Acta Cybernetica **26** (2024) 431–454.

Using Version Control Information to Visualize Developers' Knowledge*

Anett Fekete[ab] and Zoltán Porkoláb[ac]

Abstract

It is not always clear in case of a software project who has the right amount of knowledge concerning a certain module or file. Programmers frequently ask questions like "Who knows the most about this code?" or "Who can I ask for help when I work on this module?". In a large, long-term software product, knowledge is distributed in an uneven way among developers. Developer fluctuation during the product lifetime might cause some parts of the code to be known very well by a multitude of developers, while other parts might sink to the "gray zone", where developer competence is dangerously scarce. It is important for the project management to identify such critical points, to avoid the complete loss of competence. Version control repositories contain loads of useful information about the evolution of a software project. This paper presents a novel developer-centered method implemented as a plugin in the open-source code comprehension tool, CodeCompass. The method is intended to detect individual, team-bound and company-bound knowledge of large legacy projects. The competence information is computed from the extracted version control information from Git repositories. The calculated competence value is based on the number of commits per developer and their significance. The method weighs all changes according to their added value computed by a plagiarism detection software. Aggregated views for teams and companies are available based on various heuristics. The results are visualized as graph-based diagrams. Project managers and individual developers may both profit from the tool, whether it concerns software evolution, human-resource management, architecture, knowledge catch-up, or blame.

1 Introduction

In case of long-running software projects, the fluctuation of developers is inevitable. This is true for smaller projects with only a few developers at once, such as a uni-

*Prepared with the professional support of the Doctoral Student Scholarship Program of the Co-operative Doctoral Program of Ministry of Innovation and Technology financed by the National Research, Development and Innovation Fund.

[a]Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary
[b]E-mail: afekete@inf.elte.hu, ORCID: 0000-0001-8466-7096
[c]E-mail: gsd@inf.elte.hu, ORCID: 0000-0001-6819-0224

DOI: 10.14232/actacyb.299322

versity lab project, or large, industrial projects that count tens or hundreds of developers. When fluctuation is so great, there is always risk that certain components of the software suffer neglect. If all or even most developers that have knowledge about a component leave the project, maintenance problems might emerge, and it will cause much more problems to debug or develop the component than it would be in possession of decent expertise [3].

During development, lots of questions emerge that concern other developers, like "Who wrote this code?", "Why did they decide to write this code like that?", "Who understands this code the best?" and so on. These questions might be hard to answer, especially in large companies, where dozens of programmers develop a software. Nowadays, using some type of version control system for our projects like Git or SVN is fundamental. Repositories are able to store every piece of information that is bound to development since the start of the project. Naturally, the information we need is not stored explicitly but in the form of commits that can be processed and analyzed.

In this paper, we present the *competence plugin*, a tool that is capable of analyzing the commit history of a Git repository and tell various information about the project files and developers as graph-based visualizations. The plugin serves multiple different developer-related purposes with its visualizations. We would like to facilitate code comprehension for programmers by providing them a visualization that methodizes the obtained knowledge about project files. Observing the already familiar part of the code and the unknown territories, the programmer can consciously select the next comprehension target. It is also our goal to provide useful team- and company-level information, thus we present the team view and the affiliation views. They tell about the most competent developer on a certain file and their affiliation. They also provide information about the teams or companies who have the most code-related knowledge. The calculations and visualizations are all handled on file level.

For ethical reasons, all data in this paper has been anonymized. The original research has been done in a real industrial software development environment with actual development teams and programmers. We replaced the email addresses, usernames, and team or company names. We use the form *Dev email x* and *Team y* in place of the original names.

The rest of the paper is structured as follows: Section 2 describes the various methods and approaches for code comprehension supporting visualization, paying particular attention to version control related software. Section 3 presents our methodology of version control data analysis. Section 4 describes the visualization of the version control data. Section 5 provides a report of the details of implementation. In Section 6 we present the results of our work on some well-known open source projects. Section 7 discusses the possible threats to the method's validity. Finally, in Section 8 we give a heads-up about future work and conclude our paper.

2 Related work

There has been much research done before about code comprehension supporting visualization possibilities [6,32,41], and using version control data for visualizations in particular. Code comprehension supporting tools usually focus on a group of important and coherent aspects of a software, which means they do not cover every need of a developer that seeks deeper knowledge of the project. Visualization tools generally stick to 2D or 3D imaging exclusively, thus they can be divided in these two groups.

In this section, we give an overview of recent visualization tools that support code comprehension, and then we discuss tools that use version control information in their visualizations in detail.

2.1 Visualization for code comprehension support

Ever since large software projects started spreading, the need for reliable, transparent, informative code comprehension tools and visualizations has been growing [4, 30]. There are several software to facilitate source code comprehension, in the form of plugins (e.g. in IDEs) and standalone tools [8, 46]. These software always have some focus points that they put the most emphasis on, usually code metrics based on structure or content [44].

A returning approach is displaying source code metrics in the form of some type of map view, created with complex geoinformatical methods [20]. Tools with this approach usually represent source code as continents, states, cities, and buildings on a map. Metrics can be diversely shown by the size, color, borders and surroundings of the representing shapes. This technique guarantees that abstract modularity levels are depicted correctly, and transparency is provided for the user. Code comprehension software with map views can be divided into 2D and 3D tools. A couple 2D software include *CodeSurveyor* [20], *Software Cartography* [26], 3D software include *CodeCity* [47, 48] and *CityVR* [33] which is also a modern step towards code comprehension supporting visualization with an experimental usage of VR technology and gamification. Even more recent approaches combine traditional visualization techniques with virtual and augmented reality [25].

Another commonly used visualization method includes using simple shapes, such as rectangles or circles [2], and diagrammatic figures (e.g. UML diagrams, statistic diagrams) to represent code metrics. We can also apply grouping the tools by their dimensions; 2D software include *CodeCrawler* [28], *ExplorViz* [15], and *CommunityExplorer* [35], 3D software include *sv3D* [31] and *TraceCrawler* [19]. er erom

2.2 Usage of version control information

Version control information is used for various software research areas. Most frequently, in the center of these researches is the connection between commit actions and software quality, and the cost of maintenance. Naturally, this is used as the

prediction of distribution of software bugs. In [37], the authors developed a regression model that accurately predicts the likelihood of post-release defects for new entities. Similarly, in his PhD thesis [12] the author describes the connection between code maintenance activities as it is reflected by version control information and the deterioration of the code quality. In a related paper [13] the authors show that a connection between version control operations and maintainability really exists, in spite of the fact that the data is coming from different sources.

Apart from software quality, other researches target the developer's team. The authors of [22] utilize version control information for mining and visualizing networks of software developers. They detect similarities among developers based on common file changes, and construct the network of collaborating developers. The authors show that this approach performs well in revealing the structure of development teams and improving the modularity in visualizations of developer networks.

