Information Visualization in Software Maintenance and Evolution

Marjane Namavar

Abstract— Software visualization is concerned with the static visualization as well as the animation of software artifacts, such as source code, executable programs, and the data they manipulate, and their attributes, such as size, complexity, or dependencies. Software visualization techniques are widely used in the areas of software maintenance and evolution, where typically large amounts of complex data need to be understood and a high degree of interaction between software engineers and automatic analyses is required. The present work provides a survey of 23 software visualization systems in the field of software maintenance and evolution and infers a categorization based on 5 main classes including task, data, data processing, representation and availability. The results of this survey help to ascertain the current role of software visualization in software engineering from the perspective of researchers in these domains and give hints on future research avenues.

Index Terms—information visualization, software visualization, software maintenance, software evolution

Introduction

Extending Roman and Cox's [1] definition of program visualization to other software artifacts, software visualization can be defined as the mapping from software artifacts including programs to graphical representations. Software visualization is needed because software is invisible. In the simplest case., we may visualize artifacts textually. More advanced graphical visualization techniques promise to better help understanding software, usually in combination with techniques that raise the level of abstraction, reduce the amount of information to what is needed to perform the task at hand, or to ease browsing the large information space. However, whether graphical representation is really superior to textual representation is rarely proven empirically. There are empirical studies that show evidence that specific ways of graphical visualization work better than textual visualization for certain tasks [2]. In other cases, a textual presentation is likely to be the most appropriate [3, 4]. Nevertheless, many researchers believe in the value of software visualization. In particular, in the domains of software maintenance and evolution, where typically large amounts of complex data need to be understood, software visualization may play an important role. We know from empirical studies that maintenance programmers spend 50% of their time simply to understand the software to be changed [5] and it is plausible that visualization has a substantial positive effect on the time needed to comprehend large programs.

There are some main questions in this area. What are the perspectives of researchers in software maintenance and evolution visualization? What is visualized and how? What types of purposes are the main ones to apply visualization in this field? Could the developed visualization systems be accessible by a lot of users for large projects? What kinds of pre-processing methods are more popular? In order to find answers to these open questions I conducted a survey. The results of this survey are presented here and help to gain an overview of the usage of software visualization in software maintenance and evolution. The goals of this project are to review the existing literature focusing on the use of visualization for software evolution and maintenance. Then, analyzing the data from empirical experiments under a certain framework. Finally, abstract gathered information to categorize existing approaches.

This paper is organized as follows. Section 1 overviews related work. Section 2 is a brief background. Section 3 presents our classification model. Section 4 presents the analysis of cases. Section 5 discusses our results. Section 6 concludes the paper.

1 Related work

There are a limited number of survey papers that are close to this subfield. I have reviewed some survey papers in a broader field of software engineering and a few ones in similar subfields. Price et al. [6] compared 12 tools against 6 desirable features categories: scope, content, form, method, interaction and effectiveness. The tools were however not related to a single application area. Maletic et al. [7] compared 5 software tools along 5 axes: task, audience, target, representation, and medium. Similar to Price et al., the scope of this taxonomy and tools is quite broad. Since a tool's audience strongly depends on its purposes [7], evaluating similar-purpose tools would be more insightful [8]. Here, Storey et al. [9] compared 12 tools that provide awareness of human activities during software development against the categories of intent, information, presentation, interaction and effectiveness. In corrective maintenance, Baecker et al. analyzed three classes of techniques used for debugging [10]: animation, improved typographic representations, and error sonification. Sensalire et al. evaluated ten general-purpose software-understanding tools[11]. None of the above surveys are exactly in the field of software maintenance and evolution. Also this project is different from previous work in the way it analyzes all systems under what/why/how framework systematically.

2 Background

Software evolution is an important topic in software engineering and is referred to as the process of developing software initially, then repeatedly updating it for various reasons. Software Maintenance is also the modification of a software product after delivery to correct faults or improve performance. Both of them generally deal with large amounts of data, as one must look at whole project histories as opposed to their current snapshot. Software visualization is the field of software engineering that aims to help people to understand software through the use of visual resources. It can be effectively used to analyze and understand the large amount of data produced during software evolution and maintenance.

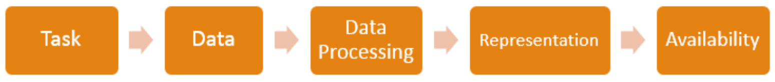


Fig. 1. Five main categories.

3 Classification

Figure 1 shows 5 main categories of this project.

3.1 Category A: Task

Visualization system mainly aim to accomplish following tasks.

Help to detect code smell: Code smells signal bad programming, design, or code, and are often used to drive refactoring. Some visualization systems help with finding code smells either directly or indirectly.

Help to analyze execution of the program: Traces are defined as data gathered during a program's execution. Some visualization systems are able to generate or show traces.

Help to perform debugging: Debugging is the process of finding and resolving defects or problems within a computer program that prevent correct operation of computer software or a system. Some visualization systems do basic debugging, e.g. breakpoints, code stepping, and simple text watches or they might show higher-level debugging facts, e.g. bug or test case data.

Help to analyze user feedback: User feedback is defined by all the information you get from your customers about whether or not they are satisfied with your product or service. Feedback could be in the shapes of ratings or comments. Some visualization systems help to organize and make sense out of a large amount of user feedback.

Help to monitor code changes: Developers continuously apply changes to add features or fix bugs. A lot of visualization systems try to illustrate these changes and various aspects of them.

Help to monitor developer activities: With the invent of version control systems every single activity of developers is recorded with detailed information. Also, the version of a project at the time of a specific activity is available. A lot of visualization systems shows interactions between developers and their contribution to a project.

Help to track bug reports: A bug report should explain how exactly the product is broken and there are some tools for bug tracking. Some visualization systems exploit this information from issue-trackers to produce insight into the whole bug reporting process.

Help to comprehend the structure of a program: A computer program has different components and when a project becomes very large, the comprehension of the program's structure will be more difficult. Some visualization systems try to visualize the structure of a program and make this process easier and faster.

3.2 Category B: Data

These are main data types that are input to the visualization systems.

Source code: The actual textual content of the program

Packages: Components of the program at the package level

Classes, objects, interfaces: Components of the program at the class level

Functions: Components of the program at the function level

Test suite and results: The content of a test suite or the result of running test cases

Bug report: The information needed to report and track a bug including text and different states

Events & sequences: A chain of occurrences to be tracked

Relationships between code components: Code components have a variety of relationships together such as containment or calling each other

User feedback: The feedback of user in the shape of ratings or comments

Metadata: Extra information such as revision data

3.3 Category C: Data Processing

Visualization systems apply following methods to derive data from raw input data.

Abstract Syntax Tree: is a tree representation of the abstract syntactic structure of source code written in a programming language. It often serves as an intermediate representation of the program that the visualization system needs to understand the structure of a program.

NLP Methods: NLP methods are concerned with the interactions between computers and human languages, in particular how to program computers to process and analyze large amounts of natural language data. Visualization systems might use them to extract information from textual content written by humans.

Static Analysis: Static program analysis is the analysis of computer software that is performed without actually executing programs. Visualization systems use it to analyze program and infer desired information.

Dynamic Analysis: Dynamic program analysis is the analysis of computer software that is performed by executing programs on a real or virtual processor. If the visualization system needs information that would be obtained during the run-time of a program, then uses dynamic analysis.

3.4 Category D: Representation

Visualization systems take advantage of a wide-range of techniques to illustrate data and they might/might not display the actual textual content corresponding to the input data being visualized.

Technique: Various techniques have been used from different channels and charts to a variety of reducing, manipulation and facet methods.

Textual Content: All software artifacts have a corresponding textual representation. It indicates whether the visualization system shows the actual textual content of the input data (whatever it is) being visualized or not.

3.5 Category E: Availability

Availability means whether the developed visualization system could be used for a large software project or not and if it is capable of being integrated into an IDE so that it's more available to the users.

Scalability: It demonstrate whether the input data (whatever it is) fed to the visualization system, could be scaled to a size so that it's one million time as large as before.

Integration: Whether authors explicitly mention that their visualization system has been integrated into an IDE or not.

4 Surveyed cases

For reviewing existing literature 23 papers were gathered. All of them were from VISSOFT conference under maintenance and evolution topics between 2003 and 2019. Only design study papers which have a visualization system with an assigned task as their central part, were chosen. The next 23 subsections provide a summarized description and a table for each paper. The table shows the analysis of a paper through our 5 main categories. The base of this analysis is the what/why/how framework from [12]. Data, task and techniques are basically what, why, how parts of the framework respectively. Each case analysis starts by an image of the main view, then a brief description and finally a table containing the analysis information.

4.1 Case 1: A Closer Look at Bugs

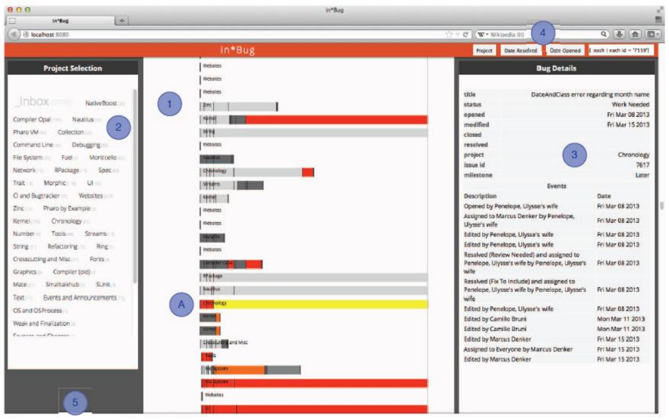


Fig. 2. Case1 main view

The main view of the visualization system developed in this paper [13] is shown in figure 2. In this paper in*Bug has been introduced which is a web-based software visual analytics platform to visualize bug reports. It allows users to navigate and inspect the vast information space created by bug tracking systems, with the goal of easing the understanding of bug reports in detail and also obtain an understanding of how bugs are reported with respect to one system or to an entire software ecosystem. Projects are taken from the bug tracker FogBugz. Bugs possess complicated life cycles, which makes them non-trivial to comprehend. Modern non-trivial software projects use bug tracking systems (also known as bug trackers), such as Jira and BugZilla, to manage the bugs that are reported. The repositories created by such bug trackers are a valuable source of information.

