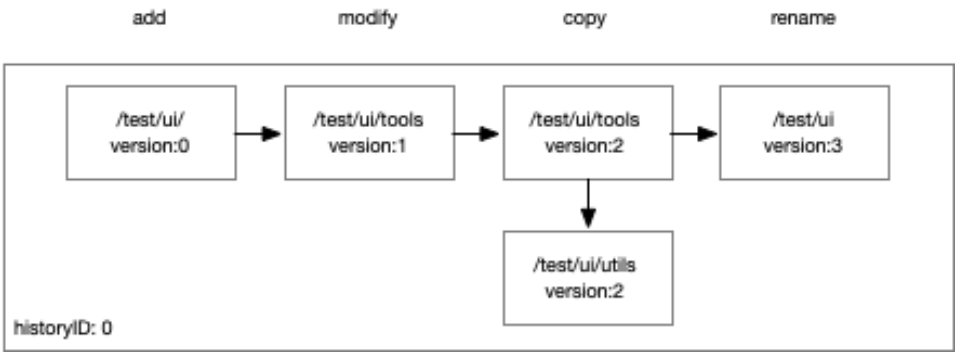


FIGURE 3.5: Histories names representation within M3TRICITY

3.1.2 Visualization

M3TRICITY is a web application for visualizing software metrics. It uses the city metaphor, a way to visualize software systems, that has been widely explored in the past years. To do so, we based our visualization on the basic idea of CODECITY. In addition to this, we also developed a history-resistant layout that leverages evolution matrix to place objects within the view.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. At the end, we will discuss the history resistant layout, the core section of the new visualization and thesis.

CODECITY: software systems as cities

In order to understand the visualization approach, we will start by describing CODECITY, a visualization for software systems 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 such as source files define the main characteristics of a system. Properties of a source file can be easily mapped to a building, for example, by setting a specific metric equal to the height of the building.

In code city, this similarity has been used to develop a visualization for software systems in which a linear mapping among those entities is created in order to transform a folder containing multiple files into a district, while files metrics are used in order to create the buildings. CODECITY didn't stopped 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 in order to understand the system itself.

CodeCity core idea is based on the fact that we need to from learn software systems. To do so, we need to transform them into something that is as similar as possible to concepts that viewer know, like a city. By creating this visual mapping, the amount of information that needs to be processed is reduced, moreover the similarities among a software systems and a city let the user explore it as it would do in real life. In order to create a meaningful mapping, CODECITY approach used a static mapping between entities that can be found in both cities and buildings (Figure 3.8). At domain level we have the two main entities, *software* and *city*. At concept level we have *class* and *package* for software systems, while for cities

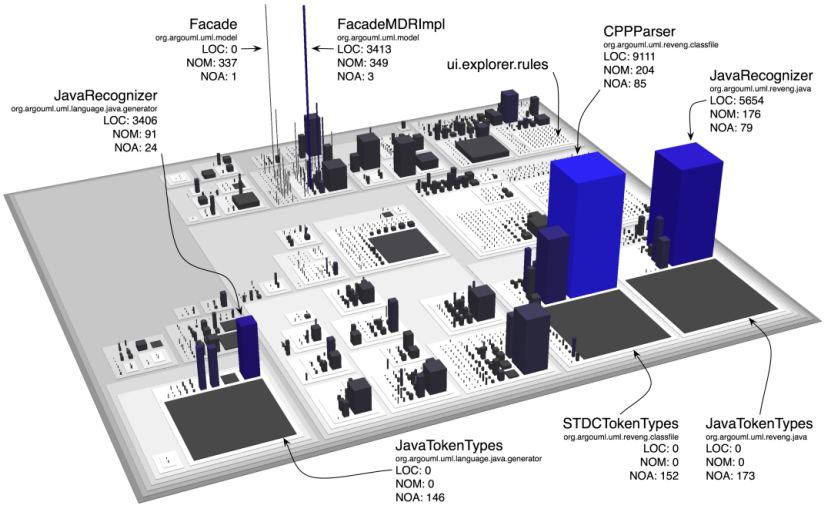


FIGURE 3.6: CodeCity with ArgoUML

we have *building* and *district*. At the lowest level we have the properties for each one of those, that can be for example *number of methods (NOM)* for a class or *height* for a building. In addition to mapping numeric values to shape properties, colors have also been used to give additional information.

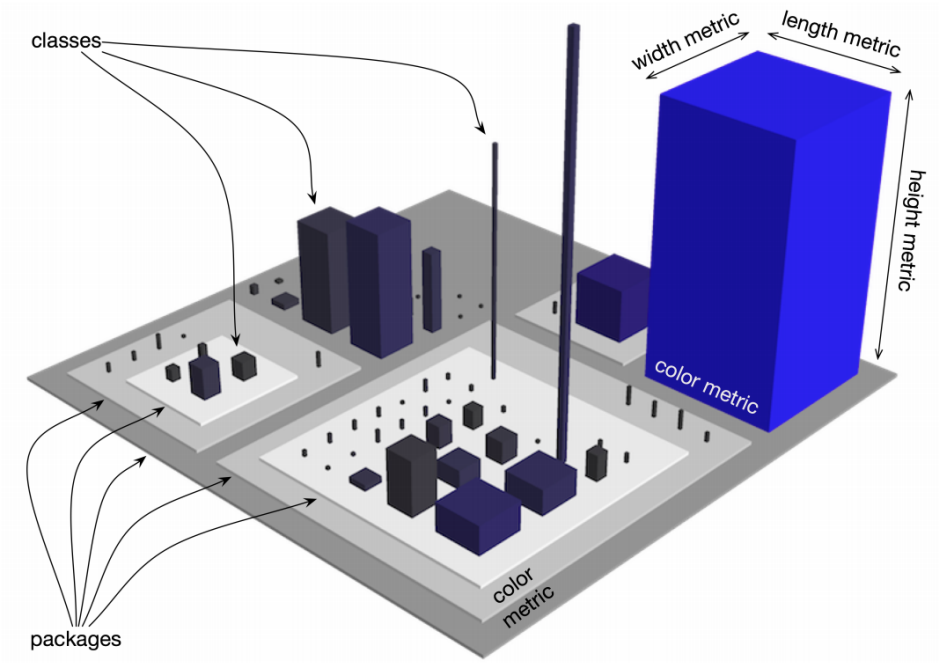


FIGURE 3.7: CODECITY classes and packages

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 3.9 shows the result of the algorithm applied on ArgoUML.

One problem of CODECITY is that it didn't use the evolution data to display the city consistently. In fact, over the evolution, objects such as buildings and districts were moved around, possibly leading to

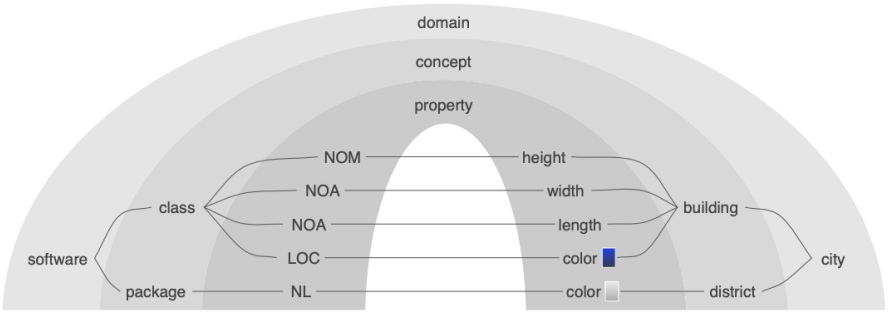


FIGURE 3.8: Mapping from software to cities in CodeCity

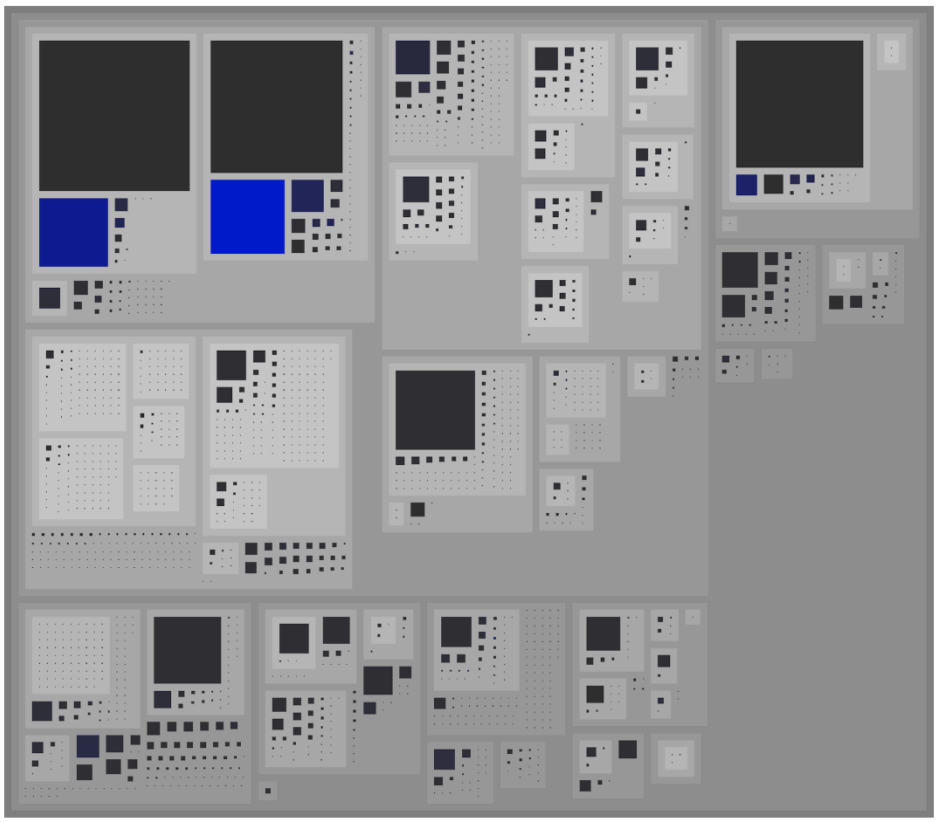


FIGURE 3.9: CODECITY layout algorithm of ArgoUML

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 place elements within the view consistently over the evolution of the software system. Similarly to CODECITY, it is composed by an algorithm consistently. A difference is that it is using that leverages the evolution matrix to understand the final size of each part of the software system within the visualization.

