A Domain-Specific Language to Visualize Software Evolution

Alison Fernandez[1], Alexandre Bergel[2]

[1]University of San Simon, Bolivia
[2]Pleiad Lab, DCC, University of Chile

Abstract*—*Context: **Accurately relating code authorship to commit frequency over multiple software revisions is a complex task. Most of the navigation tools found in common source code versioning clients are often too rigid to formulate specific queries and adequately present results of such queries. Questions related to evolution asked by software engineers are therefore challenging at answering using common Git clients.**

Objective: **This paper explores the use of stacked adjacency matrices and a domain specific language to produce tailored interactive visualizations for software evolution exploration. We are able to support some classical software evolution tasks using short and concise scripts using our language.**

Method: **We propose a domain-specific language to stack adjacency matrices and produce scalable and interactive visualizations. Our language and visualizations are evaluated using two independent controlled experiments and closely observing participants.**

Results: **We made the following findings: (i) participants are able to express sophisticated queries using our domain-specific language and visualizations, (ii) participants perform better than GitHub's visualizations to answer a set of questions.**

Conclusion: **Our visual and scripting environment performs better than GitHub's visualizations at extracting software evolution information.**

Keywords: Git, history visualization, domain-specific language

I. Introduction

Programming activities often require historical information from source code. Consider the following two software evolution tasks [1]: *"Identify the two classes someone changed the most in the past days"* and *"Identify the methods that someone else has also changed"*. Both tasks are likely to be asked by a developer in order to become familiar with someone else's work or to become aware of a team activity. It has been shown that completing these particular two tasks requires dedicated tooling and traditional code versioning systems are suboptimal in that respect [1].

This paper presents and evaluates a visualization framework to explore the evolution of a source code repository. Our approach is based on two main ingredients: (i) a domain-specific language that focuses on the notion of time, Git commit, and metric, and (ii) a visual way to stack adjacency matrices. The language we have designed aims to tailor visualizations in order to address particular questions related to software evolution.

Executing a script in our domain-specific language produces an interactive visualization. As a visual support, we employ MultiPile [2] as a compact way to summarize and navigate through a set of matrices. MultiPile was proposed as a visualization to explore temporal patterns in dynamic graphs. It employs a natural and intuitive analogy of piling adjacency matrices, each matrix representing a temporal snapshot. MultiPile was designed to help neuroscientists. Our article is about assessing MultiPile to solve software evolution problems.

Contributions. This paper makes the following contributions:

- We present GitMultipile, a domain-specific language coupled with stacked adjacency matrices to produce interactive visualizations.
- We evaluate GitMultipile using two experiments: (i) a first controlled experiment focusing on the expressiveness of our domain-specific language, (ii) a second controlled experiment to compare visualizations of GitMultipile against the ones of GitHub. For both experiments, we observed and monitored the participant activities.

Findings. We made the following findings:

- The language offered by GitMultipile is more efficient than Excel at retrieving data from a simple CSV sheet.
- Participants are more efficient at defining and using visualizations with GitMultipile than using GitHub to answer a set of software evolution questions.
- By observing participants during our experiments, we identified some obvious limitations of the navigation tools offered by GitHub.

Paper outline. Our paper is structured as follows: Section II describes Multipile, the visual foundation used in our work. Section III presents GitMultipile, our approach to assess Git-based repositories using a domain-specific language and Multipile. Section IV illustrates the use of GitMultipile on two large Git repositories. Section V discusses the methodology we use to evaluate GitMultipile. Section VI evaluates the expressiveness of our language by using a controlled experiment. Section VII compares the visualization produced by some participants against the visualizations of GitHub. Section VIII lists some observations of our participants during their activity. Section IX lists the threats to validity our work may be subject to. Section X presents the work related to this paper. Section XI concludes and outlines our future work.

II. Background: Stacking Adjacency Matrices

Matrix pile. Adjacency matrices are often used to visualize edges between software related components [3], [4], [5]. Each

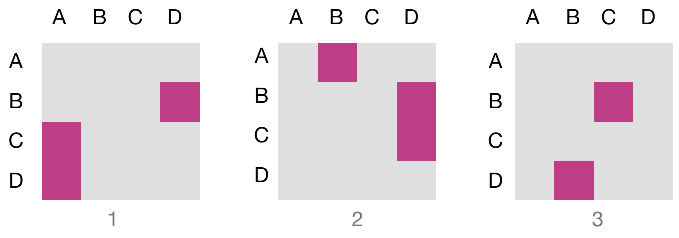


Fig. 1: Three adjacency matrices.

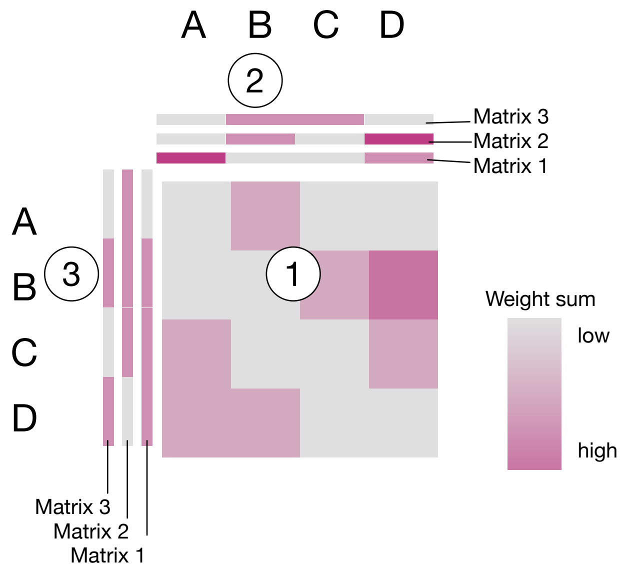


Fig. 2: A piled stack of matrices, obtained from the three matrices given in Figure 1.

element of a matrix indicates whether two elements are related. A matrix is made of edge weights. Figure 1 gives three adjacency matrices. On this contrived example, each matrix is squared and has a size of 4. Matrix 1, located on the left, indicates that element B is connected to element D, while D is connected to A and C to A. We assume that the three matrices represent the evolution in time of the graph composed of the nodes A, B, C, and D.

A matrix pile, as proposed by Bach *et al.* [2], is a structure that stacks adjacency matrices. A matrix pile is the superposition of stacked adjacency matrices. All the matrices that belong to a same pile have the same dimension and same object values for both axes.

Figure 2 represents a piled matrix obtained from the three matrices given in Figure 1. A piled matrix is made of three distinct parts. Part 1 is a matrix showing the superposition of the three matrices. This superposition is called the "coverage matrix" and the weight of element C_{ij} is the average weight of the same cell in all the matrices:

$$C_{ij} = \frac{1}{T}\left(\sum_{n=1}^{T} M_{ij}^{n}\right)$$

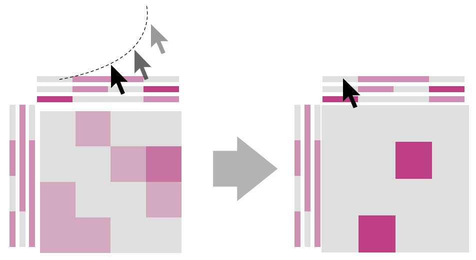


Fig. 3: The previews support navigation in a matrix stack

where M^n is a stacked matrix and T the number of stacked matrices.

Part 2 and Part 3, called top-preview and left-preview, respectively, are a small visual summary of the piled matrices. The previews summarize the content of the pile, each thin bar corresponding to a matrix. The top-preview is made of three thin horizontal bars, each representing a piled matrix. The order of piled matrices goes from bottom to top. Each horizontal bar has n parts, each summarizing a column and n is the number of columns of the coverage matrix. The summary of a part is obtained from the number of the weights greater than 0. Similarly, the left preview summarizes each row of the piled matrices.

A preview is also a navigation widget: locating the mouse above a stacked matrix summary (*i.e.,* thin bar), has the effect to replace the coverage matrix by the actual pointed matrix. Figure 3 illustrates this point: locating the mouse cursor on the top line in the preview replaces the coverage matrix with Matrix 3, given in Figure 2.

Two or more matrices can be piled to produce a pile matrix. Note that the original definition of pile matrix [2] considers the coverage matrix and the top preview (Part 1 and 2 of Figure 2). We extended this original definition with a left preview (Part 3).

Timeline. A visualization may be composed of several stacks of matrices and some non-stacked matrices. A timeline represents a summary of the whole visualization and is also a way to navigate through the different parts by highlighting parts related to the element in the timeline pointed by the mouse.

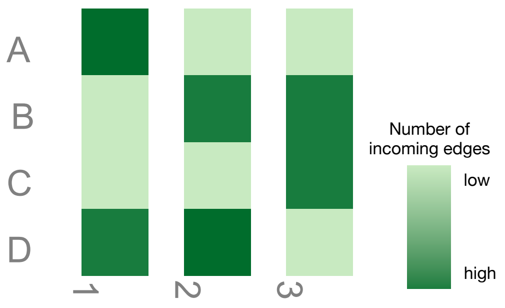


Fig. 4: Timeline for the three non-stacked matrices given in Figure 1

Figure 4 contains a timeline summarizing the three non-stacked matrices. Each stacked and non-stacked matrix has an identifier, and this identifier is used in the timeline to indicate the represented matrix.

Each vertical box of the timeline summarizes a matrix. The first column, with the id 1, represents Matrix 1 given in Figure 1. In that matrix, the element A has two incoming edges (C and D), and D receives an incoming edge from B. The intensity of the green color in (A,1), Figure 4, represents the value 2 (since A receives connections from 2 elements), and the intensity of (D, 1) represents the value 1. In Matrix 1, both B and C are not connected, thus producing cells in the timeline with no weight.

In the timeline, a vertical cell group corresponds to one time period (*i.e.,* one adjacency matrix) and is separated from other matrices. Piled matrices are represented in the timeline as joined vertical cells. Figure 4 represents the three matrices of Figure 1, kept separately, while the timeline of Figure 5 indicates that two piles are formed.