Bug lifetime panel depicts the bugs contained in the bug repository, showing their duration (as a horizontal stacked bar chart) and status (using different colors). Each bug tracking system proposes a set of statuses that an issue can acquire. These statuses are grouped into 5 categories with assigned color codes. For example, orange means the bug is still open but grey means it's closed. In the project selection panel users can pick the projects whose bugs they are interested in. Details panel provides the textual content of a bug report and all the information reported about the bug report under focus in the bug lifetime panel. Filter and options panel allows the user to sort and filter bugs. Status bar shows a quick status of the application. It displays the total number of the bugs in the repository, as for the number of bugs currently selected.

Table 1. Case1 analysis

Task	Track bug reports	
Data	Bug Report Metadata (information related to bug report) Events and Sequences (events in one bug's life cycle)	
Representation	Technique	Encode: Color, Stacked bar chart Reduce: Filter Facet: Partition into multiform views, Linked highlighting, Linked navigation, Overview-detail, Juxtaposed views Manipulate: Select, Sort
	Textual Content	Y
Availability	Scalability	Y
	Integration	N

4.2 Case 2: ClonEvol: Visualizing Software Evolution with code Clones

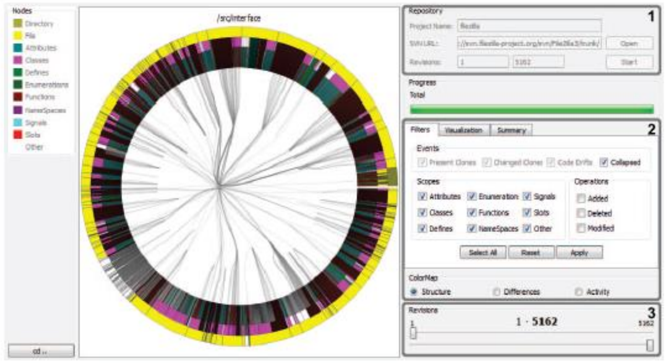


Fig. 3. Case2 main view

The main view of the visualization system developed in this paper [14] is shown in figure 3. The tool ClonEvol, that has been introduced in this paper assists in obtaining insight into the state and evolution of a C/C++/Java code base on project, file and scope level. This is achieved by combining information obtained from the software versioning system and contents of files that have changed between versions. More precisely, the tool combines the version change-logs with static analysis (of file contents) and clone detection. The consolidated information is presented to the user in a visual and interactive manner. The focus of the presented tool lies on scalability (in time and space) concerning data acquisition, data processing and visualization, and ease of use. The visualization is achieved with a mirrored radial tree to show the file and scope structures, complemented with edges that indicate the clone relations. Different hierarchies have different colors. Users can scroll through time to search for events of interest, which are highlighted by the structure color-map. For visualization a radial tree view is chosen as it can preserve the space needed for visualization. The nodes of the radial tree represent the file-scope hierarchy and the edges show clone relations. A node can be expanded as root of the visualization, to allow investigation of fine-grained details, e.g. clones between functions. The structure color-map shows object types in the code base (files, classes, functions) and the existing clones. It is used to help the user understand the visualization of the project. The right panel offers options for selection and filtering to control the visualization.

Table 2. Case2 analysis

Task	Monitor developer activities Comprehend the structure of a program	
Data	Source code Classes, objects, interfaces Functions Metadata (version and clone information) Relationships between code components	
Data (derived)	FileTree, ScopeTree	
Data Processing	Static analysis	
Representation	Technique	Encode: Radial tree with nodes and links, Color Reduce: Filter, Aggregation Facet: Overview-detail Manipulate: Select, Zoom and pan
	Textual Content	N
Availability	Scalability	Y
	Integration	N

4.3 Case 3: CVSscan: Visualization of Code Evolution

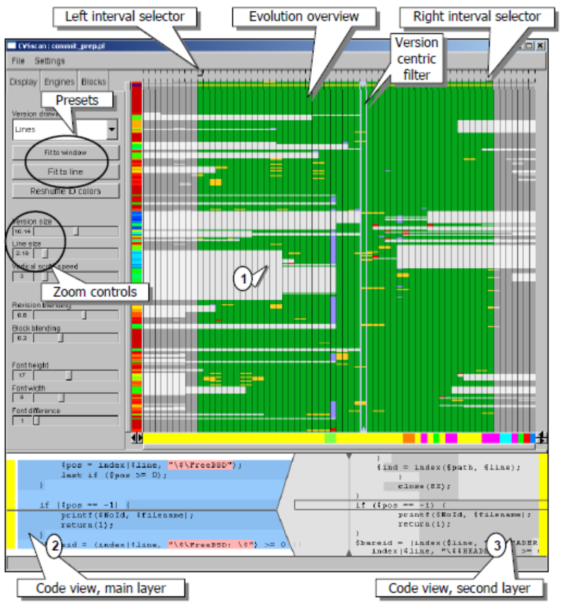


Fig. 4. Case3 main view

The main view of the visualization system developed in this paper[15] is shown in figure 4. During the life cycle of a software system, the source code is changed many times. This paper studies how developers can be enabled to get insight in these changes, in order to understand the status, history and structure better, as well as for instance the roles played by various contributors. This paper presents CVSscan, an integrated multiview environment for this. Central section is a line oriented display of the changing code, where each version is represented by a column, and where the horizontal direction is used for time, Separate linked displays show various metrics, as well as the source code itself. A large variety of options is provided to visualize a number of different aspects. The code view offers a text look at the code. Users can select the code to be displayed by sweeping the mouse in the evolution view. Vertical brushing in the code evolution area scrolls through a version's code, whereas horizontal brushing over the line-based layout goes through a given line's evolution. While the first layer (A) freezes when the user brushes over an empty region in the evolution view, the second layer (B) pops-up, and scrolls through the code that has been deleted, or will be later inserted at the mouse location. This creates a smooth feeling of scrolling continuity during brushing. In the same time, it preserves the context of the selected version (layer A) and gives also a detailed, text level peek, at the code evolution (layer B). Also, users will have several options to filter and manipulate data which is being visualized through the left panel.

Table 3. Case3 analysis

Task	Monitor developer activities Monitor code changes	
Data	Source code Events and Sequences (sequence of commits) Metadata (<id,author,date,code> for each version)	
Data (derived)	Line position, Line status	
Data Processing	Static analysis	
Representation	Technique	Encode: Dense layout, 2D matrix, Color, Position Reduce: Filter Facet: Partition into multiform views, Juxtaposed views, Linked highlighting, Linked navigation, Overview–detail Manipulate: Select, Zoom and Pan
	Textual Content	Y
Availability	Scalability	Y
	Integration	Y

4.4 Case 4: E-Quality: A Graph Based Object Oriented Software Quality Visualization Tool

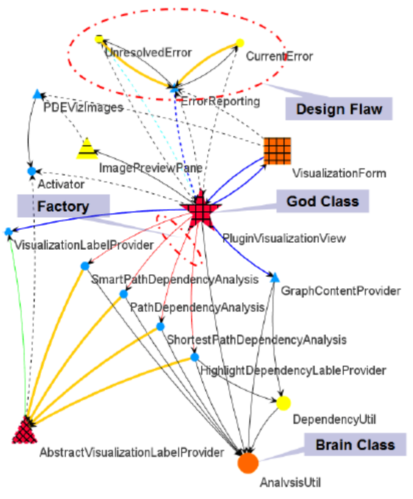


Fig. 5. Case4 main view

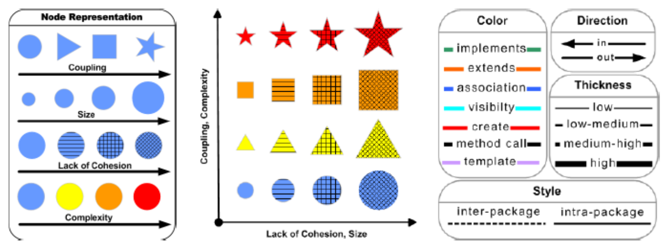


Fig. 6. Case 4 metrics

The main view of the visualization system developed in this paper[16] is shown in figure 5. In this paper, a graph based object-oriented software quality visualization tool called "E-Quality" has been introduced. E-Quality automatically extracts quality metrics and class relations from Java source code and visualizes them on a graph-based interactive visual environment. This visual environment effectively simplifies comprehension and refactoring of complex software systems. In E-Quality tool, software is represented as a weighted labeled digraph G(V, E) where V is the set of vertices that corresponds to software classes; E is the set of edges that corresponds to the different types of relations between these classes. Weights of edges are indicating the strength of relations and labels are used to identify names of classes and relations. Visual properties of a node in a graph are its color, size, shape, and fill pattern. Each quality attribute is mapped to different physical properties of a node. The user can change mapping scheme by defining different metric query statements. Main visual properties of an edge in a graph are direction, thickness, style, and color. E-Quality computes quality metrics such as cohesion, coupling and complexity. Figure 6 illustrates the various metrics and assigned channels to them. E-Quality allows filtering out some classes and relations by their name or specific attributes. The tool has an interactive graph drawing window, which allows many navigation properties such as graph editing, rotation, focusing, zooming, etc. The user can modify relation type colors in order to highlight specific types. All configurable properties, parameters, and graph models can be saved in XML format to allow further analyzing by other tools.