Unfortunately, information retrieved from version control systems has its limitations. As an example, attempts to predict code quality or developer efficiency cannot be achieved according to a research described in [36].

A frequent problem with information mining from version control systems is that they store only atomic information. In [23], the authors suggest a set of heuristics for grouping change-sets files that frequently change together. The results show that the approach is able to find sequences of changed-files.

Version control information can be used not only for extracting data from the source code for code comprehension purposes, but it is also a frequent research target for analyzing comments – either structured, or natural language text – in order to get additional information about the system [42].

Version control information is used for code comprehension purposes to connect related code modifications submitted in the same commit as described in [5].

2.3 Version control data visualization

Utilizing version control data forms a subset of code visualization techniques. While repositories contain lots of valuable information, repository mining and data analysis is an additional challenge in visualization pursuits [49].

Alcocer et al. [2] created a circular visualization called *Spark Circle* to visualize the changes in various source code metrics between commits. The visualizations vary based on the number of different metrics calculated. A spark circle consists of one annulus if only one metric is applied or multiple annulus sectors in case of multiple metrics. They also boost the visualization by using different filler and border colors and different shape sizes according to the concrete metric data. Spark Circle is a useful tool for the analysis of software evolution based on commit difference.

Not only commits, but entire Git repositories and the branches they contain can be effectively analyzed and visualized. Elsen [11] has proposed *VisGi*, a version control visualization software that is capable of displaying detailed graphs of Git branches and version structures. The software highlights structural and code-related differences. It also shows that time-bound visualization provides valuable information about software and repository evolution.

Greene et al. [17, 18] developed a browser tool that intends to give answers to collaborator- and repository-related questions among others, named *ConceptCloud*. They put focus on identifying abstract relations between objects and attributes. Their tag cloud visualization is mainly text-based, and capable of handling multiple selected tags for more accurate search results. Naturally, this method provides the user with an opportunity to find answers for code-related questions as well.

There are also attempts for the integration of version control data visualization into the development environment. The Eclipse IDE has a plugin which calculates the number of changes of methods during a given amount of time [45]. Another, less recent Eclipse plugin supports data visualization from the CVS version control system [7].

There can be correlation detected between the changes made to source code and performance. In order to facilitate discovering changes and their causes, Alcocer et al. [1] developed *Performance Evolution Matrix* which is capable of comparing multiple versions of the same software. The tool helps the user notice the modifications that might have caused changes in performance. The visualizations mostly aim changes in software metrics, such as additions and deletions to source code, call graph changes, and execution time differences.

As mentioned above, besides traditional 2D visualization techniques, modern approaches are becoming more widespread in version control visualization methods, such as virtual reality [38].

Apart from version control repositories, data from the supporting project hosting systems (e.g. GitHub, GitLab) is also useful for understanding software. Kumar et al. use Elastic search and Kibana for data visualization through mining GitHub for further information that cannot be found in plain repositories [27].

3 Methodology

3.1 Background

Repositories, especially those of long-running projects tell lots about development history and workflow. It creates an image of the gradual changes in the architectural design of the software. When it comes to maintenance and debugging, the first questions that usually emerge are "Who can I turn to for explanation of the logic, structure and objective behind this code? Who is the expert in it?". Version control systems contain all the answers to these questions in their commit history in an implicit way that requires meticulous analysis to be brought to surface. Commits provide all the information about who is responsible for each line of code ever written for that repository in their blame data. This information can be easily obtained and used in various different visual depictions. Our visualization intends to present the developer data to facilitate source code comprehension by providing developer statistics for the software.

In the competence computing, we consider one commit as a unit, this is why the plugin parses a given part of the commit history commit-by-commit, which is then

divided into *deltas* that are equivalent to the files that were modified by the commit author. The most important factor of developer competence in our calculation is the significance of modifications which is calculated using *JPlag*[1] [40], a code similarity checker. The idea is that we want to measure how important is a change and want to ignore irrelevant formal modifications. We compare the modified version of a file to its previous version in the commit history. JPlag returns a percentage value applying token-based comparison. When comparing different versions of a file, JPlag breaks down the file to tokens, and checks the changes between versions. To minor changes that do not affect the actual file content, such as adding or removing comments, or renaming identifiers, JPlag returns a 100% match. We use this number in our calculation as a threshold to detect relevant changes in the source code. If the compared versions are not 100% equal, then relevant modifications were committed to the code. This way minor changes are filtered, and the focus is put on actual content change.

3.2 Data parsing

The competence plugin consists of a parser and a service component. The parser performs code analysis, repository mining, and information extraction. Figure 1 shows the parser workflow.

The first and most important precondition of parsing is for the project to have a Git repository. Without one, the parser will finish parsing without processing any information. This is why the very first step during parsing is to find the *.git* directory. Since this directory is usually placed in the root directory of the source code, the parser looks for it in the user-provided source path. The parser can be provided with an optional input variable, n, the number of commits to parse. If n is not provided, the entire commit history is parsed.

Once the repository is found, we need to collect information about the modifications done in the commits. We traverse through each commit on the current branch, starting from *HEAD*, look through the list of modified files and calculate actual developer competence data. For this, we need to traverse all relevant commits to obtain the blame data. In order to traverse the commits, we need a revision walker that is sorted in a backward sequence in time and in topological order. The walker provides the next commit. Commit history processing is done as follows:

1. Let C be the commit history, and let c_j be the current commit that is retrieved from the walker ($C = c_1, c_2, ..., c_j, ..., c_n$). If n is provided, we check if c_j meets the conditions. If not, execution is terminated.

2. Let p_{c_j} be the direct parent of c_j which is retrieved from the repository. If p_{c_j} does not exist, we are at the beginning of the commit history ($c_j = c_n$), and execution is terminated.

