Visualizing Evolving Software Cities

Federico Pfahler, Roberto Minelli, Csaba Nagy, Michele Lanza
REVEAL @ Software Institute – USI Università della Svizzera italiana, Lugano, Switzerland

***Abstract*—Visualization approaches that leverage a 3D city metaphor have become popular. Numerous variations, including virtual and augmented reality have emerged. Despite its popularity, the city metaphor falls short when depicting the evolution of a system, which results in buildings and districts moving around in unpredictable ways.**

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.

***Index Terms*—Software visualization, Software maintenance**

I. INTRODUCTION

Researchers have explored many approaches to visualize the evolution of software systems. In 2007, Wettel *et al.* presented CODECITY, an approach to visualize software systems as interactive cities [1], which inspired many other approaches. Classes are represented as buildings, while packages are depicted as the districts in which buildings reside. The dimensions of visual elements (*i.e.,* width, height) are proportional to the values of software metrics, following the idea of polymetric views [2]. By leveraging the city metaphor, CODECITY aims at displaying software metrics in a meaningful way, while giving viewers a sense of locality in the code city. The popularity of the city metaphor has grown, going as far as utilizing virtual and augmented reality, for example with EVO-STREETS [3], CodeMetropolis [4], and VR City [5].

Despite its popularity, the city metaphor falls short when it comes to visualizing the evolution of a system. In the past researchers have either favored a small multiples approach (*i.e.,* depicting several city versions side by side) or created movies from the various snapshots. In both cases following the evolution of the system is rather hard: Using small multiples we do not see the data in the same detail as with larger visualizations while with the latter the resulting movies are not smooth since changes are depicted as "jumps" (classes and packages move in unpredictable ways from one version to the other) making it difficult to follow the evolutionary process.

We present a novel approach to visualize software systems as evolving cities that treats evolution as a first-class concept. To do so, we take inspiration from Girba's software evolution model proposed in HISMO [6] and adapt it to the Git workflow. Essentially, the model is composed of *versions* and *histories*: A version represents the set of changes that happened in the system while a history is a collection of versions. Our approach models versions and histories at class, package, and repository granularity. Once populated, the evolution model of a software system permits to construct and traverse its history.

To demonstrate the usefulness of our evolution model, we use it to depict software systems as evolving cities. To do so, we devised a *History-Resistant Layout* that leverages our evolution model. Every element in the visualization assumes a fixed position in the 3D space and remains stationary over the whole evolution, making up for the unpredictable "layout jumps" of other approaches. In CODECITY, for example, at every revision elements were free to move in the whole visualization canvas according to the heuristics employed by the underlying bin packing algorithm. As a result, our layout renders with fidelity not only artifact-specific changes (adding/removing code) but also refactoring operations (*e.g.,* moving a class to another package or renaming code artifacts).

We implemented our approach in M3TRICITY, a web application available at `https://metricity.si.usi.ch`.

II. RELATED WORK

The first 3D visualization approaches appeared in the 1990s, spreading ever faster when consumer hardware started to become both affordable and performant enough. From seminal works like PLUM by Reiss [7] and works by Young & Munro who explored virtual reality as a means to visualize software [8], the metaphor of visualizing software as a city took hold.

Knight *et al.* created Software World, where software systems are displayed as cities, with buildings, trees and streets [9]. In 2003, Panas *et al.* depicted a software system as a city with real information about static and dynamic data [10]–[12]. Langelier *et al.* presented Verso, a tool based on a landscape metaphor with cities in mind [13]. In 2007, Wettel *et al.* presented CODECITY, to visualize software systems as interactive cities, aiming to display software metrics in a meaningful way, while keeping the layout of the city consistent with the information and giving viewers a sense of locality in the city [1]. CODECITY was extended to visualize the evolution of the systems [14]. The 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. Based on the city metaphor, Steinbrückner and Lewerentz presented EVO-STREETS that takes into account also the evolution of software systems. They mapped time to the height of the hills on which classes were placed [3]. By making use of 3D blocks, Fittkau *et al.* presented ExploraViz, a tool for visualizing traces using both 3D and 2D visualizations [15]. In 2012, Erra and Scaniello proposed CodeTrees, a visualization of the software system under the form of trees [16], a concept extended by Maruyama *et al.* [17]. More recently, Vincur *et al.* presented VR City, a tool that represented a software system in a virtual reality environment [5].