Table 4. Case 4 analysis

Task	Comprehend the structure of a program	
Data	Classes, objects, interfaces Relationships between code components Metadata (size)	
Data (derived)	Coupling, cohesion, complexity	
Data Processing	Static analysis, AST	
Representation	Technique	Encode: Graph, Shape, Size, Color, Texture Reduce: Filter, Aggregation Manipulate: Select, Zoom and Pan
	Textual Content	N
Availability	Scalability	N
	Integration	N

4.5 Case 5: Evo-Clocks: Software Evolution at a Glance

The main view of the visualization system developed in this paper[17] is shown in figure 7. Evo-Clocks, the tool developed in this paper, simultaneously visualizes software structure and the evolution of its individual components. It uses localized, evolutionary representations of individual artifacts embedded within a multi-revision representation

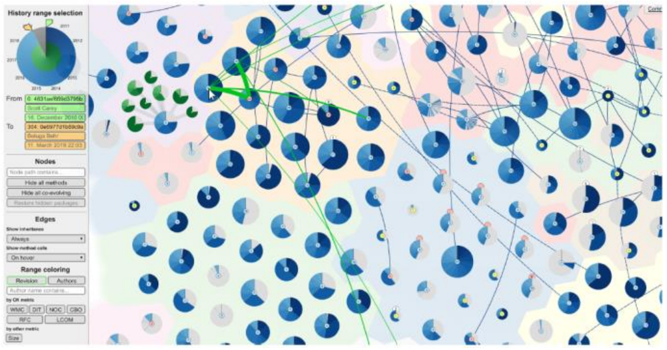


Fig. 7. Case 7 main view

of structure that has been merged into a single view. Selective information hiding, facilitated by simple effective selection and filtering features, and means to further explore hidden details, enable the observer to learn more about specific time ranges and metrics relevant to an exploratory task. A clock is really just a pie chart where different sectors represent different times in the history of the project, just like with a regular clock. If all revisions are displayed, the history begins with the oldest revision at the 12 o'clock position and runs clock-wise. Around the circle reaching the latest revisions upon returning to the 12 o'clock position approaching from the left. Classes are rendered as 1st-level nodes and methods as 2nd-level nodes (which are initially hidden). All nodes are grouped by their containing package. Size is determined from the number of lines of code. They currently use three kinds of links. Thin, blue lines indicate inheritance. Thick, transparent bands connect 2nd-level nodes to their owning 1st-level nodes. Finally, thin, green lines indicate outgoing method calls. To avoid information overload, method calls are only shown when the user hover over a calling node by default. 1st-level nodes belong to the same group to be clustered together. By default, clock sectors of 1st-level nodes (which are drawn by default) are colorized in shades of blue and 2nd-level nodes (which are hidden by default) in shades of green, darker shades representing older revisions and lighter shades representing newer ones. The user can choose to view only a specific period in the whole history of the project.

Table 5. Case 5 analysis

Task	Comprehend the structure of a program Monitor code changes	
Data	Classes, objects, interfaces Functions Relationships between code components Metadata (such as the author of a revision) Events and sequences (sequences of revisions)	
Data (derived)	Code metrics such as WMC, DIT, NOC, CBO, RFC, LCOM	
Data Processing	Static analysis, AST	
Representation	Technique	Encode: Pie chart, Hue, Size, Angle, Saturation Reduce: Filter, Aggregation Facet: Partition into multiform views, Juxtaposed views, Linked highlighting, Linked navigation, Overview–detail, Pop-up view Manipulate: Select
	Textual Content	N
Availability	Scalability	N
	Integration	N

4.6 Case 6: Exploring the Evolution of Software Quality with Animated Visualization

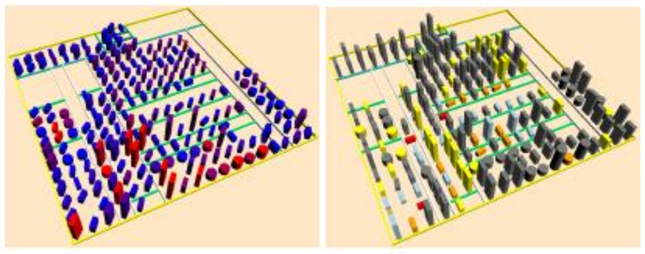


Fig. 8 case 6 main view

The main view of the visualization system developed in this paper[18] is shown in figure 8. An approach to quickly investigate programs composed of thousands of classes, over dozens of versions. Programs and their associated quality characteristics for each version are graphically represented and displayed independently. Their solution proposes to use visualization as a semi-automatic approach to analyze the quality of programs over many versions. Classes are represented as 3D boxes arranged over a 2D plane. Interfaces are differentiated from classes by using cylinders. A set of graphical characteristics are mapped to metrics: a color scale from blue to red or a set of discrete colors to represent nominal data, the box's height, and the box's rotation around the up axis. This visualization system uses following associations for metrics and graphical characteristics: color and coupling, twist and cohesion, size and height. The camera rotates on an hemisphere, can smoothly move the center of the hemisphere, as well as zoom in and out. The camera is always pointing toward the layout plane to prevent confusion. Users can also directly access the metrics numerical values or the code itself by clicking on a given class. Another mode allows users to click on a class to fetch information about its relationships. Instead of drawing links between entities, the saturation of classes not concerned by relationships is reduced. To transform from on version to another, the view animates smoothly.

Table 6. Case 6 analysis

Task	Comprehend the structure of a program Monitor code changes	
Data	Source code Classes, objects, interfaces Relationships between code components Metadata (version information)	
Data (derived)	Code metrics such as CBO, WMC, LCOM	
Data Processing	Static analysis, AST	
Representation	Technique	Encode: Shape, Hue, Saturation, Angle, Position, Size, Treemap, Animation Reduce: Filter Facet: Overview–detail, Pop-up view Manipulate: Select, Zoom and pan
	Textual Content	Y
Availability	Scalability	Y
	Integration	N

4.7 Case 7: FAVe: Visualizing User Feedback for Software Evolution

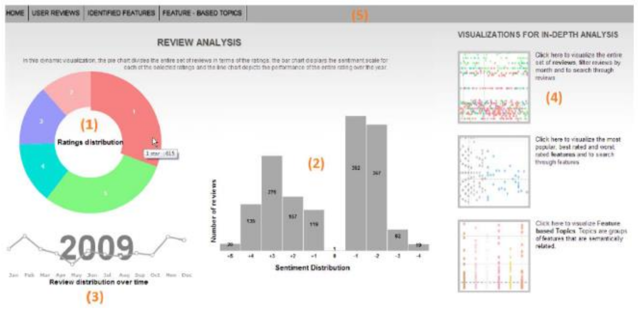


Fig. 9. Case 7 main view

The main view of the visualization system developed in this paper[19] is shown in figure 9. The visualization system developed in this paper is an interactive user feedback visualization which displays app reviews from four different points of view: general, review based, feature based and topic-feature based. To generate the data displayed by FAVe Natural Language Processing techniques have been used. The home screen of FAVe is a simple interactive dashboard. It provides a dynamic visualization of the user reviews in terms of star ratings, user sentiment associated with each review and a cumulative rating performance over the entire year. The interactive pie chart(1) shows the overall distribution of the app's ratings, in terms of the number of stars given in the user reviews. When clicking on the different ratings shown in the pie chart, the rest of the graphs in the home screen are updated to reflect the information about the selected pie chart rating. The sentiment bar graph(2) depicts the overall user sentiments of all reviews. The line graph(3) shows the month-wise distribution of all reviews. Fine-grained visualizations overview(4) provides an overview of the three different types of finer-grained user feedback views: review based, feature based and feature topic based. Hovering the mouse over each image, enlarges it, allowing the user to get a more detailed view. The navigation menu(5) provides some selection/filtering options.

Table 7. Case 7 analysis

Task	Analyze user feedback	
Data	User feedback	
Data (derived)	Sentiment score, app features, feature-based topics in app	
Data Processing	NLP	
Representation	Technique	Encode: Pie chart, Scatter plot, Hue, Brightness, Bar chart Reduce: Filter, Aggregation Facet: Partition into multiform views, Juxtaposed views, Linked highlighting, Linked navigation, Overview–detail Manipulate: Select, Search
	Textual Content	Y
Availability	Scalability	Y
	Integration	N

4.8 Case 8: GETAVIZ: Generating Structural, Behavioral, and Evolutionary Views of Software Systems for Empirical Evaluation

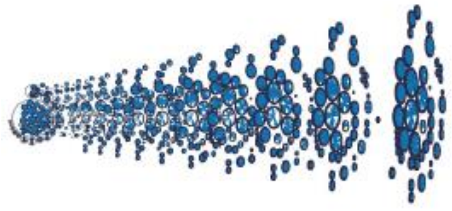


Fig. 11. Case 8 RD metaphor

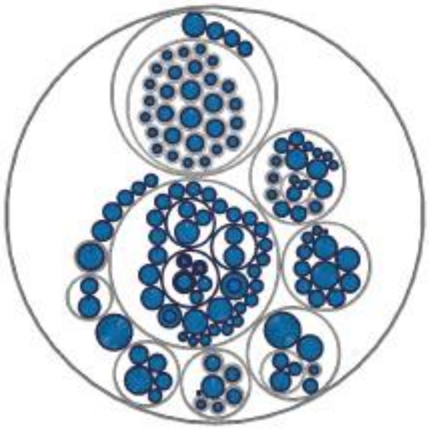


Fig. 10. Case 8 main view

The main view of the visualization system developed in this paper[20] is shown in figure 10. The Recursive Disk(RD) metaphor displays all important structural aspects of software system including packages, classes, methods, and attributes. The glyphs used by the metaphor shown in figure 11 are nested disks, rings, and segments. The nesting represents the containment relations of the visualized software entities. It shows all elements occurring in the depicted evolution period of the software system. The size of the glyphs in the base layout is determined by the maximal size of the represented entities. The versions of the visualized software system are positioned above the base layout. Only the entities occurring in the respective version of the software system are shown in the visualization. The size of the glyphs of the versions is determined by the size of the element in the depicted version. By clicking on every component, a pop-up view will be opened that presents metadata and source code corresponding to that component. By double-clicking on a plane associated with a version, a pop-up view will appear that gives information about that version.

