UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Turkhan Badalov

Software Analytics: Visualization of Source Code Evolution

Master's Thesis (30 ECTS)

Supervisor: Kristiina Rahkema, MSc
Cosupervisor: Dietmar Alfred Paul Kurt Pfahl, PhD

Tartu 2021

Software Analytics: Visualization of Source Code Evolution

Abstract:
Modern software projects are evolving with high pace. With such ever-growing software projects, reading and editing someone's code is unavoidable. Thus, it is important to have the means for analyzing source code evolution that would help understand the context of changes that have happened over time. The concept of visualization has been constantly used across many fields and industries to aid understanding and to convey one or another message. Nowadays, computer graphics with mature visualization tools and libraries present a good opportunity to build diagrams that could be used to understand the source code evolution. The goal of this thesis is to demonstrate new ways to aid source code evolution analysis by developing an open-source tool that visualizes changes made in the source code over time. While most of the available tools that visualize source code focus primarily on one state of the project, embracing the timeline in our visualization is the main difference between them.

The visualisation tool encompasses three views each concentrating on a different aspect of source code evolution analysis. Emphasising the evolution timeline in the visualisations makes it possible to detect classes and methods prone to changes. Another central point in the visualisations is the connectedness of methods and classes highlighting entities that have taken on too many responsibilities. All views contain interactive elements that provide additional information on demand.

Keywords:
source code evolution, visualization, analytics

CERCS: P170 – Computer science, numerical analysis, systems, control

Tarkvara Analüüs: Lähtekoodi Evolutsiooni Visualiseerimine

Lühikokkuvõte:

Tänapäevased tarkvara projektid arenevad suurel kiirusel. Selliste pidevalt kasvavate tarkvara projektide puhul on mõõdapääsmatu teiste arendajate poolt kirjutatud lähtekoodi lugemine ja muutmine. Sellepärast on tähtis, et oleks võimalik analüüsida lähtekoodi evolutsiooni, mis lihtsustaks muutuste konteksti mõistmist. Visualiseerimist kasutatakse erinevates valdkondades informatsiooni mõistmiseks. Tänapäeval pakub arvuti graafika oma välja arenenud visualiseerimistööriistade ja raamistikega võimaluse luua diagramme mida on võimalik kasutada lähtekoodi arengu paremaks mõistmiseks. Selle töö eesmärgiks on ehitada töörist, mis aitab uutel viisidel lähtekoodi arengut analüüsida visualiseerides lähtekoodi muutust ajas. Suurem osa olemasolevatest lähtekoodi visualiseerimis tööristadest keskenduvad ühele koodi seisule. Võttes visualiseerimisel keskpunktiks muutuste ajajoone on üheks põhiliseks erinevuseks olemasolevate tööriistadega. Visualiseerimistööriist koosneb kolmest vaatest, millest igaüks keskendub erinevale aspektile lähtekoodi evolutsiooni analüüsis. Tuues evolutsiooni ajajoone visualiseerimise

keskmesse võimaldab leida klasse ja meetodeid mis on altid muutuma. Teine keskne aspekt loodud tööriista vaadetes on meetodite ja klasside seotus mis toob esile objektid millele on antud liiga palju kohustusi. Kõik vaated on interaktiivsed ning võimaldavad küsides saada lisa informatsiooni.

Võtmesõnad:
lähtekoodi evolutsioon, visualiseerimine, analüüs

CERCS: P170 – Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaat juhtimisteooria)

Contents

1 Introduction

Using version control systems (VCS) to collaborate when developing a piece of software has been a common practice among developers for many years now. With the advent of methodologies like Agile and Extreme programming, VCSs like Git have become even more widely adopted in the development culture [1]. A VCS is suitable not only for collaboration but it is useful for keeping track of changes that the source code has undergone. First of all, it is useful for repair measures to restore previous working versions of a project in case of the deployment of a broken or a vulnerable version of the code. In addition, it is a database with answers to different questions about past and present of the project. While the source code of the project is an ultimate source of truth, it is not that helpful in telling how and why the software evolved as it is. To leave explanation or the intention behind a change, developers could use comments or commit messages that might also contain a unique issue number that is linked to an issue-tracking system. However, such ad-hoc comments in the code are generally treated as the sign of a bug or a code smell [TYKZ07]. There is even a saying broadly roaming in the internet – "Code never lies, comments sometimes do". Commit messages on the other side are not as concise as, for example, a difference between two versions of a file generated by "diff" command. Even if they might contain a unique ID that links an issue to the commit, one has to read all of the issue description and yet still miss the point what part of the code was changed. In addition, it becomes harder to maintain a mental picture of several changes as one keeps reading just description of changes or even commit contents themselves.

All in all, we can see that the intention behind leaving comments and meaningful commit messages as well as maintaining the change log is to preserve the context of changes that could be used to understand the code better. To briefly mention, it has been known that developers spend a huge part of their time reading the code. In 1984, this chunk was about 50% of a developer's time [Sta84] while recent researches report that this number grew up to 70% [MML15]. In addition, most of the development happens in maintenance phase as opposed to initial phase of the development [Leh80] which means developers usually deal with code written by someone else, hence reading someone's code is inevitable as well as all of the challenges inherent to understanding the code.

There are a lot of tools to help understand code. Some tools allow to constantly analyze source code to prevent quality degradation which improves readability and maintainability in the long term, e.g. style checks [2] or detecting code smells and potential vulnerabilities [3]. Other tools are used for debugging that could also collect run-time

[1] There are examples of other communities using Git for collaborative work as well. The book of "Homotopy Type Theory" (HoTT) is a good example of such successful projects [LK15, PTDH19]

[2] CheckStyle - `https://checkstyle.sourceforge.io/`, accessed on 14/05/2021

[3] SonarSource - `https://rules.sonarsource.com/java/type/Code%20Smell`, accessed on 14/05/2021

statistics like memory consumption, number of parallel threads and track low-level operations, e.g. Java profilers. IDEs facilitate navigation through source code. They provide features like "Find usages" for methods and classes that facilitate the reading proccess. There are also tools that present source code information in a visualized way to simplify comprehension of a specific aspect of the code.

In this thesis, we combine visualization and software evolution to explore new ways of making sense of source code evolution. Contributions of the thesis can be summarized as follows:

- It proposes a novel way of visualizing software evolution enhanced by interactive elements
- It describes used techniques and preparation work to build the diagrams
- The project is available on Github:
  `https://github.com/turok1997/CodeEvolutionVisualizer`,
  and deployed to `https://code-evolution-visualizer.herokuapp.com`.

2 Background

In Section 2.1, we briefly discuss history and motivation of software evolution and how source code evolution relates to it. Section 2.2 describes general understanding of visualization concept and techniques used to draw complex shapes. Section 2.3 talks about available software visualization tools and the way software can be visualized for different purposes without focusing on evolution aspect of software visualization. Finally, the latter is discussed in Section 2.4.

2.1 Source code evolution

Software evolution field is relatively young but mature enough to be called as a separate discipline which could be considered as a branch of larger discipline of software engineering. Its roots go up to 1956 when Benington laid a foundation for that subsequently became known as "Water-fall model" (formalized by Royce [Roy87]) based on experience of development of SAGE air-defense system (published years later in 1983). This was the first step towards understanding that selecting the right development process is vital for building reliable and cost-efficient large production systems. Those usually involved thousands of developers to write and test new features and maintain the system in the environment where "you seldom modified another person's program - you wrote your own" [Ben83]. In 1969 when Lehman, who is known as the "Father of Software Evolution" in modern days [CDR+11, Mad02], while totally unaware of Benington's aforementioned publication at that time [Leh87] released a paper named "The programming process" (also published years later in 1985). The paper was based on studies of IBM OS/360 to understand the needs for software maintenance [LB85] where he also discusses needs for better processes to develop robust systems that solve "real-world problems". Later he classifies this type of systems as E-type systems in his SPE classification of software - the type of systems that is prone to change as opposed to systems written based on a formal specification. E-type programs turn into the core focus of software evolution because such systems are exactly the place where an evolution happens. After that, it is not surprising that many more development models like V-Model, Spiral and RUP (Rational Unified Process) appeared with an attempt to bring innovation in development processes and solve known drawbacks of Waterfall model [AGW08]. With the advent of Agile methodologies, development processes shifted towards collaborative work with fast feedback loops where developers read and modify each other's code as opposed to how Benington described the processes at his time. This increased the speed of development but at the same time it brought other problems associated with fast speed like continuous deployment, frequent releases, fast iterations where all of them require quicker and easier understanding of code. That is why it is important to study source code evolution which is one of the common aspects that are studied by researchers of software evolution field.

In general, software evolution as a discipline encompasses different aspects of software and not only source code as one might expect. Development methodologies, analytic measurements and other assets like documentation and source code constitute objects of studies in this field. The root motivation behind these studies, in addition to the pure interest of understanding the evolution per se, is to reduce expenditure on software development. The common phenomenon that is in the heart of these studies is "change". It is incorporated into one of the Lehman's laws which is "Continuing Change". Software is prone to change and it needs to be continuously updated to stay relevant. This again implies costs attached to maintenance. As it was mentioned before, most of the developer's time goes to understanding the source code. So, it is reasonable to think that by facilitating comprehension of the code, we can reduce costs spent on maintenance of a software project. This thesis focuses on understanding a few aspects of the project based on the changes that the source code has undergone.

2.2 Visualization

General understanding Good visualization can explain a lot. There is a saying that "a picture is worth a thousand words" which is even more relevant when speaking of charts and diagrams that represent a data set. Data visualizations have intrinsic function to aid understanding and simplify comprehension by aggregating, often huge amount of, data into different visual shapes. They communicate a message hidden in the data and allow to get answers at a glance. Different visualizations are designed to answer different questions meaning that each visualization comes with its own set of questions that it is able to answer. Hence, it is important to select right visualizations for the right problems as visualizations can be well used to support a decision-making process [CB92, SVV03], or for education purposes [HRE03] and of course software development or just software in general. It is also important to realize that quality of the conveyed message first of all depends on the quality of underlying data – no visualization is much of an aid if the data is not complete to answer a question or even worse if the data is incorrect. On the other side, one should not assume that benefits of visualization come from oversimplification of data. Rather, benefits are achieved through conciseness that the visualization provides, thus reducing cognitive effort required to interpret the information. That is one of the principles that were used when designing visualizations for this thesis – being concise has nothing to do with oversimplification [Cai16]. Section 3.1 demonstrates a good example of simplifying the visualization yet efficiently conveying the original message.

About drawing complex shapes. Visualization shapes vary in their complexity starting from primitive geometrical shapes like lines, circles, polygons to popular symbols like a heart symbol or even a whole human face. There are programs with a rich collection of both common and non-regular shapes including emojis, different kinds of arrows to

help users build powerful visualizations. For example, Power Point has a wide gallery of icons as well as Adobe Photoshop with numerous forms of brushes and shapes varying in their complexity. Nevertheless, it is obvious that they just physically cannot provide all possible shapes due to the inconceivable variety in forms and sizes of shapes. Though the question still persists: how do we draw complex shapes?

The question is important to anyone who is building visualizations. With the wide diversity of approaches and techniques to draw complex shapes, we can observe that all of them boil down to the concept of composing complex shapes out of simpler ones. At least because the only thing one needs, in principle, is a line which is the most primitive shape. After all, every differentiable function resembles a straight line at any of its points if zoomed sufficiently, which is referred to as the tangent line of a function at that point. Some people took this concept further into practice and that is why in the internet it is possible to find plenty of examples with pictures drawn by mathematical functions, e.g. Homer Simpson drawn by functions (Figure 1) or even more complex and impressive ones drawn for the "Desmos Global Math Art Contest" competition [4].