Multipile was presented at EuroVis 2015. Multipile has several benefits, such as topological states, which are quickly spotted and compared in a scalable fashion. This paper reconsiders this work under the scope of software engineering by extending and using it in addressing some software evolution tasks.

III. GitMultiPile Matrix

This section gives an overview of GitMultiPile, our combination of Multipile and a domain-specific language. First a brief and informal description is given (Section III-A). A running example is used to illustrate various aspects of our approach (Section III-B). Each of the subsequent sections covers a particular problem addressed in analyzing software history and uses GitMultipile to define a visual support, useful to address that problem.

A. In a nutshell

The GitMultipile approach is centered around Git commits and stackable adjacency matrices. GitMultipile features the following:

- *Expressing relations* – The domain and codomain of the relation (*e.g.,* authors, files, time) are mapped to the X and Y axes of the adjacency matrices.
- *Computing cell weight* – The weight of a matrix cell is computed as the number of Git commits that match a particular condition.
- *Stacking matrices* – Matrices can be stacked using particular time ranges and conditions. Time range may be deduced by using binary relations, *i.e.,* all the consecutive months in which a particular author is active.
- *Filtering data* – Rows and columns of the matrices may be filtered out using manually set thresholds or defined conditions, *i.e.,* removing all the authors with less than a particular number of commits.
- *Adjusting visual properties* – The overall layout involves the location of matrices over an infinite two-dimensional space. Customizable layouts may be used to accommodate the overall visualization.
- *Highlighting data* – Predicates may be formulated to highlight particular pieces of data, *e.g.,* a particular author or particular relations. Information on demand may also be adjusted to reflect the meaning of each cell.

Projection and transformation metrics may be used when defining the matrices. Transformation may be useful in treating some particular outliers, which would hide particular patterns if not adequately considered.

We have designed a domain-specific language that combines the aspects listed above. Programs written in our DSL are usually short, usually less than 20 lines of code, and are supposed to support a particular analysis of a Git repository evolution. The complete GitMultiple language is described in Appendix.

B. Running example

We take as example a repository of Microsoft named mwt-ds-explore-java[1]. This repository contains 55 commits and 5 authors.

This example is relatively small to comfortably illustrate different aspects of our language, presented in the subsequent sections. Section IV discusses the scalability of GitMultipile.

C. Frequency of commit within a range of time

The frequency of code commits during a particular time range may provide a valuable insight on the effort distribution during that period. This subsection details a short script (14 LOC) that presents the activity at the end of 2015 with the beginning of 2016 of our running example. The result of the script is shown in Figure 5.

Script. The variable `git` refers to a particular Git repository and exposes the domain-specific language (DLS) through commands sent within this context. The first necessary step to initialize a visualization is to indicate the mapping of the Git model to the two matrix axes:

```
git mapFrom: #authors to: #days.
git timeMonth.
```

The lines above indicate that the adjacency matrices represent authors and commit days of Git commits. Authors are represented on the Y-axis and month-days on the X-axis. The `timeMonth` keyword sets the time range represented by a matrix: each matrix represents the activity of a month. All matrices have the same dimensions: vertically the number of authors of the whole project, and the day number of the month, horizontally. As all matrices have the same size, the matrices have 31 cells horizontally, to accommodate with the maximum number of days a month can have. In case a month has less than 31 days, cells of the missing days have a value weight of 0.

The weight of each cell *(author, day)* is determined using the instruction:

```
git
  from: [ :commit :author | commit author = author ]
  to: [ :commit :day | commit day = day ].
```

[1]`https://github.com/Microsoft/mwt-ds-explore-java`

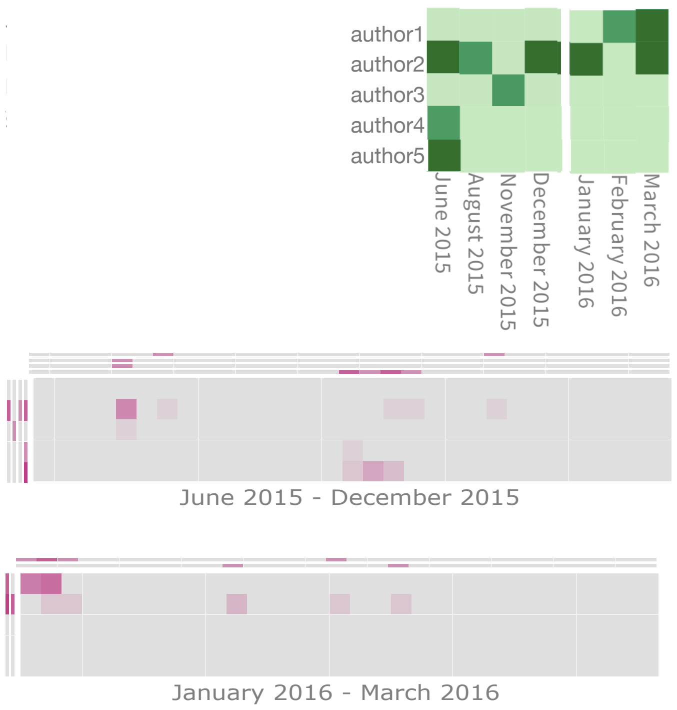


Fig. 5: Comparing the activity at the end of 2015 with beginning of 2016 [Author names are anonymized].

The weight of a cell is the number of commits matching the condition provided with `from:... to:...`. Each cell of a matrix represents all the commits made by a particular author on a particular day of the month. The weight of the cell is the number of commits authored by a particular author on a given day. Visually, a cell with a dark color represents a high number of commits while a light color intensity indicates few commits.

We will distinguish the commits made in the project according to the year. We will compare the period of Jun-Dec 2015 against Jan-Mar 2016 by using two matrix piles. The following two instructions achieve this:

```
7  git pileFrom: (Month month: 'June' year: 2015)
8          to: (Month month: 'December' year: 2015).
9
10 git pileFrom: (Month month: 'January' year: 2016)
11         to: (Month month: 'March' year: 2016).
```

A complete API for manipulating time range and forming date is available, however briefly presented in the Appendix. Each invocation of the command `pileFrom: ... to: ...` declares a new pile, composed of matrices fulfilling the provided time range.

Some parameters may be set to accommodate the visual representation.

```
12 git fromIdentifier: [ :author | author fullName , author email ].
13 git toIdentifier: [ :day | 'Day ', day asString ].
14 git layout verticalLine.
```

Lines 12 and 13 define a textual description of the cell. This description is useful when the mouse cursor pointer is above a cell to give some contextual information. Line 14 defines the layout of the matrices. A vertical layout is selected which means that the first pile is physically located above the second pile. The result of the script execution is given in Figure 5.

Analysis. The script above contrasts the activity of two periods of time for our running example and Figure 5 shows the visualization produced from its execution. The visualization shows a number of interesting facts. In particular the timeline indicates:

- Four developers (author2, author3, author4, and author5) participated in the development in 2015 while only two developers (author1 and author2) in 2016.
- Author2 is the only developer who has a relatively constant activity over the analyzed period. Other developers contributed for a short period of time.

The two piled matrices provide additional information about the overall activity. In particular they reveal:

- During 2015, author2 made several commits on the fifth day of the months. The left preview indicates that commits were realized in 3 of the 4 months composing the 2015 period.
- During 2015, author5's contributions are concentrated on only three days, 16, 17, 18.

D. *Stacking matrices to reflect author changes*

Section III-C shows that author2 made the most commits in our running example. In this section, we will focus on his activity by highlighting commits made on Java files.

Script. The script we will describe produces the visualization given in Figure 6.

```
1 git mapFrom: #authors to: #days.
2 git timeMonth.
3 git
4   from: [ :commit :author | commit author = author ]
5   to: [ :commit :day | commit day = day ]
6   weight: [ :commits | commits select: [ :c | c containsFile: [ :file | file
        fullName endsWith: '.java' ] ] ].
```

Similarly as in Section III-C, each matrix maps authors to days (Line 1) and each matrix represents a month (Line 2). The weight function for each matrix cell is slightly more complex than earlier. In this case, the weight is defined as the number of commits that modified at least one `.java` file, made on a `day` and by an `author`.

A stack of matrices superposes the activity of several months. A condition may be provided to highlight a portion of author2's work:

```
7 git pileIf: [ :matrix |
8   matrix containsAuthorNamed: 'author2' ].
```

Lines 7-8 create piles for consecutive months in which the author has committed. Figure 6 shows that only one pile is formed, between January and March 2016.

Particular authors may be highlighted:

```
9  git
10   highlightAuthorName: 'author2'
11   using: Color blue.
```

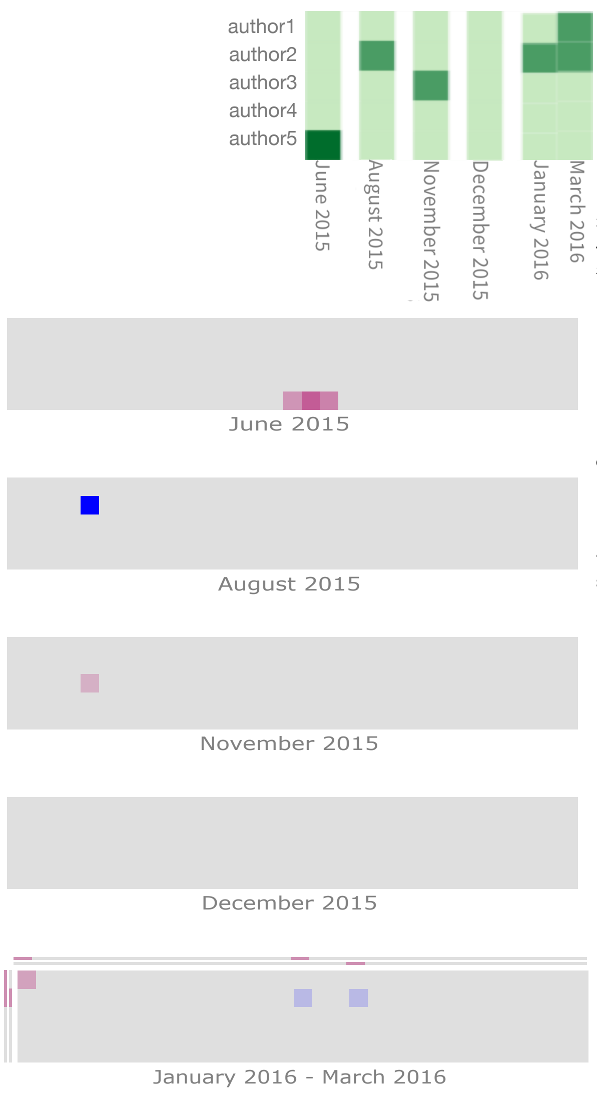


Fig. 6: Highlighting a particular author and his activity.