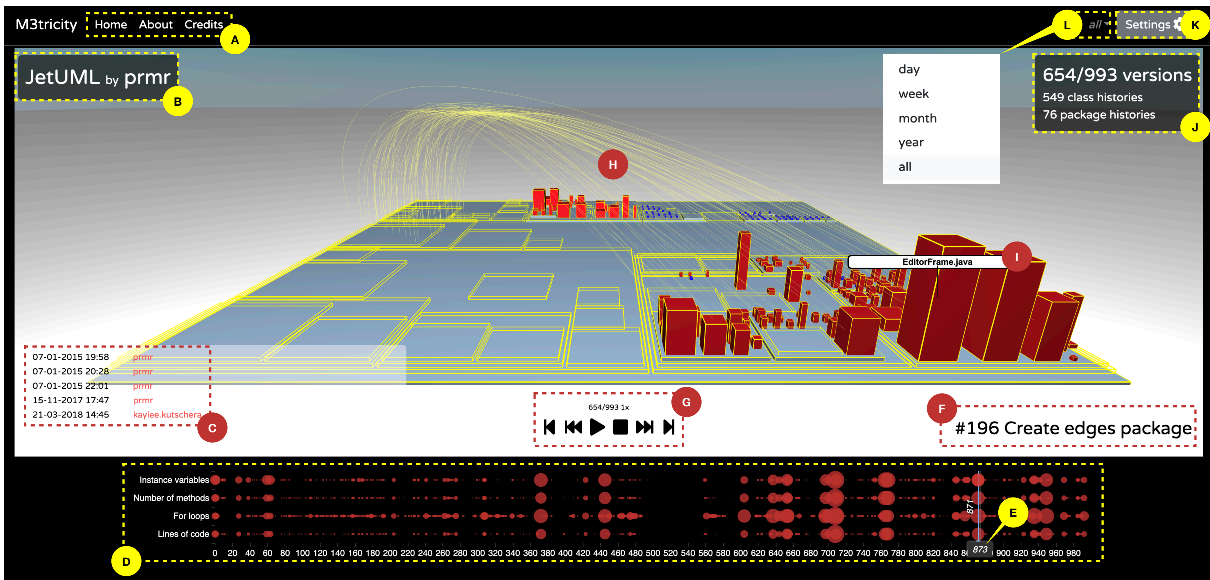

Fig. 1: Visualizing a Snapshot of JETUML with M3TRICITY

III. M3TRICITY

We present an approach that considers evolution as a first-class concept to visualize software systems as evolving cities. The approach is implemented in M3TRICITY, publicly available at `https://metricity.si.usi.ch`.

Figure 1 depicts the main user interface (*i.e.,* UI) of M3TRICITY visualizing a snapshot of JETUML.[1] From the top bar (Fig. 1-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. 1-B) and displays the timestamps and the authors of the five last commits (Fig. 1-C).

At the bottom, a timeline (Fig. 1-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. 1-E, keeps track of the snapshot being depicted in the canvas (Fig. 1-H).

On the bottom right, M3TRICITY displays the commit message of the current snapshot (Fig. 1-F). The control panel (Fig. 1-G) lets users move through time by playing, pausing, forwarding, or rewinding the visualization. The visualization canvas (Fig. 1-H) occupies the central part of the screen.

By hovering on elements in the visualization, M3TRICITY shows a tooltip with additional information (Fig. 1-I).

[1] See **`https://github.com/prmr/JetUML`**

On the top right corner of the screen, M3TRICITY summarizes information about the history of the system at hand (Fig. 1-J), lets the user choose between different ways to traverse time (Fig. 1-L), and change the settings (Fig. 1-K).

In the snapshot depicted in Fig. 1-H, JETUML is undergoing a structural refactoring [18] 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 2 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).