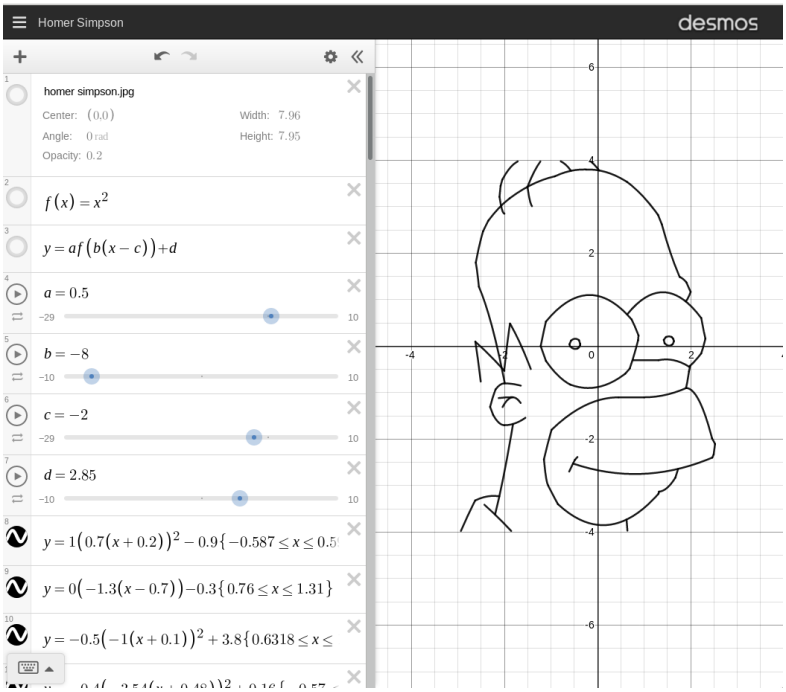

Figure 1. Homer Simpson drawn using mathematical functions by an anonymous user – `https://www.desmos.com/calculator/facoeuy4uk`, accessed on 14/05/2021

Another approach could be the composition of basic shapes with color manipulations

[4] Art competition by Desmos – `https://www.desmos.com/art`, accessed on 14/05/2021

to build a solid object. This technique is similar to, or perhaps a primitive form of, the collage technique which assembles different pieces to produce a new visual. The idea is to split the desired shape into regular shapes on hands and color them to vanish borders. For example, to draw a heart symbol we could use two tangent circles of the same radius, a triangle with two of its sides beings tangent lines of the circles, and then color them into the same color. We can also put a small rectangle in the middle to fill a tiny gap left between circles and the triangle. Figure 2 demonstrates this idea with raw shapes drawn on the left and colored shapes with vanished borders drawn on the right.

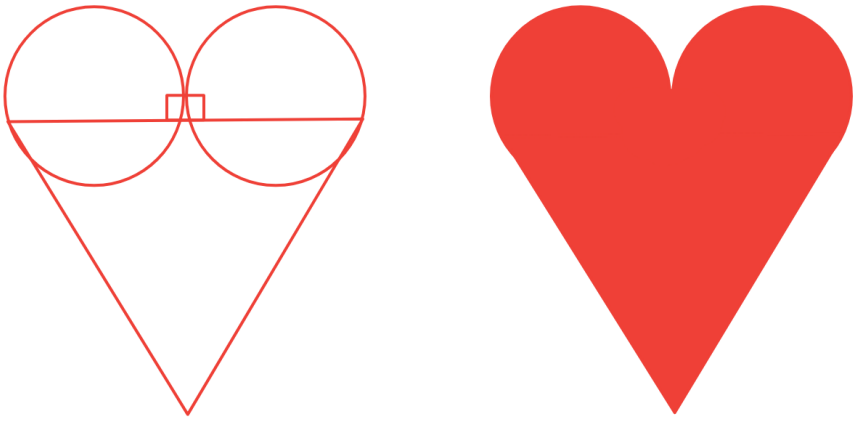

Figure 2. Heart shape composed of circles, a triangle and a rectangle drawn by eye

The latter technique is heavily used in this project to draw complex shapes as we will see later in the thesis. All of the techniques described above can be united under the "Divide and conquer" phrase – in order to conquer a complex shape, one can divide it into primitive shapes and weld them together by proper positioning and usage of colors.

2.3 Software visualization

Software visualization is a common concept utilized in the software development industry. Although, according to a systematic literature review conducted in 2016, the primary users of the visualizations tools are developers, needs of others groups such as project managers, testers and software architects are addressed as well [MIK+16]. They can use visualization to make various decisions, e.g. in the thesis "Software Runtime Data: Visualization and Integration with Development Data – A Case Study" Aytaj describes a possible scenario how a manager can decide to allocate time to work on a feature when noticing frequent crashes of an application in a dashboard after the feature had been released [Agh19]. A similar assumption about the target audience was made in a

systematic mapping study of software evolution visualization published in 2013 despite it was reported that most of the studied papers didn't explicitly specify audience for their tools [NTM+13]. The study also suggests two dimensions into which software evolution visualization efforts could be categorized based on their goals: the ones that focus on computer graphics issues and the ones that focus on helping software engineers in one or another aspect of their work. However, in our project we found that these two dimensions are intertwined with each other – not always clean and elegant representations of data help to reach a particular goal, e.g. 3D diagrams are often not very helpful when knowing precise values is important as it is described in the section "Method" , nor it is useful to build a diagram without thinking of and understanding how to solve graphics issues first, e.g. it is important to use interactive elements and provide information on-demand to reduce the level of noise instead of trying to depict all data at once.

Not all visualizations tools are built and live within boundaries of academic interest solely. There are commercial solutions that are based on visualization and companies selling visualization products. That being said, Structure 101 [5] is a visualization tool that draws a structure of an application allowing to examine, group dependencies between packages or modules and compare changes in the structure between two revisions of the application. According to testimonials on the website, architects and consultants find the tool useful because they can quickly get an overview of the architecture of an application and identify problems like cyclic dependencies between packages at one glance. Another tool, Understand by SciTools [6], is advertised as a visualization tool that can calculate and visualize metrics. It can also draw control flow graphs, declaration graphs and many other graph-based diagrams. Compared to the previous tool, Understand can help to comprehend more granular details of implementation unlike Structure 101 which is designed to understand the architecture of an application better without going into implementation details. One more tool that is commercially built is Sonargraph by Hello2morrow [7]. Quick look at a feature list tells that this tool is not designed primarily for visualization as it focuses more on enforcing quality models by duplicate code analysis, configuring basic and advanced metrics thresholds and so on. However it can also draw dependency class-level dependency graphs where clicking on a line connecting two nodes (classes) will lead a user to the exact place in code where the dependency is used. The common attribute between aforementioned applications is that they are all integrated with one or more IDEs like Eclipse, IntelliJ IDEA and so on. Perhaps, it is the path that tools emerged in academia should take in order to shift to the market and start monetizing itself.

Another family of visualization tools is used to to visualize algorithms including data-structures. These tools are heavily used in education. Although the aforementioned

[5] `https://structure101.com/`, accessed on 14/05/2021

[6] `https://www.scitools.com/`, accessed on 14/05/2021

[7] `https://www.hello2morrow.com/products/sonargraph`, accessed on 14/05/2021

systematic mapping study proposes dimensions for specifically software evolution visualizations, the field of education could also be considered as one more dimensions for software visualizations in general.

As a conclusion, despite the variety of visualization tools, many of them target analysis of one revision of an application or provide a basic comparison between two versions of the application. The goal of our project differs in a way that it aims to tell more how the application changed over time as opposed to how it is structured in the present. Tools lying in the spectrum of evolution visualization are described in the "Related work" section.

2.4 Software evolution visualization

The concept of visualization to facilitate understanding of software evolution has been an active topic of discussions and experiments. Existence of many research papers available online starting from recent studies to ones dating back to 90s [GJR99] say that. Even to this day new ideas keep appearing touching different aspects of software evolution visualization. Rapid progress of available technology for building visualizations in recent decade may have contributed to this. For instance, Tulip visualization framework [Aub04] has been used to visualize source code information as graph models ([THER09], [CAT07], [TA08]), D3.js javascript library for building interactive data-driven diagrams has been a popular choice among researchers ([APBG19], [dJSSRdSC18]) and many more. Merino et al. selected 86 papers that apply visualization to relieve concerns in software development with years ranging in 2002-2016 and mapped them to technologies that they used to produce the visualizations [MGN18].

Ideas for visualization of software evolution differ from each other trying to solve different problems. Code duplication visualizations aim to represent clones scattered across source code files of a project and incentivize to take actions for restructuring and refactoring the code, e.g. CloneEvol [Han13] and CCEvovis tools [HTY$^+$19]. The evolutionary side of them allows to see how much clones grew in the project and track if there is any effort put into reducing their amount. As such, code duplicates are usually considered as code smells. Another idea was proposed by Chevalier et al. in 2007, who described a visualization technique for structural analysis of source code traceable across revisions [CAT07]. A year later the idea was improved by Telea and Auber, co-authors of the original paper, by replacing visual tubes with splines that connect same code fragments and presenting a structure tracking technique that allows separation of (near) constant code fragments from actively changed parts, which gave the birth to CodeFlows tool [TA08].

Not every source code is compiled into an "invisible" program running in background. For example, shaders are programs used in computer graphics rendering and are designed to run on GPU. With each change the program might produce different visual output. Vis-a-Vis tool allows developers to run several versions of the program in parallel and

compare differences in the visual output rendered by the program, helping developers to see evolution of their work and be able to compare results - kind of a visualization for visualization [BB19]. Although aforementioned tools still apply to source code of shaders, e.g. you might want to find code duplicates among your shaders, they are not designed to support intrinsic goal of visualization algorithms.

3 System and diagram design

This section describes heuristics behind building diagrams. It backtracks the thought process for deciding which diagrams to implement. Also, we discuss the application architecture.

3.1 Diagram design

Selecting the right diagram requires answers to many questions. To mention the least, examples are "who is the audience", "what message should the diagram convey", "how detailed should it be and what information can be neglected". Not only selecting but designing new diagrams also requires answering those questions, however it also has an artistic side of it to create something different than what has been done before. In order to build diagrams for the thesis project, there were two main aspects - creativity and functionality of the diagrams. For each of the diagrams we will discuss both aspects together with the methods that were used to get inspired and that led us to final decisions.

Design techniques and heuristics. Designing a diagram needs trials and errors before getting it right. Usually there is some sort of heuristics involved that guides in which direction to go. By "heuristic technique" we mean a non-formal approach that is mostly based on intuition and previous experience which is still useful enough to guide thinking process, quickly reject bad ideas and focus on good ones. There were 2 main and clearly distinguishable heuristic techniques that helped in the process of designing diagrams. They are described in the following 2 paragraphs: "Data-oriented design" and "Problem-oriented design".

Data-oriented design. The idea behind this approach was to find somehow interesting to visualize variables and, perhaps not immediately obvious and not always explicit, relationships between them that have not been used in visualizations before or at least are not common among already available visualizations of source code. The next step was to map the variables and their relationships to a problem or a question which the visualization based on this data could provide answers for. If we stretch an analogy, then the we can paraphrase the technique as the following question: given a statement **A**, what is the set of questions **B** so that **A** is an answer to its elements? Then we pick one or more questions that could relate to any meaningful use case. For example, let's assume that we have a list of all method calls in some project. Also, any list carries additional intrinsic information like its size that could also be reflected in visualization. Now, the next step would be to find questions that described data could help to find answers for:

- How many classes depend on a particular class? Just by counting unique class names whose methods use a particular class to make the calls is enough to answer this question. So, the data we have provides an answer to this question.

- What method is called the most?
- How many times is each method called?
- etc.

Similar approach was taken in Evo-Clocks visualization [APBG19] where the design of visualization preceded its goal definitions, i.e. they describe the concept of their visualization but don't define practical use cases for the diagram. The idea was rather to discover strong and weak sides of the visualization itself and discuss possible types of questions that it could answer.

The next and the final step was to find usage scenarios that would depend fully or partially depend on answering some of the found questions. Although it might sound that the usefulness of the diagram has a low priority in this technique, it is not truly the case. Rather, the goal was to remove any possible obstacles during brain-storming process to get as many as possible visualization options and, obviously, eventually select the most useful ones if any found.

Problem-oriented design. This technique uses a reverse approach - we focus on the problem that we would like to solve first and then extract necessary variables that would reveal the answer. After that, we check whether we have those variables in our hand and think of a diagram that could visualize those variables in a meaningful way. For example, the "Call volume" diagram of this project was derived from the problem of finding a class that potentially violates the single-responsibility principle of OOP – the larger is the amount of calls to different methods, the higher chances that the class is a "God" object.

Diagram dimension. When designing a diagram, there is always a question of how many dimensions to use in the diagram. Should it be 2D or 3D? When recalling how often we see 2D diagrams and how widely they have been applied, 3D diagrams tend to seem as slightly exotic and somehow more appealing to the eye. Particularly, in software evolution field there is always a dimension that is inherent to its studies - timeline. Regardless of what we design, all the time we have to take into account timeline and incorporate it into our diagrams in one or another way, should it be solved by animation or a smart diagram structure – timeline is always there. Thus, it is intuitive to choose 3D first as then we would have 2 dimensions in our disposal and allocate 3rd dimension for timeline. However, not all is that straightforward with 3D diagrams. First of all, 3D diagrams tend to deceive. If we take a look at the Figure 3 where the data is artificially generated for demonstrative purposes, it might seem like a value for Class 1 at commit 1 is lower than than 160 (Figure 4).