Commits made by a particular author are colored in blue. Colors may be mixed in case more than one `highlightAuthorName: ... using: ...` clauses are defined.

Analysis. The script given above produces a visualization representing the consecutive periods of activity on Java files by a particular author. In particular, author2 committed during the periods: August 2015 and January 2016 - March 2016.

E. *Activity on files*

As we have previously seen (Section III-C), the participation of author5 is punctual in the project since he worked only in June 2015. This section delves into author5's activity during that period of time. The result is presented in Figure 7.

Script. We focus on the activity per file per authors during a month-time period. The matrix dimensions are defined as:

```
git mapFrom: #authors to: #files.
git timeMonth.
```

Each matrix represents a month, as in the previous examples. The Y-Axis represents authors while files are located on the X-axis. Each cell represents therefore the activity of an author on a particular file, during a month period.

The weight of each cell is computed as the number of commits made by `author` and that involve `file`:

```
git
  from: [ :commit :author | commit author = author ]
  to: [ :commit :file | commit containsFile: [ :f | f = file ] ].
```

Since we know that author5 has contributed in June 2015 only, we isolate this month from all the others by stacking all the remaining months (*i.e.,* the months that do not contain author5's activity):

```
git sequenceIf: [ :matrix | matrix containsAuthorNamed: 'author5' ].
```

We highlight author5's activity in green:

```
git highlightAuthorName: 'author5'
   using: Color green.
```

Analysis. Figure 7 shows the result of the script execution. The time on top of the figure indicates two piles: the first made with a unique matrix corresponding to June 2015 and the second pile to the months August 2015 to March 2016. The timeline shows that author5 made the the most contributions in June 2015. The green line in the matrix June 2015 indicates that author5 committed some changes over a large number of files. By contrasting this matrix with the stacked matrices, author5 is the only contributor who committed over so many files. Actually, since our running example project was created in June 2015, we deduce that author5 created all the green files, and later did not touch them anymore.

IV. Case Study on Large Repositories

This section uses GitMultiple to represent history of large software systems.

A. *Author's commits per day*

The repository `elastic/elasticsearch` is one of the most popular Java projects kept on GitHub. The project[2] has 769 different contributors, totaling 26,220 commits. According to gittrends.io [6], elasticsearch has a truck factor of 7 people, which is pretty high compared to other Git repositories.

We used GitMultipile to highlight the truck factors over a period of time from August 2016 to January 2017, as depicted in Figure 8. The figure is produced by the following script:

```
git mapFrom: #authors to: #days.
git timeMonth.

git
  from: [ :commit :author | commit author = author ]
  to: [ :commit :day | commit day = day ].

git fromIdentifier: [ :author | author fullName , ' ', author email ].
git toIdentifier: [ :day | 'Day ', day asString ].

authors := #(tf1' 'tf2' 'tf3' 'tf4' 'tf5' 'tf6' 'tf7').
```

[2]`https://github.com/elastic/elasticsearch`

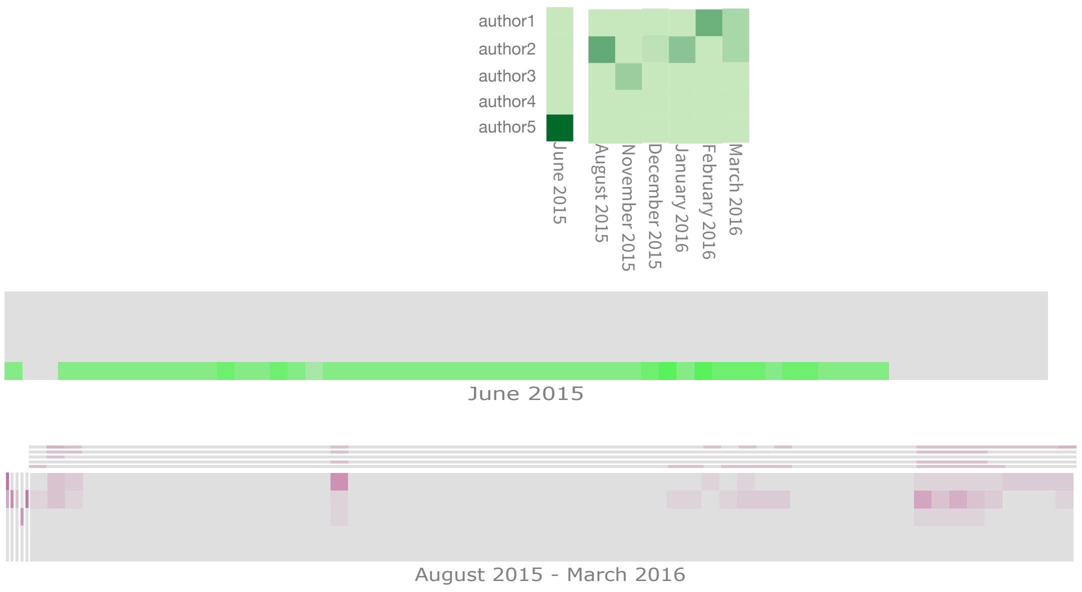


Fig. 7: The green horizontal line indicates a commit affecting many files

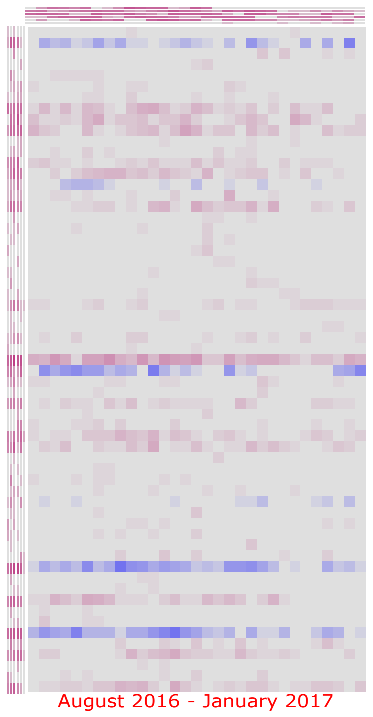


Fig. 8: Portion of the ElasticSearch history

```
git pileIf: [ :matrix |
   matrix containsAnyAuthorMatchingFrom: authors
    ].

git filterOutAuthor: [ :author | author numberOfCommits < 2 ].

authors do: [ :a |
   git highlightAuthorMatching: a  using: Color blue ].
```

In order to preserve author anonymity, we replaced the real author name by a generic term to designate a truck factor. The variable `authors` contains the authors listed by gittrends.io[3]. Figure 8 clearly indicates in blue the contribution of the truck factors over the considered period of time. The script filters out authors that have less than 2 commits.

B. *Author's commits per hour*

Brackets is an open source code editor for the web, written in JavaScript, HTML, and CSS. Brackets is a popular application, with over 27K stars and 340 contributors. Brackets has a truck factor of 5[4].

The development of Brackets began in 2012 and it is still under a sustained development. We use GitMultiple to perform two tasks: (i) see the distribution of commits along a day's hour, and (ii) contrasts two time periods, January - September in 2012[5] and in 2017[6]. The truck factor is highlighted.

We generated a visualization using the script:

```
git mapFrom: #authors to: #hours.
git timeMonth.

git
   from: [ :commit :author | commit author = author ]
   to: [ :commit :hour | commit hour = hour ].

git fromIdentifier: [ :author | author fullName , ' ', author email ].
git toIdentifier: [ :hour | 'Hour ', hour asString ].

truckFactorAuthors := #('tf8' 'tf9' 'tf10' 'tf11' 'tf12').
```

[3]http://gittrends.io/#/repos/elastic/elasticsearch
[4]http://gittrends.io/#/repos/adobe/brackets
[5]https://github.com/adobe/brackets/graphs/contributors?from=2011-12-07&to=2012-09-10&type=c
[6]https://github.com/adobe/brackets/graphs/contributors?from=2016-12-07&to=2017-09-10&type=c