3. Let $d_{c_j,p_{c_j}}$ be the difference between the two Git trees built from c_j and p_{c_j}.

[1] GitHub repo: https://github.com/jplag/JPlag

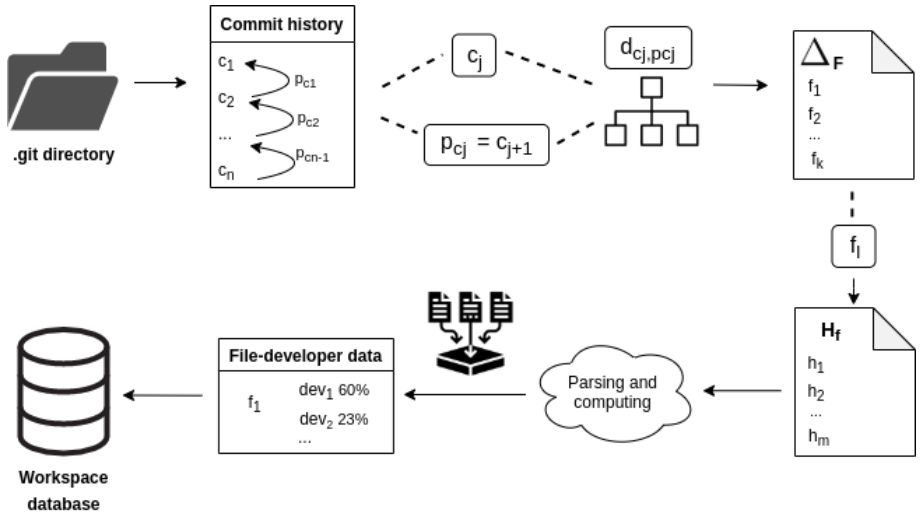

Figure 1: The methodology of commit history parsing. The commit history of a Git repository is parsed commit-by-commit. We take the diff of each consecutive pair of commits, and analyze every file that was modified in the latter one. The file versions are compared with a software similarity checker to determine the structural significance of modifications as a percentage value. The summarized data is persisted into the database.

4. As mentioned before, $d_{c_j,p_{c_j}}$ consists of *deltas*, each of which contains the modifications in one file. Let f_i be an individual file that has been modified in a commit ($F = \{f_1, f_2, ...f_k\}$).

5. $\forall f_i \in d_{c_j,p_{c_j}}$ we compare f_i with its previous version in p_{c_j} using JPlag. The returned percentage value shall be the competence data of the author for f_i.

6. Go to 4): move on to the next delta if it exists, otherwise, go to 7).

7. Go to 1), and start over with c_{j+1}, if all the conditions are met (i.e. there are further commits to parse), otherwise, go to 8).

8. Persist the calculated data in the database for every file and every commit author.

Despite not being able to extract usernames, we can, however, use author email addresses to extract affiliation, i.e. company or team names. Large companies usually have their own domain name and give a workplace email address within this domain to their employees (e.g. *microsoft.com*, *ericsson.com*). Based on this convention, we can assume, that if a developer authors their commits with a certain email address, we can conclude their affiliation.

The competence plugin automatically extracts well-known company names from email addresses during parsing time. It works with a list of possible domain names mapped up with company names prepared in advance. When the plugin is done with competence data calculation, all email addresses are checked for company name extraction. Private email addresses and others not in the company list can be completed with affiliation manually.

4 Visualization

The competence plugin is implemented as a part of CodeCompass[2] [39], which is an LLVM/Clang based open-source code comprehension framework developed by Eötvös Loránd University and Ericsson. The CodeCompass parser applies static analysis on the given source code and the corresponding build commands that are logged during compilation. Various information is stored about the project including structural data, code metrics, version control information, etc. This information is stored in the workspace database which is then accessed by the CodeCompass webserver. The webserver provides several various textual and graphic services through a web browser, such as detailed searching, structural and code-level visualizations, and Git blame data.

CodeCompass has a pluginable framework. Plugins work independently, thus a certain plugin can be easily skipped from the parsing process if it is not needed. Plugins consist of a *parser* and a *service* component. The parser component takes care of the analysis, while the service component is responsible for constructing the visualizations based on the stored information in the database and displaying it through the CodeCompass webserver. Currently, there are 4 different available diagram types.

4.1 Personal view

When browsing software components or learning the source code, it is useful for the user to keep track of which parts in the code base they have reasonable knowledge about, and what else is there to investigate and learn. The personal view intends to show this information. CodeCompass is capable of text-based authentication, thus it is possible to show the user their personal information stored in the database.

The personal view diagram shows the maximum competence percentage that belongs to the authenticated user for every file in the project. The percentage is converted to a color code that appears in the corresponding node in the diagram. Generated colors are on a scale from red, assigned to 0%, to green, assigned to 100%. Figure 2 shows an example of the personal view about the structure of the competence plugin.

This view is also useful for project managers to decide who to assign a certain task, or for other teammates to see who they can ask if a question emerges or a

[2] GitHub repo: https://github.com/Ericsson/CodeCompass

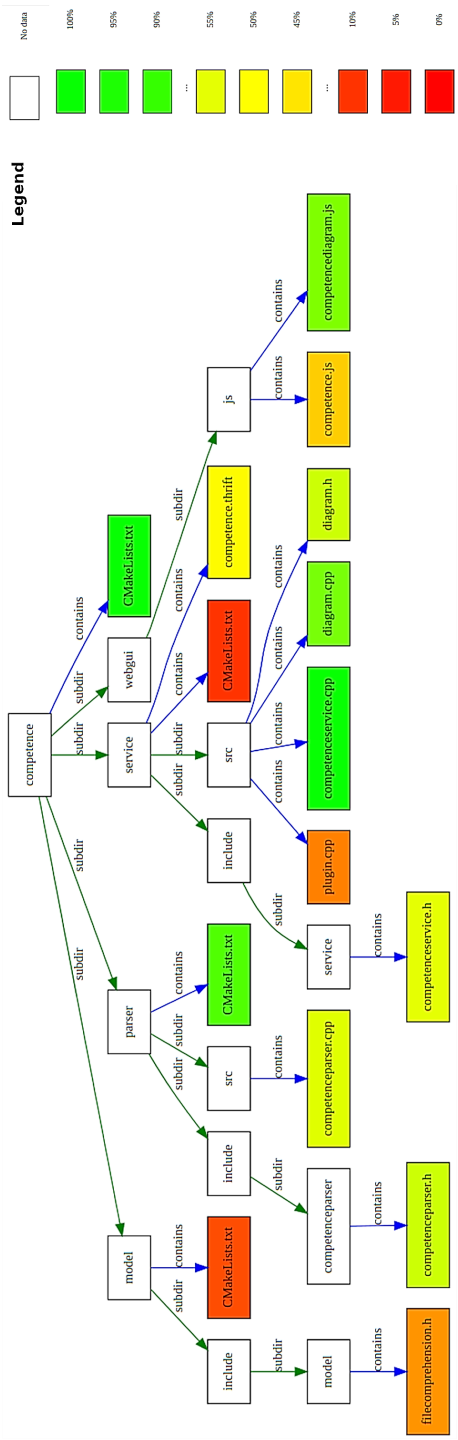

Figure 2: The personal view of a developer concerning the source code of the competence plugin. The non-white nodes represent files in the plugin. They are colored on a scale from red to green, according to the competence percentage of the developer. The containing directories are left white as they have no assigned data.

bug is found etc. However, the plugin is now only capable of showing the user their own information.