However, if we project values of the Class 1 on a 2D bar chart, we can see that what seemed to be lower than 160 is exactly equal to 160. It is depicted in the figure 5. It happens because of the perspective effect of 3-dimensional graphics – the camera is

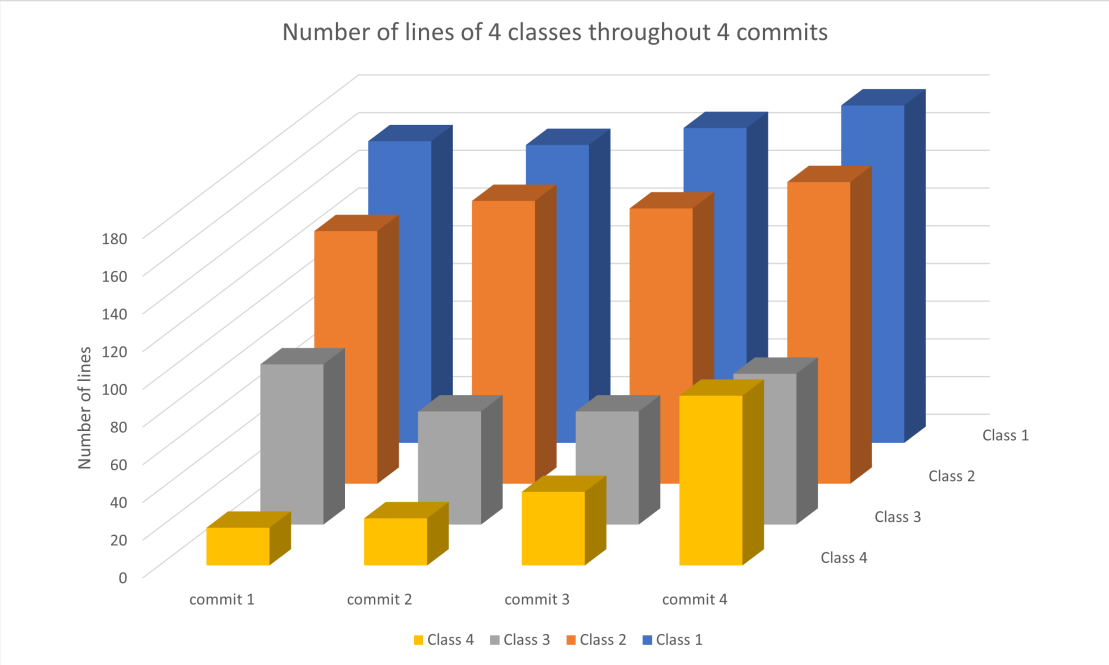

Figure 3. Example of a 3D bar chart

higher than the diagram. In addition, the 2D chart allows us easily compare values at each commit as opposed to the 3D version of the chart where it is not fully clear that the value of the Class 1 at the second commit is slightly lower than the value at the first commit. The better version of the 3D chart could be the 2D bar chart shown in the Figure 6.

Now, we can clearly compare values of every class at any commit. Nevertheless, we can still do better. The drawback of this diagram is that it depicts values at each revision like a snapshot and is not very helpful to see trends like how values for a particular class are changing. Slightly modified version of this diagram then would be as drawn in the Figure 7.

We can still improve the diagram if we pay attention that the width of bars is not really playing any role. Instead it is simply taking space in the diagram without providing value. The Figure 8 shows a version of this diagram that possesses the same features: it is possible to see precise values of each class at each commit and it is possible to follow trends for each class. In addition, we can compare trends of classes with each other while it was not very comfortable in the previous version of the diagram. For instance, we can see how the class 4 overgrew the class 3 in the last commit while classes 1 and 2 increased almost equally in the last commit.

Although, technically the trend chart is not very suitable for discrete data, because,

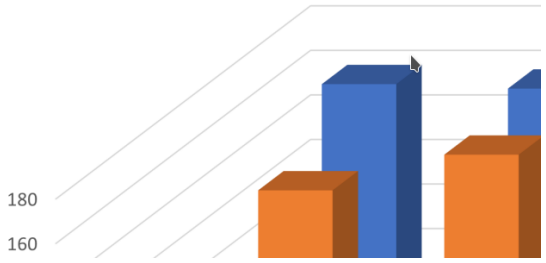

Figure 4. Effect of perspective in the 3D diagram

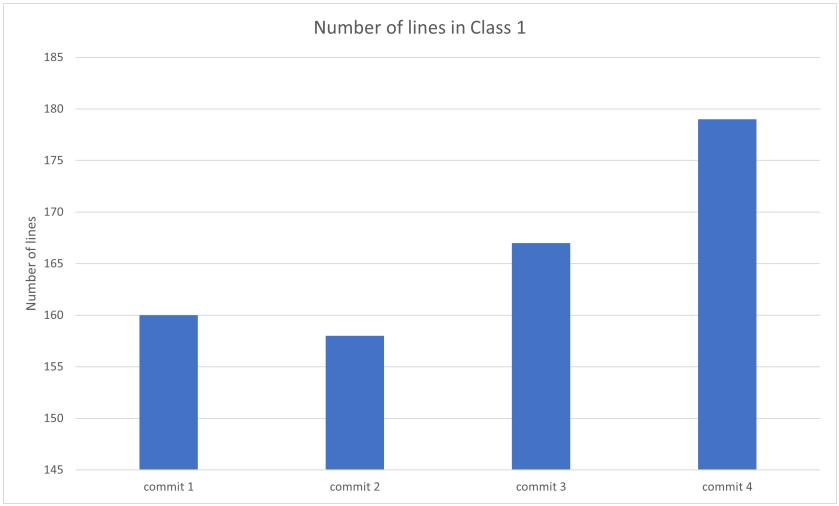

Figure 5. Projection of the values of the Class 1 onto a 2D chart

for example, values on a line between points do not make any sense for the number of source code lines, it can still be useful if we assume that lines carry no more information than just visually guiding the trends.

In addition, the fact that 2D diagrams require simpler implementation compared to the 3D equivalent advocates for using two dimensions only. Good example of 2D yet powerful visualization of a software project history is Gource [Cau10] [8].

As a conclusion, we observed how it was possible to reduce the 3D diagram into a simple trend chart. We got rid of one dimension and we used one-dimensional shape – line as opposed to a rectangle which is a two-dimensional shape. Finally, we provided value by allowing the reader to see precise values and compare those values across all commits and classes. Hence, 2D diagrams were also sufficient for visualizations in this

[8] Gource in Bloom Youtube video - `https://www.youtube.com/watch?v=NjUuAuBcoqs`, accessed on 14/05/2021

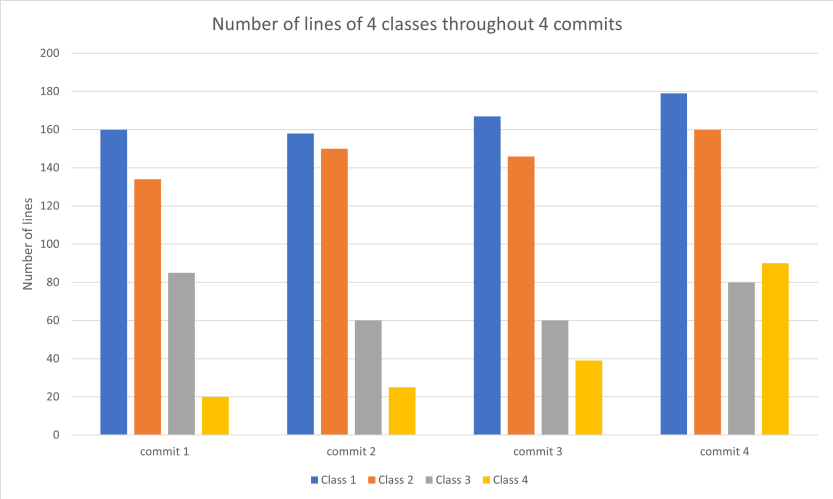

Figure 6. 3D bar chart redrawn as a 2D bar chart

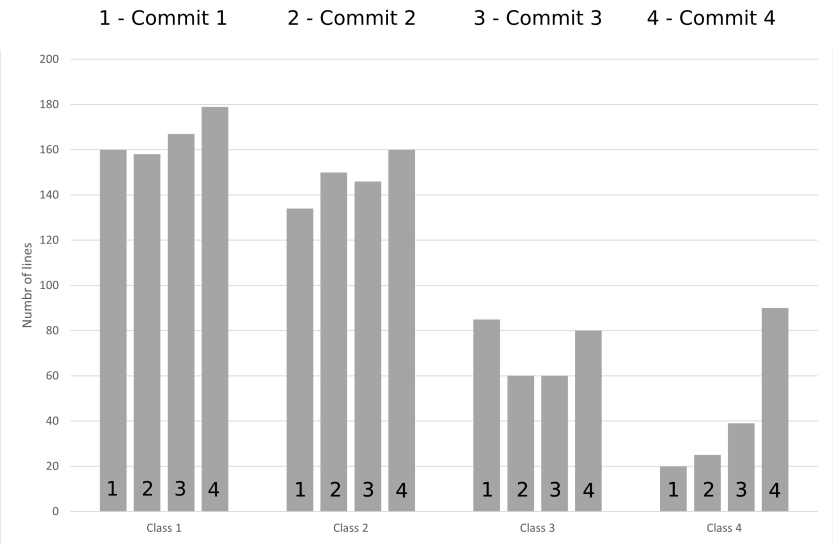

Figure 7. 2D bar chart that shows trends for each class

project. The charts in this section were drawn using Excel. As a side note, Excel has a button "Recommended charts" where it displays what kind of a chart would be suitable for the data. What is interesting is that even Excel did not suggest a 3D chart for the same data used to draw bar chart diagrams for this section. Finally, this and other techniques of optimizing charts are well described in the book "The Truthful Art: Data, Charts, and Maps for Communication" [Cai16] that helped a lot to understand different visualization

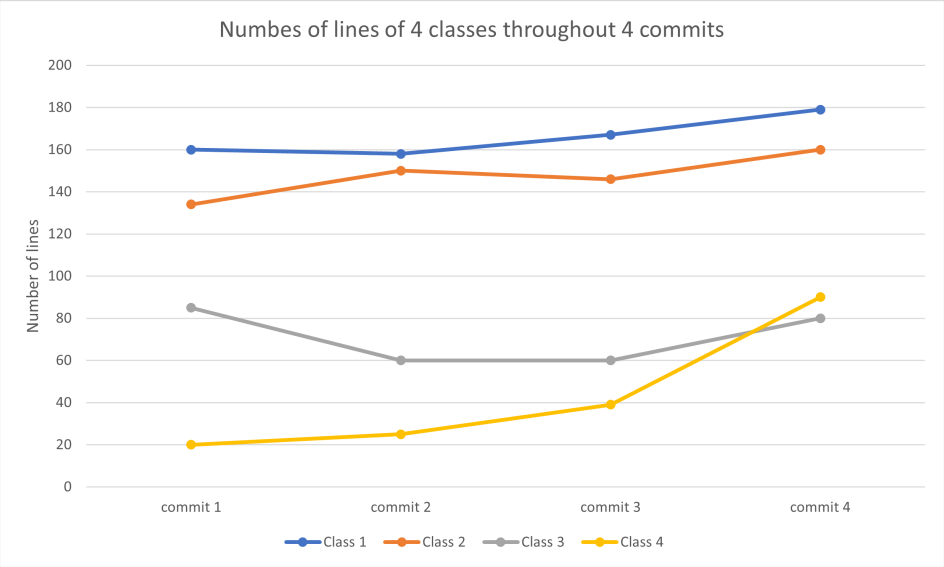

Figure 8. Trend chart redrawn from the 3D diagram

techniques even better.

3.2 Application architecture

The project has an application structure common to modern applications. It consists of loosely coupled back-end and front-end tiers that are located in the same git repository for the sake of simplicity and ease of development process. However, they could be easily separated into two repositories as well.

Back-end. The part of the system that provides APIs to diagrams to obtain underlying data is referred to as a back-end part. Neo4J, a no-sql graph database, and Express.js, a framework running on top of Node.js, constitute the technical stack of the back-end. The decision to use Node.js is rather simple:

- The project by itself does not perform heavy calculations that would require, for instance, parallelization. So, Node.js accompanied by lightweight web framework Express.js looked like the right fit.
- Javascript is the language used to build applications running in a browser, i.e. front-end part. Using the same language adds to the consistency in the project.

The underlying data of repositories is stored in Neo4J database. It is a powerful graph database that uses Cypher query language. It is produced by GraphifyEvolution which

is a modular application that can be used to analyse source code histories [RP21]. The application produces the graph representation of a software project defined in terms of nodes like classes and methods; and relationships such as "CLASS_OWNS_METHOD" or "CALLS" to indicate calls between methods and so on. Each commit points to the next one by the "CHANGED_TO" relationship. Thus, it was a straightforward decision to re-use this data for our project. In addition, pattern-matching capabilities of Neo4J are perfect for analysis of graph-based phenomena like source code evolution on top of version control systems. In and of itself, the primary goal of the back-end is to provide necessary data for drawing diagrams.

