Visualizing Evolution and Performance Metrics on Method Level as Multivariate Data

Hagen Tarner[1]
hagen.tarner@paluno.uni-due.de

Veit Frick[2]
veit.frick@aau.at

Martin Pinzger[2]
martin.pinzger@aau.at

Fabian Beck[1]
fabian.beck@paluno.uni-due.de

[1]paluno, University of Duisburg-Essen, Germany

[2]University of Klagenfurt, Klagenfurt, Austria

Abstract

Visualizing the evolution of software metrics helps understanding the project progress of a software development team with respect to code quality and related characteristics. Blending this information with performance information can provide relevant insights into crucial changes in execution behavior and their respective context from code changes. We interpret this composition of evolution and performance metrics as multivariate data and map it to a fine-grained method level. This is the basis for investigating a multivariate visualization approach consisting of a visually enriched tabular representation that provides the method-level details for all the metrics across time, a projection view that shows clusters and outliers among the methods on a higher-level of abstraction, and a timeline view to find relevant temporal changes. Interactions connect the views and allow the users to explore the data step by step.

Keywords: Software Visualization, Software Evolution, Software Performance, Multivariate Data.

1 Introduction

Software metrics provide information about code quality and other technical or socio-technical characteristics of a software system. Studying the evolution of metrics using temporal visualizations has been discussed in research in the past twenty years [5,8] and is part of established code quality tools such as *SonarQube* or *CQSE Teamscale*. These analysis approaches help better understand and track the project progress, for instance, with respect to quality and technical debt. However, the approaches are mostly limited to investigating static information, but not dynamic information reflecting also the execution of the respective software versions, a particular important dynamic property being *performance*.

In contrast, there exist approaches and tools specialized on analyzing performance information [4]. State-of-the-art performance profiling tools, among other views, show the executed methods of a program as a list referencing a small number of performance indicators per method. These lists can be sorted based on one of the metrics to find the *hottest* methods. However, code changes and other aspects of software evolution are not considered. How did the performance metrics change with respect to a previous version? Which parts of the code did change and how did they change? These are relevant questions for developers when trying to improve the runtime properties or fixing performance bugs of a program.

These shortcomings of both software evolution and performance analysis approaches led us to interpreting the evolution and performance metrics as combined multivariate data. Each of the methods of a software system is described by multiple metrics, either providing absolute characteristics of the method or relative characteristics comparing characteristics to a previous version. These metrics also have a temporal dimension as they change with every code modification. We show this evolving multivariate data and enable developers to identify clusters and outliers of methods with respect to the metrics, find relationships between evo-

In: Proceedings of the Seminar Series on Advanced Techniques & Tools for Software Evolution, Virtual conference (originally in Amsterdam, the Netherlands), 01-02 July 2020, published at http://ceur-ws.org