4.2 Team view

If we are looking to see who the most competent developer is in the current state of a file, it is likely the one who recently committed a larger significant modification to the file in question. The investigated number of commits can be set by the user before parsing. This way, the last n commits will be parsed, starting from the very last one. Team view displays the most competent developer of every file, based on the parsed information. It selects the maximum percentage stored for the file. The diagram nodes are colored according to the color code map that was previously calculated from the developers' email addresses. An example of the team view is shown in Figure 3.

4.3 Affiliation views

In case of a collaboration project or an open-source software, it is useful if we are aware which unit is the most competent in a software module. This is why it is advantageous to display affiliation-focused diagrams. The competence plugin provides 2 different affiliation views:

4.3.1 Individual affiliation view

In its logic, this type of diagram is similar to *team view*, but it is focused on affiliations. We calculate who the most competent developer is in a file, but instead of coloring the node with the personal color of the developer, we color it with the assigned color of their company or team.

4.3.2 Accumulated affiliation view

In order to learn which team is the most competent in a file, we need to consider all data that belongs to the file in question. In this diagram, we group the records of a file by developer affiliations, and take the average competence of every team. A diagram node gets the color of the team with the maximum average value.

Although they derive from the same data, the affiliation views might display different results for the same file: the first diagram focuses on the individual competence rate and the corresponding affiliation, while the second one calculates the average competence rate of the members of each team that worked on the examined file, and shows which team has the highest competence in that file. An example of the two views and their comparison is shown in Figure 4.

The latter 3 visualizations can be displayed without authentication.

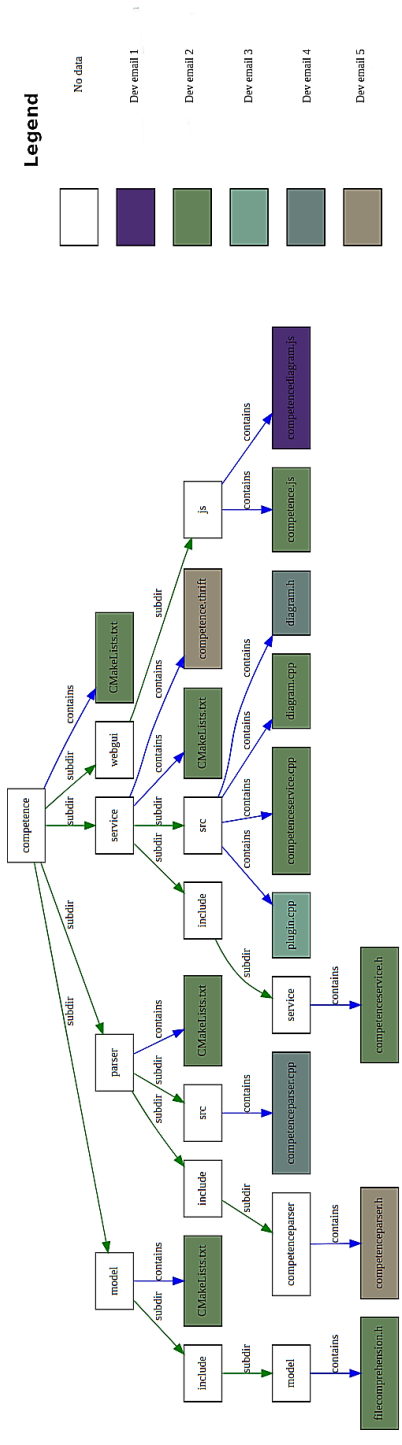

Figure 3: The team view of the competence plugin. Like in Figure 2, the white nodes represent directories, and the colored nodes represent the contained files. The file nodes are given the color of the most competent developer in that file. The colors are automatically generated from hashing email addresses.

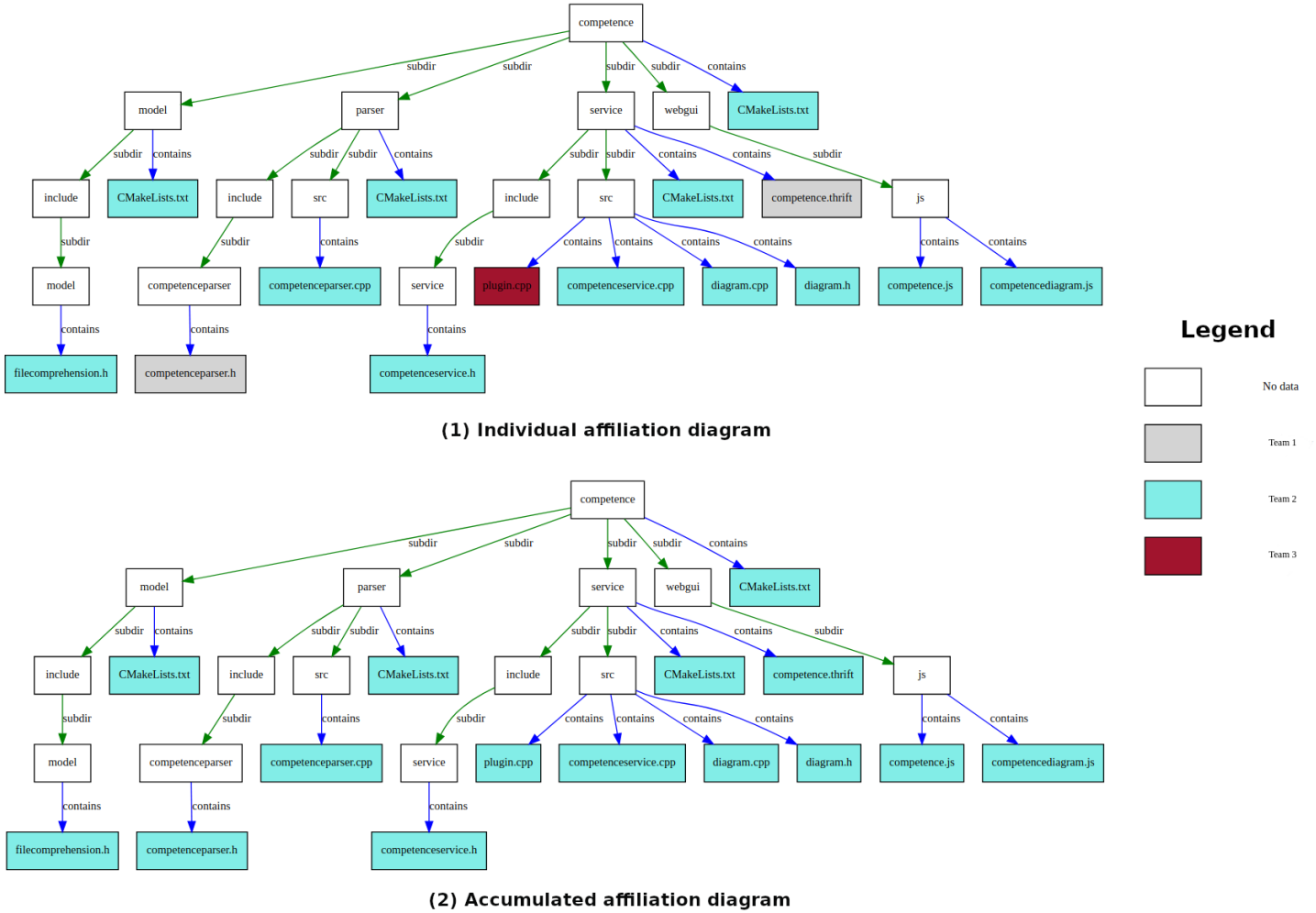

Figure 4: Comparison of the individual (1) and the accumulated (2) affiliation diagram for the source code of the competence plugin. (1) shows the affiliation of the most competent developer in a file, while (2) shows the most competent team in a file on average.