Front-end. The front-end is built as a single-page web application. There are several reasons that advocate this approach. The main reason is that it provides a great way to loosen coupling between back-end and front-end systems. If it was not built as a single-page application, it would have been implemented as multiple separate html pages or views as it is referred to by the MVC which would depend on the server to handle their state between transitions from one view to another. In fact, its configuration setup allows us to have the server and the front-end running in separate parts of the world and even more – each single endpoint consumed by the front-end could be actually a different instance of the back-end. To give a better understanding, listing 2 presents a configuration file that defines API url for each diagram, meaning that we can replace any of them without affecting the work of other diagrams.

```
module.exports = {
  commitRangeView: {
    apiUrl: '/commit_range_data',
  },
  classOverviewView: {
    apiUrl: '/class_overview',
  },
  callVolumeView: {
    apiUrl: '/call_volume',
  },
};
```

Listing 1. Diagram url configuration

URLs starting from forward slash indicate the endpoint is located relative to the root of the domain. We use this configuration by default since it points to localhost when running locally and does not initiate cross-origin requests when running in cloud. It aids to keep development environment and deployment process simple. However, the natural question might appear - if the domain that we are using to access the front-end is the same for API calls, how do we run both the back-end and front-end under the same

domain given that they are built as 2 independent projects? After all, it is not possible to run two application listening the same port. One solution could be to put a load balancer in front of the front-end and back-end, and based on the request path decide where to forward the request to. But there is a simpler solution that does not require external infrastructure setup to operate properly. As its name states, single-page applications is a single rendered page that does not require a browser reloading to switch to another view. All user interface updates and interactions are realized through asynchronous calls using Javascript. Then we could ask the server to render only one page and attach a Javascript bundle containing all of the logic of the front-end. After that, the server could run listening the port 80 and serve API calls as well as the requests targeted to the root "/" by returning a single page with Javascript bundle attached to its header. The rest is a series of communication between browser of a client and the server. It is exactly how the application is bundled for deployment. To support this behavior, we use Webpack bundler. Webpack bundles the whole front-end into one Javascript file and puts it to the "public" directory of express.js from where assets like images, CSS and JS files are served, e.g. "public/javascripts/frontend.bundle.js". In addition to that, Webpack creates an index page that has a reference to the bundle, as it is displayed below, so that a browser can download and run it:

```
1 <script src="/javascripts/frontend.bundle.js"></script>
```

Listing 2. Script tag in the index page

Obviously, we add these 2 files "frontend.bundle.js" and "index.ejs" to .gitignore to avoid committing them to the repository because they are byproducts of building the project, same as binaries of compiled projects.

To avoid low-level DOM operations in the code and leverage unidirectional data flow, React is selected as a front-end framework to enable single-page architecture.

4 Implementation

4.1 General view structure

The application is divided into multiple views. Particularly, there are 3 views that constitute the application – "Commit range", "Class overview" and "Call volume" views. We will examine each of them separately in the following sections. In spite of the differences between their functionalities and design goals, they share common characteristics like organization of user interface and its elements.

Initially, views consisted of only one canvas taking all of the available space in a browser window. Subsequent trials quickly made us realize that such a user interface is lacking of control elements at least as basic as filtering ones to allow users to control what information they want to analyze and what to focus on. Hence, we split each **view** into two parts to rectify the need for control elements - left part containing **menu items** relevant to the current view and the right part is a working area to display visualization that we refer to as a **diagram**.

4.2 Commit range view

What is it? "Commit range" view is the first view that users see when they open the application in a browser. The main role of the view is to be the entry point to the application and provide sufficient and meaningful means to switch to other and more specific diagrams. In addition, it presents a general overview of the project and a possibility to switch between applications that are available in the database.

The diagram drawn in this view is a stacked bar chart where sub-bars represent a modified class. Since each class is assigned a unique color, sub-bars have the color of the class that they are representing. The height of a sub-bar is defined by the number of changed lines in the class. Hence, the height of a full bar is the sum of heights of all stacked sub-bars which is equivalent to the number of all changed lines in the edited classes within that commit. A changed line is defined as either an added, a removed or an edited line. This number is calculated by GraphifyEvolution and is available in the database. Each bar is associated with a commit. Commit hashes play a role of the timeline as they are sorted by their author date. Figure 9 demonstrates the view. The following paragraphs describe the ways to interact with the diagram.

Switching between available applications. One of the control elements displayed at the top of a menu is a drop-down list (Figure 10). It displays the name of a software project that is currently examined in the application. If the Neo4J database contains multiple applications, then a user can switch between those by selecting the right one in the drop-down list. The application will reload the view dynamically without refreshing the browser page.

Content filter. As its name says, these elements allow filtering by the type of content

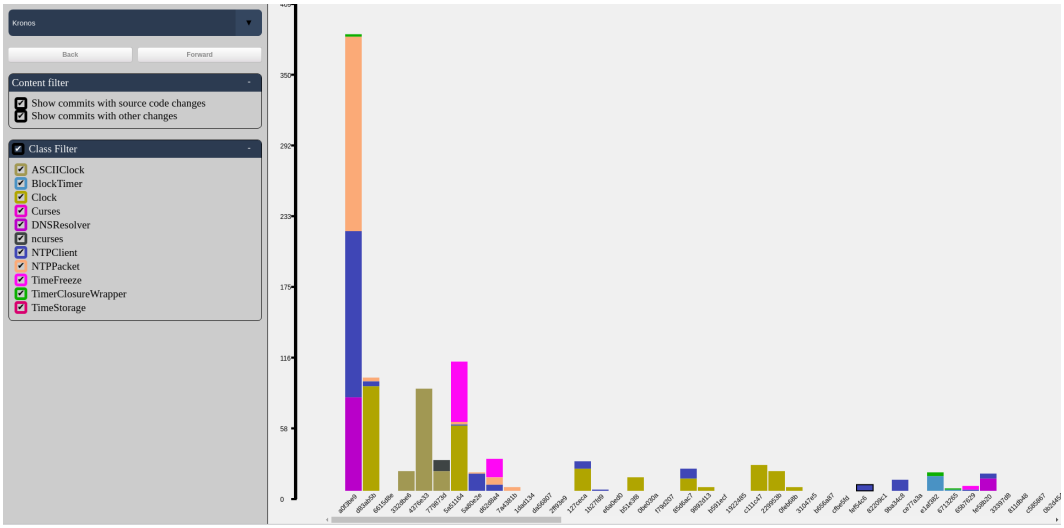

Figure 9. "Commit range" view

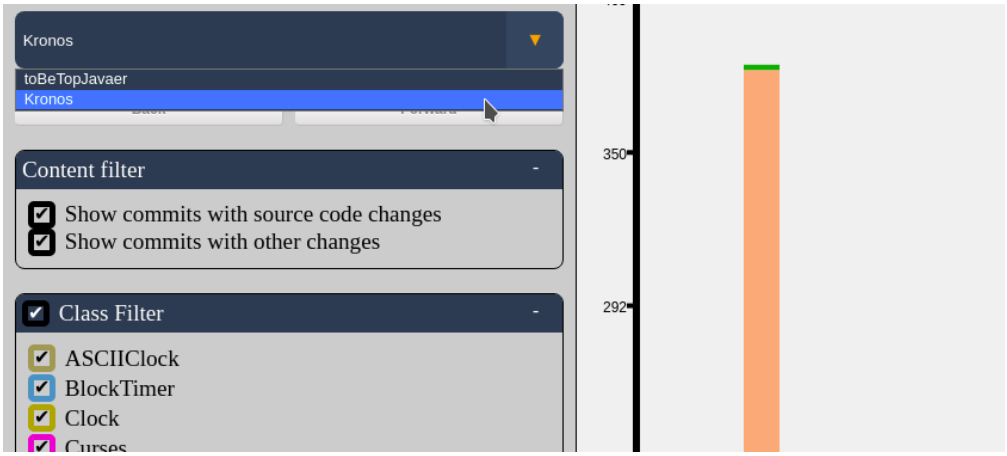

Figure 10. Drop-down list to switch between available applications

of commits. We distinguish two types of commits: the ones that change source code files of a project (those that contain at least one class in them) and the ones that change the rest of the files (dependency file, configuration files, etc.)

Class filter. Class filters allow a user to choose about which classes they want to see information. If class is disabled, all of the stacks representing the class disappear from bars meaning that the user sees information about only those changed classes that are still enabled. The filter is especially useful when some classes have abnormally large changes which force other commits to be drawn relatively small compared to the commit

where the abnormal changes happened. If the classes are out of the user's interest, then the sizes of drawn bars make it difficult to get any meaningful overview of the project. By disabling these classes, the user can bring the scale to normal.

Tooltip with more information. Interactive elements are useful to avoid polluting a diagram with too much information. Instead of drawing everything at once, the better approach is to show the most minimum and necessary information and propose additional information on-demand. This being said, a user can hover the cursor over any part of the bar to know exactly how many lines were changed in that particular class. To know more about the commit itself, hovering over the commit hash shows a tooltip with information as a commit message, date, author and the list of all classes that were changed with the number of lines affected. Prior to rendering the tooltip, the width of it is calculated to position it in a visible part of the screen. It is needed to avoid situations when only a small part of the tooltip is visible and the rest is beyond the borders of the screen which usually happens for commit hashes that are closer to the right border of the screen.

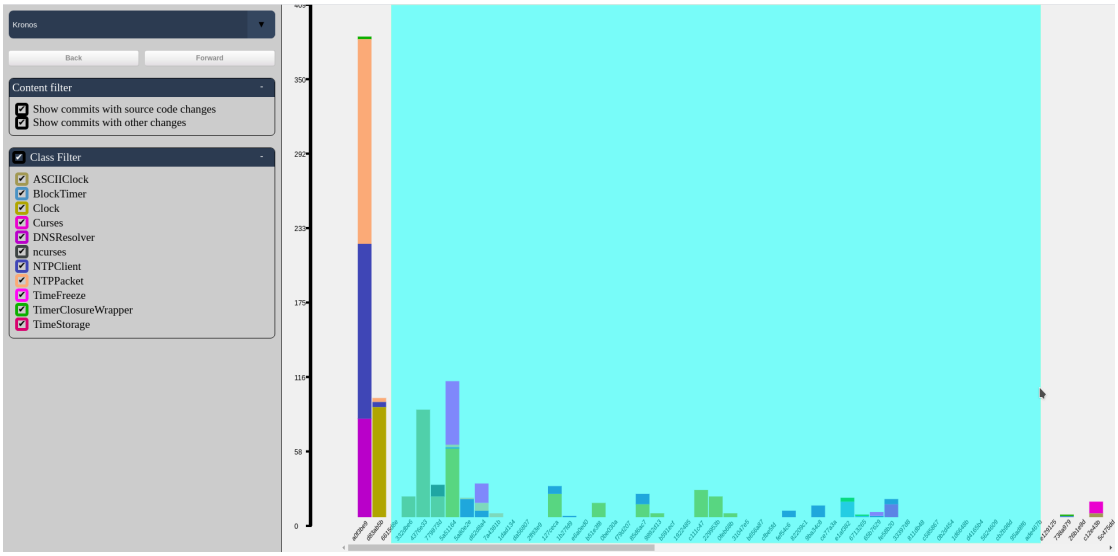

Figure 11. Selecting commits between "6615d8e" and "ade467eb" to open "Class overview" view

Zooming helps to get a broader overview of the project by scaling down the diagram and displaying more commits in the screen. To control the zoom level, the user can either use the mouse wheel or press '+' or '-' keys on the keyboard to zoom in and zoom out correspondingly.

Moving in the diagram is possible by the horizontal scroll bar. The length of the scroll bar automatically adjusts depending on the zoom level and the number of commits in the project.

We decided to use number of changed source lines of code (SLOC) as a metric system for the diagram. We can agree that it is perhaps the most common and intuitive metric system for source code [KB12]. Since the times of assembly and machine codes, programmers used it to get a feeling of complexity of a project. Even now it is possible to see open discussions about practicalities of using SLOC during code reviews in the internet [Sta17]. The metrics of number of changed lines is inherent to text files, and source version control systems like Git are built to work with text files, so it is quite natural to describe each commit in this way. Also, it is straightforward to get this data by using git commands. By mapping each commit to the number of changed lines in that commit, we do not expect users to make specific conclusions but rather to let them visually perceive size of impact that the commit makes compared to other commits. Think of it as a stock price chart - we can't say if ten thousand euros is a small or a high stock price unless we look at surrounding prices in the chart. Same for the number of changed lines, we do not intend to imply whether a commit has a small or a large impact - we show the larger picture and let users observe and make their own opinion. In the end, taking into accounts mentioned factors, it seemed like a good accompanying piece of information to provide for the interactive entry view.

4.3 Class overview view

What is it? This view is called "Class overview" view because it allows to examine a particular class on a more detailed level over several commits. The diagram in the view can be interpreted as a table where each column represents a particular commit that contains change information for all methods that have ever existed within a specific range of commits (Figure 14).

