The *Code Time Machine*

Emad Aghajani, Andrea Mocci, Gabriele Bavota, Michele Lanza
REVEAL @ Faculty of Informatics - Università della Svizzera italiana (USI), Switzerland

***Abstract*—Exploring and analyzing the history of changes is an intrinsic part of software evolution comprehension. Existing tools that exploit the data residing in version control repositories provide only limited support for the intuitive navigation of code changes from a historical perspective.**

We present the *Code Time Machine*, a lightweight IDE plugin which uses visualization techniques to depict the history of any chosen file augmented with information mined from the underlying versioning system. Inspired by Apple's Time Machine, our tool allows both developers and the system itself to seamlessly move through time.
A video of the *Code Time Machine* can be found at: https://youtu.be/meblwFO95oA

I. Introduction

Software evolution comprehension is an integral part of the software development process. According to Gîrba and Ducasse [1], a class of techniques to support software evolution analysis is *version-centered*. Such approaches target answering questions of when something happened in the history of a software project, and involve activities such as comparing two different versions, in terms of source code and/or runtime behavior. To perform such analyses, developers must be able to easily revert the system back to any given revision, perform static code inspections (*i.e.,* visit the source code and its corresponding metadata), compile and run the project, and possibly inspect the behavior at runtime.

In practice, the common facilities exposed by version control systems are rather limited. In fact, carrying out version-centered analyses is cumbersome, since it must be done through repetitive procedures using default user interfaces, such as the command line or applications with rather simple GUIs like *GitHub Desktop*[1].

Besides source code history, additional information such as code metrics are the most important resources to understand a system's evolution. As opposed to reverse engineering where one needs to understand the current version of a system, understanding a system's evolution copes with such data multiplied by the number of its revisions. Although code metrics and the versioning system's data are accessible separately, *i.e.,* through different tools, the user himself has to manually collect and correlate them to perform a given evolutionary analysis.

According to Gonzalez-Torres *et al.* [2], understanding the evolution of a software project can be effectively performed with the support of a visual representation. Several approaches have been proposed to understand a system evolution with the help of software visualization, for example by combining it with software metrics.

[1]See https://desktop.github.com/

Consequently, different tools have been developed, primarily focusing on visualizing history to support the analysis of system evolution. *Gource* [3], *CodeSwarm* [4], *SVN time-lapse View*[2] (and its Git[3] version), CVSScan [5], and *GitX*[4] are some examples of them. Such tools provide a common core of features, like a slide-bar for viewing different revisions of a file and a diff-view of changes. Although these tools could potentially be used to analyze a system's evolution, they do not uniformly integrate complementary information like code metrics into their visualizations.

More importantly, these tools do not provide any intuitive mechanism for previewing different versions of a system and navigate through them. The current practice is to perform the *checkout* operation for reverting to a specific version, which in the case of uncommitted changes would be problematic. In addition, in some scenarios, the developer analyzing a project's evolution needs to focus on a certain file in the course of time, but this is not possible because some of the tools do not provide a file-centric history view. Thus, to follow the evolution of a particular file, developers must manually find the commits which include the file.

To overcome the aforementioned problems, inspired by Apple's Time Machine, we came up with the idea of supporting the analysis of a system's evolution with a visualization that leverages the screen depth as the time axis. This concept brings a uniform, version-centered, and seamless history exploration experience to developers. Our visualization is file-centered, and it augments the familiar code editor of the IDE by enabling developers to navigate a file's history along a depth-based, perspective timeline, without losing the current context. In addition, the visualization integrates the following features:

- a *code metrics view* to illustrate the evolution of metrics like Lines of Code (LOC), number of methods, and cyclomatic complexity;
- a zoomable *timeline view* that represents the history of commits for the file, and where the user can select an active range window;
- a detailed *commit list view* that enables inspection and navigation through the commits in the selected active range.

The visualization is implemented as the *Code Time Machine*, a lightweight language-independent plug-in for the IntelliJ Idea IDE.