4.4 Coloring

In the personal view, node coloring is trivial, since a node is assigned a color on the scale from red to green depending on the corresponding competence percentage from 0% to 100% respectively. In the affiliation diagrams, each team is assigned a random color. However, node color generation in the team view is based on the corresponding email address. Email addresses are hashed, and the hash code is converted to a unique color.

5 Implementation

CodeCompass provides a stable core for a pluginable framework. The backend (parser and web server) is written in C/C++, while the frontend is written in JavaScript, using the *dojo.js* library. The competence plugin is implemented as an extension of CodeCompass as a new plugin. It relies entirely on version control data and supports only Git. Repository mining and data analysis is implemented using the *libgit2* API library. The graph-based visualizations are generated using *GraphViz* [10]. Since JPlag is capable of parsing several popular programming languages, we consider the plugin fairly language-independent, meaning that the competence calculation is applicable to all languages supported by JPlag.

6 Case studies

The plugin was tested on multiple long-running open-source projects: *Google Test*[3], *libgit2*[4], and *LLVM-Clang*[5] [29]. All of these projects are continuously developed with hundreds or even thousands of commits a month, and their developer teams consist of large numbers of programmers from several different companies. We also took CodeCompass itself[6] into the test projects, since we thoroughly know its history and structure, and it facilitated the evaluation of the results. The tests were run on an average personal computer with 16 GB RAM and 3 CPU cores.

Table 1 shows the results of parsing the test projects. All of their repositories contain hundreds, thousands, or even hundreds of thousands of commits. In order to provide easily comparable information, we parsed the latest few hundred commits of every project. Average execution time was calculated considering that 3 cores were used.

We can see significant difference in execution time, the number of modified files, and the number of committer developers between the projects. The most spectacular difference is the execution time of LLVM which can be measured in hours, compared to the other projects where hundreds of commits have been parsed

[3]GitHub repo: https://github.com/google/googletest
[4]GitHub repo: https://github.com/libgit2/libgit2
[5]GitHub repo: https://github.com/llvm/llvm-project
[6]The repository of CodeCompass was migrated from SVN to Git in 2016. The earlier version control data is not accessible.

Table 1: Competence parser results

Project	All commits	Parsed commits	Exec. time	Modified files	Devs
CodeCompass	955	500	8m 31s	553	15
Google Test	3,913	500	26m 21s	151	63
libgit2	14,550	500	92m 41s	353	33
LLVM-Clang	425,138	500	15h 34m	1340	159

in less than an hour. The cause of this phenomenon is that, although other test projects are also open-source and developed by numerous programmers, LLVM is still an edge-case compared to them; if we take a look at the LLVM's GitHub statistics, we can see that over 3000 commits are pushed to master during a single month, while other continuously developed projects are expanded by only a few hundreds of commits at most. What's more, the commits pushed to LLVM are large ones, affecting many files, and they frequently reach the size of a full value pull request[7], which means parsing one commit in this project means actually parsing several smaller commits. This also explains why the average execution time of one commit is significantly higher in LLVM.

Aside from the visualizations, the workspace database also provides a great deal of information about developers. Table 2 contains the answers to the developer-related questions asked above, "Who knows the most about the code?". In all test projects, less than a dozen developers can be named who are competent in larger parts of the code. This circumstance means increased risk from the aspect of the project's future. If any of the highly competent programmers leave the developer team for any reason, the software might suffer serious damage from a code and software quality perspective among others.

Table 2: Developer data of the test projects

Project	>25 files known	>50 files known	Most competent developer	Files known	Most competent team
CodeCompass	6	4	user1	399	Company1
Google Test	3	3	user2	87	N/A
libgit2	4	2	user3	200	N/A
LLVM-Clang	15	2	user4	107	Company4

[7] Commits on LLVM are mostly actual size pull requests where lots of commits are compressed into a patch file.

The "most competent developer[8]" at each software, while they possess a large amount of information and they are very important in the project, still have limited knowledge about the code. This can be considered a normal situation, since we cannot expect, even from a lead developer or an architect, to know everything about every component. The programmers with the highest amount of knowledge supposedly know the entire project thoroughly, and they are aware of all the important developer decisions.

We are also able to evaluate the results from the affiliations' aspect. If a project is developed by multiple teams or companies – which is usually the case in open-source projects – it might be an interesting and useful information to know which team is the most competent in certain parts of a software. As mentioned before, it is not easy to determine a programmer's affiliation by their commits only, but we can rely on their committer email addresses for some information. The competence parser is currently equipped with a list of international companies that give distinctive workplace email addresses to their employees, e.g. Apple (*apple.com*), Ericsson (*ericsson.com*), and Intel (*intel.com*). By running simple queries on the parsed data, we can identify the most competent company or team in a software project. In this case, "most competent" means the highest number of developers from a team that committed modifications to the project in the investigated time period. Of course, this value can be scaled by considering the amount and significance of the modifications.

Although the affiliation is obvious in some cases, some companies make their distinctive email domains accessible for non-employees as well, such as Google (*google.com*) and Yahoo (*yahoo.com*). This makes accurate analysis more difficult, since we cannot decide by a mere commit if the committer is an actual employee at one of these companies or not. This is why, even though the natural assumption is that e.g. Google is the most competent in Google Test, they are not clearly identifiable.

Another important information concerns the aforementioned risk factor of files that have not been modified in a long time, thus they are in danger of neglect. The risk of completely forgetting the purpose and content of these files gets significantly lower if their associated files are regularly maintained. However, this requires further deep analysis of the version control history.