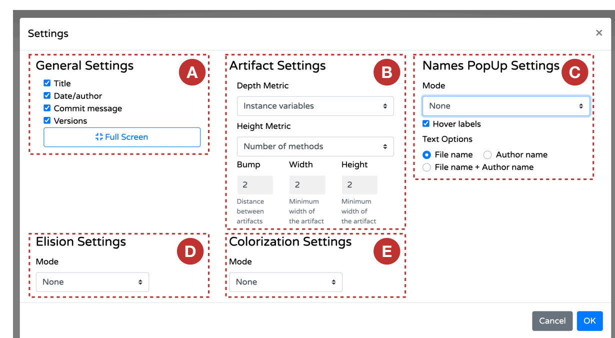

Fig. 2: The Settings of M3TRICITY

General Settings (Fig. 2-A) give users the possibility 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. 2-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. 2-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. 2-D is used for hiding objects matching a regular expression. The last setting, *Colorization* (Fig. 2-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.

Advanced Evolution Model. The goal of M3TRICITY is to model evolution as a first-class concept and leverage it for the creation of the 3D visualization. Our approach is inspired by Girba's software evolution model named HISMO [6].

In HISMO history is modeled in a matrix. The matrix represents a package history, while rows represent class histories, and columns represent different versions of the package. Being based on SVN, there was no need to have the concept of "repository history" since the system itself assigned a unique version to all files at every commit. Git stores only information about modified files, hence we had to extend HISMO to also keep track of the files at every single revision/version. Figure 3 shows the Advanced Evolution Model of M3TRICITY.

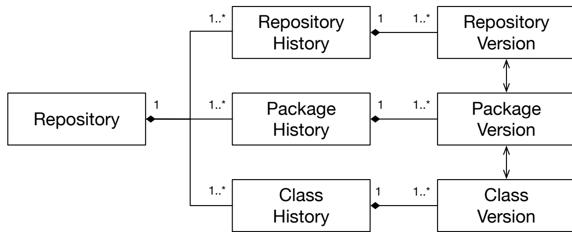

Fig. 3: The Evolution Model of M3TRICITY

Inside the model, we have the concept of *history* and *version.* History consists of a series of versions, where a version represents changes made in one or more objects within the system. The RepositoryHistory is a linear succession of RepositorVersions. Similarly, ClassHistory and PackageHistory also have a linear succession of versions, each one representing their changes over time. To understand the linking between RepositoryVersion, PackageVersion, and ClassVersion, each version entity points to one or more associated entities. RepositoryVersion is the root element of this structure and ClassVersion is the last one in the hierarchy. By doing so, we can construct all histories of all components or the whole repository history. The linking permits to understand and pair a PackageVersion with a specific ClassVersion among the ones contained in a ClassHistory. The model also makes it possible to merge rows (= histories) and columns (= versions) when renaming or copying happens. Figure 3 shows how data can be retrieved: Starting from the Repository, we can access its history but also its package and class histories. RepositoryHistory links to one or more RepositoryVersions. This last one contains the PackageVersions within the Repository at that specific time. Similarly, PackageVersion references a list of unique ClassVersions. By traversing the RepositoryVersions, we can extract all the changes that happened over time.

History-Resistant Layout. The evolution model permits to construct, traverse, and ultimately visualize the history of the software system. In M3TRICITY, we implemented a layout that leverages the evolution model to create a history-resistant layout. The positioning of every element within the visualization is computed based on the information provided by the model. By using this technique, every element is assigned a position. The evolution model also includes information about metric values at every revision. Therefore, we can infer the maximum real estate needed for each class. In this way, we are confident that the class "building" will not run into overlapping problems with other classes. Recursively, the same applies to packages. To place the individual elements within the visualization, we use a recursive bin-packing algorithm similar to other city visualizations. Figure 4 shows the history-resistant layout (left side) as opposed to a standard bin-packing layout (right side), like the one used by CODECITY, where the goal was to optimize space.