The construction of the layout starts with the analysis of the history of each class (row in the matrix) that is found within the software system. This information is represented directly through ClassHistory entities that are stored within all Repository models. By iterating over all versions of class histories, we can understand how the class evolved over time (Figure 2.5). 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 in order to define a placeholder in the layout for that class, that is the maximum width and depth the object. Software systems do not only contain classes, and most of the time contain a hierarchy of folders mixed up with source files. This can lead to problems within the visualization itself, requiring an additional step in order to define the final size of the packages.

To define the layout of a package within the view, first, given a PackageHistory, we need to iterate over all ClassHistories in it. This can be achieved by by using the inverse property found within each Version entity. At this point, recursively we need to extract and keep track of each maximum ClassVersion contained in each package. In the end, by summing 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 placeholder for packages. It can be seen as a class in the system, but in reality it contain other classes. This is used in order to simplify the creation of the layout, because for each nesting level we will 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, for each package, we will have a list of versions, 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 package to a specific spot inside the layout, so that it will occupy less space as possible. In this thesis we decided to use a simple but fast 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. With this solution we know that the class will not overlap with any other classes because that each spot is assigned only once.

Figure 3.10 shows the differences between a simple history-resistant layout and a bin-packed history resistant layout. In the left side, we can see our implementation of history-resistant layout as opposed to a standard bin-packing algorithm, similar to the one used in CODECITY. As we can see, both algorithms optimize the space.It seems that the standard bin-packing does a better job compared to our history-resistant layout. However, our layout algorithm places the entities in a way that is consistent over time, with the position that stays the same over all revisions.

3.2 Tool

In this section, the implementation background of M3TRICITY and the design choices that have been made throughout its development. In Section 3.2.1 we first describe the architecture of the tool, by looking at the backend and frontend. In Section 3.2.2 we shows and discuss the design requirements. Concluding, Section 3.2.3 we will presents the final user interface of the tool.

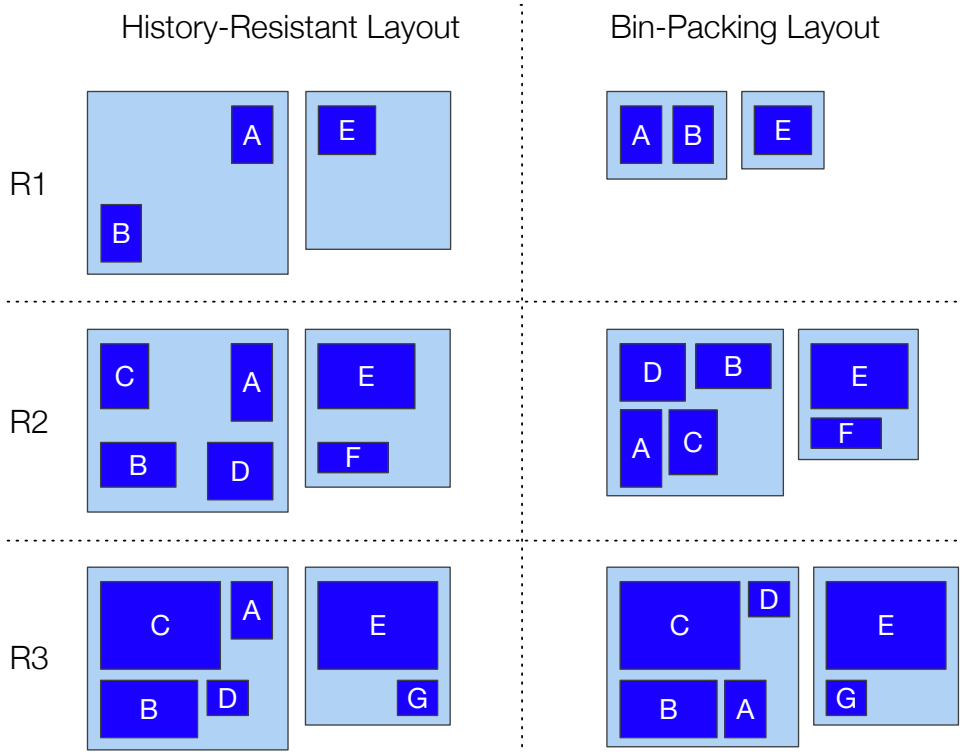


FIGURE 3.10: History-Resistant Layout vs. Bin-Packing

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 3.11 shows a high-level diagram of the application.

The backend is written in Java and constructed using Spring Boot [2] that is Spring's convention-over-configuration [11] 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, view is responsible for preparing the data for users who wants to visualize the evolution of a software system. This module generates the layout given the history of the system that has been previously analyzed and generates a layout that is resistantto 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 the REST API and for the WebSocket communication protocol.

The frontend application is developed in Vue.js [3] an open-source model–view–viewmodel JavaScript framework for building user interfaces and single-page applications. The view itself is simple: the home page consists of a list of analyzed software systems that can be visualized with the possibility of starting a new analysis. The other view is the city metaphor with the resistant layout in which the software evolution can be watched in 3D. The frontend communicates with the backend though some REST endpoints together with sockets which manage the evolution of the city. The frontend also has settings that can be personalized and a chart that shows where changes have been made over time.

[2] See Spring Boot
[3] See Vue.js

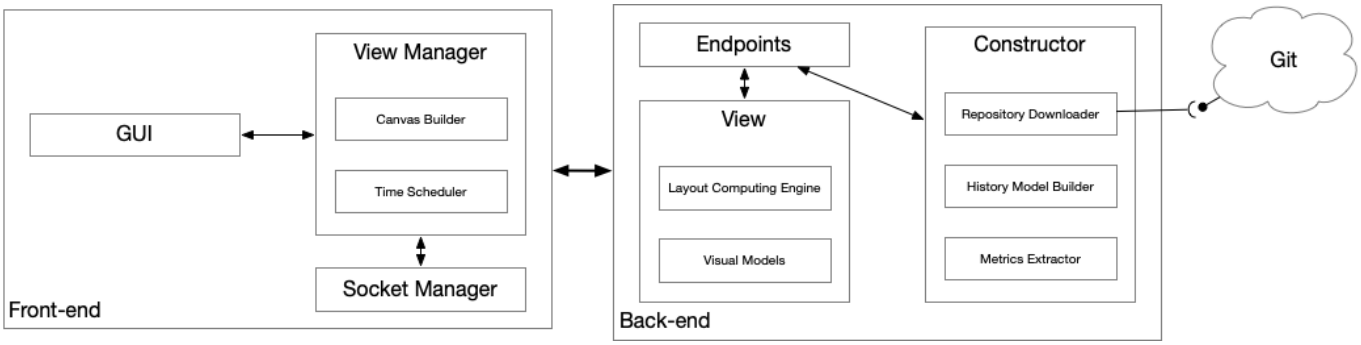


FIGURE 3.11: Architecture of M3TRICITY

Backend

The backend service is implemented using Java [4]. 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.

Spring Boot can be seen as a level of abstraction over the Spring framework. It provides easy-to-integrate functions that can be accessed by using annotations. As an example, SpringBoot has easy access to Services and Components by using the annotations @Service or @Component but also thanks to the ComponentScan. Services and components are singletons that provides general functionalities and are automatically recognized by the application. Dependency injection is used in order to access any of this components.

REST API This type of communication is used in M3TRICITY for retrieving information based on the interactions of the users. In our application, two main HTTP methods that 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 the only POST mapping. It is used to start a new analysis and only requires an URL to 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 in order 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.

[4] See Java

⇒ !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, and paging guarantees fast responses when required.

⇒ !GET /api/repository/repoOwner/repoName/version/terminal! This last endpoint is only used by the visualization itself and is used by the terminal component, which shows the history of the 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 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 evolution of the software system. 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! transfers data about the evolution timeline. It consists of a series of points for each metric. The values for each point are normalized between 0 and 1 and represent the amount of changes for each version.