6.1 Validation

The results of the plugin have been verified by applying the calculations on *CodeChecker*[9], an open-source source code analyzer which also applies static analysis to detect errors and malfunctions in programs. We conducted an experiment where we asked the developers of CodeChecker to evaluate their knowledge of the various modules of the project. The development team of CodeChecker consists of a small group of full-time developers and students and interns. The project is de-

[8] Personal names, email addresses and company names are anonymized due to privacy reasons.
[9] GitHub repo: https://github.com/Ericsson/codechecker

veloped in a multi-language environment with Python, C++ and JavaScript being the primary languages, which makes CodeChecker an excellent test project.

First, we determined the main modules of CodeChecker, then the entire commit history of more than 4800 commits was analyzed by the our method. Afterwards, the participating developers had to answer the question, *To what extent do you know the source code of this module: (module name)?*. The participants graded their knowledge in each module on a five-point scale, from 1 if they were unfamiliar with the module to 5 if they had detailed knowledge of the source code of the module. Each point on the scale corresponds to an interval of 20% understanding: 1 corresponds to 0-20% of source code knowledge, 2 to 20-40%, and so on.

After the analysis and the data collection from developers, we accumulated the results of competence calculation to match modules instead of files, and compared the results to the data given by developers. The experiment showed that the tool gave correct results in 48% of cases, with an 18% average deviation from the answers to our questions. The data indicated that the participants overestimated their knowledge in 50% of the time, and underestimated in 1.8%. The high value of overestimation suggests that developers on average have more knowledge of the source code than the commit history data shows. We also concluded that applying the results to modules instead of files could provide more useful results in everyday usage.

The average deviation of 18% indicates that the scale of proportions should be recalibrated to show more accurate and more detailed representation of knowledge in a software. We also concluded that the "unseen" knowledge which is inevitable for contribution should also be included in the method. More information of developer knowledge can be extracted from additional input, such as contributions in the project hosting system (e.g. GitHub, GitLab).

More details of the experiment and its results can be found in our previous paper [14].

6.2 Evaluation

Considering earlier studies about the possible aspects of data visualization tool evaluation [21, 24, 34, 43], we evaluated the usefulness of our tool can be evaluated based on the following criteria system:

Relevance: Most version control visualization tools focus on the actual source code (its architecture, evolution, change-proneness, etc.), and omits analyzing developer-related data (see Section 2 for examples). Our tool puts focus on collecting and visualizing information about the project-related knowledge of developers and teams, which can be used for developer-centered support within the development team.

Usability: The user interface of the plugin is integrated with the core frontend of CodeCompass. The functionalities (diagrams) are available through right-click menus on the source files, which is intuitive and easy to learn. The diagrams are generated by the graphic tools of GraphViz, which provides

a versatile and clean tool set for graph-based diagrams. However, the entire current frontend of CodeCompass is obsolete. User-friendliness will be improved by a new, modern web frontend in the near future.

Functionality: The plugin is capable of providing a broad overview of the most knowledgeable developers and teams in a software project, as well as mapping the overall familiarity of individuals with every source file. This information can be used by developers, team leaders, scrum masters, project owners, etc. to improve the efficiency of task distribution, and provide better support for less experienced team members. The extracted data leaves more space for improvement, as the plugin could offer support for further developer- and knowledge-related questions. For example, based on the frequency of modifications in a source file, and the number of active developers who have contributed to the file, the tool could calculate which files are in danger of forgetting in case some developers leave the team.

Scalability: In Table 1 we can see that 500 commits were parsed from each test repository. However, the tool was tested on the reopsitory of CodeChecker which contained more than 4800 commits at the time of testing. Furthermore, during the development of the plugin, we continuously executed testing on a larger (10000) set of commits from the repository of LLVM repository. LLVM receives hundreds of contributions every day which makes it an excellent test project because the commits include several edge cases and exceptions that had to be handled to guarantee secure operation. Thus, the plugin is capable of handling large repositories.

Performance: Table 1 shows the runtime of the tool on the four test projects. We can see that the larger and the more complex commits get in a project, the more time the plugin takes to analyze them. The performance of the plugin can be improved by using an extension or wrapper library instead of the raw libgit2 API. The API includes many memory issues that wrappers tackle which may improve performance and reduce runtime significantly.

Flexibility: The plugin itself can be easily switched on and off during the usage of CodeCompass. On development level, the skeleton of diagram generation is readily provided in case of implementing new features. However, in order to improve or extend parsing, the algorithm needs to be understood first. Currently, commits from one branch can be analyzed at one parse. If a user, for example, wants to compare branches, the parser component of the plugin must be extended for which the user has to understand the appropriate libgit2 API elements.

Integration: The plugin is integrated with CodeCompass. The user may provide the number of commits they wish to analyze, and include the repository with the source code. This way, the plugin can be easily integrated with continuous integration systems. Future work includes analyzing data from project hosting systems.

7 Threats to validity

Although token-based comparison of edited files guarantees high-level accuracy in our change analysis, there can be some distorting factors. Just because a programmer has committed to a file some time ago in the past, does not mean that they are still competent in the file in its current state. For example, what if the programmer wrote a big chunk of a file, but someone came a couple days later and entirely refactored it? The two programmers would get similar results for this file, but the deleted content or overlap of modifications can cause distortions in the results. Refining the calculations from file level to smaller units, like classes or functions may help eliminate such anomalies in the future.

Another question that might come up is also content-related: what if someone modifies every file in a project by adding some minor change, such as license or copyright information to the comments? This way, this person is granted to have some percent of comprehension for every file for a while, even though they have not contributed to the project in its function and might not know anything about the code at all. Fortunately, software similarity checkers take care of this problem with token-based comparison. Renaming a variable, reordering a file, or adding comments do not count as significant modifications, there are no semantic differences are detected between file versions.

A more positively distorting factor lies in the nature of programming, which is particularly present when a programmer makes changes to the previously written work of someone else, let that be debugging, maintenance or further development: programmers can hardly (correctly) contribute to code without understanding at least some of it. That means actual competence in a file is most likely higher than our calculations say it to be.

In our calculations, we focus on exclusively the objectively measurable data, the structural significance of modifications. However, expertise and knowledge calculations include more subjective human factors, such as the capability of memory retention of a developer. According to the work of Ebbinghaus [9], people tend to forget detailed information quite quickly after hearing or reading said information. The forgetting curve has been tested by making people read and remember unrelated information like random words. Program code consists of more tightly connected units to which the forgetting curve may not apply. Future research includes human factors in the competence calculation.

Another threat to validity is that certain code modifications which seem like serious modifications to a code similarity checker might not really require deep knowledge of the code in question. Trivial refactoring patterns such as the extract function [16] can be applied with minimal understanding of the purpose of the code. However, we might usually assume that such tasks are assigned to a person who can take responsibility of the target module.