Table 8. Case 8 analysis

Task	Monitor code changes	
Data	Packages Classes, objects, interfaces Functions Relationships between code components Metadata (version information)	
Representation	Technique	Encode: Pie chart, Size, Containment, Position Facet: Superimposed views, Juxtaposed views, Overview-detail, Pop-up view Manipulate: Select, Zoom
	Textual Content	Y
Availability	Scalability	N
	Integration	N

4.9 Case 9: Multiscale and Multivariate Visualizations of Software Evolution

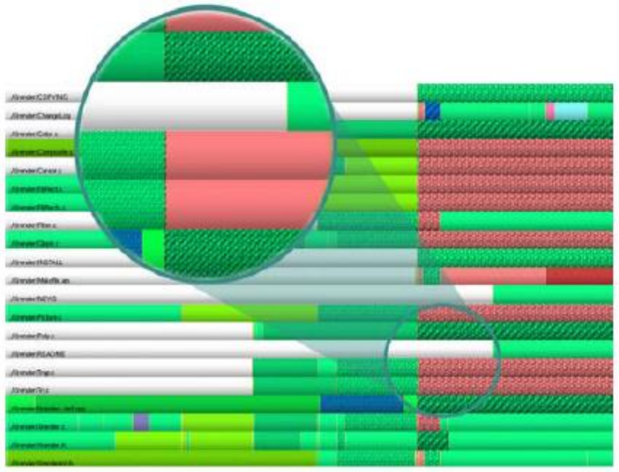


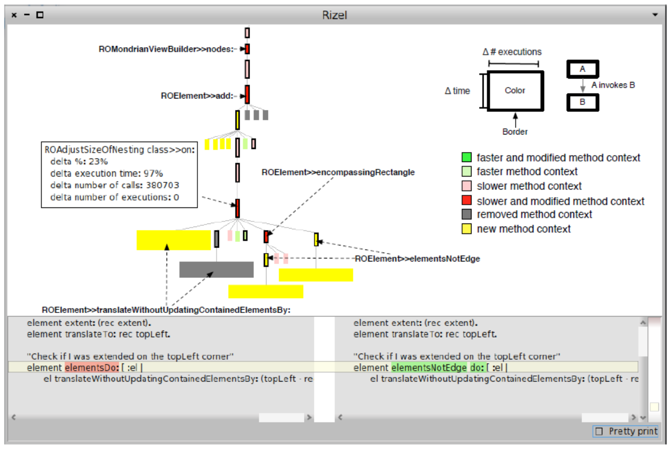
Fig. 12. Case 9 main view

The main view of the visualization system developed in this paper[21] is shown in figure 12. This paper addresses two aspects of the problem of coping with the large software size to be visualized: many data elements (e.g. files and file versions in a repository) and many attributes per element (e.g. file size, type, and author, and commit time and comments). They address the first problem by using a multiscale (or hierarchical) software decomposition and a new visual widget for displaying this hierarchy and letting users choose from its relevant levels of detail. They address the second problem by a new visual approach that enables complex visual correlations over multivariate data Each file is depicted along a time horizontal axis as a sequence of segments. Each segment shows one file version. The version creation time and the duration decide the position of the segment in the sequence and its length. The segment color shows version attributes, e.g. author ID, or functions defined on attributes, e.g. code size. To build complete visualizations of software evolution, individual file representations on the vertical axis are stacked so they share the same time scale, and use the same color encoding. Users can select attribute(s) they would like them encoded and create a visualization of their own. For example, figure 12 shows an example of visualizing several attributes. Here, bubble patterns are used to indicate revisions belonging to a given system release, and a diagonal hatch pattern for files containing the word ‘tag’ in their commit logs. Color shows author ID. Files can be easily recognized if they belong to the selected release and contain the word ‘tag’. By clicking on each segment, a pop-up window shows source code and detailed information about that segment.

Table 9. Case 9 analysis

Task	Monitor code changes Comprehend the structure of a program	
Data	Source code Events and sequences (sequence of revisions) Metadata (version information)	
Data (derived)	Files containing word “tag”	
Data Processing	Static analysis	
Representation	Technique	Encode: Stacked bar chart, Color, Position, Size, Texture Facet: Juxtaposed views, Pop-up view Manipulate: Select
	Textual Content	Y
Availability	Scalability	Y
	Integration	N

4.10 Case 10: Performance Evolution Blueprint: Understanding the Impact of Software Evolution on Performance


Rizel
ROMondrianViewBuilder>>nodes:
ROElement>>add:
ROElement>>encompassingRectangle
ROElement>>elementsNotEdge
ROElement>>translateWithoutUpdatingContainedElementsBy:
Δ # executions
Δ time
Color
Border
A
A invokes B
B
Pretty print

🟩 faster and modified method context
🟩 faster method context
🟥 slower method context
🟥 slower and modified method context
⬛ removed method context
🟨 new method context

```
ROAdjustSizeOfNesting class>>on:
  delta %: 23%
  delta execution time: 97%
  delta number of calls: 380703
  delta number of executions: 0
```

```
element extent: (rec extent).
element translateTo: rec topLeft.

"Check if I was extended on the topLeft corner"
element elementsDo: [ :el |
   el translateWithoutUpdatingContainedElementsBy: (topLeft - re
```

```
element extent: (rec extent).
element translateTo: rec topLeft.

"Check if I was extended on the topLeft corner"
element elementsNotEdge do: [ :el |
   el translateWithoutUpdatingContainedElementsBy: (topLeft - re
```

Fig. 13. Case 10 main view

The main view of the visualization system developed in this paper[22] is shown in figure 13. They propose performance evolution blueprint, a visual support to precisely compare multiple software executions. Suppose that after a chain of modifications the performance of the system has been reduced. Unfortunately, identifying which of the changes contained in these versions are responsible for this performance drop is difficult. To address this issue, this paper proposes a new approach to visualize the performance difference. A blueprint is obtained after running two executions. Each box is a method context. Edges are invocations between methods (a calling method is above the called methods). Height of a method is the difference of execution time between the two executions. If the difference is positive (i.e., the method is slower), then the method is shaded in red, otherwise it is green. The width of a method is the absolute difference in the number of executions, thus always positive. Light red / pink color means the method is slower, but its source code has not changed between the two executions. If red the method is slower and the source code has changed. Light green indicates a faster non-modified method. Green indicates a faster modified method. Yellow indicates new methods and gray indicates removed methods. Tooltip gives an extended list of data for the particular methods, including its name, its defining class and the numerical values of the differences.

Table 10. Case 10 analysis

Task	Monitor code changes Comprehend the structure of a program Analyze execution of the program Perform debugging	
Data	Functions Events and sequences (sequence of calls) Relationships between code components	
Data (derived)	delta time, delta # of executions	
Data Processing	Dynamic analysis	
Representation	Technique	Encode: Link, Size, Color Facet: Partition into multiform views, Juxtaposed views, Linked highlighting, Linked navigation Manipulate: Select
	Textual Content	Y
Availability	Scalability	N
	Integration	N

4.11 Case 11: Visualization of Program-Execution Data for Deployed Software

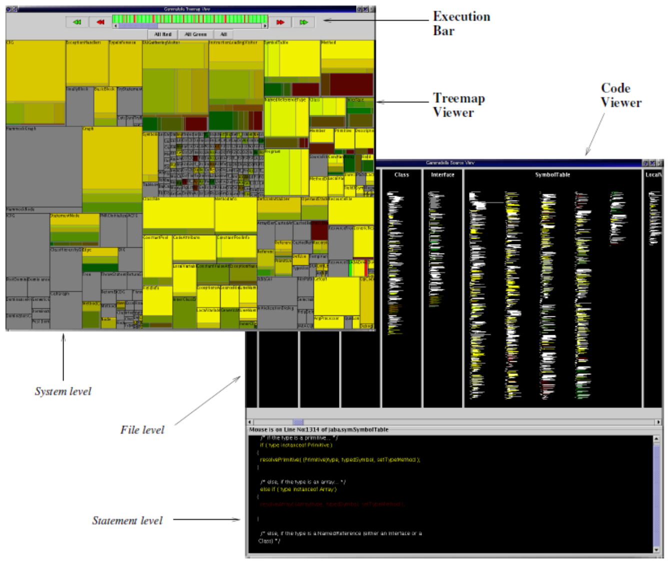


Fig. 14. Case 11 main view

The main view of the visualization system developed in this paper[23] is shown in figure 14. In this paper, a new technique has been introduced for collecting, storing, and visualizing program-execution data gathered from deployed instances of a software product. The approach is defined for a context in which a number of instances of a program are continuously monitored. Statement level is the lowest level of representation. At this level, the actual source code is represented, and each line of code is suitably colored. The representation level provides a miniaturized view of the source code. It maps each line in the source code to a short, horizontal line of pixels. This zoomed away perspective lets more of the software system be presented on one screen. The system level is the most abstracted level in this visualization. For the representation at this level treemap view. The root node represents the entire system. The intermediate non-leaf nodes represent modularizations of the system (e.g. Java packages). The leaf nodes represent source files in the system. To represent executions, they use an execution bar: a virtually infinite rectangular bar, of which only a subset is visible at any time. Each band in the execution bar represents a different execution of the monitored program in the field. The Code Viewer displays both the file-level view and the statement-level view. Right-clicking on a statement in the file-level view causes a context menu to appear that permits the viewing of different types of information about the statement, such as the number of executions that covered it or the types of exceptions that were thrown by the executions that covered it. The statement-level view shows a small number of statements in its full-sized text, at the bottom of the Code Viewer window. Moving the mouse cursor over the file-level view causes the statement-level view to display those statements under the cursor. The idea is to assign a color to each statement in the program to represent how likely it is for the statement to be responsible for the behavior that led to the throwing of an exception. Red, yellow, and green are used in this case to represent very likely, possibly, and unlikely, respectively. Selecting an execution or a set of executions causes the other displays to update their views to show only the information pertaining to the selected executions. Executions can be selected by left-clicking with the mouse on the corresponding band(s).