⇒ !GET /api/repository/owner/name/version/class/timeline/series! is used by the evolution timeline to highlight where changes of a specific class version have happened.

⇒ !GET /api/repository/owner/name/version/package/timeline/series! is used by the evolution timeline in order to highlight where changes involving that specific package version have happened.

⇒ !GET /api/repository/owner/name/timeline/series/mode! This special endpoint is used when bucketing by time is selected. This feature is used in order to create a more compact visualization that by merging events that have happened daily, weekly, monthly or yearly.

⇒ !GET /api/repository/owner/name/version/class/timeline/series/mode! is used to highlight the point when one class is selected in the view and bucketing is active.

⇒ !GET /api/repository/owner/name/version/package/timeline/series/mode! is used to highlight the point when one package is selected in the view and bucketing is active.

To communicate with the backend through WebSockets, the following endpoints are available for subscription:

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

⇒ /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 these sub-modules. Figure 3.12 shows the composition of this module.

M3ConstructStarter is the SpringBoot 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

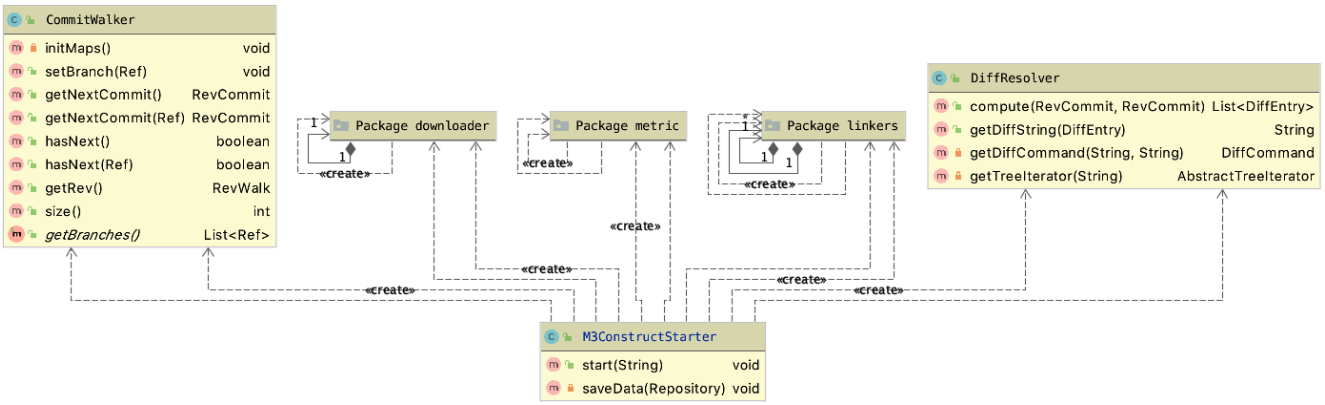


FIGURE 3.12: Constructor UML diagram

with their linkers. This component constructs the RepositoryHistory and save it in the memory. *CommitWalker*, together with *DiffResolver*, is used to iterate over the commits using JGit[5] . DiffResolver makes the diff between the two commits while also using the RenameDetector, which is responsible of detecting renames and copies within two commits. This solution although it is not perfectly precise, aims to solve some problems with git original solution which is not able to detect these two cases safely. This components is also responsible for generating Version entities, since for every commit there is a new version of a file.

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 to extract and compute the metrics from source code that has been checked out locally, and *linker*. Linking histories is a crucial part of this modules. Figure 3.13 shows the internal structure of a linker together with their methods.

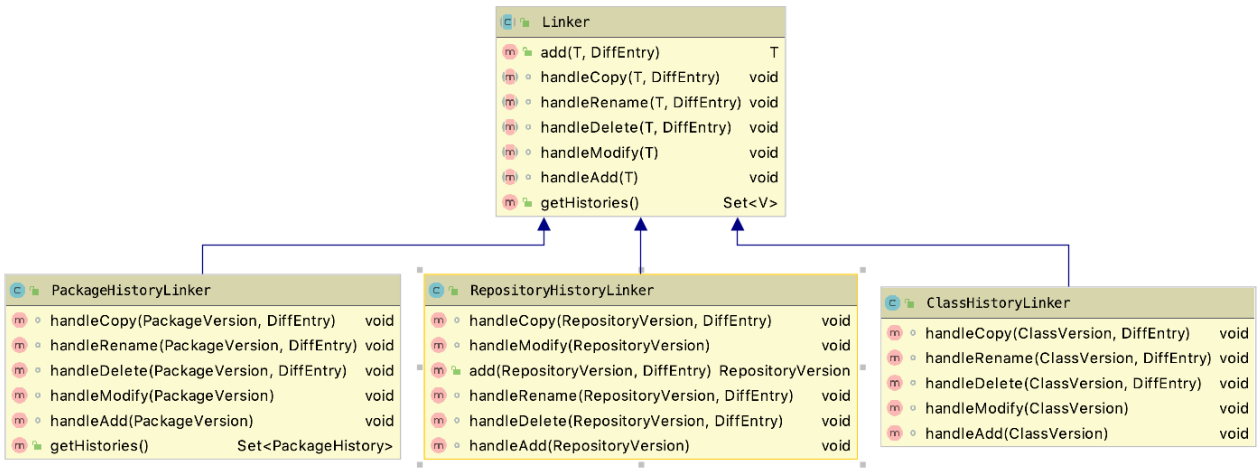


FIGURE 3.13: Linkers UML diagram

This 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. By using Generics, Linker exposes only two 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 path of the file that generated the version. When `add`

[5]See JGit

is called, it will look for the type of Git operation that the version represents and then call the appropriate method.

Figure 3.14 shows the sequence diagram for the creation of the history of a repository. When the controller receives a new analysis requests, it contacts the ConstructStarter that initializes a repository. The acknowledgement is set back to the user as soon as the repository is downloaded. From that point on, the analysis its straightforward with the commit walker that 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.

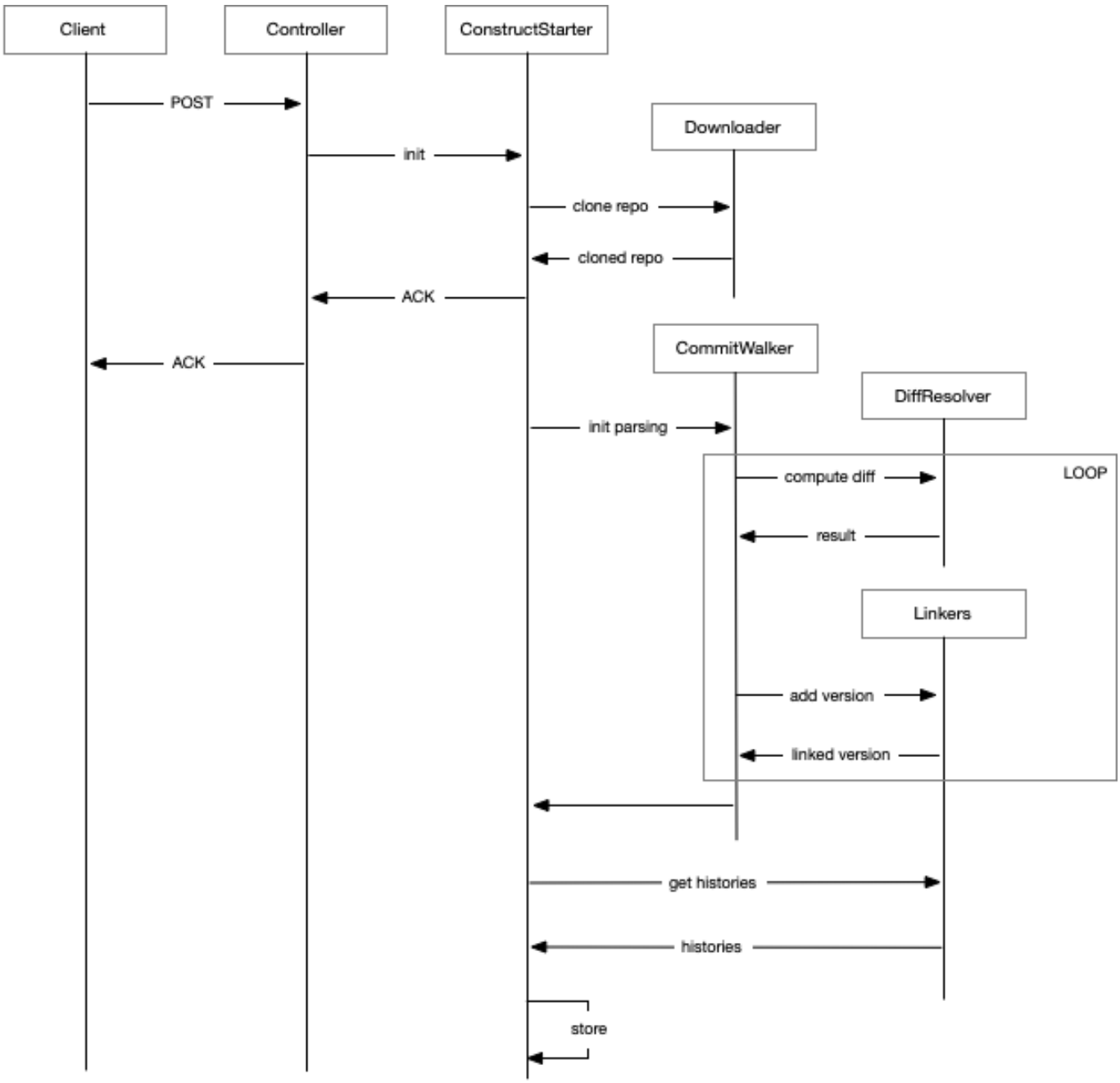


FIGURE 3.14: Sequence diagram for the creation of histories

View module The view module is the second core component of the backend. It is responsible for the analysis of the history, this one 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.