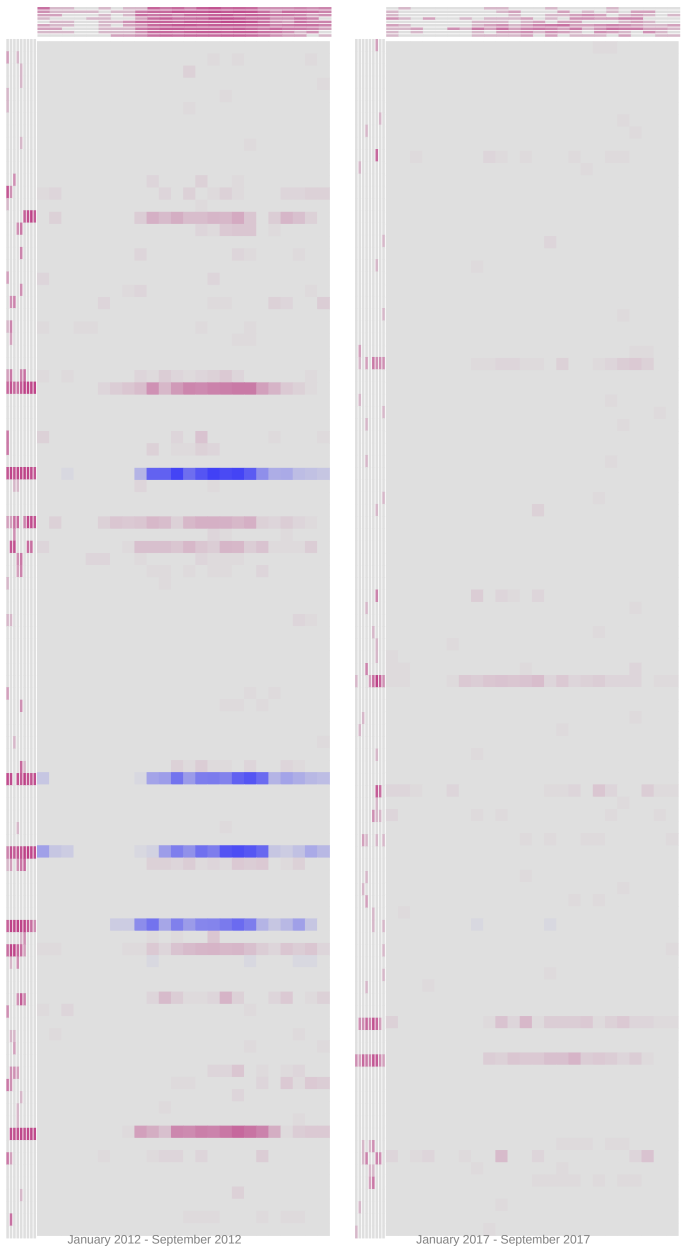


Fig. 9: Two time periods of Adobe's Brackets

```
git pileFrom: (Month month: 'January' year: 2017)  to: (Month month: '
        December' year: 2017) .
git pileFrom: (Month month: 'January' year: 2012)  to: (Month month: '
        December' year: 2012) .

truckFactorAuthors do: [ :a |
     git highlightAuthorMatching: a using: Color blue ].

git layout horizontalLine.
```

The script produced the visualization Figure 9, which is composed of two large matrix piles. The left matrix covers the months January until September in 2012 and the right pile covers the same months in 2017. Each line in a preview (*i.e.,* a matrix) represents a month time period. We have anonymized contributor names.

We draw the following conclusions:

- Most commits happen after 8:00 am, as the top preview indicate. Commits made before may indicate contributor living in a different time-zone.
- The activity in 2017 is considerably reduced when compared with the same time period in 2012.
- The truck factor is composed of 5 developers[7], colored in blue in the piled matrices. In the period January - September 2012, only 4 developers belong to the truck factor, thus indicating that the fifth one became active later on. We also see that the truck factor was very active during the 2012 period, and did very little in 2017.
- The left preview indicates many developers were constant in their effort. However, not all of these authors belong to the truck factor.

V. Evaluation Methodology

GitMultipile combines a domain-specific language, a visualization framework, and an environment in which scripts may be interpreted. These three facets therefore structure our methodology to evaluate GitMultipile.

A. *Three pillars*

DLS expressiveness. The linguistic constructions given in Section III are the basic blocks in formulating queries over a dataset – a Git repository in our case. GitMultipile offers several constructs to filter and query the history of a repository. The first research question we will evaluate is

Q1 - "How expressive are the querying facilities offered by GitMultipile's domain-specific language?"

More specifically, we will assess the expressiveness of the offered language constructions to formulate queries.

Visualization. Visualizations play a major role when navigating and retrieving data from the history of a Git repository. The second research question we will assess is

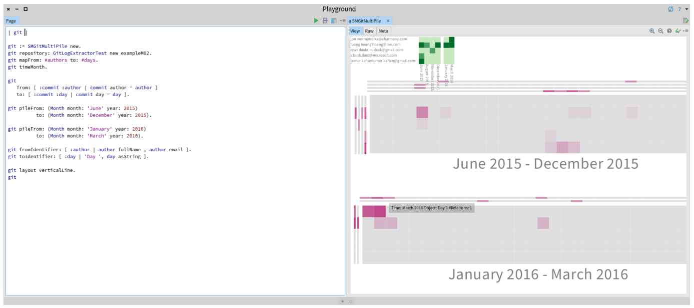
Q2 - "How effective is the GitMultipile visualization?"

In particular, whether the produced visualizations reduce the search time and increase the accuracy of the participant answers.

Usage. We built a simple but effective environment to edit and run GitMultipile scripts, built on top of GTInspector [7]. Figure 10 illustrates the GitMultipile execution environment. Scripts are typed in the textual panel located on the left-hand side. The script is then executed by clicking on the green triangle button, above the text pane, to produce the visualization on the right hand side.

Observing user activities and identifying questions that are raised during our experiments is proven to be effective. We will apply the methodology formulated by Sillito *et al.* [8]: we will identify the questions that a user is answering based

[7]http://gittrends.io/#/repos/adobe/brackets



```
| git |

git := SMGitMultiPile new.
git repository: GitLogExtractorTest new exampleM02.
git mapFrom: #authors to: #days.
git timeMonth.

git
   from: [ :commit :author | commit author = author ]
   to: [ :commit :day | commit day = day ].

git pileFrom: (Month month: 'June' year: 2015)
         to: (Month month: 'December' year: 2015).

git pileFrom: (Month month: 'January' year: 2016)
         to: (Month month: 'March' year: 2016).

git fromIdentifier: [ :author | author fullName , author email ].
git toIdentifier: [ :day | 'Day ', day asString ].

git layout verticalLine.
git
```

Fig. 10: The GitMultiple environment in action

on the activity being carried out. We will then classify these questions and measure their frequency.

B. Controlled experiments

We will conduct two controlled experiments [9] in which the measured and tested dependent variables are the productivity of a particular task and the independent variable reflects the treatment (the used tool, GitMultipile, Excel, GitHub, in our case) used to carry out the task.

Experimental design decisions. We will consider the following: (i) each experiment will use a distinct set of participants to avoid learning effects, (ii) we will use a within-subject design (*i.e.,* all participants are exposed to every condition), (iii) only experienced professional developers will be used as participants, (iv) task assignment to subjects is randomized.

Baseline for the first experiment. A baseline is a treatment that is used as a standard of comparison. In order to have meaningful results, it is important to use a *fair* (*i.e.,* impartially selected to not favor our results) and *representative* (*i.e.,* match current practices) baselines in our experiments.

GitMultipile provides a domain-specific language to formulate queries. Measuring the expressiveness of our language requires a baseline to which our language has to be compared. Unfortunately, there is no obvious choice for such a comparison. We reviewed the different candidate baselines for our experiment:

- gitql[8] is a SQL like query language for a Git repository. Although appealing at first glance, gitql does not provide a tutorial nor a solid documentation. Making our experiment fair requires a teaching material of the system we will compare GitMultpline against.
  However, comparing GitMultipile against gitql will be (i) biased toward GitMultipile if we consider the rather poor documentation of gitql, or (ii) biased if we designed the gitql documentation ourself (*e.g.,* we could unintentionally design a sub-optimal documentation that would naturally favor our system).
- GitHub Developer API[9] is an API accessible via network requests, which return JSON descriptions as results. Both the network query formulation and the JSON description manipulation require dedicated tools. The choices we may have in picking these tools as the "glue" between the network and JSON aspect are significant biases. In addition, this tool chain cannot be considered as a natural solution that matches current practices to query Git repositories: this API is designed to build Git clients and not solve particular software evolution tasks.
- GitPython[10] is a full-fledged API to query Git repositories. GitPython is a complex and large framework in which repository queries are expressed by creating and combining classes offered by the framework. Most of queries expressed using GitMultipile involve significantly more lines of code using GitPython. GitPython is not made to write short script, instead, it is a solution to perform sophisticated Git repository manipulations.

We also looked into commercial products. GitPlex[11], developed by the PMease company, is a sophisticated Git repository management server. The purpose of GitPlex is to manage Git repositories (including issue management and build pipeline), and not directly query the history of a repository.

Microsoft Excel is commonly used to explore structured and unstructured data. It is also used as a baseline in another controlled experiment around software evolution tasks [10]. In that related work, Excel is used as a baseline against CodeCity to solve tasks that code metrics. Moreover, Excel offers a large number of functions to filter and manipulate data sets.

In our first controlled experiment, we pick Excel as the baseline to compare GitMultipile against. Excel is well known among practitioners and frequently employed to manipulate data. Comparing Excel's querying and filtering features against GitMultipile is therefore relevant. We will provide the data ready to be processed by Excel.

Baseline for the second experiment. In our second controlled experiment, we use GitHub's visualizations and navigation tools as the baseline. GitHub offers several visualizations that let one crawl over the activity of a Git repository. Comparing the visualizations offered by GitHub against the one produced by GitMultipile is therefore relevant. The way stacked matrices are defined and structured will be evaluated against the GitHub visualizations.

Oracle. In order to unequivocally determine if a participant answer is correct or not, we need an oracle. We have defined the oracle manually, using all the available treatments participants will be exposed to (Excel, GitHub, GitMultipile). We have determined each answer using one treatment and checked with the other treatments. No discrepancies were found between the answers obtained from the different treatments. We can therefore conclude that all the correct answers can be found using the treatments we will use in our experiment. We collected answers for both research questions that way.

Scoring. Each question has to be answered with one or more textual items (*e.g.,* dates, author names, number of commits).

[8]https://github.com/cloudson/gitql
[9]https://developer.github.com/v3/search/
[10]http://gitpython.readthedocs.io/en/stable/
[11]https://www.pmease.com/gitplex

Each answer to a question provided by a participant can be correct, incorrect, or partially correct. For each answer, we compute the precision (fraction of items contained in a participant answer that are correct) and the recall (fraction of the correct items contained in the answer from all the expected correct items). We then compute the F-measure that combines precision and recall (F = $2.\frac{precision.recall}{precision+recall}$). The score of each question answered by a participant is the F-measure of the answer. A score of 1.0 means that the answer has both the precision and recall equal to 1.0. A score of 0.0 means that either the precision or the recall is equal to 0.0. Scoring each participant answer with the F-measure has the benefit of simplifying the analysis since only one numerical value is associated. Alternatively, we could have used the precision and recall separately, however no clear benefit would be gained by doing so.