Since the diagram reflects changes over several commits, to switch to it, the user needs to select a range of commits in the first "Commit range" view as it is shown in the Figure 11. In the figure 14, on the side of a diagram, we can see the list of method names, e.g. "drawClock", "loop", "onTick", etc. While commit hashes are displayed at the top, e.g. "6615d8e", "332dbe6", etc. At the intersection of each commit hash and a method name there is a status indicator in the form of a circle which belongs one of the following types: blue-colored circles, green-colored circles, gray-colored circles, half-colored green circles, omitted circles.

The blue color tells that a method was just created, e.g. the loop() method was created in the commit "4376e33".

The green color tells that a commit changed a method, e.g. "drawClock", "start" and "tick" methods were changed in the commit "779873d".

The gray color indicates that a method was not changed in the particular commit, e.g. starting from the commit "5a51164" to the right no methods were changed.

A half-colored green circle indicates a merge commit to the master branch that does not change a method itself. Rather the change was done in one of the branches pointed

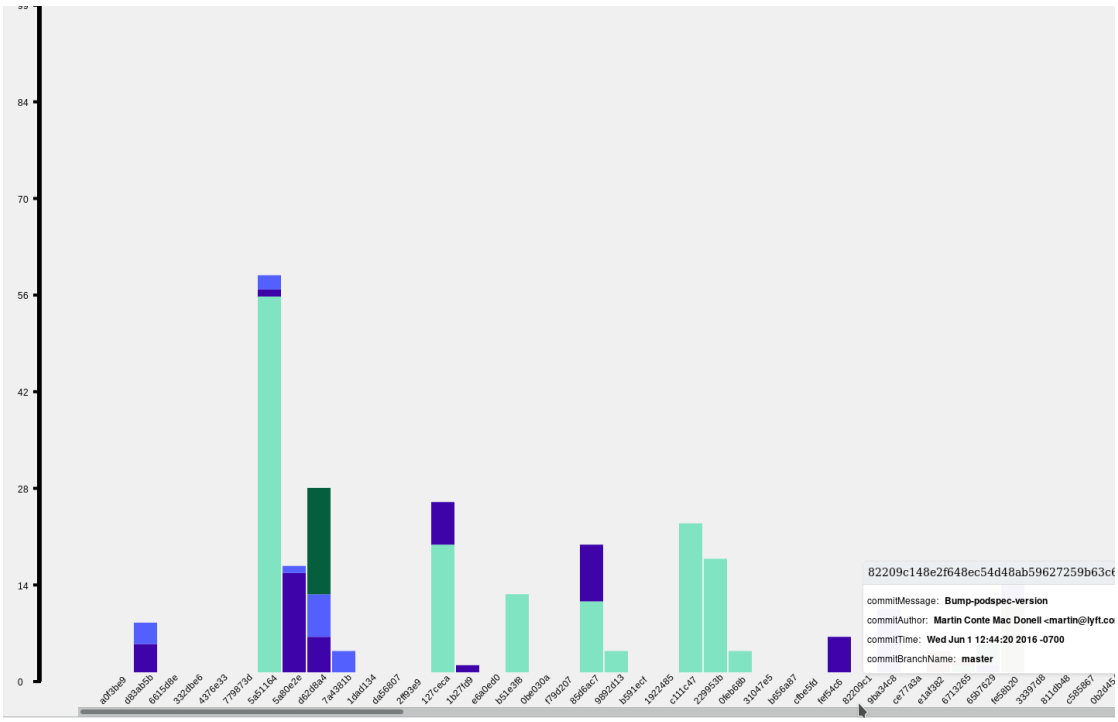

Figure 12. Hovering mouse over a commit displays a tooltip with more information

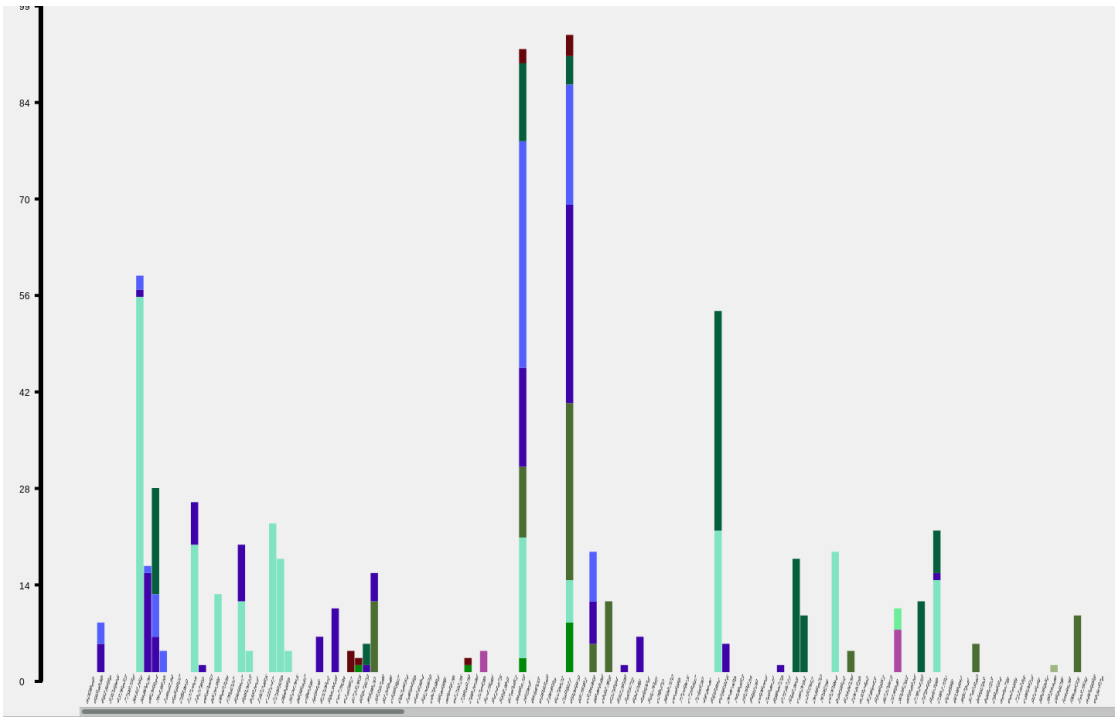

Figure 13. Zooming out allows to see a wider picture of activity

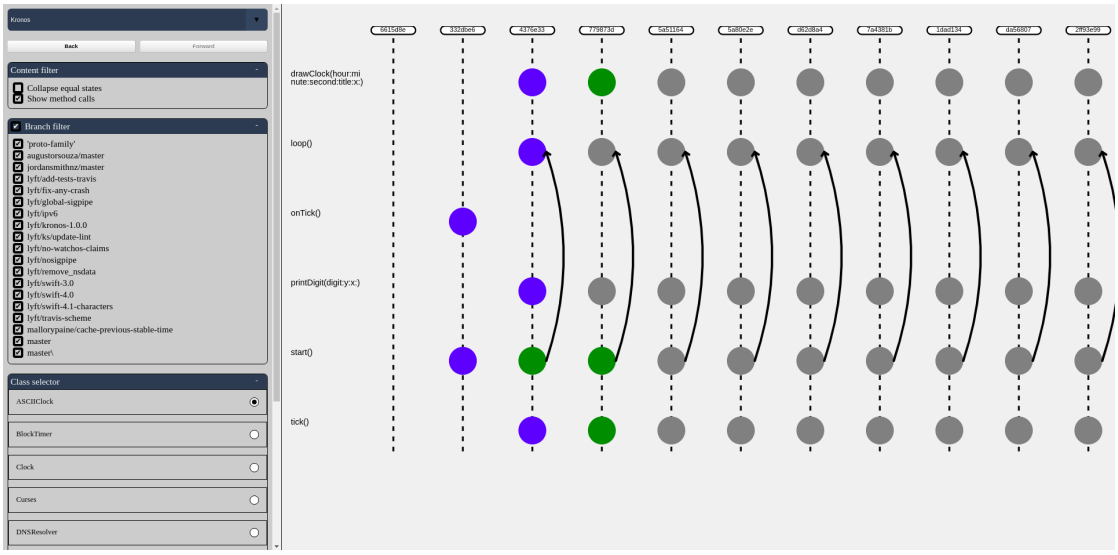

Figure 14. "Class overview" view visualizing the "ASCIIClock" class starting from the commit "6615d8e"

by the merge commit which effectively causes the master branch to contain the changed version of the method. It is especially useful when filtering out other branches from the view. The indicator lets the user know that some of the hidden branches contain a change that leads to the method change in the master branch.

A missing circle denotes that a method did not exist in that commit. It could mean that either the method had not been created yet or it was removed if there are preceding visible indicators. For example, according to the figure 14, method "onTick" didn't exist in the first commit, then it was created in the next commit and removed right after that, leaving an empty row for that method.

Arrows between methods represent method calls. Status indicators that are not connected by an arrow mean none of the methods is calling another one. For example, in the figure 14 the method "start" is calling the method "loop" throughout the whole displayed range of commits.

The "Class overview" diagram provides several ways to interact with it as described in the following paragraphs.

Moving. If visualized data is large enough to not fit the screen, it is possible to scroll the diagram in both horizontal and vertical directions. The diagram makes sure that method names and commit hashes stay always visible and scrolls them according to the scroll position of the diagram.

Tooltip with more information. Same as in the commit range view, it is possible to move the cursor over a commit hash and see more information about the given commit.

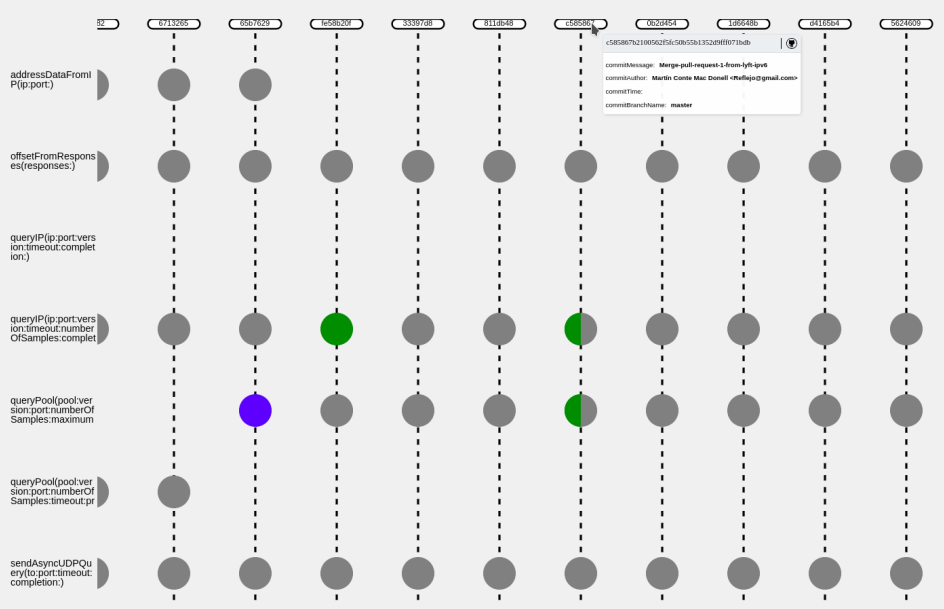

Figure 15. Half-colored status indicators for the methods "queryIP" and "queryPool"

This can provide additional information about context around a particular change such as the list of other changed classes in the same commit. It is demonstrated in the figure 15.

Available filters and selectors are described below.

Collapsing equal states. It is easy to spot repetitive visual elements in the figure 14. Commits that do not change the observed class result in the same indicators. To provide more value to the user by allowing to skip same states of commits, the content filter contains a filter saying "Collapse equal states". Once enabled, commits with the same state are merged into one column with commit hash being replaced by ellipsis "...". Based on the Figure 16, we can see that the class is never changed after the commit "779873d" as the rest of commits are collapsed into one column that indicates no change.

Displaying method calls. This filter is enabled by default and is used to control whether the diagram should draw arrows indicating calls between methods. Disabling the filter results in the same diagram but without the arrows drawn on top of it.

Class selector allows the user to switch between classes. Class names displayed in the list depend on a selected range of commits and class filters – if a class does not exist withing the specific range, it is not listed in the class selector; as class filters are global, if a class is disabled by the user in the class filter, then it is not listed in the class selector either. Class selector is displayed in the figure 14 at the bottom of the menu.