View is an abstract class that contains all the logic for visualizing the software system as it was a city. During the development cycle of M3TRICITY, multiple iterations of different layouts have been developed. Abstract classes cannot be instantiated, thus by extending it, we were able to create a structure that was supporting multiple visualizations. Over the time, the following visualizations have been developed:

- EMView. With this view all classes are placed with a flat layout, i.e., there is no concept of the package. 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 package that contains other classes or packages. With this view, we also started to try to model the visualization while making it resistant. The idea behind this is that we can create a layout using the bin-packing algorithm. We will discuss this in the next paragraph.
- EMVBinPackedViewDate. Extends EMVBinPackedView and has added the function of bucketing the evolution data by day, week, month and year, reducing the amount of information processed inside the view.

The view uses the *packing* module to compute the smallest layout for the view. 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 dependent on their evolution. The other module, *glyphs*, contains the cuboids that will be displayed on the frontend as the result of the computations of the view.

Figure 3.15 shows the sequence diagram in 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 component 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 assign a location to every package and class of the system. When the /update is ready, it will provide the 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 discussed model is the result of a series of multiple iteration of the model itself, in which at every iteration the structure became more manageable and apparent. Still, the resulting approach needed to be implemented using Object-Oriented principles. For this reason, in this paragraph, we discuss more in detail the data contained within each of the entities inside the model.Figure 3.16 shows the model derived from the evolution model.

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

History entities consitute the evolution of single versions that are found within the software system. In M3TRICITY, we have RepositoryHistory, which is a single entity, and multiple PackageHistories and ClassHistories, each one representing a package or a class. Entities will contain a set of Version entities,

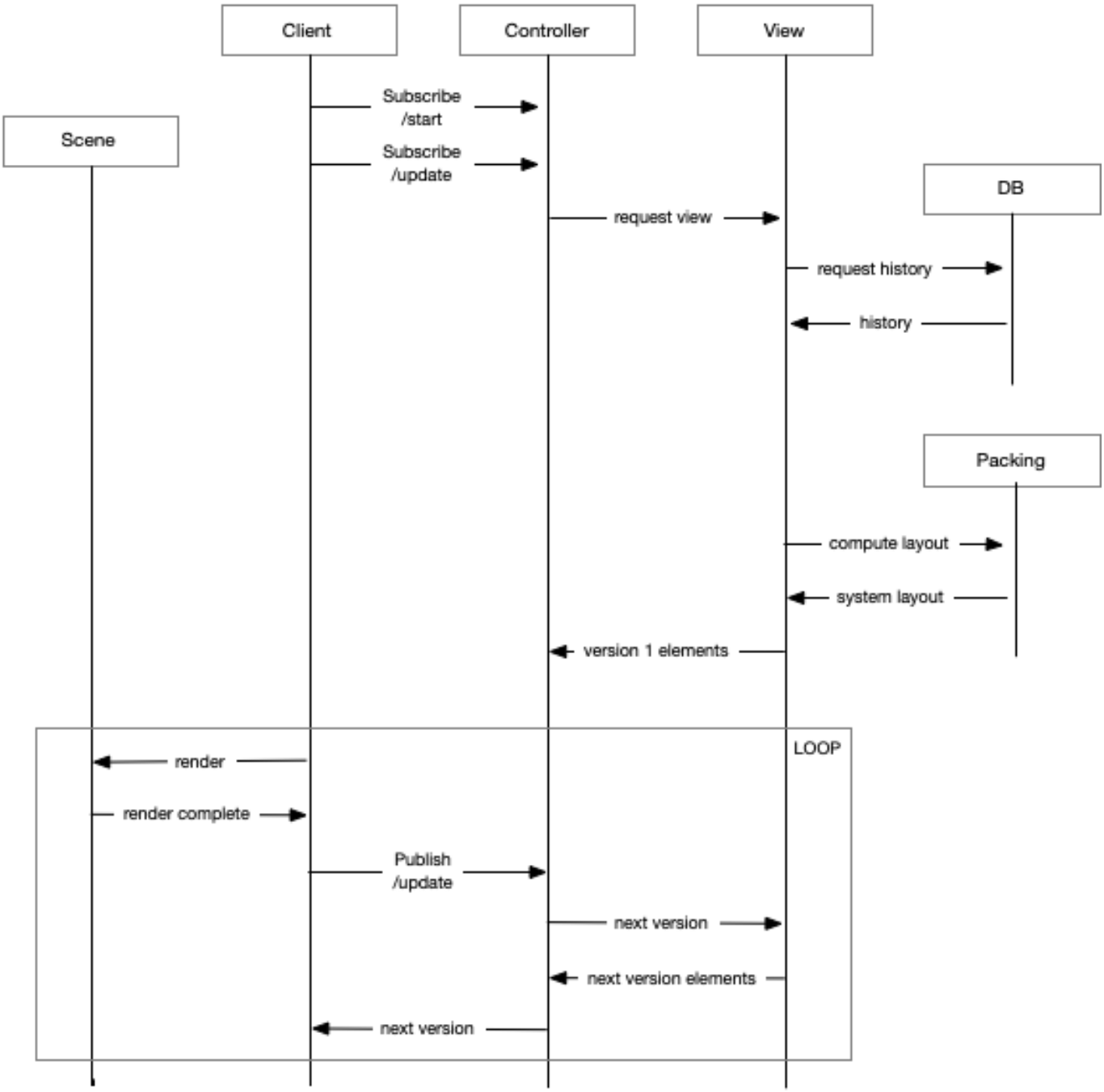


FIGURE 3.15: Sequence diagram for the visualization of software systems evolution