Work session. Both controlled experiments will follow a work session as follows:

1) *Learning material A* – We provide a short description, written in a mini-tutorial fashion that illustrates how to use treatment A. Examples used in the description are smaller and different from the tasks T1 and T2.
2) *Task T1 on A* – A number of questions (4 or 5, depending on the task) are asked to a participant. Answers are written on a sheet of paper.
3) *Learning material B* – We provide learning material for treatment B.
4) *Task T2 on B* – Similarly as earlier, questions are asked to the participant and answers are written on a sheet of paper.
5) *Experiment feedback* – Open comments are gathered informally and orally to not pressure the participant into giving an answer that we expect.

We will observe and monitor the execution of each work session. Participants performed the two tasks on a computer, and reported their answers on paper.

Software evolution questions. It is known that developers often implicitly or explicitly formulate questions when performing a software-evolution activity [8]. We have produced a set of questions (see our additional material, Section VI, and Section VII) to define a benchmark to measure the effectiveness of an evolution-related activity. Our questions are inspired from existing attempts at proposing such a benchmark [1], [10] and are restricted to the notion of time, authors, commits.

Pilot Study. Before carrying out our experiments, we performed for each experiment a pilot run. In this pilot run, two professional engineers from a local Chilean software company have run the experiments described later. This pilot run led us to make some adjustments in our experimental design:

- *Improved our questionnaire* – Our participants had some troubles to precisely understanding some questions in each of our two experiments. We therefore simplified these questions to remove the ambiguities in their formulation. Note that the questions are formulated in English while our participants are native Spanish speakers.
- *Improved keywords of our DLS* – The participants had to often look for the meaning of some keywords from the teaching material we provided. This means that our original set of keywords was not intuitive. We therefore improved our DLS by simplifying the set of keywords defining the language.

The two participants agreed with our improvements of the experiment. Note that the two participants did not participate in the full experiment as the knowledge acquired in the pilot run could be source of biased measurement.

Additional material. The following sections detail our experiments. Since we cannot provide every detail of our experimental measurements, we therefore invite the reader to access our additional material, available online (`https://www.dropbox.com/s/ki7uyml24yeq1jd/Material.zip?dl=0`).

C. Participants

We have a pool of twenty participants (two women and eighteen men): eight of them are based in Bolivia, while the remaining twelve are from a local company in Chile. Participants are all young professional software engineers (all with less than 8 years of working experience and the oldest participant is 39 years-old), making their the largest part of their earning gross from professional software development. None of the participants is known to be color blind: we did not conduct any test and none of the results may cast suspicion about whether some participants are color blind.

VI. Evaluating the Expressiveness and Usability

When defining a new language syntax, it is important to keep the right balance between the expressiveness of the language and the cognitive effort needed to learn it.

By expressiveness, we refer to the ability of the language to express queries and GitMultiple visualizations about a historical dataset. The intuition we are building on, is that before solving some software tasks, we assess whether our domain-specific language is fit to solve those tasks.

We evaluate the expressiveness and usability of GitMultipile by studying how users formulate queries for a given set of tasks. We measure the performance of each participant based on the formulated queries. As a consequence, we do not identify and characterize all the possible queries that may be formulated using a particular treatment (GitMultiple or Excel), which we consider as outside the scope of this work. Furthermore, we employ a set of tasks that may be solved using both treatments.

Motivation. In this first evaluation, we will compare our DLS with Excel. We will use a reduced version of GitMultipile by stripping out the Git aspect. Users will therefore have to formulate queries using the construct `pileFrom:to:`, `pileIf:`, and `sequenceIf:` without referring to Git. By removing the Git aspect, conditions used in the script are simply formulated as checking the presence or absence of particular values in the matrices.

We chose to strip the Git aspects when evaluating the expressiveness since (i) Excel does not natively support queries

Part.	EX - Score	EX-Time	GM - Score	GM - Time
P1	3.8	36	4.8	23
P3	3.6	37	5	19
P5	2	44	4.8	30
P7	3.3	40	4.8	25
P2	3.5	32	5	28
P4	3.1	33	4.8	23
P6	3.6	35	4.5	30
P8	4.6	53	5	27

TABLE I: Score and time (in minutes) of Excel (EX) and GitMultipile (GM). Gray cell indicates Dataset 1, white cell indicates Dataset 2.

over a Git history and (ii) using Git may be a confounding variable (*e.g.,* a negative result may be due to the complexity of Git and not to the DSL). We use the term GitMultipile^T to refer to GitMultipile trimmed from the Git aspect.

Datasets. We use two datasets and each dataset is expressed as a tab-separated values file that relate some identifiers to other identifiers at a given time. Here is an excerpt of a dataset:

```
time  author  class  weight
1     aut1    RT1    1
1     aut1    RT2    1
2     aut2    RT20   1
2     aut3    RT5    1
```

Both datasets are similar in their size. Our first dataset contains 524 data points and the second dataset contains 642, spanned over 19 time values. The dataset represents an evolving graph for which the edges and their weights evolve over time.

Questions. We formulated five questions that have to be answered by identifying values of the `time`, `author`, or `class` columns that match a particular condition. For example, one question asks about the biggest number of classes having incoming edges from only one author during the first four `time` periods. Another question is about the a `class` name that appears in each `time`. The questions we considered are similar to the ones we used in our previous study [11] and in the second experiment (Section VII, but without referring to Git).

The maximum score a participant can have is 5 since answering a question gives a value between 0.0 and 1.0.

Running the experiment. As described earlier, the work sessions are structured into two tasks: Task *T1* on treatment *A*, and Task *T2* on treatment *B*. This experiment was run with 8 participants, totaling 8.5 hours. With four participants we have *A* = Excel, *B* = GitMultipile^T. Treatments *A* and *B* are swapped with the remaining four participants.

With four participants, we have *T1* that uses the first dataset and *T2* uses the second dataset. The remaining four participants have *T1* and *T2* that are swapped.

Results & Analysis. Table I contains the score and time taken by each participant. Figure 11 represents the distribution of the time and score using a Tukey boxplot.

We compare both the score and the time to complete the tasks. Using Excel participants have a median score of 3.5 (average = 3.4, standard deviation σ = 0.7) and a median score of 4.8 using GitMultiple^T (average = 4.8, σ = 0.1).

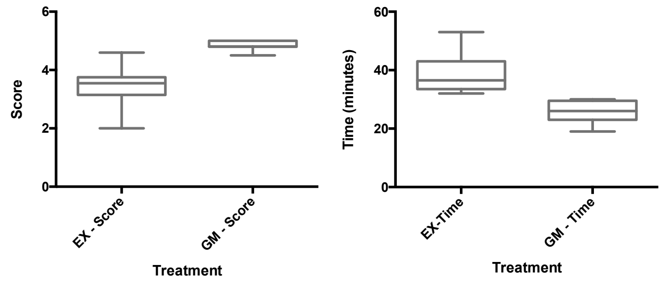


Fig. 11: Tukey boxplot of score and time of Excel and GitMultipile

We use the non-parametric test Mann-Whitney to measure the significance between our two data sets. A two-tailed test indicates that they are significantly different, with P = 0.0003.

Since our time values are normal, we use the classical t-test to estimate if the two score values are different. A two-tailed test gives $P = 0.0004$, which is lower than 0.05, thus indicating that the score values are significantly different.

We provided two datasets. We need to verify if these two sets are comparable in their difficulty to complete. The idea is to discard a bias that may be due to a dataset significantly more difficult to answer the questions than when using the other one. We compare the values indicated with gray cells with the white cells, for both the column "EX - Score" and "GM - Score". The Mann-Whitney test indicates that comparing Dataset 1 and Dataset 2 with Excel and GitMultiple^T results in $P > 0.9999$, U = 8.0. We therefore conclude that both datasets have a similar difficulty to process.

The effect size is a way to quantify the size of the difference between two groups. Cohen's d indicates a standardized difference between two means, obtained from two groups of values. Cohen's value expresses this difference in standard deviation units. Cohen's d = 1.874 for the score and d = 2.458 for the time. In both cases, Cohen's d is bigger than 1, which means that the difference between the means for both the score and time is larger than one standard deviation. We conclude that the effect size is large (if d is larger than 0.6 then it is usually considered as a large effect).

Since the difference is significant, the null-hypothesis is rejected, the effect size is large, we conclude that:

- Participants graded significantly better using a trimmed version of GitMultipile compared with Excel.
- Participants completed the questionnaire significantly faster when using a trimmed version of GitMultipile than Excel.
- Both datasets are similar in their difficulty and time to complete for a given treatment

VII. Evaluating the Visualization

Motivation. GitHub offers several visualizations and navigation tools to let one browse and reason about the history of a Git project. Figure 12 illustrates three GitHub tools, indicated as A, B, and C on the figure.