[2]See https://code.google.com/archive/p/svn-time-lapse-view/
[3]See https://github.com/JonathanAquino/git-time-lapse-view
[4]See http://gitx.frim.nl/

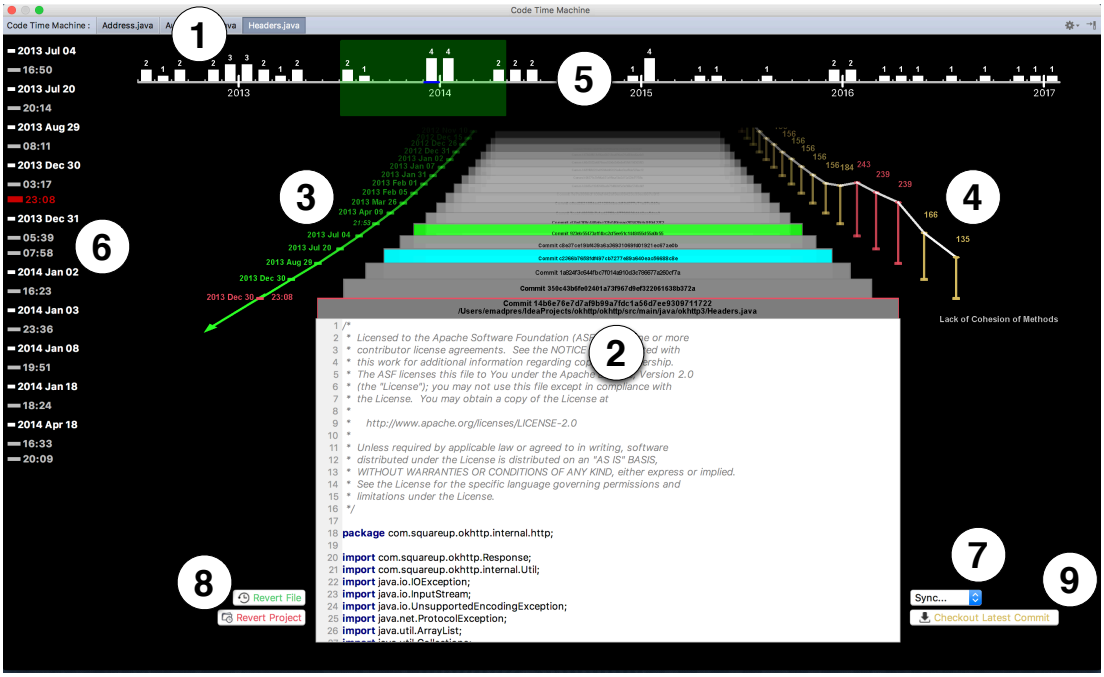

Fig. 1. *Code Time Machine* main window.

II. The Code Time Machine in a Nutshell

Figure 1 depicts the main window of the *Code Time Machine*. A tab list represents the list of running *Code Time Machine* instances on different files (Figure 1-①). The *commits stack view* (Figure 1-②) depicts the history of underlying commits for the file and enables developers to simultaneously explore the evolution of source code and corollary code metrics. In case the file has uncommitted changes, a *virtual commit* is created for the sake of consistency. Each commit window in the stack represents a single version of the file. By hovering on a commit window, details about that commit are displayed on the top of the window (Figure 2).

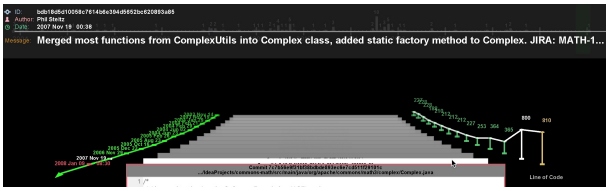

Fig. 2. Displaying commit information by hovering over one.

The *perspective timeline view* displays the commit's time (Figure 1-③) and the *metrics view* shows the evolution of values of code metrics (Figure 1-④). The developer can pick 14 metrics such as Lines of Code (LOC), number of methods, and cyclomatic complexity. The metric values are computed on the fly with no need for any preprocessing.

The *timeline view* (Figure 1-⑤) can be used to explore the commits at different levels of granularity. At the highest level (Figure 3-Ⓐ), a vertical bar represents the total number of commits for each month.

By zooming in, the timeline gets more detailed and commits are displayed individually (Figure 3-Ⓑ).