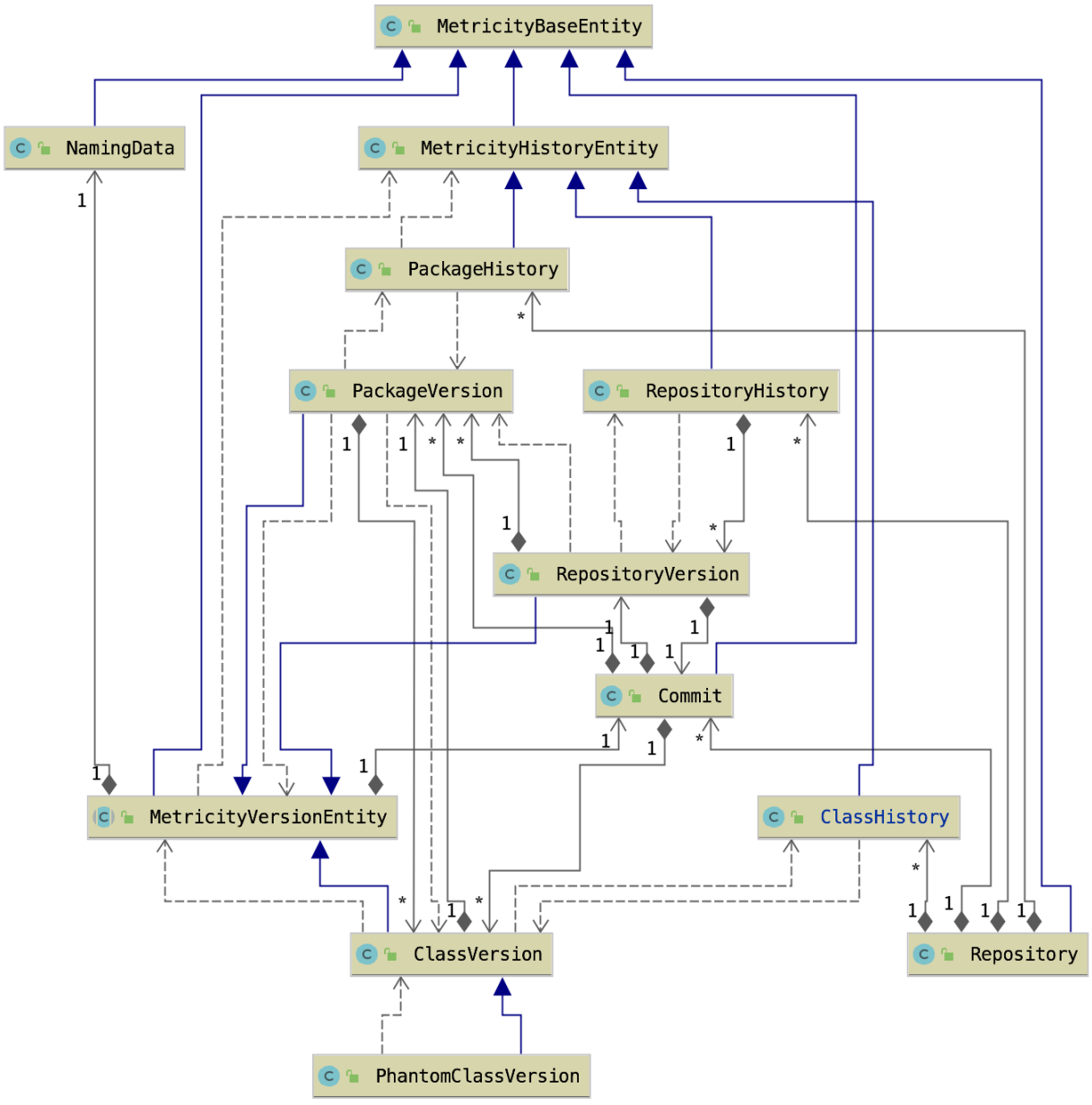


FIGURE 3.16: Evolution model entities modeled as objects

that represent the evolution of the selected history. Due to the evolution, packages and classes can be "merged". This information is extracted from the path of the file that has been changed from version to version. It is important to mention that Histories do not have a name, but either an ID that defines them as discussed in **??**. A 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 name into another one, therefore naming the history as the class would have been a wrong choice. As result, the linking of a history to a name is done only at version level. PackageHistories on the other hand also contains a list of sub-packages, that are the result of the hierarchy of MXNode. Figure 3.17 shows the resulting hierarchy

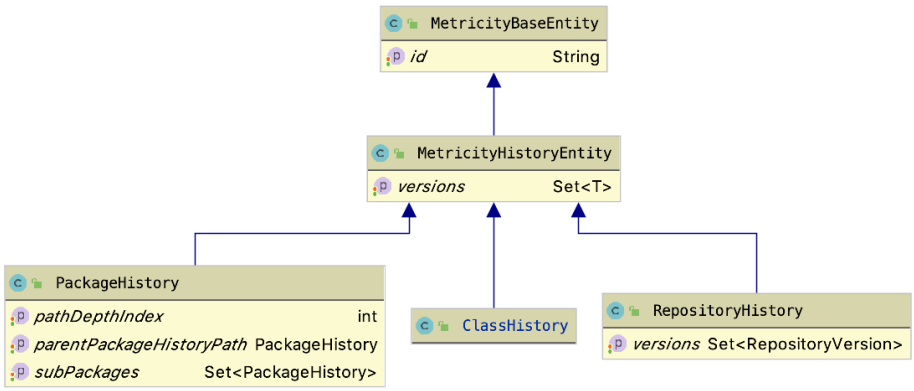


FIGURE 3.17: History entities hierarchy as objects

Version entities represent the leaves of the evolution. A version, as the name says, it is a single point in the history of the software 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 in that point in time. To understand to which history it belongs, we also have an inverse relationship with the HistoryEntity. A parent-child relation represent a simple pointer between two consecutive versions, while the set of MetricData contain 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 it difficult 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 3.18 shows the final structure of the version entities.

Frontend

The frontend is developed using Typescript[6], and uses Vue.js, an open-source model–view–viewmodel[7] JavaScript framework for building user interfaces on the web. Regarding the styling framework, we decided to use Boostrap[8], an open-source CSS framework directed at responsive, mobile-first front-end web development. Regarding the visualization of the software systems, we used Babylon.js[9], is a real-time 3D engine using a JavaScript library for displaying 3D graphics in a web browser.

In the previous sections we have seen the sequence diagrams describing the construction of a layout of the city in the backend. In Figure 3.19, instead, we see 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,

[6]See Typescript
[7]See model-view-viewmodel
[8]See Boostrap
[9]See Babylon.js

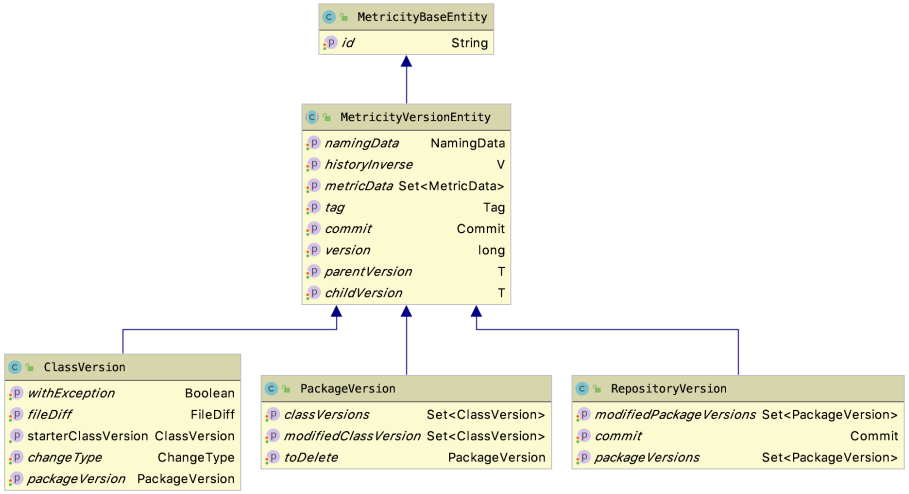


FIGURE 3.18: Version entities hierarchy as objects

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 Id representing the same ID that is computed 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, so as to ensure a correct view of the evolution.

3.2.2 Design considerations

In this Section we will the design requirements of M3TRICITY.

- *Easily extensible to new languages*. M3TRICITY has the limitation of being able to extract metrics only for Java projects. However, we designed it extensible to support additional languages by extending the MetricExtractor class developers can implement their logic to extract metrics from files, together with the possibility of adding new metrics.
- *Portable*. The portability of the system lies in the fact that it 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 guarantees a better representation of the evolution of the system.
- *Navigable*. The evolution of any software 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, you can jump in time but also go forward faster.

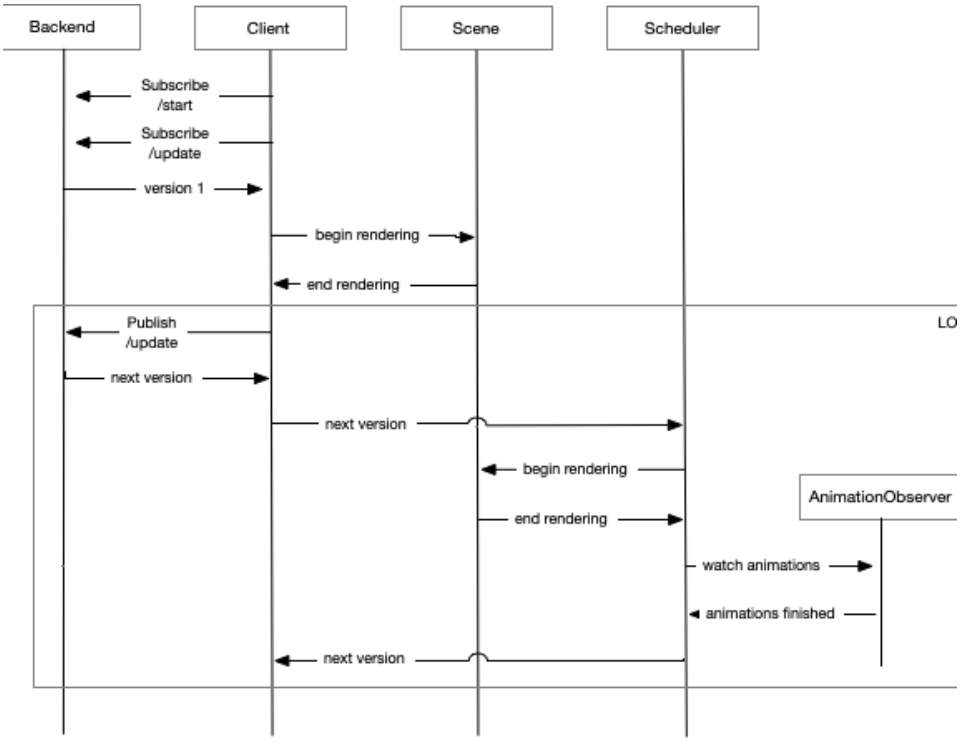


FIGURE 3.19: Sequence diagram for the visualization of software systems evolution in the frontend

- *Customizable*. The visualization is not static. Users can personalize the metric and, therefore, the scene as well. In addition, the user can use specific functions in order to highlight different areas of interest within the city.

- *Interactive*. As difference with previous implementations of this system, M3TRICITY was designed to be easy to interact with. It provides functionalities to access elements in focus just by hovering, together with several additional interactions when the scene is clicked (e.g. highlighting of the evolution-timeline and details about the selected class) .