Table 11. Case 11 analysis

Task	Analyze execution of the program Perform debugging	
Data	Source code Packages Classes, objects, interfaces Test suite and results Events and sequences (run-time events) Relationships between code components Metadata (statement information)	
Data (derived)	Level of suspiciousness	
Data Processing	Dynamic analysis, Static analysis	
Representation	Technique	Encode: Dense layout, Treemap, Position, Size, Color, Bar chart, Stacked bar chart, Containment Reduce: Filter, Aggregation Facet: Partition into multiform views, Juxtaposed views, Linked highlighting, Linked navigation, Overview-detail Manipulate: Select
	Textual Content	Y
Availability	Scalability	Y
	Integration	N

4.12 Case 12: Software Evolution Storylines

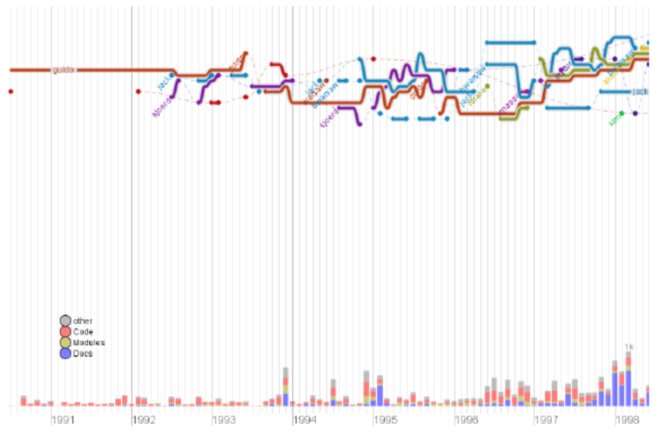


Fig. 15. Case 12 main view

The main view of the visualization system developed in this paper[24] is shown in figure 15. This paper presents a technique for visualizing the interactions between developers in software project evolution. The goal is to produce a visualization that shows more detail than animated software histories, but keeps the same focus on aesthetics and presentation. Instead of thin lines, they use metro maps (i.e. schematic diagrams of public transportation routes) to thicken the lines and use bold colors. The amount of space between connected lines was decreased, to mimic the metro map convention showing collinear routes. A fairly common occurrence in open source development is when a developer does not commit during a timestep, but resumes work in a future one. These furloughs from activity ought to be visually differentiated from a developer who leaves the project permanently. They use dashed lines to connect develoeprs' timesteps during their temporary absence. As the storylines indicate the number of developers but not the amount of commits, a commit histogram is placed at the bottom. This shows the number of file-commits (the sum of files in each commit). Each bar is one timestep and is divided into color categories, defined by the user. In the histograms in this paper, the colors are red for core source code, yellow for modules, and blue for documents. a storyline, only that developer's storyline is colored and the rest are turned to grayscale (pictured below). In addition, the selected developer's activity in the commit histogram is highlighted through this interaction.

Table 12. Case 12 analysis

Task	Monitor code changes Monitor developer activities	
Data	Events and sequences (sequence of commits) Metadata (version information)	
Representation	Technique	Encode: Line chart, Dot marks, Size, Color, Stacked bar chart, Texture Reduce: Filter Facet: Partition into multiform views, Juxtaposed views, Linked highlighting Manipulate: Select, Zoom
	Textual Content	N
Availability	Scalability	N
	Integration	N

4.13 Case 13: Software Visualization in the Large

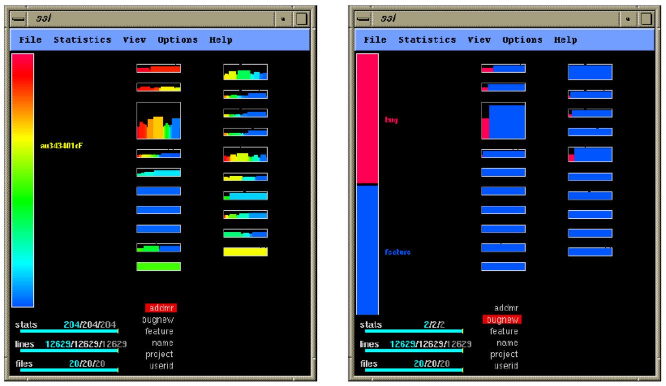


Fig. 16. Case 13 main view

The main view of the visualization system developed in this paper[25] is shown in figure 16. Software is invisible, disappearing into files on disks. The invisible nature of software contributes to low programmer productivity by hiding system complexity, particularly for large team-oriented projects. Visualization can help software engineers cope with this complexity and thereby increase programmer productivity. The summary representation presents file-level statistics. Each file is represented by a rectangle. There are four possible rectangle heights, corresponding to the four quartiles of file size (as measured by number of lines). Because file sizes may vary from a few lines to tens of thousands of lines, grouping the sizes by quartiles ensures that all of the files are always visible. This visualization shows the summary representations of the same files in two different panes, corresponding to two different statistics. In this case, only three of the four size quartiles are represented in the data set. The left pane shows the code age as miniature time series within each rectangle, while the right pane shows the amount of code added for bug fixing and new functionality. Other possibilities for color encoding include software metrics such as Halstead's program volume measure or McCabe's cyclomatic complexity. By hovering over different parts of a file a pop-up view shows detailed information.

Table 13: Case 13 analysis

Task	Comprehend the structure of a program Monitor code changes	
Data	Source code Relationships between code components Metadata (statement information)	
Data (derived)	Halstead's program volume measure, McCabe's cyclomatic complexity	
Data Processing	AST, Static analysis	
Representation	Technique	Encode: Histogram, Color, Bar chart, Size Facet: Partition into multiform views, Juxtaposed views, Overview-detail, Pop-up view
	Textual Content	Y
Availability	Scalability	Y
	Integration	N

4.14 Case 14: Supporting the understanding of the evolution of software items

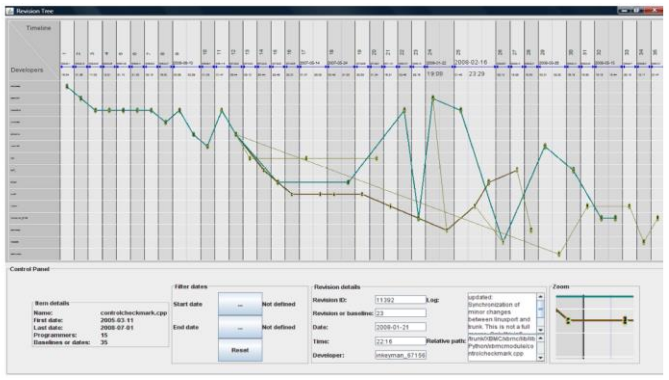


Fig. 17. Case 14 main view

The main view of the visualization system developed in this paper[26] is shown in figure 17. It is important to highlight that the representation of software items evolution helps to visualize the contributions of team members through several revisions. The green line connecting yellow ovals represent the main code versioning and the light brown lines represent branches. It also provides the possibility to select a branch and highlight its path. The visualization main components are the revision tree, the timeline and the control panel. A grid based structure is used for the revision tree and its correlation with the timeline for providing an intuitive mechanism to visualize relationships between developers, baselines and revisions; the rows represent authors and the columns represent timeline elements. On the other hand, the control panel displays item and revision details and allows filtering by date ranges and zooming into visualization areas. The item details provided by the control panel are the item name the creation date, the update date, the number of programmers, the number of baselines or dates of the evolution. When one revision is under focus, the control panel displays the log and the path of that revision. The timeline uses variable width columns to accommodate baselines, dates and creation time of revisions; the column width depends on the number of revisions associated to the baseline.

Table 14. Case 14 analysis

Task	Monitor developer activities Monitor code changes	
Data	Events and sequences (sequence of revisions) Metadata (revision information)	
Representation	Technique	Encode: Grid, Color, Line chart, Size Reduce: Filter Facet: Partition into multiform views, Juxtaposed views, Linked highlighting, Overview-detail, Superimposed line charts Manipulate: Select, Zoom
	Textual Content	Y
Availability	Scalability	N
	Integration	N

4.15 Case 15: Towards Anomaly Comprehension

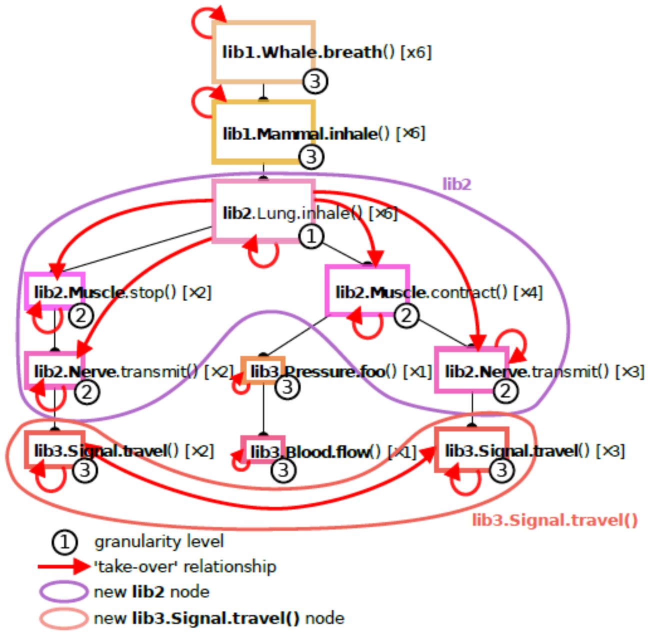


Fig. 18. Case 15 main view

The main view of the visualization system developed in this paper[27] is shown in figure 18. In this paper, a localized approach to navigate and analyze the CPU usage of little-known programs and libraries has been introduced. This method exploits the structural information present in profiling call trees to selectively raise or lower the local abstraction level of the performance data The traditional approach for navigating a profiling tree consists selectively hiding or showing subtrees. The represented information remains however at the same level of abstraction: each node corresponds to the invocation a method along a particular call path starting at the tree's root. This paper proposes to explore an alternative approach by varying the level of abstractions at which different parts of the profiling tree are represented. Developers might however wish to zoom-in by lowering the abstraction of one particular part of the graph, while maintaining the rest of the graph in its compacted form. The technique presented in this paper further extends this approach by allowing users to select local levels of abstraction that only apply in one part of the profiling tree. As a result, the same program element might be expanded at different granularity levels in different parts of the graph. Figure 18 shows how the right-hand side lib3 package is locally expanded, while the same left-hand side package remains compacted.