How is it useful? Typically, version control systems work on a file level or, if speaking of source code files, a text level. Although such behavior is useful in order to be language-

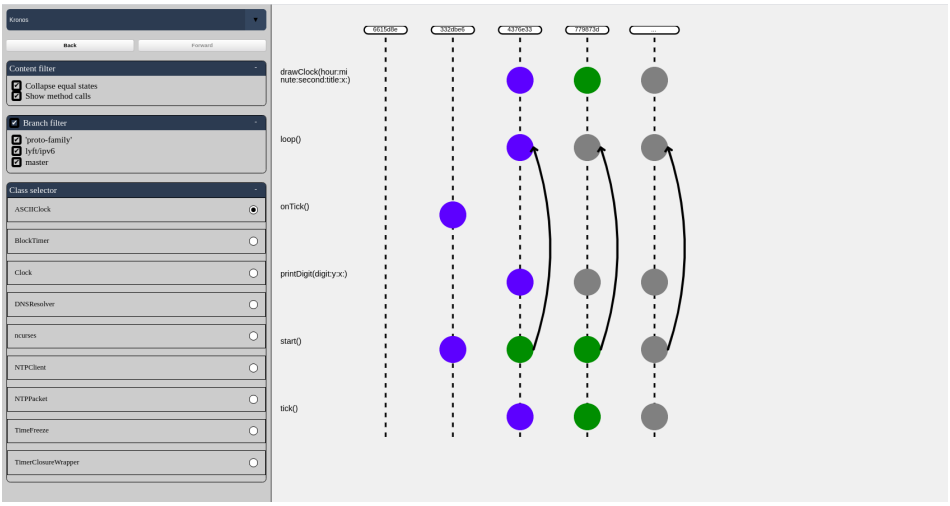

Figure 16. Same view from the Figure 14 after enabling "Collapse equal states" filter

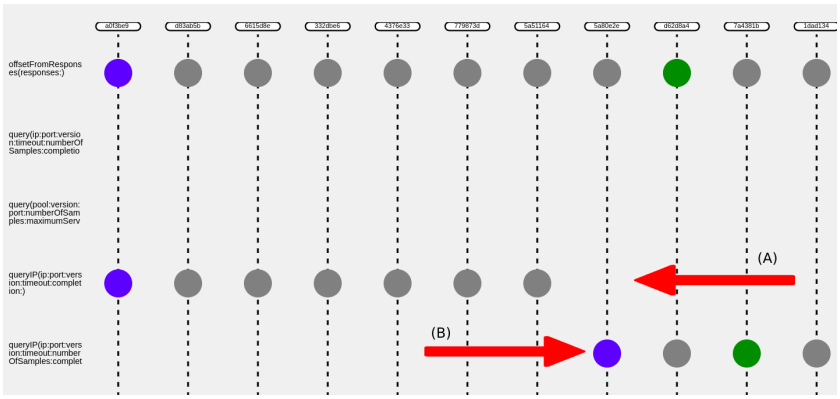

Figure 17. (A) - indicator for the method "queryIP" disappears, (B) - blue indicator tells us a new method with the name "queryIP" was created

independent, it also means that revisions (a.k.a. commits) provide limited information about a change which is lacking of language abstractions like class fields, methods and calls between them. Developers have to figure out all these details on their own by reading the source code. As a side note, it is also one of the limitations that are used to argue that Git and similar VCSs are incomplete and better VCSs are needed to support higher quality evolution research [RL07, NVC+12], while proposing alternatives that are directly integrated in IDEs and are tracking low-level modifications of a developer or that use abstract syntax tree (AST) to track changes on a language level [YMK13]. Visualization of commit changes at language-level abstraction reduces the effort required

to understand which commit changes which methods. For example, in the Figure 17 (red arrows are not part of the diagram and drawn to bring the reader's attention to the discussed spot), it is straightforward to notice how the method "queryIP" disappears in the commit "5a80e2e" and a new method with the same names appears, so it must be a signature change in the method.

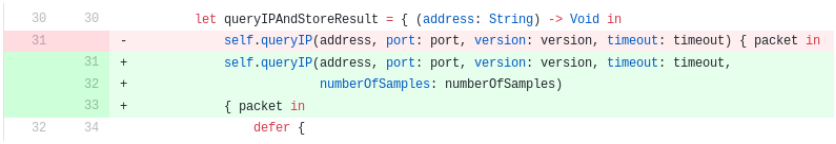

Figure 18. Signature change in the commit where the method "queryIP" re-appears according to the Figure 17

When looking at the commit, we can see that indeed one of the changes involves a signature change (Figure 18). Moreover, having all of the change indicators in one screen allows to visually see frequency of updates done in class methods. It can help to make decisions as it is described in the "Usage scenarios" section.

4.4 Call volume view

What is it? The last view of this project is the "Call volume" view. It is named so because it represents the amount of calls that methods receive. As GraphifyEvolution performs static analysis, this amount is equivalent to the number of call statements. So, if the method A and the method B have correspondingly x and y statements calling the method C, the call amount for the method B is equal to the sum of calls which is $x + y$ in this case. The sum is mapped to the diameters of circles – the larger the circle is, the more calls that method is receiving. Hence, circles in the diagram represent individual methods. The total sum of call amounts of each method that belongs to a particular class defines the thickness of the pipe. Therefore, each pipe represents one class and circles that stem from that pipe are methods that belong to that class. The thickness of pipes helps to quickly identify the most used class in terms of calls by finding the pipe with the largest width. The calls could be one of both: calls coming outside of the class and internal call withing the class. Figure 19 depicts the view where we each pipe has a color that corresponds to each class. Based on the image we can say that the yellow pipe which represents NTPPacket class has the largest width together with a discernibly large circle.

A circle with a hole in it represents a method with zero calls. Correspondingly, pipes with an empty body mean that none of the methods in that class receive a call. It is possible when the class is just created and committed to the repository, or when it is a class that ovverrides callback methods that are called by some framework, e.g. methods like doGet() and doPost() in Java Servlets. For example, in the previous figure, the class

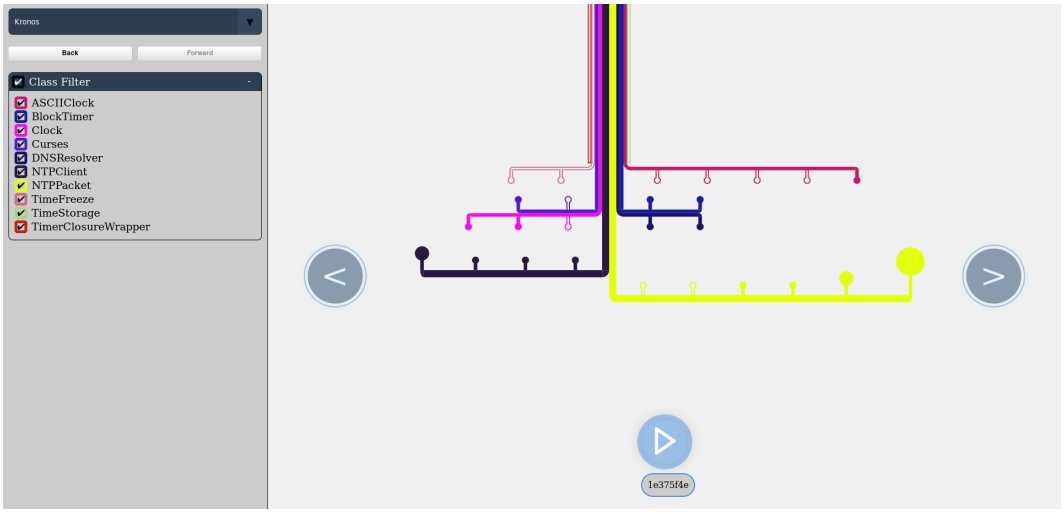

Figure 19. "Call volume" view

TimeFreeze has 2 methods as there are 2 circles growing from the pipe and both of them have a hole. If a pipe does not have any circles then it vacuously has an empty body, e.g. TimeStorage and TimerClosureWrapper classes from the previous figure.

As it was said earlier, diagrams in this project are built with interactivity in mind and the call volume diagram is not an exception. Below are listed all possible ways one can interact with the diagram.

Zooming. The quantity of drawn classes as well as the size of nodes can grow rapidly as a user moves towards the latest commits. Fitting everything into one screen without loosing details becomes impossible after some moment. Although losing details is not perilous since the goal of the diagram is to allow to differentiate the most discernible large entities like classes and their methods easily at one glance, the user still may want to control the level of details drawn in the diagram. For that, they could zoom in to and zoom out from any point of the diagram by using a mouse wheel, touch-pad scroll or '-' and '+' keys on a keyboard.

Moving. To provide more flexibility and support even smooth experience when zooming in and out, diagram can be moved around by drag & drop technique. For aesthetic purposes, the diagram is locked from the top border of a screen to avoid situations when the diagram is floating in the screen after being dragged. Moving together with zooming the diagram allows the user to fully control position and scale of the diagram which facilitates the analysis.

Switching commits. The diagram has three control buttons: left, right and play. Click on the left or right buttons switches the current commit to the previous or the next one correspondingly. The transition is accompanied by a smooth fade-in transition

effect. To run the diagram through commits, the user can press the play button which will automatically switch commits resulting in animation of the diagram. The user may pause the animation at any commit by clicking the pause button. The label under the play button indicates the current commit hash that is visualized in the diagram.

Tooltips with more information. Hovering the cursor over the current commit hash displays the tooltip with commit information. It is the same tooltip that is displayed in the previous views. In addition, hovering the cursor over pipes enables a tooltip with the list of methods that belong to the class represented by the hovered pipe. Each method has the amount of calls with the names of classes where the calls are coming from. Hovering over a specific circle displays the same tooltip but only with the information of that particular method represented by the circle. Class-level and method-level tooltips are displayed in the Figure 20 (attention to where the cursor is).

Figure 20. A. Tooltip that appears when hovering over the pipe. B. Tooltip that appears when hovering a specific circle.

How is it useful? The diagram data is based on the amount of calls made to the methods. Such information can help in several ways. First, one can estimate how difficult it could be to change or refactor a method by looking at the diagram. Large circles would

mean more places in the code that depend on the method which might imply more work required to finish the task. Second, if the large width of a pipe caused by several places in the code calling different methods of the class, this could be a sign of a very large class usually known as "God" object or "God" class.

In addition, similar metric **Number of Calling Functions – "CALLING"** is defined by Hersteller Initiative Software (HIS) source code metrics. HIS is a set of recommended metrics for efficient project and quality management originally defined for automotive systems. This metric is said to address such attributes of the project like resource utilization, modularity, reusability, testability and others [VKC+20]. The currently used metric differs in a way that a calling method could be counted twice if it has two statements calling the same method. Although not supported out-of-box, this feature could be easily added and then the visualization could be useful to identify problems within the scope of HIS metrics. Looking at the diagram would be sufficient to detect potential problems where the large shapes could indicate the symptoms.

Shape composition. The diagram is leveraging the visualization technique that was mentioned earlier. We compose such a complex diagram from basic shapes that are provided by Konva library - circles, rectangles and arcs. You can see decomposition of the diagram in the Figure 21. It is worth to mentioned that dynamic programming technique was used to calculate coordinates of each shape; and to avoid redundant calculation, memoization of positions was used.

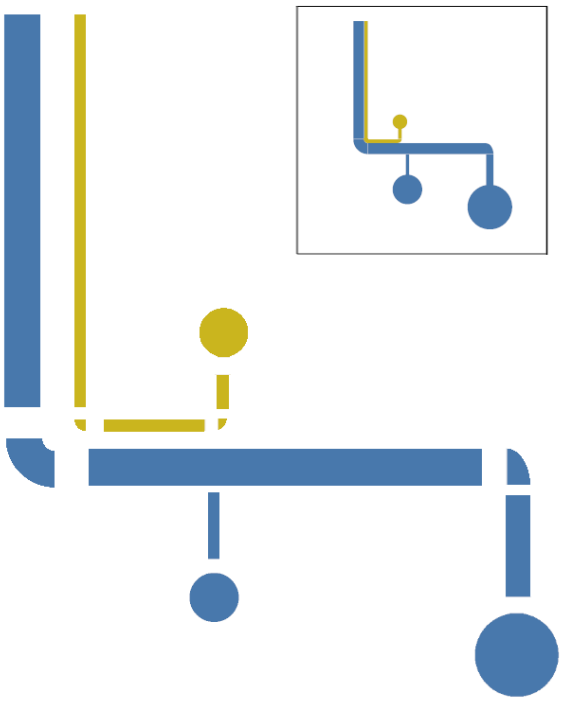

Figure 21. Building blocks of the call volume diagram

5 Usage scenarios

5.1 Commit activity

Questions.

- How to know which commits change source code files and which commits change other files?
- What is the distribution between these 2 types of commits?
- What is the usual size of a commit?