3.2.3 UI

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.

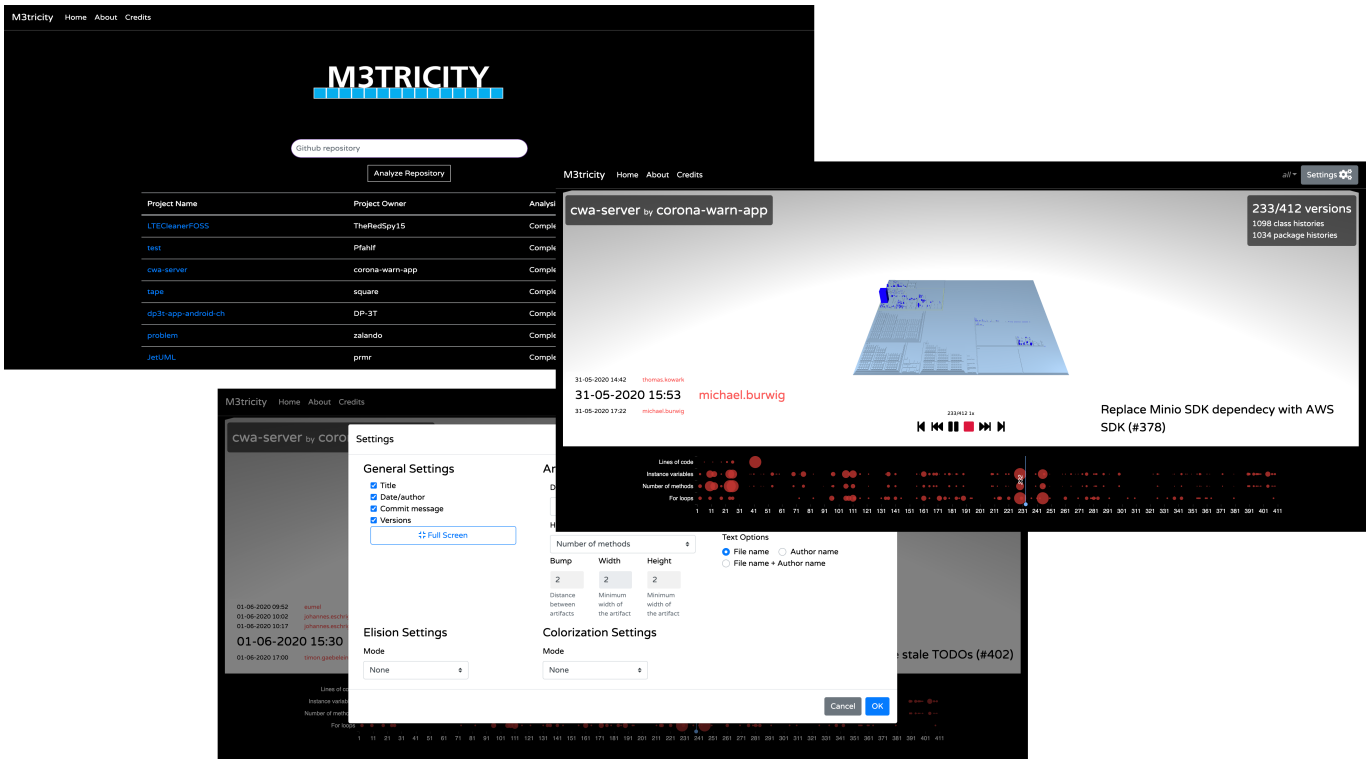


FIGURE 3.20: M3TRICITY in Snapshots

Web-pages

M3TRICITY has two main pages: one is the home page, and the other one is the visualization of a city. Both pages have a nav-bar for styling and navigation purposes. The nav-bar has also contain the settings for the visualization page, that we will discuss later.

Figure 3.20 shows some screenshots of the web application. In the top left corner, we see the home page, with an input field that can be used for analyzing new repositories by clicking on the 'Analyze Repository' button. There is also a list of pre-analyzed repositories that can be directly visualized. In the middle, we have the actual visualization of the software system, that we will discuss in the next section. In the bottom left corner, we have the settings that can be used to customize the scene.

City view

Figure 3.21 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 another repository, he needs to submit a new request through the appropriate field. From the top bar (Fig. 3.21-A), 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 (Fig. 3.21-B) and displays the timestamps and the authors of the five last commits (Fig. 3.21-C).

At the bottom, a timeline (Fig. 3.21-D) 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 cursor, highlighted in Fig. 3.21-E, keeps track of the snapshot being depicted in the canvas (Fig. 3.21-H).

On the bottom right, M3TRICITY displays the commit message of the current snapshot (Fig. 3.21-F). The control panel (Fig. 3.21-G) lets users move through time by playing, pausing, forwarding, or rewinding the visualization. The visualization canvas (Fig. 3.21-H) occupies the central part of the screen.

By hovering on elements in the visualization, M3TRICITY shows a tooltip with additional information (Fig. 3.21-I).

On the top right corner of the screen, M3TRICITY summarizes information about the history of the system at hand (Fig. 3.21-J), lets the user choose between different ways to traverse time (Fig. 3.21-L), and change the settings (Fig. 3.21-K).

In the snapshot depicted in Fig. 3.21-H, JETUML is undergoing a structural refactoring [19] 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.

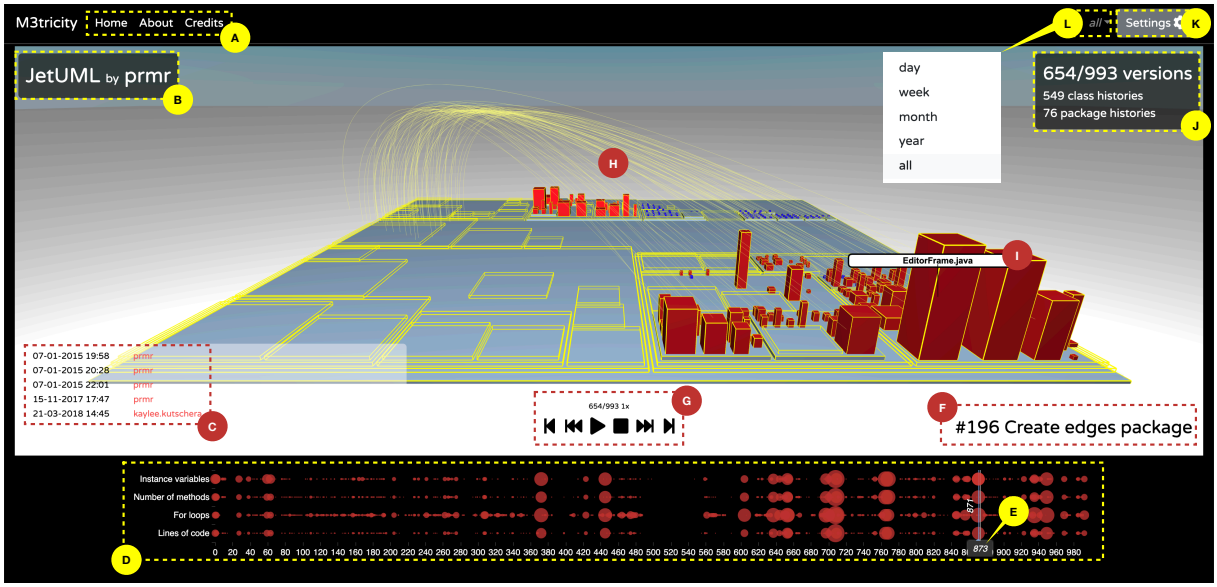


FIGURE 3.21: Visualizing a Snapshot of JETUML with M3TRICITY

Settings view

Figure 3.22 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 (Fig. 3.22-A) enable the user to fine-tune the visualization. It is possible here to hide some elements on the canvas or switch to full-screen mode. *Artifacts Settings* (Fig. 3.22-B) 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* (Fig. 3.22-C) let users toggle automatic tooltips to show the names of the elements (*i.e.*, classes and packages) that have been modified in the current snapshot. Fig. 3.22-D is used for hiding objects matching a regular expression. The last setting, *Colorization* (Fig. 3.22-E) let 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. We next discuss M3TRICITY's Advanced Evolution Model and its History-Resistant Layout.

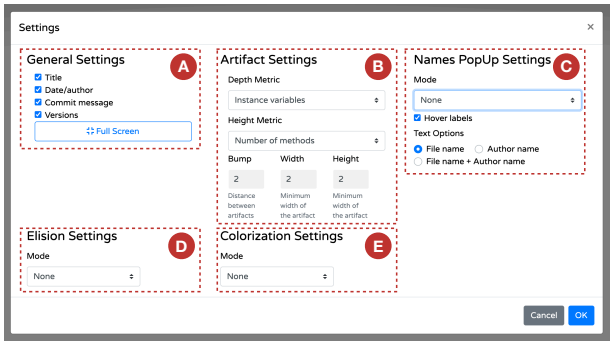


FIGURE 3.22: The Settings of M3TRICITY

3.3 Limitations

The purpose of this section is to discuss the limitations of M3TRICITY. By limitations we mean problems that we are aware of but that the current structure does not allow to solve quickly. When possible, we will also propose possible solutions to these problems.