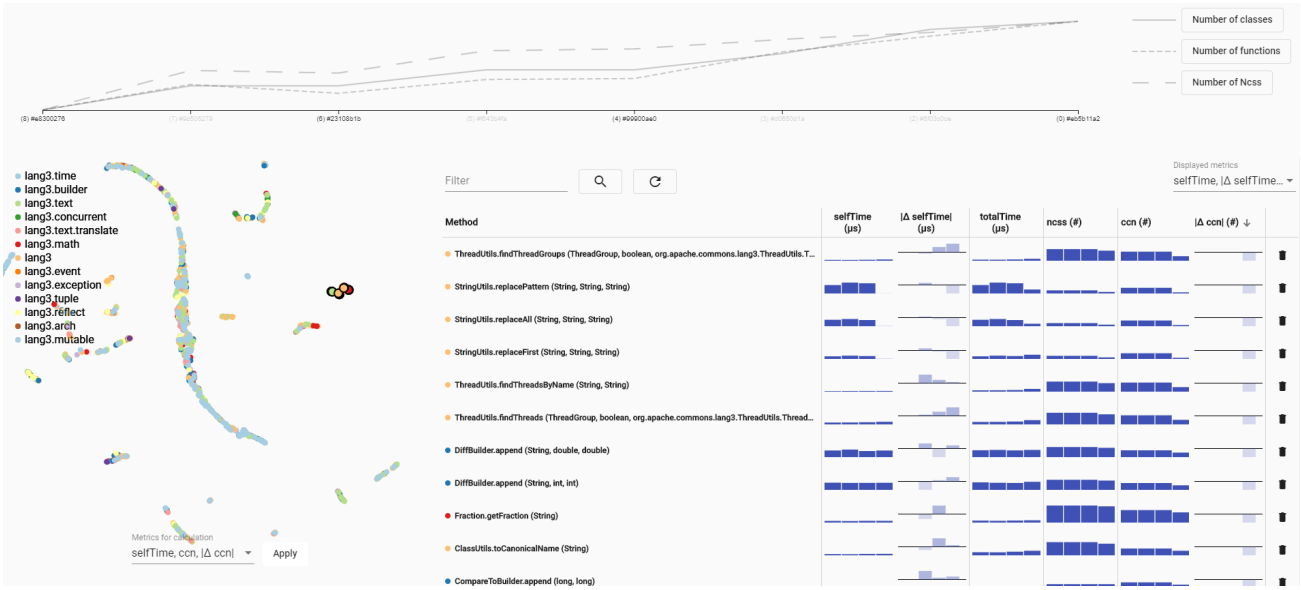

Figure 1: The main interface of our software consists of three components: (I) the Timeline View at the top, visualizing various aggregated static metrics per uploaded commit, (II) the Projection View, showing the UMAP-projection of the gathered metrics as a scatterplot, and (III) the Tabular View on the right side, showing both static and dynamic metrics (columns) per method (rows).

lution and performance metrics, and note code changes that affected runtime behavior and performance.

The specific approach we developed consists of three views as shown in Figure 1. They comprise

- **the Tabular View**, a visually enriched tabular representation with detailed method-level statistics, also representing temporal changes as small-scale bar charts,
- **the Projection View** arranging methods with similar evolution and performance characteristics into clusters while highlighting outliers, and
- **the Timeline View** providing an overview of more versions and forming the basis to select the most interesting ones for detailed analysis.

This is work in progress as—though we present first examples for its applicability—it is yet unclear what are further relevant questions that can be answered with this approach in practical application. We will exemplify some directions as part of Section 6 but also hope to trigger a broader discussion.

2 Related Work

Since a single metric only provides a limited picture, visualizing software metrics as multivariate data is a common approach [1, 6, 7]. Such multivariate metrics have also already been visualization along software evolution [2, Section 5.3.2]. For instance, *Evolution Matrix* [5] suggest to represent the evolution of class-level metrics as rectangular glyphs on a timeline. Software cities—extending the idea of a glyph to represent the multivariate data to three dimensions and combining it with a map-like layout—can be animated to show software evolution [12, 14]. In contrast, multiple variables of a code artifact can be visualized as a line connecting values across multiple axes that are either organized in a parallel (*parallel coordinates*) or radial fashion (*star/spider plot*, *Kiviat diagram*). Pinzger et al. [8] provide an example with radial charts using different colors to show evolutionary changes. These approaches, however, lack an integration of dynamic runtime information. One exception is the work by Dal Sasso et al. [3], where a visualization blends evolution metrics with debugging information.

The visualization of software performance data is also an established area [4] but only few approaches consider the evolution of performance metrics. *Performance Evolution Blueprint* [10] compares two executed software versions with respect to a small set of performance and code change metrics in rectangular glyphs. Instead of discerning different software artifacts, small glyph-based representations can also augment the commit history of a project [11]. The *Performance Evolution Matrix* [9] approach provides a timeline showing multiple versions with expandable packages and classes to investigate performance regressions in detail including method call and activation pattern information. In contrast to our approach,

these works focus on a small set of predefined metrics. Tarner et al. [13] discuss different general visualization approaches for comparison of arbitrary method-level multivariate performance metrics, but allow only a comparison of two versions; still, this work informed the design decisions of the multivariate visualization techniques presented in the following.