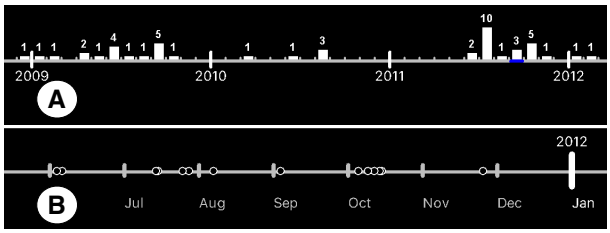

Fig. 3. Different levels of detail in *timeline view*

The *commit list view* (Figure 1-⑥) along the *timeline view* provides an alternative mechanism to explore commits. A developer may select a specific period of time, *i.e.,* an *active range* displayed in green, by clicking and dragging on *timeline view*. Thus, the *commit list view* gets updated to show only the commits pertaining to the active range (Figure 4).

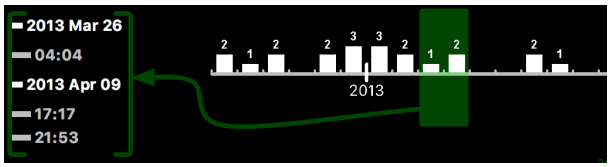

Fig. 4. Updating the *commit list view* by specifying active range.

A developer can hover over any commit in the *commit list view* to inspect the commits' details on top of the window.

A. History Exploration

Our tool supports many ways to explore commits. First, a developer can use *timeline view* and *commit list view* to focus on a specific range of commits. In addition, she may start flying over all commits looking for interesting value changes in any of the available code metrics. After finding a range of interest, she can start navigating through commits one by one either using the keyboard or the mouse wheel.

Diff view. In the commit stack view, commits can be marked using keyboard keys 'B' and 'N': Figure 1-② shows an example of marked commits, which are colored in cyan and green. After marking any two commits and pressing the space bar, a fully featured *Diff window* is displayed in a separated window, without losing the current context (see Figure 5).

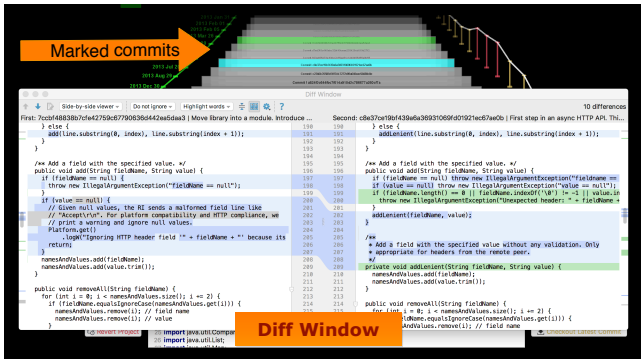

Fig. 5. The *Diff window*.

Distinguishing Commit Authors. This feature allows developers to distinguish the commits made by a specific author. After enabling it, the header of the commits stack windows will be displayed in the same color when it belongs to the same author (see Figure 6).

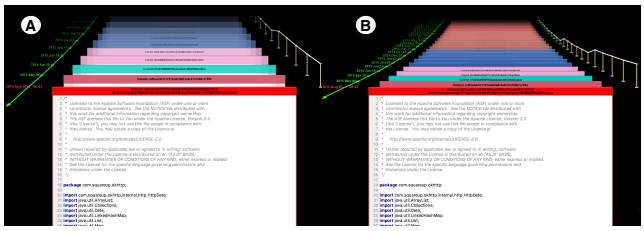

Fig. 6. Example of *commits stack view* customizability and colorful mode.

Commits Stack view Customization. Using keyboard keys 'I'/'K' and 'O'/'L', a developer can adjust the perspective of the *commits stack view* in terms of the maximum visible depth and the distance between commits windows, respectively. Figure 6 also shows two different configurations of the perspective.

Commit Sync. The sync drop-down list (Figure 1-⑦) can be used to synchronize time between two *Code Time Machine* instances representing two files. By pressing the "Sync" button and choosing one of the other available instances, the current view flies to the same commit (or the nearest one in the past if it does not exist) where the other instance is focused on.

B. Time Traveling

Figure 1-⑧ shows the UI components dedicated to time traveling. Pressing the *Revert File* button reverts the underlying file to the currently selected commit, *i.e.,* the topmost window's source code. The *Revert Project* button reverts the whole project to the selected commit.