Improving history-building Building the history 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 possibly a node. In order to increase the performance, one possible solution for the future is not to checkout 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 into keeping the current structure while trying to parallelize the extraction of the commits. This second solution requires to rethink how to connect histories due to the fact that results might not be in order anymore.

More resistant layout The creation of new histories for every package does not represent the evolution of a software system correctly. With a more resistant layout we mean that the final structure of the layout can be achieved by actually detecting package renames within the evolution of software system. The problem with the current implementation of M3TRICITY is that renames at packages level are not detected. The reason is that packageHistories can be merged to other other histories, can be split into multiple new histories

and they might even reappear in the future. During this thesis we decided to create a new PackageHistory every-time we were feeling to in order to simplify the visualization, since we weren't able to focus on all cases. As soon as we can detect renames and construct a hierarchy among histories

Space reuse In order to achieve a more compact layout, free space should be reused in the future by newly created objects, in order to avoid to have dead spots within the cities. To do so, we need to track the evolution when we compute the layout by reassigning space that is not used anymore to new objects when possible.

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 important cases. It uses math in order to understand if two files are the same or not. Still, this is far from optimal because some files might pass undetected. The library itself has parameters that can be changed in order to try to increase the match score. In the future, we should try to increase the detection of those cases, probably going also at method level in order to have a more in-depth knowledge of the changed that happened over the time.

Metric Extraction M3TRICITY approach for metric extraction is far from optimal. For each commit, it generates an XML for each source file. 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 to use external tools for this pass and should work directly within its object. By doing so, the performance should increase. For this, a parser for multiple languages should be integrated like Antlr, adding a layer of complexity.

Database integration Due to the internal complexity of the evolution model, one of the first simplification of our tool was to drop the remote database in favor of a in-memory one. The number of entries that each repository was requiring was overkilling not only when storing it, but especially when trying to retrieve it for the visualization. The model itself contains cycles between data, due to complexities and requirements of the application itself. Those cycles, given the structure of the model, should be avoided.

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 visualization part 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 tool needs to retrieve the whole history of the repository to compute the layout. By moving this operation in the construction phase, the amount of time spent in this operation would be decreased since no requests would be made to the database. In addition, the position of each class will be already defined, not requiring any computation.

Redefine PackageHistories PackageHistories have a big problem, as they represent is the history of a series of entities. When we think of a history of something, we imagine it 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 that every-time when this situation happens, a new history with no relation is created. 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 in order to preserve a simple and linear structure.

As for the future, we should try to model this entities with those possible cases in mind. In fact, PackageHistories might connect to other histories since what define a PackageHistory are the class contained in them. A better representation for this would be a graph, in which a history is not unique but can accept that at a certain point gets connected to other histories, creating a hierarchy of histories.

Chapter 4

Case of studies

4.1 JetUML

JETUML[1] is a desktop application for fast UML diagramming. The history of this system starts in early 2015.

Violet, Violetta, and JetUML. Figure 4.1 shows six key snapshots of JETUML visualized with M3TRICITY.

Fig. 4.1-A depicts the first available version [1] 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 whole source code of the VIOLET[2] project contained in a package called `com.horstmann.violet`.

Fig. 4.1-B shows the 28^{th} [2] 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 Fig. 4.1-C [3], 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 [4] few changes happen inside the main package.

Fig. 4.1-D 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 [5], depicted in Fig. 4.1-E: packages were renamed by removing the acronym "`stg`" from their names (*e*, `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 Fig. 4.1-F is at revision 1005 [6] when numerous classes are affected by a package renaming where the word *"graph"* was substituted by *"diagram."*

4.2 cwa-server

4.3 M3TRICITY

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

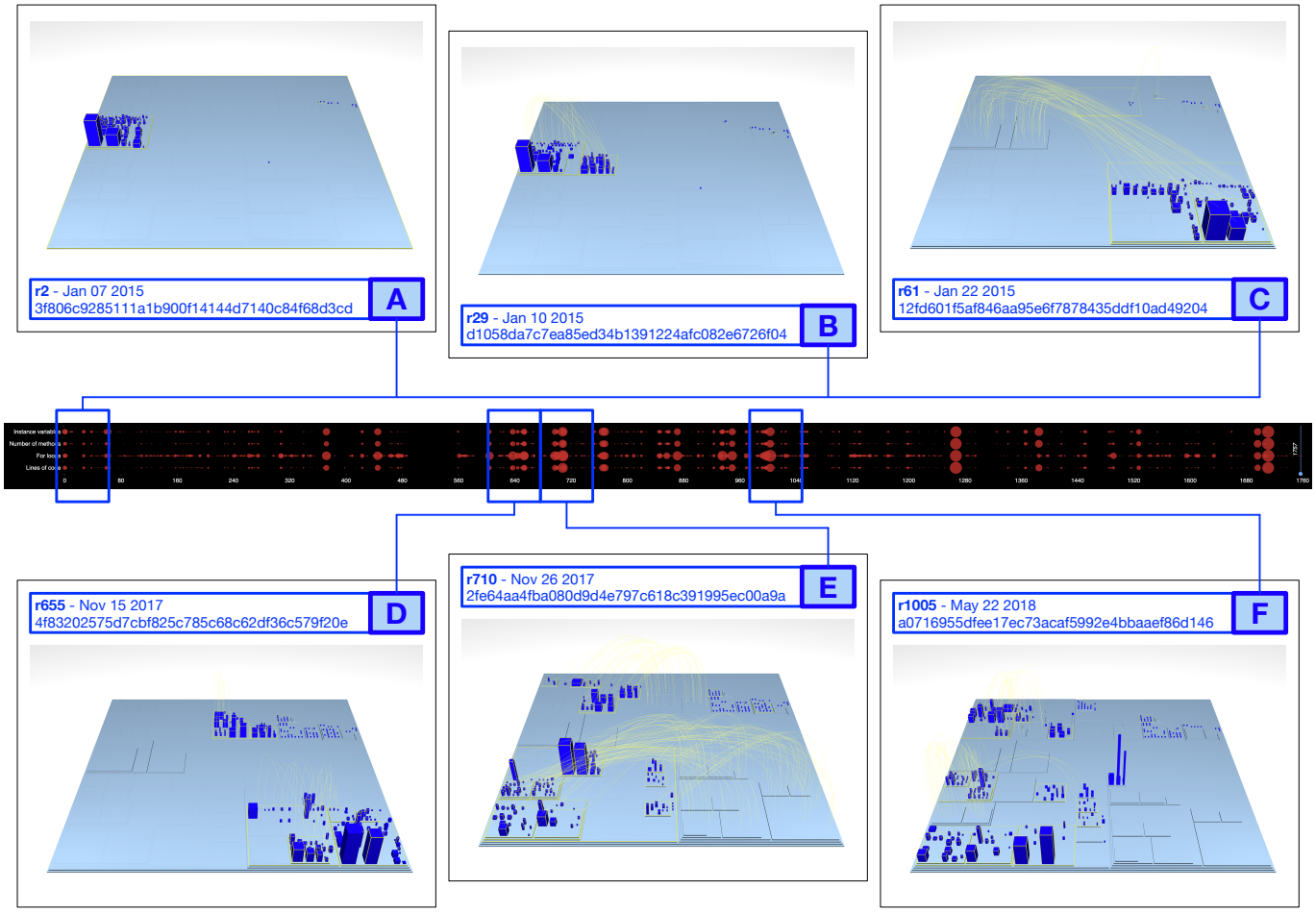


FIGURE 4.1: From Violet to Violetta to JetUML

Chapter 5

Conclusions and Future Work

I need to better sell my product csaba-f1

In this chapter we summarize our thesis, discuss key lessons learned throughout the development of M3TRICITY and present directions for future work. Section 5.1 makes a recap of what has been done concisely. Section 5.2 reflects on what have 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, based on what has been learned and developed up to now.

5.1 Recap

The main goal of this thesis was to visualize software systems using the city metaphor with respect to their evolution. We wanted to show that creating a more resistant layout would therefore make the visualization more understandable to the user who studies the evolution of a system as a developing city. Although we had no resources to conduct a thorough user study, but we gathered a more in-depth knowledge and understanding on how evolution must be modeled, together with a better awareness on how the city must be visualized given the current limitations imposed by web technologies.

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 the evolution model in order to make it more accessible and explorable. 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 that need to be care of .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 reliable development point, we decided it was time to add functionalities to the visualization. Part of these functionalities aimed at letting the user customize the type of visualization, such as the highlight of specific elements of the class but also customizing the actual metrics that were defining the city itself.

After having worked on M3TRICITY, we used it to demonstrate how software systems around the web have been developed over time. We discovered that none of the applications we decided to analyze were stale. All of them, developed uniquely, with some common visual patterns. For example, large refactorings happened but also many renames.