3 Software Versions as Multivariate Data

We study software projects as a sequence of versions of method-level multivariate data consisting of software metrics derived from static and dynamic analysis. In the context of this work, the term *version* refers to a specific commit in the *master* branch of each of the projects.

Static data focuses on size and complexity metrics as two of the most important categories of static metrics. It is collected using *JavaNCSS*[1] and comprises the number of non-commenting source statements (`ncss`), and the *Cyclomatic Complexity Number* (`ccn`) per method and version.

Dynamic data relates to different perspectives of software performance in this work. It is gathered by running all unit tests (JUnit) of a project and collecting relevant metrics via *VisualVM*[2]. To account for inaccuracies in measurement we report the average over five repeated measurements. For each method we count the number of invocations (`invocationCount`) and four temporal metrics in milliseconds: a method's self- and total time as wall-clock times (`selfTime` and `totalTime`) and as CPU time measures (`selfTimeCPU` and `totalTimeCPU`).

Metric changes for adjacent versions are calculated as absolute differences (or short: *diffs*) for every metric. To denote the diff variant of a metric, we use Δ as a prefix to the variable name (e.g. Δ`cnn`). Diffs are calculated between temporally consecutive versions in a user selection. The diff metrics, hence, represent a set of evolution metrics.

In addition, few project-level metrics are recorded summarizing all code per version to give an overview of the evolution of the whole project over time. They are gathered by static analysis via *JavaNCSS* and include the number of classes (`#classes`), the number of methods (`#methods`), and the number of non-commenting source statements (`ncss`).

[1] `http://www.kclee.de/clemens/java/javancss/`
[2] `https://visualvm.github.io/`

4 Visualizing the Evolution of Multivariate Software Characteristics

Our visualization approach consists of the three main views, as depicted in Figure 1, which are linked through interactions. A prototype was implemented as a Web application using *D3.js* and *Angular*.

The Tabular View is the main component of the application, as it provides the most detail on method-level. Each selected method in the dataset is represented by one row in the table, while each selected metric forms one column. The selection of displayed metrics is configurable via the drop-down list in the upper right. Each table cell contains a small bar chart visualization (also called *sparkline*), based on the selected metric: (I) absolute metrics are displayed as juxtaposed bar charts, whereas (II) diff metrics are visualized as diverging bar charts (e.g. `selfTime` and Δ`selfTime` in Figure 1). The bar charts for the absolute metrics contain one bar per version. The diverging bar charts for diff metrics contain one bar per diff, that is, per pair of adjacent versions in the selection. Table rows can be sorted by a single metric (by clicking the respective column header). As a basis to sorting, we identify the maximum absolute value of a metric across all versions for each method. The maximum of these values is used to scale the bar heights of a column, to allow for metric-wide comparison of methods. Table rows are color-coded (circle in front of the method name) according to package membership.

The Projection View (scatterplot on the lower left side of Figure 1) is used for visual cluster and outlier detection. It projects the methods described by a set of metrics in an n-dimensional space to a two-dimensional space trying to preserve the spatial neighborhood. We decided to use UMAP as a state-of-the-art projection algorithm because it provides stable results in short time. The input data for the projection algorithm consists of one vector per method, constructed by chaining all selected metrics across all selected versions together. Hence, for four versions and three metrics, each method is described as a 12-dimensional vector. The dimensionality of the data is then reduced to two-dimensional space for plotting. The selection of the metrics relevant for the projection is interactive, and can include static, dynamic, and diff metrics. It is done in the drop-down list below the scatterplot. Each point in the scatterplot is colored according to the package the method is contained in, and follows the same color scale as the circles in the Tabular View. Clicking an item in the scatterplot legend selects all methods of the package for further examination. When a single method is selected, the Projection View also offers nearest neighbor search for a user-defined number of k neighbors, either applied

in the original high-dimensional space or the projected two-dimensional space using Euclidean distance.