8 Conclusion and future work

In this research, we have developed the competence plugin, a visualization tool which uses information obtained from version control repositories to make developer-related information accessible for the user. The plugin answers some frequently asked questions during development, such as "Who knows the most about this part of the code?" and "Who can I ask for help in this task?". Also, the tool is essential to better plan the use of human resources, e.g. detecting when the knowledge in some part of the code dropped below a certain threshold, or when the knowledge is unevenly distributed between developers. In such cases, the project management can apply preventive actions to avoid complete loss of information about certain parts of the code. Aggregated views can be constructed from individual developers' knowledge information, using various heuristics. We tested the plugin as part of the CodeCompass open-source code comprehension framework, on several long-term open-source software projects that are continuously developed with frequent commits in their repositories. Our study has shown that the plugin is an effective code comprehension supporting tool that is useful for individual developers and project teams as well.

The competence plugin will be further developed by implementing new visualizations focusing on other programmer-related questions and utilizing more information from the version control system. Mapping the comprehension visualizations to project dependency graphs could more effectively help the developer in the process of learning about the software in a more conscious way. In the future, we will develop the calculation to not only apply to files, but modules as well. We plan to evolve the available visualizations with more subtle coloring, mouse hover functionality, and some logical diversity in node shapes. We also plan to implement an interactive interface where the users can map parsed email addresses and affiliations to their user accounts, and fill in the missing user data.

References

[1] Alcocer, J. P. S., Beck, F., and Bergel, A. Performance evolution matrix: Visualizing performance variations along software versions. In *Proceedings of the 2019 Working Conference on Software Visualization (VISSOFT)*, pages 1–11. IEEE, 2019. DOI: 10.1109/VISSOFT.2019.00009.

[2] Alcocer, J. P. S., Jaimes, H. C., Costa, D., Bergel, A., and Beck, F. Enhancing commit graphs with visual runtime clues. In *Proceedings of the Working Conference on Software Visualization (VISSOFT)*, pages 28–32. IEEE, 2019. DOI: 10.1109/VISSOFT.2019.00012.

[3] Bao, L., Xing, Z., Xia, X., Lo, D., and Li, S. Who will leave the company?: A large-scale industry study of developer turnover by mining monthly work report. In *Proceedings of the 2017 IEEE/ACM 14th International Conference*

on Mining Software Repositories (MSR), pages 170–181. IEEE, 2017. DOI: 10.1109/MSR.2017.58.

[4] Bassil, S. and Keller, R. K. Software visualization tools: Survey and analysis. In *Proceedings of the 9th International Workshop on Program Comprehension*, pages 7–17. IEEE, 2001. DOI: 10.1109/WPC.2001.921708.

[5] Brunner, T. and Porkoláb, Z. Advanced code comprehension using version control information. *IPSI Transactions on Internet Research*, 16(2):47–54, 2020. URL: http://ipsitransactions.org/journals/papers/tir/2020jul/p7.pdf.

[6] Chotisarn, N., Merino, L., Zheng, X., Lonapalawong, S., Zhang, T., Xu, M., and Chen, W. A systematic literature review of modern software visualization. *arXiv preprint arXiv:2003.00643*, 2020. DOI: 10.1007/s12650-020-00647-w.

[7] da Silva, I. A., Mangan, M. A., and Werner, C. M. CVS Watch: A group awareness tool applied to collaborative software development, 2004. URL: https://www.researchgate.net/profile/Marco-Mangan/publication/266471809_CVS_Watch_a_Group_Awareness_Tool_Applied_to_Collaborative_Software_Development/links/55a79ec308aea2222c747c4f/CVS-Watch-a-Group-Awareness-Tool-Applied-to-Collaborative-Software-Development.pdf.

[8] de F Carneiro, G., Magnavita, R., and Mendonça, M. Combining software visualization paradigms to support software comprehension activities. In *Proceedings of the 4th ACM Symposium on Software Visualization*, pages 201–202. ACM, 2008. DOI: 10.1145/1409720.1409755.

[9] Ebbinghaus, H. *Über Das Gedachtnis*. 1885. URL: https://home.uni-leipzig.de/wundtbriefe/wwcd/opera/ebbing/memory/GdaechtI.htm.

[10] Ellson, J., Gansner, E., Koutsofios, L., North, S. C., and Woodhull, G. Graphviz—open source graph drawing tools. In *Proceedings of the International Symposium on Graph Drawing*, pages 483–484. Springer, 2001. DOI: 10.1007/3-540-45848-4_57.

[11] Elsen, S. Visgi: Visualizing GIT branches. In *Proceedings of the 2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–4. IEEE, 2013. DOI: 10.1109/VISSOFT.2013.6650522.

[12] Faragó, C. *Maintainability of Source Code and its Connection to Version Control History Metrics*. PhD thesis, Department of Software Engineering, University of Szeged, Hungary, 2016.

[13] Faragó, C., Hegedűs, P., Végh, A. Z., and Ferenc, R. Connection between version control operations and quality change of the source code. *Acta Cybernetica*, 21(4):585–607, 2014. DOI: 10.14232/actacyb.21.4.2014.4.

[14] Fekete, A., Cserép, M., and Porkoláb, Z. Measuring developers' expertise based on version control data. In *Proceedings of the 2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*, pages 1607–1612. IEEE, 2021. DOI: 10.23919/MIPRO52101.2021.9597103.

[15] Fittkau, F., Krause, A., and Hasselbring, W. Software landscape and application visualization for system comprehension with ExplorViz. *Information and Software Technology*, 87:259–277, 2017. DOI: 10.1016/j.infsof.2016.07.004.

[16] Fowler, M. *Refactoring*. Addison-Wesley Professional, 2018. URL: https://martinfowler.com/books/refactoring.html.

[17] Greene, G. J., Esterhuizen, M., and Fischer, B. Visualizing and exploring software version control repositories using interactive tag clouds over formal concept lattices. *Information and Software Technology*, 87:223–241, 2017. DOI: 10.1016/j.infsof.2016.12.001.

[18] Greene, G. J. and Fischer, B. Interactive tag cloud visualization of software version control repositories. In *Proceedings of the IEEE 3rd Working Conference on Software Visualization (VISSOFT)*, pages 56–65. IEEE, 2015. DOI: 10.1109/VISSOFT.2015.7332415.

[19] Greevy, O., Lanza, M., and Wysseier, C. Visualizing feature interaction in 3-D. In *Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–6. IEEE, 2005. DOI: 10.1109/VISSOF.2005.1684317.

[20] Hawes, N., Marshall, S., and Anslow, C. Codesurveyor: Mapping large-scale software to aid in code comprehension. In *Proceedings of the IEEE 3rd Working Conference on Software Visualization (VISSOFT)*, pages 96–105. IEEE, 2015. DOI: 10.1109/VISSOFT.2015.7332419.

[21] Isenberg, T., Isenberg, P., Chen, J., Sedlmair, M., and Möller, T. A systematic review on the practice of evaluating visualization. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2818–2827, 2013. DOI: 10.1109/TVCG.2013.126.