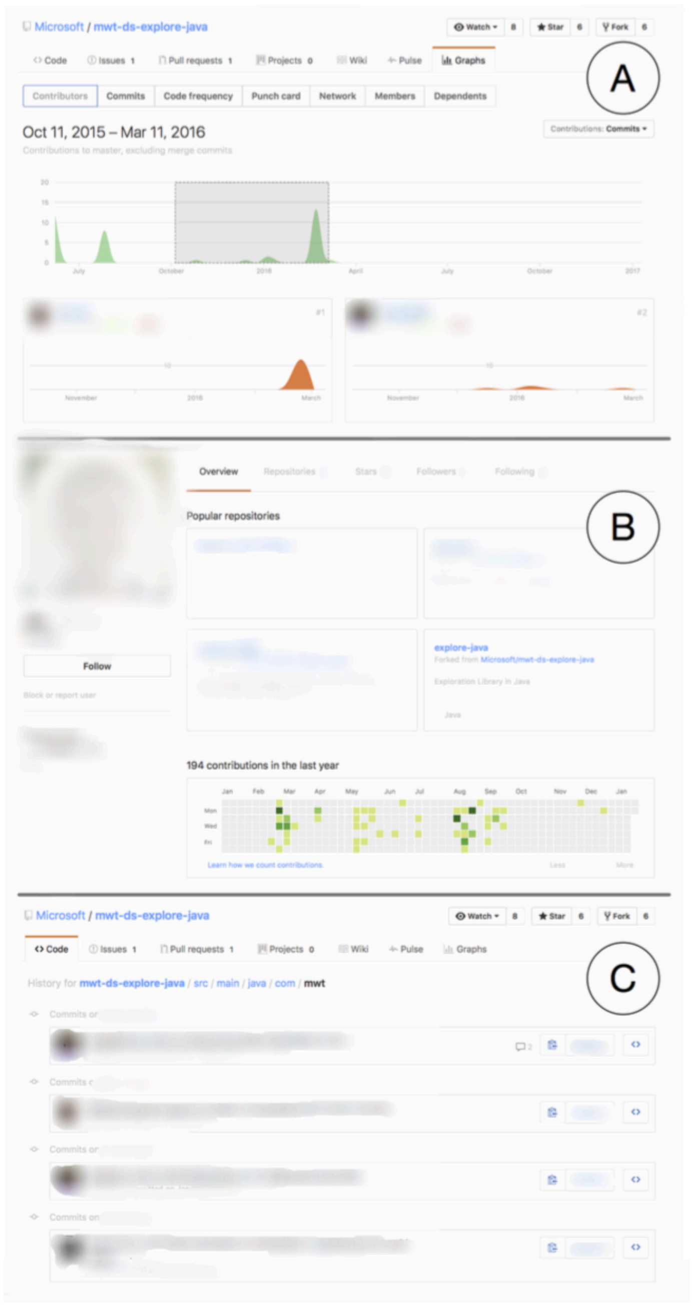


Fig. 12: Visualizations and navigation tools offered by GitHub

Part A offers an overview of the contribution to the master branch over time. A graph indicates the activity over a period of time. One can select a portion of time to see those who contributed in the selected time period. Contributors are listed and ranked according to their number of commits. Clicking on a contributor leads us to his or her personal page (Part B).

Part B summarizes the activity of a contributor. Popular repositories are listed in a calendar heatmap visualization to summarize the overall contributions across all repositories.

Part C illustrates the file inspector. The content of each file may be browsed via the Git interface. A `history` button gives the list of all the commits and contributors related to that file.

Part.	GH - Score	GH - Time	GM - Score	GM - Time
P9	2.5	21	4	15
P11	3	38	4	28
P13	2.5	30	3.5	22
P15	2.5	34	4	27
P17	4	30	4	18
P19	2.5	32	3.5	24
P10	2.8	32	4	17
P12	2.9	44	3.8	18
P14	3.8	33	3.73	18
P16	1.9	40	4	20
P18	2.9	44	3.5	20
P20	3.8	47	3.8	22

TABLE II: Score and time (in minutes) of GitHub (GH) and GitMultipile (GM). Gray cell indicates the use of mwt-ds-explore-java, while cell indicates mssql-jdbc.

GitHub offers several additional tools (*e.g., blame* to see the author of each line of code and *pulse* to track the active pull requests and issues). However, we consider them as out of the scope for this work. Instead, we have presented the three views that are related to our effort.

Datasets. We use two GitHub repositories to conduct our experiment: `Microsoft/mwt-ds-explore-java` (Section III-B) and `Microsoft/mssql-jdbc` (https://github.com/Microsoft/mssql-jdbc), another Java project from Microsoft.

Questions. We formulated four questions to describe a task. We have a set of questions for each dataset. For example, for the exercise involving mwt-ds-explore-java, we use the following set of questions:

Q1 - Who and when did someone commit a change that modified the file named `Test.java`?

Q2 - Who worked during the most days during the same month on a file named `Test.java`? For how many days did the development take place?

Q3 - When and who has committed a change on a file that belongs to the path `src/main/java/com/mwt/sample`?

Q4 - What is the greatest number of commits pushed during one single day by a single contributor during the months when Jon Morra contributed?

Questions for the other application may be found in our additional material (see Section V-B).

Running the experiment. In total, 12 participants were involved in this second controlled experiment, totaling 10 hours and 59 minutes. With six participants we have A = GitHub, B = GitMultipile. The six other participants have the treatment swapped.

With six participants we have *T1* using `mwt-ds-explore-java` and six others we have *T2* using `mssql-jdbc`. The other other participants have the tasks swapped.

Results & Analysis. Table II contains the score and time taken by each participant. Figure 13 represents the distribution of the time and score using a Tukey boxplot.

Applying the non-parametric test Mann-Whitney indicates that data are significantly different both for score and time (P = 0.0011, $P < 0.0001$, respectively)

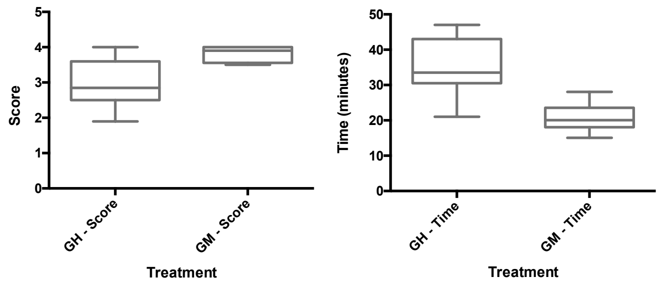


Fig. 13: Tukey boxplot of score and time of GitHub and GitMultipile

Since the difference is significant, the null-hypothesis is rejected, and we therefore conclude that GitMultipile performs better (both in score and in time) than the visualizations of GitHub at answering the questions we formulated.

Participants evaluated two Git repositories (mwt-ds-explore-java and mssql-jdbc) using two treatments (GitHub and GitMultipile). We compared the scores obtained by using the two repositories with each treatment. The two-tailed Mann-Whitney test applied to the scores obtained with GitHub results P = 0.5887, U = 14. The same test applied to the scores obtained with GitMultipile results P = 0.6104, U = 15. These two tests indicate that the two Git repositories we considered lead to similar results per treatment. We can therefore conclude that the two Git repositories are similar in their difficulty to analyze.

We compute the effect size: Cohen's d = 2.440 for the score and d = 2.342 for the time, indicating a large effect size.

We conclude the following:

- Participants graded significantly better using GitMultipile than using GitHub visualizations and navigation tools at answering our set of questions about software evolution.
- Participants completed the questionnaire significantly faster using GitMultipile than using GitHub visualizations.
- The two analyzed projects are similar in their difficultly to answer our set of questions. We therefore exclude the presence of unbalanced task difficulties, which could be assimilated as a plausible bias.

VIII. Observations

We have closely observed each participant activity in the second controlled experiment, the one involving GitHub and GitMultipile. We used two data collection techniques: the think-aloud protocol and user interactions. In the think-aloud protocol, we asked participants to verbalize their thoughts while answering their tasks. This protocol helps us understand developer activities and identify questions that participants were asking while trying to solve the tasks.

Questions. We identified 7 questions participants asked themselves during the experiment about GitHub and GitMultipile. Questions are given below and each is annotated with the number of occurrences during our experiment. We recall that 12 subjects participated in our experiment.

GitHub. We identified three questions that identify some limitations or surprising behavior of GitHub:

1) *"Why is the number of the contributors at the initial page different than the number of contributors shown in the graph page?"* [Occurrence = 6] Each GitHub project main page indicates the number of contributors, and time to time, this number differs when accessing the list of contributors. For example, https://github.com/Microsoft/EMDocs.hu-hu indicates that 11 contributors are part of the project. However, clicking the the graph tab lists 8 contributors. The fact that merge commits are not listed in the graph page was confusing for the participants.
2) *"The Graph tab is not contextual. For example, I am on the page of a folder of the project, and I press the Graph button, the metrics that are given are for the whole project, and not the folder as I expected."* [Oc. = 6] Participants were expected to have metrics per folder, and not for the whole project, even if nested folders are currently presented.
3) *"Why searching for a file gives me so many answers that I don't require?"* [Occurrence = 5] Consider Question Q1 about the file `Test.java`. Many participants have entered `Test.java` in the search text field of GitHub. However, GitHub returns the list of file having `Test.java` in their name (*e.g.,* `PRGTest.java`, `MurMurHash3Test.java`). Many participants could not use the search facility because GitHub does not solely return the exact match.

GitMultipile. Three questions were identified when participants used GitMultipile:

1) *"Can I highlight relations with a condition involving code comments?"* [Oc. = 5]
2) *"Can I directly see the code modified by a contributor on a file?"* [Oc. = 3]
3) *"Can I know how many lines of code were changed on a file?"* [Oc. = 3]
4) *"Can I modify the relations shown in the timeline?"* [Oc. = 3] Currently, the metric shown in the timeline is set and cannot be changed.

Post-experiment Feedback. After the experiments, we informally asked the opinion of each participant about GitMultipile. Here are some comments:

- *"GitMultipile is powerful for these kinds of questions while GitHub forced me to do a lot of manual searches"*
- *"GitMultiPile with `weight:` allowed me to reduce the commits to focus on, thus easing the searches. First, I focus on the timeline, which let me identify the stacks of matrices and on who is working. Secondly, I* zoom *in on the data by using the relevant matrices"*
- *"GitMultiPile is an interesting tool. I like the idea to stack matrices using a condition I defined. At first it was not clear to me the difference between using `weight:` and not using it. I used a lot the highlight facilities in the questionnaire".*

All the comments made by the participants focus on the functionalities offered by GitMultipile.

Results. A number of interesting facts can be deduced from observing our participants. First, some obvious limitations of GitHub were spotted. Half of our participants complained about (i) the mismatch of list of contributors, (ii) the Graph button giving a global analysis when a local one was expected, and (iii) the search facility is suboptimal.

Regarding GitMultipile, a notable result is to not have any questions about the meaning of the adjacency matrices and their piles. This is confirmed by the post-experiment feedback. Moreover, no participants questioned the interaction we provided. The GitMultipile language did not seem to be difficult to learn and no negative surprises were experienced by the participants. We can therefore conclude that GitMultipile is intuitive for the experiments we designed and for the subjects who participated. All but one comment is about the limitation of GitMultipile. Currently, GitMultipile does not operate directly on the application source code. This perceived limitations will shape our future work, as described in Section XI.