Both functionalities keep the developer's uncommitted changes safe. After clicking either buttons, the *Code Time Machine* window will get closed to let developers compile and run the project, for example to inspect the runtime behavior of the application in the selected revision.

To revert the whole project back to the latest commit, a developer can click the *Checkout Latest Commit* button (Figure 1-⑨). Any previously uncommitted change is still available in the commit list, and a developer may restore it for a single file or for the whole project by selecting the corresponding virtual commit and using the revert buttons (Figure 1-⑧).

III. The Code Time Machine In Action

Consider a scenario where a developer, Emma, wants to analyze the evolution of a class within the *Apache Commons Math*[5] project. As part of a reviewing activity, she focuses on the *Complex.java* file, that contains the class with the same name modeling a complex number.

By using the *Code Time Machine* she analyzes the evolution of specific code metrics in the history of the file: Starting from the very beginning of the file evolution, she spots a dramatic increase in the LOC, as shown in Figure 2. Comparing the source code using the *Diff window*, she notices that the change relates to a major refactoring, moving methods from the *ComplexUtils* class to the *Complex* class.

To understand more, Emma decides to open the related files, the *ComplexUtils.java* file which is merged with the subject class, and the corresponding test classes *ComplexTest.java* and *ComplexUtilsTest.java*. She moves to the same commit time corresponding to the refactoring in the subject class using the "Sync" button in all of them.

Comparing the *ComplexUtils.java* at that moment, she finds out that a large number of methods have been deprecated and are refactored to call back the *Complex* methods, as she expected. Besides, she finds out that the moved methods' unit tests have been added to *ComplexTest* class correspondingly.

However, she notices that the *ComplexUtilsTest.java* file was not modified, though the unit tests there could be safely deleted as *ComplexUtils* methods are just recalling the *Complex*'s methods and *ComplexTest* is now covering them.

To find out whether this class is refactored later or the developers forgot, she decides to move through *ComplexUtilsTest.java* LOC evolution. She spots the moment when the *ComplexUtilsTest.java* LOC has a dramatic drop. Comparing the source code, she confirms that the unit tests are deleted at this moment, 5 months after the fusion.

[5]https://github.com/apache/commons-math

At this moment, Emma is wondering whether there was some logic behind the decision of keeping *ComplexUtilsTest* unit tests or not. She speculates that the developers forgot to delete them. To support her hypothesis, she decides to run the pruned *ComplexUtilsTest* on the *ComplexUtils* class of the early refactoring time. Thus, first, she reverts the project to the commit time corresponding to the fusion time. She runs the tests and they all pass, as it is expected. Then, she reverts the *ComplexUtilsTest.java* file to the 5-months-newer commit when unit tests are removed. By running the tests again, she observes that the tests still pass, which it means most probably the developers could have refactored the *ComplexUtilsTest.java* right after fusion.

IV. Related Work

There are a number of studies that leverage visualization techniques to understand system evolution. The *Revision Towers* approach [6] models the evolution of a file telling by whom and to what extent a file has been changed. The *RepoGrams* [7] provides a metric-based visualization model to understand the metrics evolution during the history of a software project. Khan *et al.* [8] surveyed the research related to software architecture visualization. Ogawa and Ma [9] use a heuristic approach to present repository evolution focusing on the developers which are involved.

Some researchers have used the matrix as the baseline for their visualizations, such as a two-dimensional matrix to present metrics changes and to improve the software evolution comprehension [10], [11], [12]. Tymchuk *et al.* [13] propose a visualization to detect quality fluctuations using a three dimensional matrix.

A popular technique has been the leveraging of a city metaphor to depict software systems in 3D [14]. There are also a number of other works that use the time concept to enable one to move back and forth through changes [15], [16].

To the best of our knowledge, our tool is the first to bring the code and code metrics seamlessly next to each other for system evolution understanding.

V. Conclusion and Future Work

The Code Time Machine aims to help system evolution comprehension with the support of visualization, fully integrated in the IntelliJ IDE. It provides a uniform code and code metrics history exploration and enables developers to revert a file or a system to a specific version seamlessly.

The main view of the tool integrates a variety of different code metrics right next to the source code. The tool also enables developers to mark two versions and compare their source code to figure out how or why the value of a code metric has changed at some point in time.

As part of our future work, we plan to extend the file history view and enable developers to have the same seamless history navigation experience in terms of package and project scope.