Why is it useful? Not all of the commits made in a project necessarily change behavior of the application. Such changes include but not limited to

- Updating project dependencies due to security patches or just simply upgrading to stay up to date, e.g. pom.xml for Maven or package.json for NPM

- Changes in visual layout, e.g. HTML, CSS or XML like for Android application layouts
- Changes in documentation, e.g. updating README.md or a license file, adding a new translation to the documentation
- DevOps configuration changes, e.g. travis.yml for Travis CI configuration
- Application configuration changes, e.g. changing the default value of some parameter
- Plugin configuration updates, e.g. checkstyle or linter parameters
- Updating .gitignore file

A team might be updating dependencies manually. By being able to see how many commits you perform to keep the project up to date compared to commits that evolve the project source code, one might decide that it is time to integrate the repository with automated dependency updating tools like Renovate boot [9]. Or, one could use the activity distribution to check if documentation is updated constantly after new changes in the source code.

Answer. You can also filter out other commits to see only source code file changes and vice versa. Zooming out helps to see a wider distribution of the commits.

5.2 Violation of the single-responsibility principle of OOP

Questions.

- How to know if there are signs of potential "God" object, a.k.a. "God" class, that knows too much and violates the single-responsibility principle of OOP?

Why is it useful? The single-responsibility principle of OOP states that classes in a project should be responsible of only one aspect of a problem and solve that problem well. Now, when a class has a lot of public methods called from different parts of the project, the class starts growing in size incurring several problems, e.g. it becomes hard for a developer to read and understand what the class is doing; debugging becomes challenging when it involves a large state of the class, which also might lead to hard-to-write tests; large objects might unnecessarily increase memory consumption.

[9] `https://www.whitesourcesoftware.com/free-developer-tools/renovate`, accessed on 14/05/2021

Answer. Since the size of pipes and circles in the diagram depends on the number of calls, it is easy to spot a large entity that represents a class that is being used too much which can be a sign of a potential "God" object. Obviously, there is no specific size that would ultimately tell if we are dealing with such a class, rather the observation is based on relative comparison with the sizes of other classes. In addition, as sizes of circles vary, it is possible to see what are the most "popular" methods. If only one method of a class has a large circle, it could be just a single method doing too much work which could be an indication to split the method. For the typical scenario, a developer willing to analyze the project for present issues would start from the most recent comment. It will visualize the last state of the project. However, it is possible to visualize any commit of the project. Possibility to switch back and forth between commits helps to see how the sizes of shapes representing classes and their methods are changing over time. It is helpful to visually see if there was any progress in refactoring due to aforementioned problems – the width of pipes and circles should become smaller and perhaps new classes might emerge as some of the responsibilities are being extracted from the "God" class.

5.3 Writing tests for untested code base

Questions.

- What methods to start writing tests from?

Why is it useful? When dealing with unfamiliar code base, it is not fully clear where to start writing tests from. One of the ways to approach the problem could be to start from the most "fragile" parts towards stable parts of the code, providing value as early as possible. Fragility could be assessed by different criterias, e.g. business importance of a part of the code, number of reported bugs. Another criteria could be the change frequency of methods. Software is made by humans so every change is a potential cause for a new bug, even if it the change itself is a fix to an existing bug [YYZ+11]. So by starting from the methods changing the most, we can provide value early in the beginning of work.

Answer. Class overview is the ideal place to see frequency of method changes in a particular class. By selecting the whole range of commits, we can observe change statuses of methods from the first up to the latest commit in the project. It is easy to spot changes depicted by green-colored circles while unchanged methods are depicted by the gray color. In case of writing tests, the developer could decide to start writing tests from the most recent green-colored methods, or a person who is responsible for work allocation like a manager can simply look at the diagram and based on frequency decide which methods should be tackled first and leave the rest to other sprints or time frame.

5.4 Splitting a large class into smaller classes

Questions.

- What methods can be extracted together?
- What was the original concern of the class?

Why is it useful? For this type of refactoring, it is important to know what concerns the class is trying to resolve. Understanding concerns can tell which methods form a group that belongs to one domain and could potentially be extracted together. It can help to form a mental picture of the class.

Answer. The "Class overview" view shows arrows that depict method calls. Such calls can indicate coupled methods that belong to the same concern and could be extracted together. For the typical scenario, a developer would select the whole range of commits and look at the latest commit to see current method calls. In addition, the developer can enable collapsing equal states. It will allow to quickly see how method calls were changed over time. If some method was initially called by a group of methods and then the call was removed, it might be also worth to examine that method before extracting the group.

Also, when dealing with large classes it can be naturally interesting to know what were the original methods in the class before it grew up so much in size. Such information could also give an idea about the original concern of the class before the rest of the methods had been added to the class. For that the developer could select an arbitrary range of commits starting from the commit of interest and scroll to the left of the diagram to see original methods within that range.

6 Discussion

6.1 Evaluation.

Comparison of tools. This project differs from previously created tools in several ways. We will describe main differences in comparison to the most related tools that were mentioned in Section 2.4. First of all, as opposed to the Vis-a-Vis that focuses on graphics source code like shaders, this visualization tools concentrates on projects written using general-purpose programming languages like Swift or Java. Another main difference is the way of accessing the tool. Most of the similar tools are binary executables and need corresponding set-up to be done before running the tool. Our tool, on the other hand, is accessed by using a browser making it easier to access. The only requirement is that it should be deployed to the cloud and prior work like analysis of a software repository with GraphifyEvolution is done to create a database for the tool. Also, our tool provides 3 views that are connected by meaningful transitions through clicking on a commit or selecting a range of commits. These transitions and available user interface elements like filters and tooltips tell about interactivity of the tool. Other visualizations like CCEvovis or CodeFlows typically provide only one view. As opposed to our tool, nothing was mentioned about leveraging of interactive elements either in those tools.

Quality of the software. The important aspect of the quality is the clear separation of back-end and front-end as it was discussed in Section 3.2 "Application architecture". In addition, using React and Konva libraries provided a clean way of rendering Canvas shapes and DOM elements separately. The testing of software was minimum due to time constraints and fast pace of development that required constant updates. It would be unfeasible to maintain tests in such early stage of the project while we were experimenting with different implementations. So the decision was to reduce the priority of covering the project with tests. However, there are still a few tests that cover core functionality like position calculation of visualization shapes. Mainly, testing was performed manually in two browsers "Mozilla Firefox" and "Google Chrome". Writing tests could be the next item as part of the future work.

Performance. There are three main important questions that would tell about performance of the application: "How many commits can the application handle?" and "How fast are queries returned from the server?". Although we haven't tried with the real repository that would contain such number of commits, the test was performed with duplicated commits of a real repository. Each trial made the server respond with larger number of commits than in the previous. It was observed that the "Commit range" view can visualize about 1 million commits before its performance drastically degrades. It is safe to assume that it is more than sufficient for an average software project. Linux

kernel repository [10] which is a quite large and old repository has slightly more than 1 million commits. "Class overview" view on the other hand is not well-optimized, so it can display at most about 1000 commits. However, taking into account that the user has to manually select the range of commits, 1000 commits is a reasonable threshold. No tests were conducted for the "Call volume" view. Regarding the speed of the server, the largest repository that we have tried contained 246 commits. The average response time of the API for "Commit range" view is 45 ms. For the API of "Class overview" view, the average response time is 524 ms. For the "Call volume" view with 10 classes that time is 15ms. This time includes querying Neo4J and returning the JSON response.

6.2 Future work

Call arrows in the "Class overview" view. It is easy to make the diagram look complex by having many intersecting arrows pointed to different directions. Figure 22 depicts the issue in the leftmost column and alternative solutions that avoid intersections in the middle and the right columns. We can see how circles with originally intersecting lines were re-ordered to avoid crossing lines.

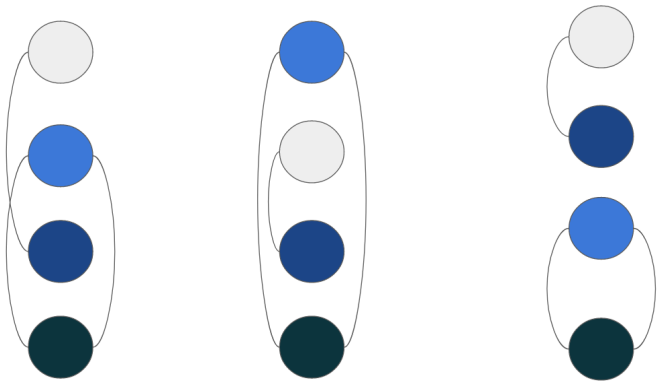

Figure 22. Demonstration of intersected lines in the left column and possible improvements in the middle and the right columns

Currently, there is no specific rule about how arrows are drawn. The Figure 23 shows how intersecting arrows look like in the diagram. To improve this, we can pay attention that we are dealing with graphs; and the property that we want to detect is planarity because planar graph is a graph that can be drawn without crossing edges. By checking if

[10]Linux kernel repository on Github – `https://github.com/torvalds/linux`, accessed on 14/05/2021

a graph is planar we could potentially order method names so that arrows do not intersect in the view.

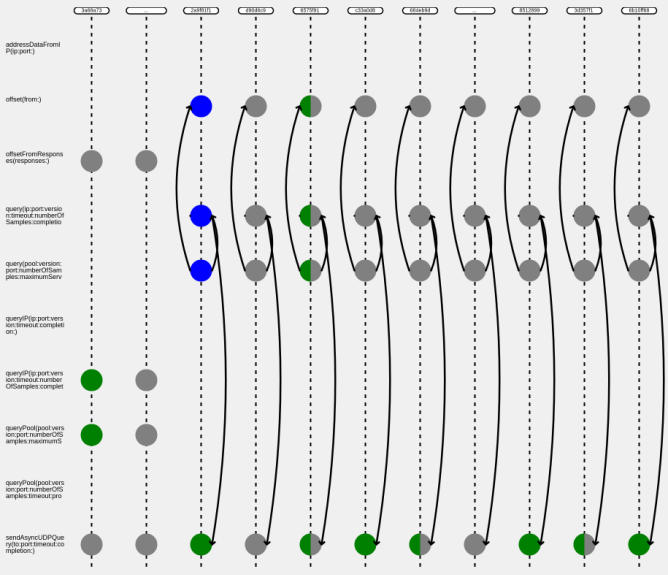

Figure 23. Intersecting call arrows in the "Class overview" view

Support for multiple branches. In the current scope, we focus only on one "master" branch, a.k.a. main branch. Although the tool displays commits from other branches, these are the branches that were eventually merged into the master branch. However, in some cases a branch has not been merged to the master yet, for example, for recent development ongoing in a separate branch. The tool does not display such branches, and it could be improved in the next versions by allowing dynamically switching between available branches.

Execution paths between methods. Based on the data produced by GraphifyEvolution, it is possible to build a diagram that would depict all possible ways that runtime execution flow might reach the method B from the method A. The inspiration came from the visualization shown in the Figure 24 that depicts how the trace of the river Mississippi changed over time.

Similar idea could be implemented for methods to see how the "trace" between methods has been changed. Figure 25 demonstrates a proposal diagram for this idea. Here, the method A is calling methods D and C that both eventually end up calling the method B. The pointer displays the current commit position that can be moved by the

Figure 24. The diagram at the top represents traces of the river in different years. The picture below is an actual satellite picture of that part of the river [NAS]

.

user to switch between commits. Removed calls could have a greyed-out traces while the active ones might have unique colors.

Such a diagram could be useful for debugging to understand dependency between methods. Also, it could be used to find potentially affected methods in case of discovered vulnerabilities in the source code.

Pulse of the project. As it was discussed earlier, some commits may contain no source code changes. Rather, they contain maintenance changes like dependency or "Readme" file updates. Based on this information, commit range view provides a general overview of activity in the repository like how many of the commits are dedicated to the actual code change. It gives an idea to the "digital signal" chart of zeros and ones where zeros and ones would indicate non-source code and source code changes correspondingly.

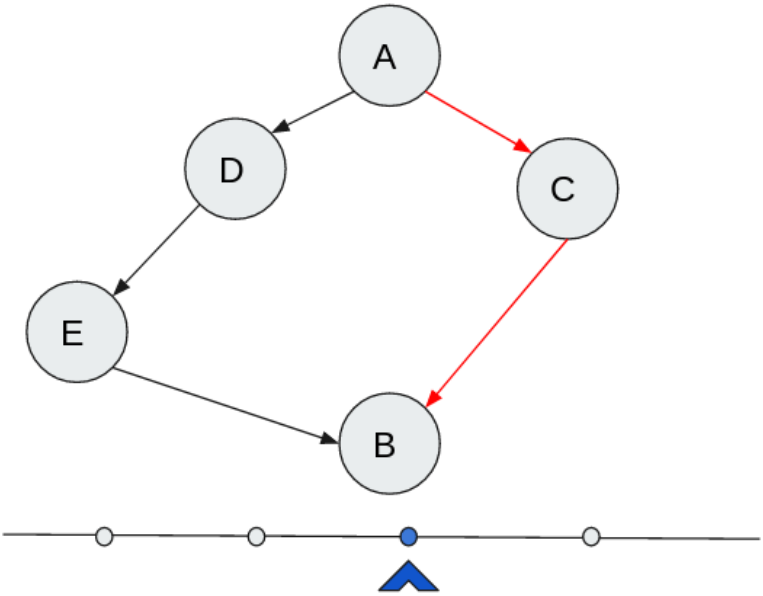

Figure 25. Visualization of execution flow between two methods. The blue pointer shows the current commit

Figure 26 depicts the concept of such a chart.