Selection and filtering of one or more methods is linked across the Tabular View and the Projection View. Clicking a row of the table or a point in the scatterplot selects that method and updates both views. It is also possible to select more than one method by holding down the *Ctrl*-key. Filtering of the selected methods is done by typing parts of the method signature into the text field above the Tabular View.

The Timeline View (line chart in the top of Figure 1) orders versions by date from left to right and enables the selection of relevant versions. The idea is that more versions are shown in this view than being selected for further exploration with the two views described above. Each version is visualized by three project-level metrics: `#classes`, `#methods`, and `ncss`. The visibility of each metric can be toggled via the buttons on the right side of the line chart. The labels on the x-axis serve as version selection: clicking a label (de)selects the version for further analysis.

5 Interactive Exploration Process

To demonstrate the interactive analysis process, we describe a typical visual data exploration scenario as an application example illustrated in Figure 1.

We load eight pre-processed versions of the *Commons Lang*[3] project into our application to find simultaneously appearing changes in performance and evolution metrics across methods. In the resulting line chart of the timeline view, we select four versions (based on the observed changes in displayed metrics) for further exploration. After version selection, the Projection View and the Tabular View get updated. The table initially shows all methods ordered by their signature. Ordering by Δ`ccn` and selecting only the top row, yields the method with the highest change in `ccn` (`ThreadUtils.findThreadGroups()`) and highlights a single point in the scatterplot. As the initial projection is based on all absolute metrics, we have to compute the projection again to find methods with a similar change in `ccn` in the respective version. As we are also interested in dynamic metrics, we run the projection on `selfTime`, `ccn`, and Δ`ccn` (as seen in the screenshot). In the resulting scatterplot we select the nearest neighbors to the previously selected method and inspect them in the table. Via the *Displayed metrics* selection (top right of the Tabular View), we show only the metrics we are currently interested in. The result of this exploration is the screenshot shown in Figure 1.

The Tabular View now shows a set of methods with a decrease in `ccn` for the last two versions. The methods of the `StringUtils` class also show a decrease in the other metrics for the same two versions. Further investigation into the source code reveals: functionality has been moved to a different class and methods have become deprecated and are now only stubs that call external methods. Furthermore, methods of the classes `ThreadBuiler`, `DiffBuilder`, `Fraction`, and `ClassUtils` show a decrease in `ccn` combined with an increased `totalTime`. That means, while containing less complex code, the time it took to execute these methods increased. The source code reveals that functionality has been replaced by a generalized (and thus more complex) version that takes longer to execute.

[3] `https://commons.apache.org/proper/commons-lang/`

6 Discussion and Future Work

We presented an approach to visualize a blend of static and dynamic software metrics on method level along software evolution. The above example already provides some hints how this can be leveraged for investigating the interplay of code changes and software performance. We have discovered relevant and significant code modifications that led to changed runtime behavior and performance. Identifying performance regressions in the code is also possible by examining the recorded dynamic metrics, and especially their diffs, to see how they change over time. The static metrics provide relevant context to interpret potential root causes of these regressions. However, as there might be other use cases and benefits of this approach, we want to exemplify further possible directions for analysis and extensions of the tool.

Visually exploring the history of an application sooner or later leads to the point where the question of *Why?* comes up: *Why is there a change in metrics for this method?* While this might be trivial to answer for static size and complexity metrics (e.g. `ncss`, `cnn`), it gets harder to do for dynamic metrics. Our approach currently does not answer these kind of questions, but only provides a starting point for further research into the source code. One way to mitigate this could be to show source code and source code changes directly in the application, e.g. clicking the diverging bar of a visualized diff metric could show the actual changes that were made to the source code of the selected method in the two corresponding versions (in a unified diff view, similar to existing visualizations of version control systems).

As we have tested the approach so far only with rather small projects, the scalability of the visual encodings still needs to be evaluated. We expect problems in the Tabular View and the Projection View as their visual complexity increases with the amount of methods. A solution could be to first show the data on

higher-level abstractions (e.g. class or package level), before drilling down to method-level interactively.