IX. Threats to Validity

Our experiments and results are subject to validity threats. Since such threats may be a source of false negatives and false positives, it is important to carefully identify possible threats and analyze how their impact may be mitigated.

Conclusion validity. Our conclusions are founded on two experiments for which their strong statistical results favor GitMultipile over Excel and GitHub. However, our conclusions are based on the result of only 20 participants, which is relatively low. Although we had no indication that increasing the number of participants may invalidate our result, the strength of the statistic results may be affected.

Internal validity. Our experiments shows strong evidences that GitMultiple significantly performs better and faster than Excel and GitHub to formulate queries and solve the software evolution tasks we have designed. We therefore conclude that the changes in the independent variables (the employed treatment and the tasks we designed) cause the observed changes in the dependent variables (score and time).

Construct validity. Can our results be generalized to other software evolution tasks? Unfortunately, we are not aware of any recognized standard benchmark for software evolution. Thus, it could be that GitHub would perform better than GitMultiple for a different set of tasks. We were careful to identify research questions that reflect software evolution tasks, based on existing work [1], [8], [10].

External validity. Threats to external validity are conditions that limit the generalization of our result to industrial practices. GitMultiple is a better solution than GitHub for the tasks we designed. GitHub offers tools to help address software evolution activity. We picked GitHub as the most representative baseline, although it is not primarily designed to support software evolution tasks. Thus, it could be that we wrongly picked GitHub. As discussed in Section V-B, we took care to pick GitHub as the most intuitive and natural choice.

Participants in the second experiment belong to the same company and have experience from having worked on a common codebase. Thus, it may be that their common knowledge could put in question the random heterogeneity of the participants. However, we could not see any hint supporting this threat.

X. Related Work

Software visualization techniques are commonly employed when analyzing software history. Our approach, which combines a domain-specific language with stacked matrices, is unique as far as we are aware of. This section summarizes the work in the field of software visualization.

A. Visualizations

In an attempt to classify software visualization research tools, we have produced a taxonomy (Table III). Our taxonomy follows the taxonomy proposed by Diehl [24], but adjusted to the field of visualizing software evolution. The two dimensions of the matrix are the abstraction layers of software systems (whole systems, repository activity, and code clone), and the associated phenomena to these layers. The remaining of this section details each reference.

Repograms [12] is a visualization technique to qualitatively compare and contrast software projects over time. Each project is represented as a horizontal bar, in which portions of that bar are colored to indicate values of a metric. Values can either be numerical (in that case the color saturation is linear to the metric values), or numeral (in that case, each item has a unique color, *e.g.,* designating code authorship by representing author names).

ClonEvol [23] represents evolution of code clones across multiple software versions. A radial tree and a variant of hierarchical bundle edges are combined to indicate source code clone relations. A dedicated color schema for the tree nodes and edges indicates differences from a baseline. The schema may indicate structural differences or variation in the activity.

CVSscan [21] is an integrated multiview environment in which each software version is represented by a column, and where a source code line is represented as a horizontal line. Metrics and software versions are represented using configurable color maps. One color map indicates textual modification: source code lines that have been added, modified, and deleted are represented in a distinct color. Another color map involves authorship. CVSscan may be seen as an improvement of SeeSoft [25].

Revision Tower [17] represents the activity of a control version system repository. It uses a vertical layout of boxes in which each box represents a historical information of a file contained in the repository.

3DSoftVis [19] is a 3D visualization tool in which each software release history is articulated over time, the software structure, and the module version numbers.

Multiple Visualization Strategies [13] emphasizes that software evolution may be analyzed in many different ways, *e.g.,* along the time dimension or structural dimensions. SourceMiner

	Static structure	Component logical coupling	Evolution of static structure	Source code	Metric evolution
Systems	[12]		[13]	[14] [10][15]	[16]
Activity Repository	[17]	[18]	[19]	[20] [21]	[22]
Code clones	[23]				

TABLE III: Taxonomy of software evolution visualizations

Evolution tool offers different visualizations, namely treemap, tree-like polymetric view, graph showing dependency, timeline matrix, to navigate through a software history.

History Slicing [14] is a scalable visual metaphor in which a file history is a horizontal time line. Any part of this time line can be zoomed in to reveal modification of the source code.

Evolution metrics [16] uses a Kiviat diagram with superposed data sets: each axis in the diagram presents a metrics and a data set corresponds to a particular system snapshot. This superposition of multiple revision lets patterns emerge in the system evolution.

Chronia [20] is a visualization in which each file is represented as a horizontal line. Time goes from left to right. Authorship is represented with a color. Chronia indicates authorship of commit during each file lifetime. Commits are indicated as a colored circle, and size of the commit is reflected in the size of the circle.

Spectographs [22] represents the evolution of software components on a particular property measurements. The visualization is a 2D chart in which the X-Axis represents software versions and the Y-Axis represents files. Each element of the chart indicates the spectrum (similar to the one used in sound decomposition) and is colored accordingly to reflect variation of metrics.

Evolution radar [18] visualizes the logical coupling of one module with the others. A selected module is placed at the center of a pie chart (similar to a "radar") and each sector represents a module the central module depends on. The size of each sector represents the size of each module as indicated with the number of files. Time intervals are represented using a user-defined position-color mapping.

CodeCity [10], [15] is a 3D visualization of the structure of a software system using a city metaphor. In a city, each building represents a class for which the height represents the number of methods of that class and the building top area represents the number of variables. Underneath flat squares, indicating a district, represent packages. CodeCity has evolved with two time lines represent structural evolution of a system [26]: a coarse-grained time in which each visualization represents a snapshot of a system in a given point in time, and a fine-grained time in which a visualization shows the evolution in time.

Dependency Structure Matrix is commonly used to represent evolution of dependencies between software components. Lattix [3] offers a navigable adjacency matrix. Enriched DSM [4] augments each matrix cell with information about the indicated dependency.

B. Domain specific languages

Boa [27] is a domain-specific language and infrastructure that eases mining software repositories. Boa runs over an "ultra-large dataset" and the language is designed in such a way that scripts are relatively shorts (a few lines long). Output are textual and a visualization tool is necessary to produce compact representations.

Feature-based DSL construction [28] has been proposed as a way to build DSL families. A DLS family is a series of DLSs having a commonality in their domain. Such encoding of DSL families is expressed using Alloy[12], a language to describe explorable structures.

Contrary of these domain specific languages to mine software repositories, GitMultipile's domain-specific language is designed to produce stacked matrices.

XI. Conclusion and Future Work

We designed a visualization framework and a domain-specific language to produce software visualizations from a Git repository. We performed two controlled experiments, involving 20 professional software engineers for nearly 20 hours. We made three findings: (i) the participants performed better at formulating queries using GitMultipile than using Excel, (ii) our participants performed better using GitMultipile than GitHub's visualizations to answer a set of questions on the history of two Git repositories, and (iii) we identified some oddities in the way GitHub behaves.

We conclude that GitMultipile represents a better alternative than GitHub's visualization to crawl and extract historical information.

As future work, we plan to expand our work as follows:

- Consider application source code. Currently, GitMultipile does not consider the actual source code commit.
- Making the timeline more flexible by considering user defined metrics.

These two aspects will address some of the situations our participants have faced during our experiment.

References

[1] L. Hattori, M. D'Ambros, M. Lanza, M. Lungu, Answering software evolution questions: An empirical evaluation, Information and Software Technology 55 (4) (2013) 755 – 775. doi:10.1016/j.infsof.2012.09.001.
URL http://www.sciencedirect.com/science/article/pii/S095058491200184X

[2] B. Bach, N. Henry-Riche, T. Dwyer, T. Madhyastha, J.-D. Fekete, T. Grabowski, Small multipiles: Piling time to explore temporal patterns in dynamic networks, Computer Graphics Forum 34 (3) (2015) 31–40. doi:10.1111/cgf.12615.
URL http://dx.doi.org/10.1111/cgf.12615

[12]http://alloy.mit.edu/alloy

[3] N. Sangal, E. Jordan, V. Sinha, D. Jackson, Using dependency models to manage complex software architecture, in: Proceedings of OOPSLA'05, 2005, pp. 167–176.

[4] J. Laval, S. Denier, S. Ducasse, A. Bergel, Identifying cycle causes with enriched dependency structural matrix, in: Proceedings of the 2009 16th Working Conference on Reverse Engineering, WCRE '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 113–122. doi:10.1109/WCRE.2009.11.
URL http://dx.doi.org/10.1109/WCRE.2009.11

[5] S. Rufiange, G. Melançon, Animatrix: A matrix-based visualization of software evolution, in: Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on, 2014, pp. 137–146. doi:10.1109/VISSOFT.2014.30.

[6] H. Borges, A. Hora, M. T. Valente, Understanding the factors that impact the popularity of GitHub repositories, in: Proceedings of the 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME), 2016, pp. 334–344.

[7] A. Chiş, T. Gîrba, O. Nierstrasz, A. Syrel, GTInspector: A moldable domain-aware object inspector, in: Proceedings of the Companion Publication of the 2015 ACM SIGPLAN Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH Companion 2015, ACM, New York, NY, USA, 2015, pp. 15–16. doi:10.1145/2814189.2814194.
URL http://scg.unibe.ch/archive/papers/Chis15b-GTInspector.pdf

[8] J. Sillito, G. C. Murphy, K. De Volder, Asking and answering questions during a programming change task, IEEE Trans. Softw. Eng. 34 (2008) 434–451. doi:10.1109/TSE.2008.26.
URL http://portal.acm.org/citation.cfm?id=1446226.1446241

[9] S. Easterbrook, J. Singer, M.-A. Storey, D. Damian, Selecting empirical methods for software engineering research, in: F. Shull, J. Singer, D. Sjoberg (Eds.), Guide to Advanced Empirical Software Engineering, Springer London, 2008, pp. 285–311. doi:10.1007/978-1-84800-044-5_11.
URL http://dx.doi.org/10.1007/978-1-84800-044-5_11

[10] R. Wettel, M. Lanza, R. Robbes, Software systems as cities: a controlled experiment, in: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, ACM, New York, NY, USA, 2011, pp. 551–560. doi:10.1145/1985793.1985868.
URL http://doi.acm.org/10.1145/1985793.1985868

[11] T. Schneider, Y. Tymchuk, R. Salgado, A. Bergel, Cuboidmatrix: Exploring dynamic structural connections in software components using space-time cube, in: 2016 IEEE Working Conference on Software Visualization (VISSOFT), 2016, pp. 116–125. doi:10.1109/VISSOFT.2016.17.