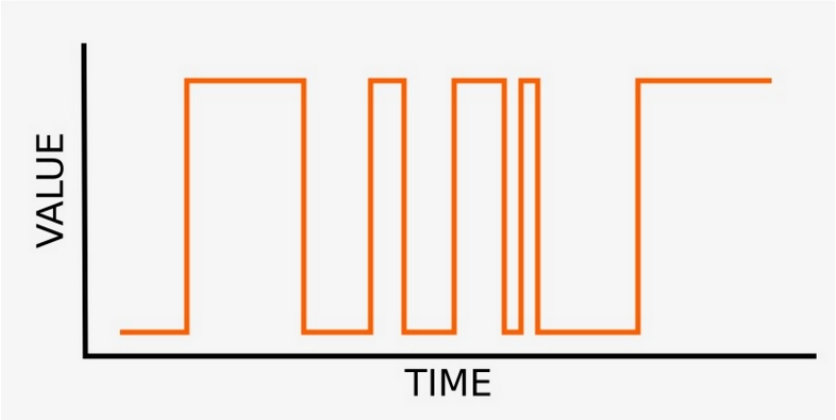

Figure 26. 0/1 - digital signal chart

Particular reason to leverage this diagram could be the diagram scale limitation – we can't zoom out infinitely. At some point of zooming out, stack bars will be squeezed so much that one physical pixel of the screen would have to depict several stack bars that represent commits. For such situations, the stack bar chart could have been switched to

the signal chart; because after some level of the scale details like commit hash become irrelevant. What the user who is trying to zoom out wants to see is the general view of the activity in the project. The signal chart allows aggregation of several commits by mapping them to one of the values – 0 or 1, hence representing the "pulse" of the project. For example, one of the ways to aggregate commits could be checking if majority of the aggregated commits within one pixel contain source code changes, then map them to the value of 1, otherwise to 0.

7 Conclusion

In this thesis, we applied visualization techniques to produce 3 interactive diagrams to visualize software evolution. To motivate the project, we discussed the background for this work that includes history of software evolution, visualization techniques and states of the art. From the software evolution history we could observe the shift of development process from when developers rarely ever wrote someone's code towards agile development where it is inevitable to modify code written by someone else. Hence, the source code remains the most important asset for software evolution studies which leads to the need of better understanding the code. Then, we talked about visualization techniques: composing complex shapes from simpler ones and reducing diagram dimension while retaining the original message. These techniques accompanied by the project architecture and heuristic techniques that made the visualizations possible laid the foundation required to design diagrams that convey one or another message. It was also within the scope of interest to see what visualizations are made in the software industry in general. So we mentioned prominent visualization tools available where some of them are built commercially and others as part of academic research. Since most of them focus on only one state of a project which is typically the latest revision, we dedicated a separate discussion for software evolution visualization specifically.

After that we focus on the actual implementation. We discuss results by describing each of the 3 views. "Commit range" views allows to get an overview of the activity in the project repository at one glance. For example, it is possible to see the distribution between commits that change the actual source code and commits that change other files like dependency list or configuration files. To examine each class individually, "Class overview" diagram visualizes the frequency of method changes and calls between those methods. Among other usages this makes it possible to easily detect most recently changed methods of a class. "Call volume" diagram depicts an individual revision of the project in terms of the number of calls done to methods with possibility to switch between revisions and animate the transitions. It is possible to see potential symptoms of violation of the single-responsibility principle of OOP like "God" classes. Usage scenarios describe in more details how each of these views can be useful to understand source code evolution better, to identify potential anomalies in the source code and

support decisions based on the frequency of changes. As a result, the tool could be useful for both developers, e.g. when refactoring large classes, and managers who have to make decisions, e.g. what methods to cover with tests first in an untested code base.

Finally, we talk about the future work that includes improvement suggestions and other possible visualizations. Each of these items has short motivation of why it can be useful to implement. They could potentially help generate new ideas and inspire to find new problems that could be solved by the means of visualization.

References

[Agh19] Aytaj Aghabayli. Software runtime data: Visualization and integration with development data – a case study. Master's thesis, University of Tartu, 2019.

[AGW08] Noura Abbas, Andrew M. Gravell, and Gary B. Wills. Historical roots of agile methods: Where did "agile thinking" come from? In Pekka Abrahamsson, Richard Baskerville, Kieran Conboy, Brian Fitzgerald, Lorraine Morgan, and Xiaofeng Wang, editors, *Agile Processes in Software Engineering and Extreme Programming*, pages 94–103, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[APBG19] C. V. Alexandru, S. Proksch, P. Behnamghader, and H. C. Gall. Evoclocks: Software evolution at a glance. In *2019 Working Conference on Software Visualization (VISSOFT)*, pages 12–22, 2019.

[Aub04] David Auber. *Tulip — A Huge Graph Visualization Framework*, pages 105–126. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[BB19] F. Bolte and S. Bruckner. Vis-a-vis: Visual exploration of visualization source code evolution. *IEEE Transactions on Visualization and Computer Graphics*, pages 1–1, 2019.

[Ben83] H. D. Benington. Production of large computer programs. *Annals of the History of Computing*, 5(4):350–361, 1983.

[Cai16] Alberto Cairo. *The Truthful Art: Data, Charts, and Maps for Communication*, page 5. New Riders Publishing, USA, 1st edition, 2016.

[CAT07] Fanny Chevalier, David Auber, and Alexandru Telea. Structural analysis and visualization of c++ code evolution using syntax trees. In *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*, IWPSE '07, page 90–97, New York, NY, USA, 2007. Association for Computing Machinery.

[Cau10] Andrew H. Caudwell. Gource: visualizing software version control history. In *SPLASH/OOPSLA Companion*, 2010.

[CB92] Bruce G. Coury and Margery D. Boulette. Time stress and the processing of visual displays. *Human Factors*, 34(6):707–725, 1992. PMID: 1292992.

[CDR+11] Gerardo Canfora, Darren Dalcher, David Raffo, Victor R. Basili, Juan Fernández-Ramil, Václav Rajlich, Keith Bennett, Liz Burd, Malcolm Munro, Sophia Drossopoulou, Barry Boehm, Susan Eisenbach, Greg Michaelson, Darren Dalcher, Peter Ross, Paul D. Wernick, and Dewayne E. Perry. In memory of Manny Lehman, 'Father of Software Evolution'. *Journal of Software Maintenance and Evolution: Research and Practice*, 23(3):137–144, 2011.

[dJSSRdSC18] I. d. J. Silva, M. S. R. Santos, L. L. Ramos, and L. P. d. S. Carvalho. Vismells: An interactive visualization for identifying and evaluating the effects of code smells on software projects. In *2018 XLIV Latin American Computer Conference (CLEI)*, pages 40–49, 2018.

[GJR99] H. Gall, M. Jazayeri, and C. Riva. Visualizing software release histories: the use of color and third dimension. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, pages 99–108, 1999.

[Han13] A. Hanjalić. Clonevol: Visualizing software evolution with code clones. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–4, 2013.

[HRE03] Anne Nielsen Hibbing and Joan L. Rankin-Erickson. A picture is worth a thousand words: Using visual images to improve comprehension for middle school struggling readers. *The Reading Teacher*, 56(8):758–770, 2003.

[HTY+19] H. Honda, S. Tokui, K. Yokoi, E. Choi, N. Yoshida, and K. Inoue. Ccevovis: A clone evolution visualization system for software maintenance. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 122–125, 2019.

[KB12] Pushpraj Patel Kaushal Bhatt, Vinit Tarey. Analysis of source lines of code(sloc) metric. International Journal of Emerging Technology and Advanced Engineering, Volume 2, Issue 5, 2012. `https://www.researchgate.net/publication/281840565_Analysis_Of_Source_Lines_Of_CodeSLOC_Metric`.

[LB85] M. M. Lehman and L. A. Belady. *Program Evolution: Processes of Software Change*. Academic Press Professional, Inc., USA, 1985.

[Leh80] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[Leh87] M. M. Lehman. Process models, process programs, programming support. ICSE '87, page 14–16, Washington, DC, USA, 1987. IEEE Computer Society Press.

[LK15] Justin Longo and Tanya M. Kelley. Use of github as a platform for open collaboration on text documents. In *Proceedings of the 11th International Symposium on Open Collaboration*, OpenSym '15, New York, NY, USA, 2015. Association for Computing Machinery.

[Mad02] N. Madhavji. Panel introduction. In *2013 IEEE International Conference on Software Maintenance*, page 0066, Los Alamitos, CA, USA, oct 2002. IEEE Computer Society.

[MGN18] Leonel Merino, Mohammad Ghafari, and Oscar Nierstrasz. Towards actionable visualization for software developers. *Journal of Software: Evolution and Process*, 30(2):e1923, 2018. e1923 smr.1923.

[MIK+16] Anna-Liisa Mattila, Petri Ihantola, Terhi Kilamo, Antti Luoto, Mikko Nurminen, and Heli Väätäjä. Software visualization today: Systematic literature review. In *Proceedings of the 20th International Academic Mindtrek Conference*, AcademicMindtrek '16, page 262–271, New York, NY, USA, 2016. Association for Computing Machinery.

[MML15] R. Minelli, A. Mocci, and M. Lanza. I know what you did last summer - an investigation of how developers spend their time. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 25–35, 2015.

[NAS] NASA. Mississippi meanders. Publisher: NASA Earth Observatory, `https://earthobservatory.nasa.gov/images/6887/mississippi-meanders`, accessed on 14/05/2021.

[NTM+13] Renato Lima Novais, André Torres, Thiago Souto Mendes, Manoel Mendonça, and Nico Zazworka. Software evolution visualization: A systematic mapping study. *Information and Software Technology*, 55(11):1860–1883, 2013.

[NVC+12] Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph E. Johnson, and Danny Dig. Is it dangerous to use version control histories to study source code evolution? In James Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, pages 79–103, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[PTDH19] Ei Pa Pa Pe-Than, Laura Dabbish, and James D Herbsleb. Collaborative writing at scale: A case study of two open-text projects done on github. 2019.

[RL07] Romain Robbes and Michele Lanza. A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science*, 166:93–109, 2007. Proceedings of the ERCIM Working Group on Software Evolution (2006).

[Roy87] W. W. Royce. Managing the development of large software systems: Concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering*, ICSE '87, page 328–338, Washington, DC, USA, 1987. IEEE Computer Society Press.

[RP21] Kristiina Rahkema and Dietmar Pfahl. Graphifyevolution - a modular approach to analysing source code histories. 2021.

[Sta84] Thomas A. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, SE-10(5):494–497, 1984.

[Sta17] StackOverflow. How important is it to reduce the number of lines in code? `https://softwareengineering.stackexchange.com/questions/185925/how-important-is-it-to-reduce-the-number-of-lines-in-code`, accessed on 14/05/2021, 2017.

[SVV03] Cheri Speier, Iris Vessey, and Joseph S. Valacich. The effects of interruptions, task complexity, and information presentation on computer-supported decision-making performance. *Decision Sciences*, 34(4):771–797, 2003.

[TA08] Alexandru Telea and David Auber. Code flows: Visualizing structural evolution of source code. *Computer Graphics Forum*, 27(3):831–838, 2008.

[THER09] A. Telea, H. Hoogendorp, O. Ersoy, and D. Reniers. Extraction and visualization of call dependencies for large c/c++ code bases: A comparative study. In *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 81–88, 2009.

[TYKZ07] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*icomment: Bugs or bad comments?*/. *SIGOPS Oper. Syst. Rev.*, 41(6):145–158, October 2007.

[VKC+20] Martin Vogel, Peter Knapik, Moritz Cohrs, Bernd Szyperrek, Winfried Püschel, Haiko Etzel, Daniel Fiebig, Andreas Rausch, and Marco Kuhrmann. Metrics in automotive software development: A systematic literature review. *Journal of Software: Evolution and Process*, 33, 05 2020.

[YMK13] YoungSeok Yoon, Brad A. Myers, and Sebon Koo. Visualization of fine-grained code change history. In *2013 IEEE Symposium on Visual Languages and Human Centric Computing*, pages 119–126, 2013.

[YYZ+11] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 26–36, New York, NY, USA, 2011. Association for Computing Machinery.

II. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Turkhan Badalov**,
*(*author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Software Analytics: Visualization of Source Code Evolution,
   *(*title of thesis)

   supervised by Kristiina Rahkema and Dietmar Alfred Paul Kurt Pfahl.
   *(*supervisor's name)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Turkhan Badalov
14/05/2021