[22] Jermakovics, A., Sillitti, A., and Succi, G. Mining and visualizing developer networks from version control systems. In *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '11, pages 24—31, New York, NY, USA, 2011. Association for Computing Machinery. DOI: 10.1145/1984642.1984647.

[23] Kagdi, H., Yusuf, S., and Maletic, J. I. Mining sequences of changed-files from version histories. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, page 47–53, New York, NY, USA, 2006. Association for Computing Machinery. DOI: 10.1145/1137983.1137996.

[24] Kienle, H. M. and Muller, H. A. Requirements of software visualization tools: A literature survey. In *Proceedings of the 2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 2–9. IEEE, 2007.

[25] Krause-Glau, A., Bader, M., and Hasselbring, W. Collaborative software visualization for program comprehension. In *Proceedings of the 2022 Working Conference on Software Visualization (VISSOFT)*, pages 75–86. IEEE, 2022. DOI: 10.1109/VISSOFT55257.2022.00016.

[26] Kuhn, A., Erni, D., Loretan, P., and Nierstrasz, O. Software cartography: Thematic software visualization with consistent layout. *Journal of Software Maintenance and Evolution: Research and Practice*, 22(3):191–210, 2010. DOI: 10.1002/smr.414.

[27] Kumar J., M., Dubey, S., Balaji, B., Rao, D., and Rao, D. Data visualization on github repository parameters using elastic search and kibana. In *Proceedings of the 2018 2nd International Conference on Trends in Electronics and Informatics (ICOEI)*, pages 554–558, 2018. DOI: 10.1109/ICOEI.2018.8553755.

[28] Lanza, M., Ducasse, S., Gall, H., and Pinzger, M. Codecrawler: An information visualization tool for program comprehension. In *Proceedings of the 27th International Conference on Software Engineering*, pages 672–673, 2005. DOI: 10.1145/1062455.1062602.

[29] Lattner, C. and Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 75–86. IEEE, 2004. DOI: 10.1109/CGO.2004.1281665.

[30] Löwe, W., Ericsson, M., Lundberg, J., and Panas, T. Software comprehension-integrating program analysis and software visualization. *Software Engineering Research and Practice*, 2002. URL: http://arisa.se/files/LELP-02.pdf.

[31] Marcus, A., Feng, L., and Maletic, J. I. Comprehension of software analysis data using 3D visualization. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, pages 105–114. IEEE, 2003. DOI: 10.1109/WPC.2003.1199194.

[32] Mattila, A.-L., Ihantola, P., Kilamo, T., Luoto, A., Nurminen, M., and Väätäjä, H. Software visualization today: Systematic literature review. In *Proceedings of the 20th International Academic Mindtrek Conference*, pages 262–271, 2016. DOI: 10.1145/2994310.2994327.

[33] Merino, L., Ghafari, M., Anslow, C., and Nierstrasz, O. CityVR: Gameful software visualization. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 633–637. IEEE, 2017. DOI: 10.1109/ICSME.2017.70.

[34] Merino, L., Ghafari, M., Anslow, C., and Nierstrasz, O. A systematic literature review of software visualization evaluation. *Journal of Systems and Software*, 144:165–180, 2018. DOI: 10.1016/j.jss.2018.06.027.

[35] Merino, L., Seliner, D., Ghafari, M., and Nierstrasz, O. Communityexplorer: A framework for visualizing collaboration networks. In *Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies*, pages 1–9, 2016. DOI: 10.1145/2991041.2991043.

[36] Mierle, K., Laven, K., Roweis, S., and Wilson, G. Mining student CVS repositories for performance indicators. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, page 1–5, New York, NY, USA, 2005. Association for Computing Machinery. DOI: 10.1145/1083142.1083150.

[37] Nagappan, N., Ball, T., and Zeller, A. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, page 452–461, New York, NY, USA, 2006. Association for Computing Machinery. DOI: 10.1145/1134285.1134349.

[38] Oberhauser, R. VR-Git: Git repository visualization and immersion in virtual reality. In *Proceedings of the the Seventeenth International Conference on Software Engineering Advances*, pages 9–14, 2022. URL: https://www.thinkmind.org/index.php?view=article&articleid=icsea_2022_1_20_10032.

[39] Porkoláb, Z., Brunner, T., Krupp, D., and Csordás, M. Codecompass: An open software comprehension framework for industrial usage. In *Proceedings of the 26th Conference on Program Comprehension*, pages 361–369, 2018. DOI: 10.1145/3196321.3197546.

[40] Prechelt, L., Malpohl, G., Philippsen, M., et al. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, 8(11):1016–1038, 2002. URL: https://pdfs.semanticscholar.org/6281/93dbaa4b88101b8d7dd0a7c2eee86af5e32c.pdf.

[41] Shahin, M., Liang, P., and Babar, M. A. A systematic review of software architecture visualization techniques. *Journal of Systems and Software*, 94:161–185, 2014. DOI: 10.1016/j.jss.2014.03.071.

[42] Shinyama, Y., Arahori, Y., and Gondow, K. Analyzing code comments to boost program comprehension. In *Proceedings of the 2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 325–334, 2018. DOI: 10.1109/APSEC.2018.00047.

[43] Shneiderman, B. and Plaisant, C. Strategies for evaluating information visualization tools: Multi-dimensional in-depth long-term case studies. In *Proceedings of the 2006 AVI workshop on Beyond time and errors: Novel evaluation methods for information visualization*, pages 1–7, 2006.

[44] Slater, J., Anslow, C., Dietrich, J., and Merino, L. CorpusVis–Visualizing software metrics at scale. In *Proceedings of the 2019 Working Conference on Software Visualization (VISSOFT)*, pages 99–109. IEEE, 2019. DOI: 10.1109/VISSOFT.2019.00020.

[45] Svitkov, S. and Bryksin, T. Visualization of methods changeability based on VCS data. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 477–480, 2020. DOI: 10.1145/3379597.3387451.

[46] Teyseyre, A. R. and Campo, M. R. An overview of 3D software visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(1):87–105, 2008. DOI: 10.1109/TVCG.2008.86.

[47] Wettel, R. and Lanza, M. Visualizing software systems as cities. In *Proceedings of the 2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 92–99. IEEE, 2007. DOI: 10.1109/VISSOF.2007.4290706.

[48] Wettel, R. and Lanza, M. Codecity: 3D visualization of large-scale software. In *Companion of the 30th International Conference on Software Engineering*, pages 921–922, 2008. DOI: 10.1145/1370175.1370188.

[49] Williams, C. C. and Hollingsworth, J. K. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31(6):466–480, 2005. DOI: 10.1109/TSE.2005.63.