Table 15. Case 15 analysis

Task	Analyze execution of the program Comprehend the structure of a program	
Data	Source code Packages Classes, objects, interfaces Functions Relationships between code components Metadata (statement information)	
Data (derived)	Halstead's program volume measure, McCabe's cyclomatic complexity	
Data Processing	AST, Static analysis	
Representation	Technique	Encode: Shape, Size, Color, Link, Size Reduce: Aggregation Facet: Overview-detail Manipulate: Select, Zoom
	Textual Content	N
Availability	Scalability	Y
	Integration	N

4.16 Case 16: Visually Exploring Software Maintenance Activities

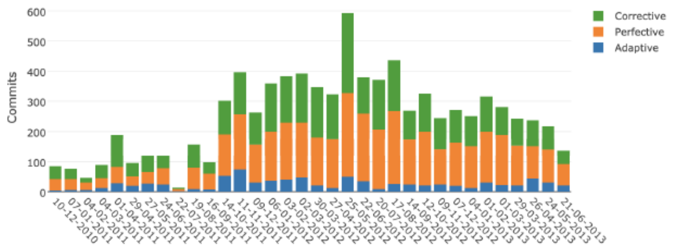


Fig. 19. Case 16 main view

The main view of the visualization system developed in this paper[28] is shown in figure 19. Each maintenance activity is encoded using a different color, and the three activity types are stacked on top of one another. The x-axis is the time-line, and the y-axis is the activity (commit) count. Stacked bar diagrams facilitate comparisons between maintenance activities within a given stacked bars column (e.g., what maintenance activity dominated a given time frame), as well as between different stacked bars columns (e.g., which of the time frames had more of a given maintenance activity). In addition, bar diagrams often allow for an easy detection of anomalies such as peaks and deeps, as well as trends. Users can zoom on a specific time period by clicking the left mouse button and dragging the mouse. Maintenance activities can be filtered by a number of parameters: project's name and time period. In the developer centric view, maintenance activities can also be filtered by a developer identifier, which can be a name, an email address, or both. By hovering over an area of a given stacked bars column, the corresponding maintenance activity's aggregate information is displayed. This additional numerical information helps in situations where the segmentation within a single stacked bars column is seemingly equal, and visually comparing the areas is not accurate enough. Users can obtain a detailed view of the commits pertaining to a specific maintenance activity and time frame by clicking its color in the corresponding stacked bars column.

Table 16. Case 16 analysis

Task	Monitor code changes Monitor developer activities Detect code smells	
Data	Events and sequences (sequence of commits) Metadata (commit information)	
Data (derived)	Different types of maintenance	
Data Processing	Static analysis	
Representation	Technique	Encode: Stacked bar chart, Color Reduce: Aggregation, Filter Facet: Overview-detail, Pop-up view Manipulate: Select, Zoom
	Textual Content	N
Availability	Scalability	Y
	Integration	N

4.17 Case 17: Using HTML5 Visualizations in Software Fault Localization

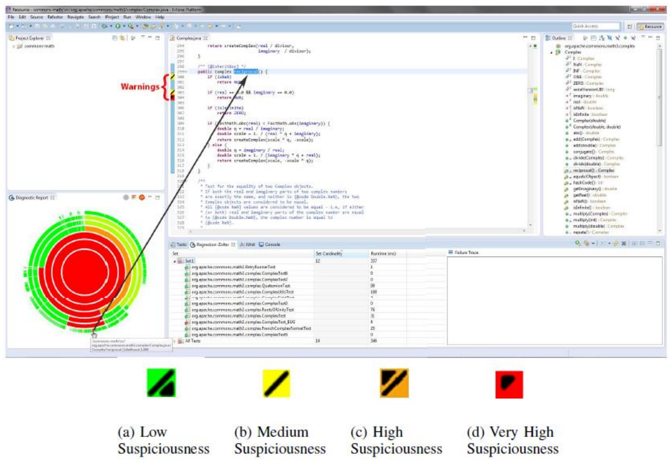


Fig. 20. Case 17 main view

The main view of the visualization system developed in this paper[29] is shown in figure 20. In this paper, GZOLTAR toolset is introduced to display the diagnostic reports yielded by spectrum-based fault localization. The GZOLTAR toolset is a plug and play plugin for the Eclipse IDE to ease world-wide adoption. This visualization helps to drastically reduce the time needed in debugging. The generated visualization is interactive, and the user is able to navigate through the project structure to analyze it in detail. The intention of the visualizations have the main goal of representing the analyzed project in an hierarchical way to allow a faster and easier debugging process. In the Sunburst visualization, each ring denotes an hierarchical level of the source code organization (from the inner to the outer circle). All visualizations obey to a color gradient ranging from green (low suspiciousness) to red (very high suspiciousness). The suspiciousness is computed by a diagnostic algorithm. The Sunburst visualization uses arcs as solid areas which represent the nodes. The radius of each one proportionally varies with the size of the respective subtree. The root element is drawn always at the center of the visualization, and the children are expanded outward from it. This visualization uses polar coordinates to properly position each arc. The GZOLTAR toolset also places warnings on the vertical ruler of the code editor next to the lines that are most likely to contain the fault. This list of warnings aid the developer in the process of pinpointing the faulty statement. The warnings can be of four types: (1) red for the top lines most likely to contain a fault, (2) orange for high suspiciousness, (3) yellow for medium suspiciousness, and (4) green for low suspiciousness. Each warning embeds a ColorADD symbol3, aimed at aiding color-blind people distinguish between the and which use JUnit test cases.

Table 17. Case 17 analysis

Task	Analyze execution of the program Detect code smells Perform debugging	
Data	Source code Test suite and results	
Data (derived)	Level of suspiciousness	
Data Processing	AST, Dynamic analysis	
Representation	Technique	Encode: Sunburst diagram, Color, Glyph Reduce: Aggregation, Filter Facet: Linked highlighting, Juxtaposed views Manipulate: Select, Zoom
	Textual Content	N
Availability	Scalability	Y
	Integration	Y

4.18 Case 18: Declarative and visual debugging in Eclipse

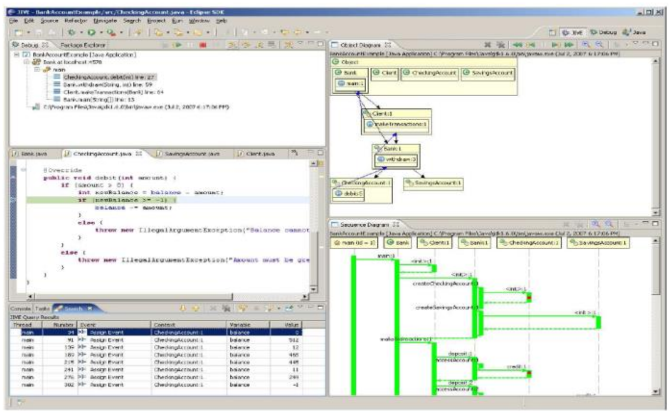


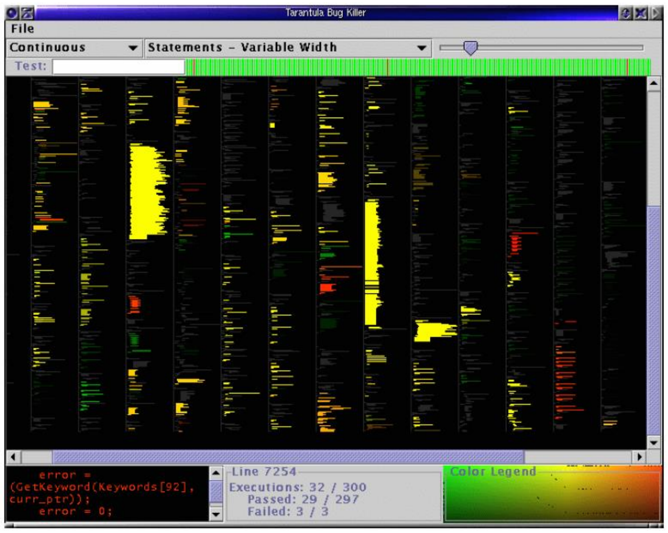
Fig. 21. Case 18 main view

The main view of the visualization system developed in this paper[30] is shown in figure 21. JIVE, the visualization system introduced in this paper, is a declarative and visual debugging environment for Eclipse. Traditional debugging is procedural in that a programmer must proceed step-by-step and object- by-object in order to uncover the cause of an error. They present a declarative approach to debugging consisting of a flexible set of queries over a program's execution history as well as over individual runtime states. This runtime information is depicted in a visual manner during program execution in order to aid the debugging process. The current state of execution is depicted through an enhanced object diagram, and the history of execution is depicted by a sequence diagram. This methodology makes use of these diagrams as a means of formulating queries and reporting results in a visual manner. The object model represents the program's execution state, while the sequence model details its history of execution. An object diagram depicts the program's execution state by showing objects and their structural links as well as outstanding method activations. The JIVE sequence diagram is constructed interactively at execution time. In JIVE, every point on the sequence diagram is correlated with the object diagram that would have been in effect at that execution point. JIVE also supports interactive forward as well as reverse stepping of the program. Through the sequence diagram, a user may direct the JIVE engine to any previous point in the execution history in order to inspect the object diagram at that execution point.