Acknowledgements

We gratefully acknowledge the financial support of the Swiss National Science foundation for the project "HI-SEA" (SNF Project No. 146734).

References

[1] T. Gîrba and S. Ducasse, "Modeling history to analyze software evolution," *Software Maintenance and Evolution: Research and Practice*, vol. 18, pp. 207–236, 2006.

[2] A. González-Torres, F. J. García-Peñalvo, and R. Therón, "A framework for the evolutionary visual software analytics process," in *Proceedings of WSKS 2011 (4th World Summit on the Knowledge Society)*. Springer, 2011, pp. 439–447.

[3] A. H. Caudwell, "Gource: visualizing software version control history," in *Proceedings of OOPSLA 2010 (25th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications)*. ACM, 2010, pp. 73–74.

[4] M. Ogawa and K.-L. Ma, "code_swarm: A design study in organic software visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1097–1104, 2009.

[5] L. Voinea, A. Telea, and J. J. Van Wijk, "Cvsscan: visualization of code evolution," in *Proceedings of SOFTVIS 2005 (2nd ACM Symposium on Software Visualization)*. ACM, 2005, pp. 47–56.

[6] C. M. Taylor and M. Munro, "Revision towers," in *Proceedings of VISSOFT 2002 (1st IEEE International Workshops on Visualizing Software for Understanding and Analysis)*. IEEE, 2002, pp. 43–50.

[7] D. Rozenberg, I. Beschastnikh, F. Kosmale, V. Poser, H. Becker, M. Palyart, and G. C. Murphy, "Comparing repositories visually with repograms," in *Proceedings of MSR 2016 (13th International Conference on Mining Software Repositories)*. ACM, 2016, pp. 109–120.

[8] T. Khan, H. Barthel, A. Ebert, and P. Liggesmeyer, "Visualization and evolution of software architectures," in *Proceedings of IRTG 1131 Workshop 2011, VLUDS 2011 (2nd Visualization of Large and Unstructured Data Sets: Applications in Geospatial Planning, Modeling and Engineering)*, ser. OpenAccess Series in Informatics (OASIcs), vol. 27. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012, pp. 25–42.

[9] M. Ogawa and K.-L. Ma, "Software evolution storylines," in *Proceedings of SOFTVIS 2010 (5th ACM symposium on Software visualization)*. ACM, 2010, pp. 35–42.

[10] M. Lanza, "The evolution matrix: Recovering software evolution using software visualization techniques," in *Proceedings of IWPSE 2001 (4th International Workshop on Principles of Software Evolution)*. ACM, 2001, pp. 37–42.

[11] M. Lanza and S. Ducasse, "Understanding software evolution using a combination of software visualization and software metrics," in *Proceedings of Langages et Modèles à Objets (LMO'02)*. Lavoisier, 2002, pp. 135–149.

[12] M. Lanza, S. Ducasse, H. Gall, and M. Pinzger, "Codecrawler — an information visualization tool for program comprehension," in *Proceedings of ICSE 2005 (27th IEEE International Conference on Software Engineering)*. ACM, 2005, pp. 672–673.

[13] Y. Tymchuk, L. Merino, M. Ghafari, and O. Nierstrasz, "Walls, pillars and beams: A 3d decomposition of quality anomalies," in *Proceedings of VISSOFT 2016 (4th IEEE Working Conference on Software Visualization)*. IEEE, 2016, pp. 126–135.

[14] R. Wettel and M. Lanza, "Codecity: 3d visualization of large-scale software," in *ICSE Companion '08: Companion of the 30th ACM/IEEE International Conference on Software Engineering*. ACM, 2008, pp. 921–922.

[15] L. Hattori, M. Lungu, and M. Lanza, "Replaying past changes on multi-developer projects," in *Proceedings of IWPSE-EVOL 2010 (Joint 11th International Workshop on Principles of Software Evolution and 5th ERCIM Workshop on Software Evolution)*, 2010, pp. 13–22.

[16] L. Hattori, M. D'Ambros, M. Lanza, and M. Lungu, "Software evolution comprehension: Replay to the rescue," in *Proceedings of ICPC 2011 (19th IEEE International Conference on Program Comprehension)*, 2011, pp. 161–170.