A way to improve the Projection View would be to also leverage the temporal aspect of the data more clearly. Instead of aggregating the metrics across time, the projection can be computed per version. Then, the projection can be played as animation to show the software evolution, the different projections can be plotted side by side or overlaid. If challenges on stabilizing the projection across the versions are addressed, one might observe changing clusters of methods and different outliers in different versions.

One option to broaden the application of our program is to extend the set of visualized metrics to other aspects of software development. For example, adding metrics from the software testing domain (e.g. code coverage per line, number of passed/failed tests) would support further analysis scenarios, such as relating changes in runtime behavior to additional tests or bug fixes. Other candidates are metrics related to static properties such as code duplication, coding style quality, and dependencies or dynamic properties such as memory allocation and I/O usage.

Acknowledgments

This work was made possible by the dedicated students of the Master project group *ViVaSD* held in summer 2019 at the University of Duisburg-Essen—Daniel van den Bongard, Nicklas Heuser, Cedric Krause, Jan Reichl—who implemented the prototype. This work has been partly funded by Deutsche Forschungsgemeinschaft (DFG) and Austrian Science Fund (FWF) as part of joint research grant 288909335 (DFG) and 2753-N33 (FWF).

References

[1] F. Beck. Software Feathers: Figurative visualization of software metrics. In *Proceedings of the 5th International Conference on Information Visualization Theory and Application*, IVAPP, pages 5–16. SciTePress, 2014.

[2] P. Caserta and O. Zendra. Visualization of the static aspects of software: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 17(7):913–933, 2011.

[3] T. Dal Sasso, R. Minelli, A. Mocci, and M. Lanza. Blended, not stirred: Multi-concern visualization of large software systems. In *2015 IEEE 3rd Working Conference on Software Visualization*, VISSOFT, pages 106–115. IEEE, IEEE, 2015.

[4] K. E. Isaacs, A. Giménez, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, and P.-T. Bremer. State of the art of performance visualization. In *EuroVis - STARs*, EuroVis, pages 141–160. Eurographics Association, 2014.

[5] 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, pages 37–42. ACM, 2001.

[6] M. Lanza and S. Ducasse. Polymetric Views — a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, 2003.

[7] H. Mumtaz, S. Latif, F. Beck, and D. Weiskopf. Exploranative code quality documents. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):1129–1139, 2020.

[8] M. Pinzger, H. Gall, M. Fischer, and M. Lanza. Visualizing multiple evolution metrics. In *Proceedings of the 2005 ACM Symposium on Software Visualization*, SoftVis, pages 67–75. ACM, 2005.

[9] J. P. Sandoval Alcocer, F. Beck, and A. Bergel. Performance Evolution Matrix: Visualizing performance variations along software versions. In *Proceedings of the 2019 Working Conference on Software Visualization*, VISSOFT, pages 1–11. IEEE, 2019.

[10] J. P. Sandoval Alcocer, A. Bergel, S. Ducasse, and M. Denker. Performance Evolution Blueprint: Understanding the impact of software evolution on performance. In *Proceedings of the 1st IEEE Working Conference on Software Visualization*, VISSOFT, pages 1–9. IEEE, 2013.

[11] J. P. Sandoval Alcocer, H. Jaimes Camacho, D. Costa, A. Bergel, and F. Beck. Enhancing commit graphs with visual runtime clues. In *Proceedings of the 2019 Working Conference on Software Visualization*, VISSOFT. IEEE, 2019.

[12] F. Steinbrückner and C. Lewerentz. Understanding software evolution with software cities. *Information Visualization*, 12(2):200–216, 2013.

[13] H. Tarner, V. Frick, M. Pinzger, and F. Beck. Exploring visual comparison of multivariate runtime statistics. In *9th Symposium on Software Performance 2018*, 2018.

[14] R. Wettel and M. Lanza. Visual exploration of large-scale system evolution. In *Proceedings of the 15th Working Conference on Reverse Engineering*, WCRE, pages 219–228. IEEE, 2008.