Table 18. Case 18 analysis

Task	Analyze execution of the program Detect code smells Perform debugging	
Data	Source code Test suite and results Packages Classes, objects, interfaces Functions Relationships between code components Events and sequences (runtime events) Metadata (running information)	
Data (derived)	Run-time information	
Data Processing	Dynamic analysis	
Representation	Technique	Encode: UML, Link, Color, Sequence diagram, Size, Reduce: Aggregation, Filter Facet: Linked highlighting, Juxtaposed views Manipulate: Select, Zoom
	Textual Content	N
Availability	Scalability	N
	Integration	Y

4.19 Case 19: Visualization of Test Information to Assist Fault Localization



```
error =
(GetKeyword(Keywords[92],
curr_ptr));
    error = 0;
```

```
Line 7254
Executions: 32 / 300
    Passed: 29 / 297
    Failed: 3 / 3
```

Fig. 22. Case 19 main view

The main view of the visualization system developed in this paper[31] is shown in figure 22. The Tarantula system, developed in this paper, is a software engineering tool for visualizing test coverage. Dense displays using line marks have become popular for showing overviews of software source code. In these displays the coloring of the lines encodes an attribute of interest. Most of the screen is devoted to a large and dense overview of source code using one-pixel tall lines, color coded to show whether it passed, failed, or had mixed results when executing a suite of test cases. The small source code view in the lower left corner is a detail view showing a few lines of source code at a legible size. The dense display scales to around ten thousand lines of code, handling around one thousand vertical pixels and ten columns. Tarantula computes two derived quantitative attributes that are encoded with hue and brightness. The brightness encodes the percentage of coverage by the test cases, where dark lines represent low coverage and bright ones are high coverage. The hue encodes the relative percentage of passed versus failed tests.

Table 19. Case 19 analysis

Task	Comprehend the structure of a program Detect code smells Perform debugging	
Data	Source code Test suite and results	
Data (derived)	Test execution information	
Data Processing	Dynamic analysis	
Representation	Technique	Encode: Dense layout, Hue, Brightness, Position, Size Reduce: Filter Facet: Partition into multiform views, Juxtaposed views, Linked highlighting, Overview-detail Manipulate: Select
	Textual Content	Y
Availability	Scalability	Y
	Integration	N

4.20 Case 20: Visualizing Interactive and Shared Debugging Sessions

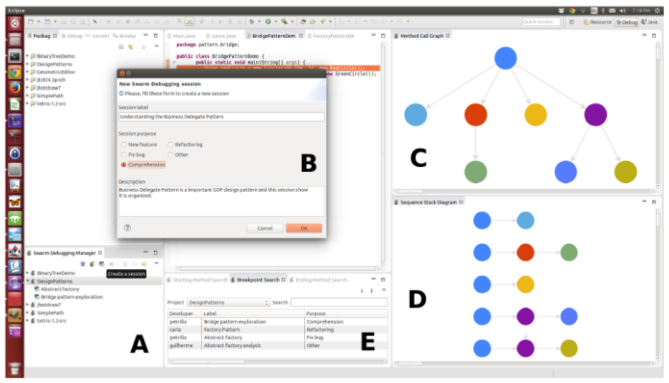


Fig. 23. Case 20 main view

The main view of the visualization system developed in this paper[32] is shown in figure 23. Debugging sessions require a methodical process of finding causes and reducing the number of software problems. During such sessions, developers run a software project, traversing method invocations, setting breakpoints, stopping or restarting executions. In these sessions, developers explore different parts of the code and create knowledge about them. When debugging sessions are over, it is likely that such knowledge is lost, and developers cannot use it in other sessions or sharing it with collaborators. Swarm Debugging, the visualization developed in this paper, is a new approach for visualizing and sharing information obtained during debugging sessions, providing interactive and real-time visualization techniques, and several searching tools. The Sequence stack diagram is a novel diagram to represent a sequence of methods invocations. They use circles to represent methods and arrows to represent invocations. Each line is a complete stack trace, without returns. The first node is a Starting point (non-invoked method), and the last node is an Ending point (non-invoking method). If the invocation chain continues to a non-starting point method, a new line is created, the current stack is repeated, and a dotted arrow is used to represent a return for this node. Code exploration features are provided so that developers can directly go to a method in the Eclipse Editor by double-clicking on its node in the diagram. The Dynamic method call graph is a diagram based on directed call graphs. This visualization uses circles to represent methods (nodes) and oriented arrows to express invocations (edges). Each session generates a call graph and all invocations collected during the session are shown in this visualization. The starting points (non-invoked methods) are represented at the top of a tree, and the adjacent nodes represent the invocation sequence. As an interactive feature, the developer can navigate along the sequence of invocation methods by pressing the F9 key (forward) or the F10 key (backwards). Finally, developers can go directly to a method in the Eclipse Editor by double clicking on its node in the diagram. The Swarm dashboard is an online panel to display project information.

Table 20. Case 20 analysis

Task	Comprehend the structure of a program Detect code smells Perform debugging	
Data	Source code Functions Events and sequences Relationships between code components Metadata	
Data (derived)	Debugging session information	
Data Processing	Dynamic analysis, AST	
Representation	Technique	Encode: Color, Link, Position, Reduce: Filter Facet: Partition into multiform views, Juxtaposed views, Linked highlighting, Overview-detail Manipulate: Select
	Textual Content	Y
Availability	Scalability	Y
	Integration	Y

4.21 Case 21: Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams

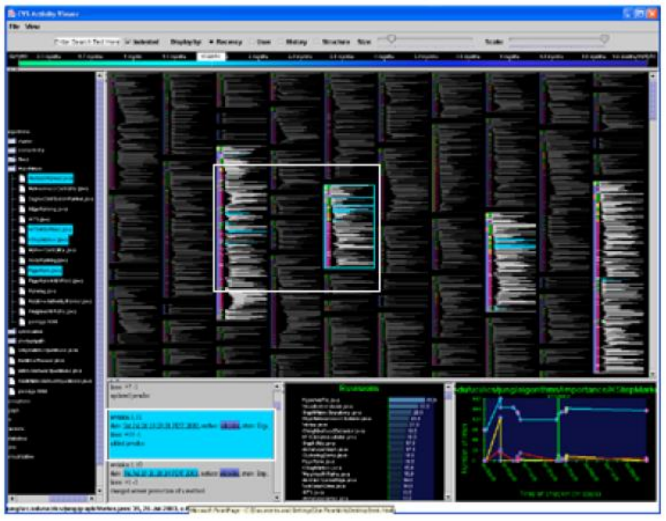


Fig. 24. Case 21 main view

The main view of the visualization system developed in this paper[33] is shown in figure 24. Augur is a visualization tool developed in this paper that supports distributed software development processes. Augur creates visual representations of both software artifacts and software development activities, and, crucially, allows developers to explore the relationship between them. Augur is designed not for managers, but for the developers participating in the software development process. Augur provides a set of linked visualizations displaying different characteristics of the software system under examination. Each pane displays a different aspect of the system being examined and changes in one view are immediately reflected in the others. The large central pane shows the line oriented view of the source code. In this view, the color of each pixel line indicates how recently it was modified; this allows a developer, at a glance, to see how much activity has taken place recently and where that activity has been located.

Table 21. Case 21 analysis

Task	Comprehend the structure of a program Monitor code changes Monitor developer activities	
Data	Source code Events and sequences Metadata	
Data (derived)	Line position, Line status	
Data Processing	Static analysis, AST	
Representation	Technique	Encode: Dense layout, Position, Size, Color, Bar chart, Stacked bar chart, Line chart Reduce: Filter, Aggregation Facet: Partition into multiform views, Juxtaposed views, Linked highlighting, Linked navigation, Overview-detail Manipulate: Select, Zoom
	Textual Content	Y
Availability	Scalability	Y
	Integration	N

4.22 Case 22: Visualizing Software Systems as Cities

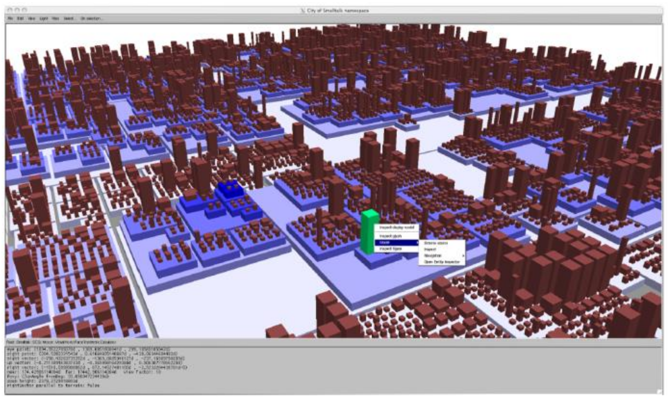


Fig. 25. Case 22 main view

The main view of the visualization system developed in this paper[34] is shown in figure 25. This paper presents a 3D visualization approach which gravitates around the city metaphor, i.e., an object-oriented software system is represented as a city that can be traversed and interacted with: the goal is to give the viewer a sense of locality to ease program comprehension. The key point in conceiving a realistic software city is to map the information about the source code in meaningful ways in order to take the approach beyond beautiful pictures. The brown buildings represent classes and interfaces, placed in blue tiles representing the packages. The color saturation of the tiles is proportional to the nesting level of the corresponding packages. The height of the buildings represents their number of methods (NOM), while the width and length represents the number of attributes (NOA). On the left of the figure, at the far end of the city, we see two external suburbs, which represent libraries. The visualization allows us to easily spot some patterns.

This city hosts a number of disproportional buildings, such as two antenna-shaped constructs, which represent classes with a large number of methods and very few attributes, as well as a number of buildings that look like parking lots, which represent classes with lots of attributes and very few or no methods (potential data classes). There are also a lot of small houses, which make up entire districts. The visualization is interactive and navigable using the keyboard, i.e., it is easy to zoom in on details of the city or to focus on one specific district Right-clicking any of the items brings up a context menu to perform a variety of tasks, such as inspecting the model entity, accessing the represented source code, etc. The initial level of granularity is set to the class level, the visualization scales up well in terms of the size of the system that it can display as a code city. However, in cities representing very large systems the interactivity and navigability can be substantially slowed down.