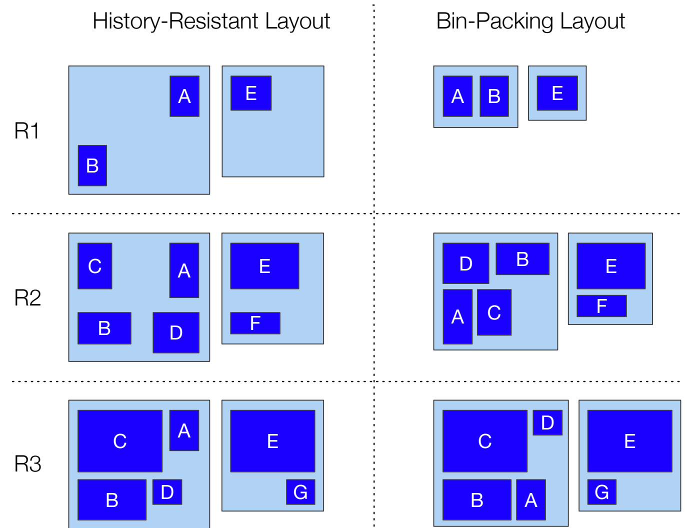

Fig. 4: History-Resistant Layout vs. Bin-Packing

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

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

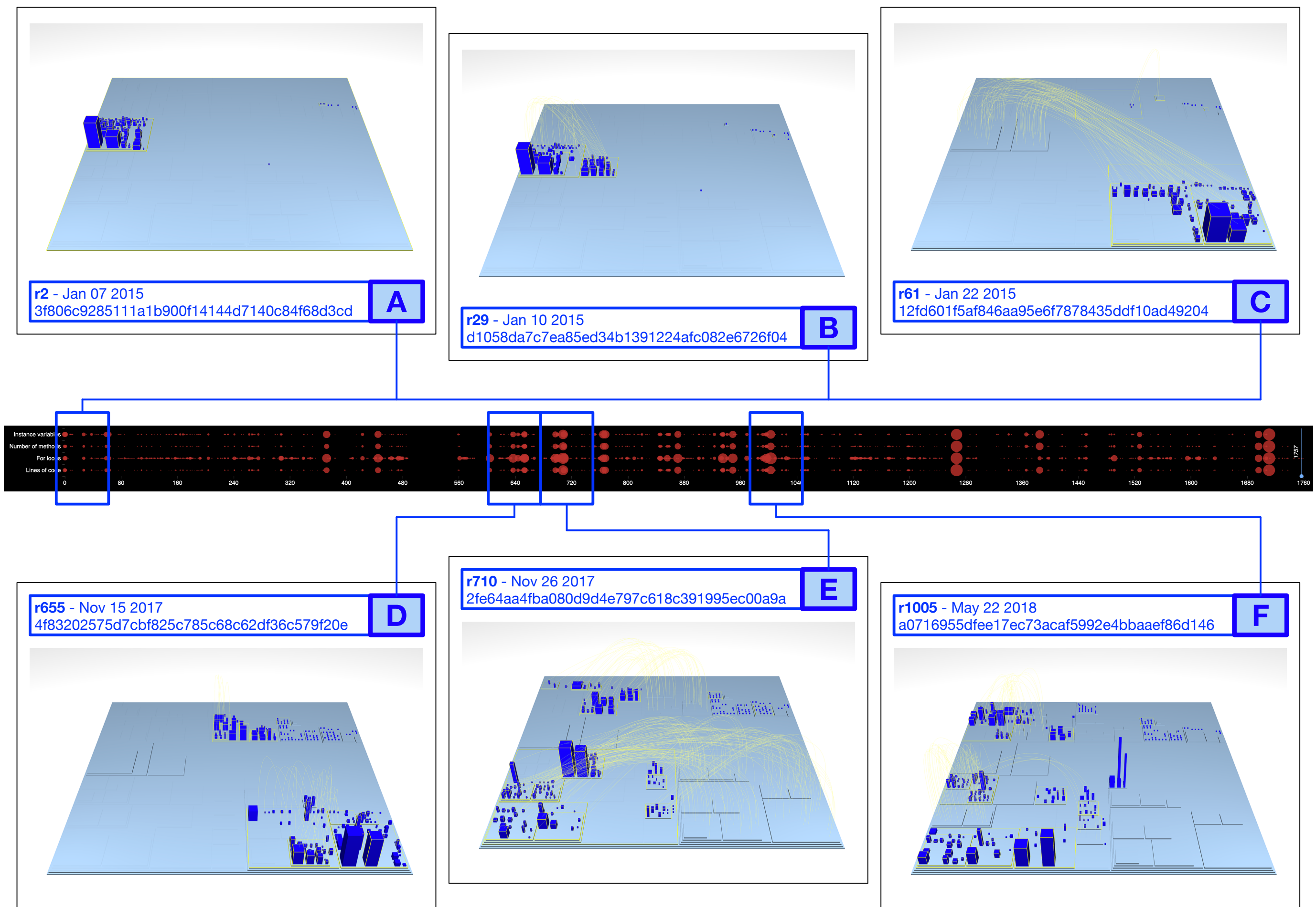

Fig. 5: From Violet to Violetta. From Violetta to JetUML

IV. M3TRICITY IN PRACTICE

In this section, we use M3TRICITY to explore the 5 years of evolution of an open-source project: JETUML.

A. *The Case of* JETUML

JETUML[2] is a desktop application for fast UML diagramming. The history of this system starts in early 2015.

Violet, Violetta, and JetUML. Figure 5 shows six key snapshots of JETUML visualized with M3TRICITY.

Fig. 5-A depicts the first available version [19] 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[3] project contained in a package called `com.horstmann.violet`.

Fig. 5-B shows the 28th revision [20] 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. 5-C [21], 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 [22] few changes happen inside the main package.

Fig. 5-D is the result of a move refactoring where all the classes concerning nodes and edges are moved into dedicated packages. A major change happens at version 710 [23], depicted in Fig. 5-E: Packages were renamed by removing the acronym "`stg`" from their names (*e.g.,* `ca.mcgill.cs.stg.jetuml` became `ca.mcgill.cs.jetuml`), leading to a new placement of the classes. While this looks like a simple renaming, which would not be displayed as a relocation, in fact it is a restructuring and M3TRICITY displays it as an explicit movement of classes. The last structural change we depict in Fig. 5-F is at revision 1,005 [24] when numerous classes are affected by a package renaming where the word *"graph"* was substituted by *"diagram."*

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

B. Reality Check

To verify whether M3TRICITY succeeds at explaining evolutionary processes, we contacted Prof. Martin P. Robillard, the owner of the JETUML project, and pointed him to the M3TRICITY website. He tried out M3TRICITY (*"nice tool, with great usability"*) providing the following quote:

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

C. (More than) One More Thing

M3TRICITY also includes several additional features:

Time Bucketing. As visualizing each commit may be too fine-grained, users can bucket time by day, week, month, and year. All commits in a bucket are then displayed at once.

Evolution Timeline. An interactive user interface component below the main visualization (see Fig. 1-D). Users can jump to a specific moment in development by clicking on the timeline. When users select an object in the visualization, the timeline highlights all commits involving the entity.

Interactivity. Users can hover and click on each object to obtain additional information, such as metrics, the version of the object, or the type of change.

Changes Highlighting. To provide information about differences between versions, changed objects are highlighted in yellow, and their name is displayed. Moreover, when software artifacts are moved, for example from one package to another, the movement is displayed as an arc (see Fig. 1-H), and the object is moved following the path of the displayed arc.

Elision. Sometimes users are not interested in seeing specific objects, for example, when a big class obstructs the main view. With the elision feature, users can select particular objects and remove them from the view.

V. CONCLUSIONS & FUTURE WORK

We presented an approach to layout software cities that make the cities more coherent when one visualizes their evolution. By treating evolution as a first-class entity, we can augment our layout algorithm about changes that happen during the history of a system. Moreover, our tool M3TRICITY supports software evolution understanding by depicting restructurings within the system as explicit move animations.

There are still short-comings that need to be addressed: When a complete part of a system is deleted from its history it will keep occupying the real estate assigned to it while it existed. Moreover, the packages themselves and the classes within the packages are still placed according to space saving heuristics stemming from the bin-packing algorithm, which is not necessarily optimal. As part of our future work, we will investigate more compact layouts (*e.g.,* [25]).

Acknowledgements. We are grateful for the support by the Swiss National Science foundation (SNF) and the Fund for Scientific Research (FNRS) (Project "INSTINCT"); and SNF and JSPS (Project "SENSOR"). We thank Prof. Robillard for trying out M3TRICITY and providing us with his feedback.