[12] D. Rozenberg, I. Beschastnikh, F. Kosmale, V. Poser, H. Becker, M. Palyart, G. C. Murphy, Comparing repositories visually with repograms, in: Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16, ACM, New York, NY, USA, 2016, pp. 109–120. doi:10.1145/2901739.2901768.
URL http://doi.acm.org/10.1145/2901739.2901768

[13] R. Novais, J. A. Santos, M. Mendonça, Experimentally assessing the combination of multiple visualization strategies for software evolution analysis, Journal of Systems and Software 128 (2017) 56–71. doi:10.1016/j.jss.2017.03.006.
URL http://www.sciencedirect.com/science/article/pii/S0164121217300572

[14] F. Servant, J. A. Jones, History slicing: Assisting code-evolution tasks, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, ACM, New York, NY, USA, 2012, pp. 43:1–43:11. doi:10.1145/2393596.2393646.
URL http://doi.acm.org/10.1145/2393596.2393646

[15] R. Wettel, M. Lanza, Visualizing software systems as cities, in: Proceedings of VISSOFT 2007 (4th IEEE International Workshop on Visualizing Software For Understanding and Analysis), 2007, pp. 92–99. doi:10.1109/VISSOF.2007.4290706.
URL http://dx.doi.org/10.1109/VISSOF.2007.4290706

[16] M. Pinzger, H. Gall, M. Fischer, M. Lanza, Visualizing multiple evolution metrics, in: Proceedings of SoftVis 2005 (2nd ACM Symposium on Software Visualization), St. Louis, Missouri, USA, 2005, pp. 67–75.

[17] C. Taylor, M. Munro, Revision towers, in: Proceedings 1st International Workshop on Visualizing Software for Understanding and Analysis, IEEE Computer Society, Los Alamitos CA, 2002, pp. 43–50.

[18] M. D'Ambros, M. Lanza, M. Lungu, The evolution radar: Integrating fine-grained and coarse-grained logical coupling information, in: Proceedings of MSR 2006 (3rd International Workshop on Mining Software Repositories), 2006, pp. 26 – 32.

[19] C. Riva, Visualizing software release histories with 3dsoftvis, in: Proceedings of the 22Nd International Conference on Software Engineering, ICSE '00, ACM, New York, NY, USA, 2000, pp. 789–. doi:10.1145/337180.337644.
URL http://doi.acm.org/10.1145/337180.337644

[20] T. Gîrba, A. Kuhn, M. Seeberger, S. Ducasse, How developers drive software evolution, in: Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2005), IEEE Computer Society Press, 2005, pp. 113–122. doi:10.1109/IWPSE.2005.21.
URL http://scg.unibe.ch/archive/papers/Girb05cOwnershipMap.pdf

[21] L. Voinea, A. Telea, J. J. van Wijk, CVSscan: visualization of code evolution, in: Proceedings of 2005 ACM Symposium on Software Visualization (Softviz 2005), St. Louis, Missouri, USA, 2005, pp. 47–56.

[22] J. Wu, R. Holt, A. Hassan, Exploring software evolution using spectrographs, in: Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004), IEEE Computer Society Press, Los Alamitos CA, 2004, pp. 80–89.

[23] A. Hanjalić, Clonevol: Visualizing software evolution with code clones, in: 2013 First IEEE Working Conference on Software Visualization (VISSOFT), 2013, pp. 1–4. doi:10.1109/VISSOFT.2013.6650525.

[24] S. Diehl, Software Visualization, Springer-Verlag, Berlin Heidelberg, 2007.

[25] S. G. Eick, J. L. Steffen, S. Eric E., Jr., SeeSoft—a tool for visualizing line oriented software statistics, IEEE Transactions on Software Engineering 18 (11) (1992) 957–968, depth.

[26] R. Wettel, M. Lanza, Visual exploration of large-scale system evolution, in: Proceedings of WCRE 2008 (15th IEEE Working Conference on Reverse Engineering), IEEE CS Press, 2008, pp. 219–228.

[27] R. Dyer, H. A. Nguyen, H. Rajan, T. N. Nguyen, Boa: A language and infrastructure for analyzing ultra-large-scale software repositories, in: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 422–431.
URL http://design.cs.iastate.edu/papers/ICSE-13/icse13.pdf

[28] C. Huang, Y. Kamei, K. Yamashita, N. Ubayashi, Using alloy to support feature-based dsl construction for mining software repositories, in: Proceedings of the 17th International Software Product Line Conference Co-located Workshops, SPLC '13 Workshops, ACM, New York, NY, USA, 2013, pp. 86–89. doi:10.1145/2499777.2500714.
URL http://doi.acm.org/10.1145/2499777.2500714

Appendix: GitMultiple Language

A. Program Structure & Language Constructions

A program written in the GitMultiple Language has to begin with instructions that (i) map a domain and co-domain to the matrices; (ii) set a time period per matrix; (iii) define the weights of each matrix cell; (iv) ordering, filtering, and highlighting instructions.

Our domain-specific languages features the following constructions:

- The domain S1 and co-domain S2 used in the matrices is defined using `mapFrom: S1 to: S2`. Values commonly given to S1 and S2 are `#authors`, `#files`, `#days`, `#months`, or `#hours`. These domains are exposed by the meta-model we used to represent the Git repository data. This construction is allowed only once in a script.
- Each matrix represents a time period, for which its granularity may be determined: `timeYear`, `timeMonth`, `timeHour`, `timeDay` define the time interval represented by one matrix. Only one keyword is allowed in a script (*i.e.,* all the matrices represents the same period of time).
- The weights of each cell is defined using `from: B1 to: B2 weight: B3`. B1 and B2 are two-args block predicates used to select the relevant commits. In case the domain are

authors and the co-domain are days, we can have B1 = `[ :commit :author | commit author = author ]` and B2 = `[ :commit :day | commit day = day ]` to define the weight of a matrix cell as the number of commits made by an author in a given day.
B3 is an one-arg block function that accepts a list of commits and a new list of commits. The function B3 may do some manipulations of the selected list of commits, for example, B3 = `[ :commits | commits select: #hasNoComment ]` selects commits without any comment. A variant of this construction is `from: B1 to: B2` for which the weight is the number of commits that matches B1 and B2.

- Each matrix cell has a popup text, activated when the mouse is located above the cell. The popup value depends on the domain and co-domain. The constructions `fromIdentifier: B1` and `toIdentifier: B2` build the popup using a textual description resulting from B1 and B2, both being an one-arg block function. A typical one-arg block function will project and manipulate some attributes of the domain and co-domain. For example, in case the domain are authors, B1 = `[ :author | author fullName , author email ]` will define a popup as the full author name concatenated with author's email.
- A particular author represented in matrices may be highlighted using `highlightAuthorName: N using: Col`. N represents the author name and Col a color.
- Matrices may be piled up using `pileIf: C`. Matrices that match the condition C are piled. The condition C is expressed as a predicate evaluated on a matrix, described below.
- Matrices representing time periods within the interval D1 and D2 may be piled using `pileFrom: D1 to: D2`. Time may be a year, a month, a day, or an hour and are specified using a dedicated syntax. For example:
  - `Year year: 2017` represents the year 2017
  - `Month month: 'January' year: 2017` represents the month January of the year 2017. A month index may be provided instead of the month name.
  - `Date year: 2017 month: 'March' day: 23` represents March 23, 2017.
- Matrices may be avoided to be piled using the construction `sequenceIf: C`. Matrices matching a condition C are not included in a pile. This construction is useful in case that matrices piled using `pileIf: C` should remain in sequence.
- Some authors may be filtered out from the visualization using `filterOutAuthor: C`, according to a particular condition C. This condition is expressed for a given author.
- Matrices and piles of matrices may be ordered using a layout. Several layouts are available, the commonly used are `layout horizontalLine` and `layout grid`.

B. Predicates

Many of the constructions given in the previous section requires a predicates. Our DSL offers a large set of constructions to build predicates.

- Piling matrices involves conditions, which may involve some dedicated predicates. A matrix offers various predicates, including `containsAuthorNamed: anAuthorName` indicating whether or not a matrix contains data for a particular author. Another predicate is `containsAnyAuthorMatchingFrom: authors` accepting a collection of author names.
- A commit offers some predicates. For example `containsCommentMatching: aContent` indicates whether a commit comment contains a text portion `aContent` . Some predicates are dedicated to files, for example, `containsFile: C` indicates whether a commit contains a file whose matches a particular condition C. For example, C = `[ :file | file fullName endsWith: '.java' ]` is a predicate that matches file whose name are ending with `.java`. The condition C may be arbitrary complex and be combined with other conditions using boolean operator (`&` and `|`).
Another predicate is `containsFileNamed: aFilename` indicates whether a commit directly modify a file with a specific name. Similarly, `containsAnyFileMatchingFrom: someFilenames` matches commits that contain a file that matches any of the set of names `someFilenames`.

C. DLS Program Input & Host Language

A script written using the GitMultiple language operates on a log file obtained from a Git repository. GitMultiple is expressed as a domain-specific language embedded in the Pharo programming language[13]. Pharo therefore acts as a host language. Many libraries supported by Pharo may therefore be used in GitMultiple scripts, including the collection and data manipulation facilities. As we employed in our example, the `select: P` construct filters a collection of elements using a predicate P.

[13]http://pharo.org