Finally, we discovered what we did right and what we should have done differently. We took paths that had already been taken in the past, and we tried to add new trails in order to make the whole experience a little better. Nothing is perfect, and some choices that are made in the past often influence those of the future, and this thesis is a proof of this.We have learned much and there is still much to do in the future, but in the end, it is the result that counts, and in our case it is the awareness that the evolution of software system is complex, but at the same time easy to handle

5.2 Reflections

The evolution of M3TRICITY reflects the difficulties that we had to face during each stage of development. We learned many things during this time, and we also had many problems. If the evolution model was already complicated, trying to visualize it was even more complex. 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 on iterating and constructing history, reflections about extracting metrics efficiently and reflections on web-based visualizations.

5.2.1 Reflections on the evolution model

We saw that evolution models have 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 Girba[22].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 model itself, making it less understandable.

The model proposed a concept of Snapshot in order 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 have 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. Another problem from the model was that it was directly linking a ClassVersion to a PackageVersion. With our model we tried to address the problem that ClassVersion entities might belong to multiple PackageVersion of different PackageHistories. In fact, renames might occur in which a file is moved from one package to another.

We do not believe that our model is perfect or that the problems we tried to address are solved correctly. A PackageHistory in which a package is split into multiple sub-packages can lead to severe problems with this model, since, conceptually, the new packages have their own history still being related to the old one. This problem needs to be investigated in the future while trying to address the fact that a history might not be linear over time for packages.

5.2.2 Reflections on exploring software systems histories

M3TRICITY is constructed on top of Java and used JGit to access the history of software systems. One major limitation of this solution is that the whole commit needed to be checked out in order to be able to compute the metrics. In addition to this, JGit does not support concurrency in those operations, leading to performance limitations. Another problem with this solution was that even if we tried to add concurrency in this operation, constructing the history could become even more complicated. In fact, the structure itself for storing and linking worked fine if the history was explored incrementally, but failed as soon as it was explored randomly as it would be happening in a threaded program.

We believe that increasing the construction performance is a crucial part in order to make M3TRICITY more useful in a real scenario. Particular attention must be given to the structures responsible for storing the information about filenames and paths in order to overcome the problem generated by the random exploration.

5.2.3 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 system. We tried several solutions, some less flexible than others. The first solution was to analyze Java projects using libraries for analyzing Java software systems metrics. In the second

attempt, we tried to understand how to generalize an interface that was capable of analyzing any type of language, regardless of whether it was Java. This led to a second problem, finding a Java library that would allow this. Contrary to what we expected, there were not many possibilities and the libraries were almost always language dependent, creating the need to develop specific classes 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 manually, but this came at the price of performance.

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, needs 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.

5.2.4 Reflections on web-based visualizations

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

In recent times WebGL, the locomotive behind each 3D library has made great strides, allowing for good development and especially good performance. We are not talking about developing complex games, but we can undoubtedly view software systems. After several tests, we understood that a limiting factor would have been the number of objects that can be displayed together in the same view. A large software system often consists of many classes, many packages and many refactoring. This can lead to scenes with thousands of objects. In addition, we wanted to make the scene as interactive as possible, with animations and information already available for each class. As a result, we had to create unique objects for each class, giving up intrinsic optimizations of the library we were using

Alternative solutions to visualize thousands of objects exist and we have tried them, but each of these solutions comes at the expense of something else, which can be usability or interactivity.

5.3 Future Work

Improve this section

In this section we will discuss some future possibilities for M3TRICITY. We will start by what should be probably redeveloped and what extensions can be added.

Rebuilding frontend The logic behind the frontend is quite complex and should be refactored in order to implement ad-hoc solutions in order to increase the amount of objects that can be rendered. Knowledge on the web-gl framework has increased over the time and current implementation do 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 favor 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 interesting. The same logic could apply during packages split.

Learning from history By analyzing multiple software systems, we could try to learn if common patterns exist within their evolution. This information could be derived by looking at PackageHistories contents but also at ClassHistory level. In fact, the evolution mode provide many information that we are not using and that can be used to, for example, determine how classes that have similar name develop over time in different systems, or if given a package name multiple systems have specific class in it that can ease the future development of the system.

Exporting RepositoryHistory One important requirement of M3TRICITY was being able to export the evolution history of software system under CSV format. By doing so, we were able to export the information provided by the versioning system in order to be used by different applications or visualization that are not based on the city metaphor.

Voronoi based visualization The layout of the city could be determined not only by bin-packing algorithms but also from Voronoi diagrams. The new visualization will consist into a more dynamic city in which districts sizes are defined and balanced among each others, all this while reducing the space.

5.4 Final Words

During the past four months we developed M3TRICITY, an application that leverages the 3D city metaphor with a resistant layout. We implemented it as a freely accessible web application.

The tool itself is the result of the implementation of multiple ideas and concepts which have been designed, developed and modified in the past few months. We believe that what we learned in the past should be used in order to create a better evolution model together with a more resistant layout.

We hope that the ideas and information provided within this thesis about modeling evolution and visualizing it in a compact layout can be used in the future in order to develop a better knowledge about software systems.

Bibliography

[1]

[2]

[3]

[4]

[5]

[6]

[7] 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)*, pages 12–22, 2019.

[8] 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.*, pages 31–40. IEEE, 2004.

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

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

[11] N. Chen. Convention over configuration. *h ttp://softwareengineering. vazexqi. com/files/pattern. htm l*, 2006.

[12] M. D'Ambros and M. Lanza. Reverse engineering with logical coupling. In *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE '06, pages 189–198, Washington, DC, USA, 2006. IEEE Computer Society.

[13] M. D'Ambros, M. Lanza, and M. Lungu. Visualizing co-change information with the evolution radar. *IEEE Transactions on Software Engineering*, 35(5):720–735, Sep. 2009.

[14] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. *ACM SIGPLAN Notices*, 35(10):166–177, 2000.

[15] S. C. Eick, J. L. Steffen, and E. E. Sumner. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, 1992.

[16] 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*, SAC '12, pages 981–988. ACM, 2012.

[17] M. Fischer and H. Gall. Visualizing feature evolution of large-scale software based on problem and modification report data. *Journal of Software Maintenance*, 16:385–403, 11 2004.

[18] 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)*, pages 1–4, Sep. 2013.

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

[20] 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*, pages 160–166. IEEE, 1997.

[21] 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'*, pages 99–108, Aug 1999.

[22] T. A. Girba. *Modeling history to understand software evolution*. PhD thesis, University of Bern, 2005.

[23] 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*, IWPSE-Evol '09, pages 25–30, New York, NY, USA, 2009. ACM.

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

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

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

[27] 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*, pages 47–54, 2007.

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

[29] D. E. Knuth. Computer-drawn flowcharts. *Commun. ACM*, 6(9):555–563, Sept. 1963.

[30] D. E. Knuth. Literate programming. *Comput. J.*, 27(2):97–111, May 1984.

[31] 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*, ASE '05, pages 214–223. ACM, 2005.

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

[33] 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.

[34] M. Lanza, S. Ducasse, and S. Demeyer. *Combining Metrics and Graphs for Object Oriented Reverse Engineering*. PhD thesis, Universität Bern, 12 1999.

[35] M. Lungu, M. Lanza, T. Gîrba, and R. Robbes. The small project observatory: Visualizing software ecosystems. *Science of Computer Programming*, 75(4):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] 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*, ICPC 2014, pages 207–211. ACM, 2014.

[37] 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*, pages 1–6, Sep. 2005.

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

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

[40] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *SIGPLAN Not.*, 8(8):12–26, Aug. 1973.

[41] 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*, ICSE '12, pages 1044–1053, Piscataway, NJ, USA, 2012. IEEE Press.

[42] 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.*, pages 314–319, July 2003.

[43] 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)*, pages 217–228, July 2007.

[44] T. Panas, R. Lincke, and W. Löwe. Online-configuration of software visualizations with vizz3d. In *Proceedings of the 2005 ACM Symposium on Software Visualization*, SoftVis '05, pages 173–182. ACM, 2005.

[45] S. P. Reiss. An engine for the 3D visualization of program information. *J. Vis. Lang. Comput.*, 6(3):299–323, Sept. 1995.

[46] M. J. Rochkind. The source code control system. *IEEE transactions on Software Engineering*, (4):364–370, 1975.

[47] 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)*, pages 1–11, Sep. 2019.

[48] 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,*, pages 273–280. INSTICC, SciTePress, 2018.

[49] M. Sondag, B. Speckmann, and K. Verbeek. Stable treemaps via local moves. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):729–738, 2017.

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

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

[52] Y. Tymchuk, A. Mocci, and M. Lanza. Vidi: The visual design inspector. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 653–656, Piscataway, NJ, USA, 2015. IEEE Press.

[53] 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)*, pages 509–516, July 2017.

[54] R. Wettel. Visual exploration of large-scale evolving software. In *2009 31st International Conference on Software Engineering-Companion Volume*, pages 391–394. IEEE, 2009.

[55] R. Wettel. *Software systems as cities*. PhD thesis, Università della Svizzera italiana, 2010.

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

[57] Z. Xing and E. Stroulia. Understanding phases and styles of object-oriented systems' evolution. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 242–251. IEEE, 2004.

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