REFERENCES

[1] R. Wettel and M. Lanza, "Visualizing software systems as cities," in *Proceedings of the International Workshop on Visualizing Software for Understanding and Analysis*, 2007, pp. 92–99.
[2] M. Lanza, "Codecrawler-lessons learned in building a software visualization tool," in *Proceedings of the European Conference onSoftware Maintenance and Reengineering*, 2003, pp. 409–418.
[3] F. Steinbrückner and C. Lewerentz, "Representing development history in software cities," in *Proceedings of the 5th International Symposium on Software Visualization*, ser. SOFTVIS '10. ACM, 2010, pp. 193–202.
[4] G. Balogh and A. Beszedes, "Codemetropolis - code visualisation in minecraft," in *Proceedings of SCAM 2013*. IEEE, 2013, pp. 136–141.
[5] J. Vincur, P. Navrat, and I. Polasek, "VR City: Software analysis in virtual reality environment," in *Proceedings of the International Conference on Software Quality, Reliability and Security*, July 2017, pp. 509–516.
[6] T. A. Girba, "Modeling history to understand software evolution," Ph.D. dissertation, University of Bern, 2005.
[7] S. P. Reiss, "An engine for the 3D visualization of program information," *Journal of Visual Languages & Computing*, vol. 6, no. 3, pp. 299–323, 1995.
[8] P. Young and M. Munro, "Visualising software in virtual reality," *Proceedings of the International Workshop on Program Comprehension*, pp. 19–26, 1998.
[9] C. Knight and M. Munro, "Virtual but visible software," in *Proceedings of the International Conference on Information Visualization*. IEEE, 2000, pp. 198–205.
[10] T. Panas, R. Berrigan, and J. Grundy, "A 3d metaphor for software production visualization," in *Proceedings of the International Conference on Information Visualizationı*, 2003, pp. 314–319.
[11] T. Panas, R. Lincke, and W. Löwe, "Online-configuration of software visualizations with vizz3d," in *Proceedings of the ACM Symposium on Software Visualization*. ACM, 2005, pp. 173–182.
[12] T. Panas, T. Epperly, D. Quinlan, A. Saebjornsen, and R. Vuduc, "Communicating software architecture using a unified single-view visualization," in *Proceedings of the International Conference on Engineering Complex Computer Systems*, 2007, pp. 217–228.
[13] G. Langelier, H. Sahraoui, and P. Poulin, "Visualization-based analysis of quality for large-scale software systems," in *Proceedings of the International Conference on Automated Software Engineering*. ACM, 2005, pp. 214–223.
[14] R. Wettel, "Visual exploration of large-scale evolving software," in *Proceedings of the International Conference on Software Engineering*. IEEE, 2009, pp. 391–394.
[15] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring, "Live trace visualization for comprehending large software landscapes: The explorviz approach," in *Proceedings of the Working Conference on Software Visualization*, Sep. 2013, pp. 1–4.
[16] U. Erra and G. Scanniello, "Towards the visualization of software systems as 3d forests: The codetrees environment," in *Proceedings of the Annual Symposium on Applied Computing*. ACM, 2012, pp. 981–988.
[17] K. Maruyama, T. Omori, and S. Hayashi, "A visualization tool recording historical data of program comprehension tasks," in *Proceedings of the International Conference on Program Comprehension*. ACM, 2014, pp. 207–211.
[18] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.
[19] JetUML [`3f806c9`]. https://github.com/prmr/JetUML/commit/3f806c9.
[20] JetUML [`d1058da`]. https://github.com/prmr/JetUML/commit/d1058da.
[21] JetUML [`12fd601`]. https://github.com/prmr/JetUML/commit/12fd601.
[22] JetUML [`4f83202`]. https://github.com/prmr/JetUML/commit/4f83202.
[23] JetUML [`2fe64aa`]. https://github.com/prmr/JetUML/commit/2fe64aa.
[24] JetUML [`a071695`]. https://github.com/prmr/JetUML/commit/a071695.
[25] M. Sondag, B. Speckmann, and K. Verbeek, "Stable treemaps via local moves," *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, no. 1, pp. 729–738, 2018.