Table 22. Case 22 analysis

Task	Comprehend the structure of a program Detect code smells	
Data	Source code Packages Classes, objects, interfaces Functions Relationships between code components Metadata	
Data (derived)	NOM, NOA	
Data Processing	Static analysis	
Representation	Technique	Encode: Shape, Position, Size, Hue, Saturation, 3D Bar chart Reduce: Filter, Aggregation Facet: Overview-detail Manipulate: Select, Zoom
	Textual Content	Y
Availability	Scalability	N
	Integration	N

4.23 Case 23: Blended, Not Stirred: Multi-concern Visualization of Large Software Systems

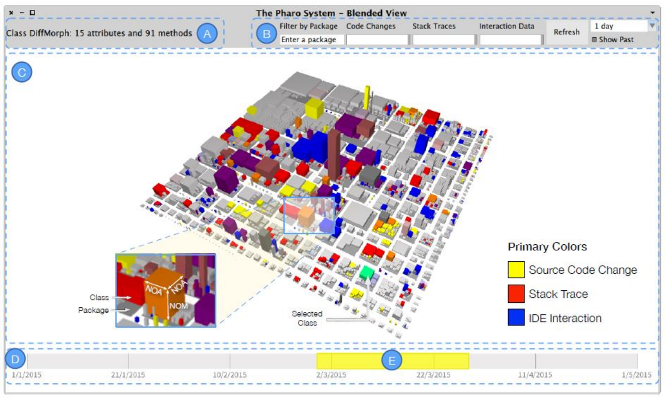


Fig. 26. Case 23 main view

The main view of the visualization system developed in this paper[35] is shown in figure 26. While constructing and evolving software systems, developers generate directly and indirectly a large amount of data of diverse nature, such as source code changes, bug tracking information, stack traces, etc. Often these diverse data sources are processed and visualized in isolation, leading to a partial view of systems. This paper presents a blended approach to visualize several data ingredients at once to enable a quick and comprehensive assessment of what happened to a software system in any given time frame. The Blended City is the tool developed in this paper. It is composed of four main parts: A status bar to display additional information on the selected entity, a toolbar to customize the visualization, the view canvas, and a timeline slider. With the timeline slider the user chooses the visualized data timespan. The width of this slider can be adapted using the dropdown menu on the right part of the toolbar. Moreover, the user can click on an entity and get additional information on the status bar. Selected entities are colored with a bright green.

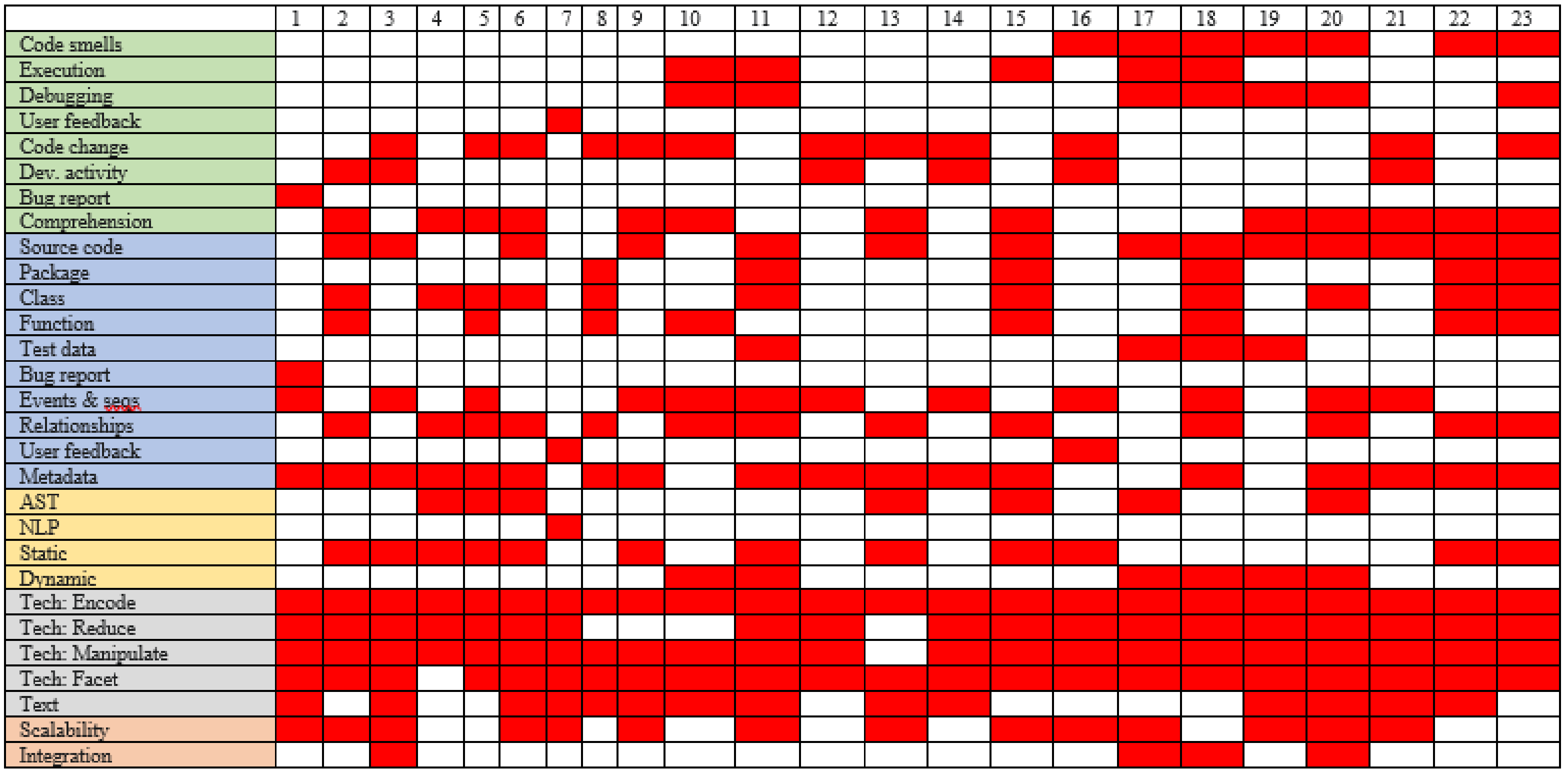
Table 23. Case 23 analysis

Task	Comprehend the structure of a program Detect code smells Perform debugging Monitor code changes	
Data	Source code Packages Classes, objects, interfaces Functions Relationships between code components Metadata	
Data (derived)	NOM, NOA	
Data Processing	Static analysis	
Representation	Technique	Encode: Shape, Position, Size, Hue, Brightness, 3D Bar chart Reduce: Filter, Aggregation Facet: Overview-detail Manipulate: Select, Zoom
	Textual Content	N
Availability	Scalability	N
	Integration	N

5 Discussion of Results

The comprehensive result of the analysis phase is in table 24. First row is the number of papers. The number is in the ordered they were analyzed and appeared in this paper. Each row show shows one of the subcategories in our 5 main categories. To be able to put the whole table in one page, the names of subcategories are shortened. To ease the comprehension, each category has a different color. Because the variety of techniques was high, the table only show higher-level techniques such as manipulation. If one paper in a column is associated to one of the subcategories in a row, then the cell placed in the intersection of that column and row, is red.
Table 1 includes a lot of information. We can see that program comprehension and monitoring code changes are the most investigated tasks, while tracking bug reports and analyzing feedback are less investigated. Because of that, bug report and user feedback are data types with the least usage. Metadata is the kind of data that is present in most of the cases. Also, the table illustrates that the combination of <source code, package, class, functions> is common and most of the time they are visualized together because in a hierarchical view of a project they are related.

Table 24. Analysis results



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Code smells																🟥	🟥	🟥	🟥	🟥		🟥	🟥
Execution										🟥	🟥				🟥		🟥	🟥					
Debugging										🟥	🟥						🟥	🟥	🟥	🟥			🟥
User feedback							🟥																
Code change			🟥		🟥	🟥		🟥	🟥	🟥		🟥	🟥	🟥		🟥					🟥		🟥
Dev. activity		🟥	🟥									🟥		🟥		🟥					🟥		
Bug report	🟥																						
Comprehension		🟥		🟥	🟥	🟥			🟥	🟥			🟥		🟥				🟥	🟥	🟥	🟥	🟥
Source code		🟥	🟥			🟥			🟥		🟥		🟥		🟥		🟥	🟥	🟥	🟥	🟥	🟥	🟥
Package								🟥			🟥				🟥			🟥				🟥	🟥
Class		🟥		🟥	🟥	🟥		🟥			🟥				🟥			🟥		🟥		🟥	🟥
Function		🟥			🟥			🟥		🟥					🟥			🟥				🟥	🟥
Test data											🟥						🟥	🟥	🟥				
Bug report	🟥																						
Events & seqs	🟥		🟥		🟥				🟥	🟥	🟥	🟥		🟥		🟥		🟥		🟥	🟥		
Relationships		🟥		🟥	🟥	🟥		🟥		🟥	🟥		🟥		🟥			🟥		🟥		🟥	🟥
User feedback							🟥									🟥							
Metadata	🟥	🟥	🟥	🟥	🟥	🟥		🟥	🟥		🟥	🟥	🟥	🟥	🟥			🟥		🟥	🟥	🟥	🟥
AST				🟥	🟥	🟥							🟥		🟥		🟥			🟥			
NLP							🟥																
Static		🟥	🟥	🟥	🟥	🟥			🟥		🟥		🟥		🟥	🟥						🟥	🟥
Dynamic										🟥	🟥						🟥	🟥	🟥	🟥			
Tech: Encode	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
Tech: Reduce	🟥	🟥	🟥	🟥	🟥	🟥	🟥				🟥	🟥		🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
Tech: Manipulate	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥		🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
Tech: Facet	🟥	🟥	🟥		🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
Text	🟥		🟥			🟥	🟥	🟥	🟥	🟥	🟥		🟥	🟥					🟥	🟥	🟥	🟥	
Scalability	🟥	🟥	🟥			🟥	🟥		🟥		🟥		🟥		🟥	🟥	🟥		🟥	🟥	🟥		
Integration			🟥														🟥	🟥		🟥			

NLP methods are used only in one case. Another three methods are applied almost equally. Among combinations <AST, Static analysis> happens more than the other possibilities.

Obviously, all the cases have "Encode" and most of the times all four high-level methods happen together. In a few cases "Reduce" or "Manipulate" are absent. More than half of the cases display textual content and it shows that this is a common approach in visualizing software evolution and maintenance.

More than half of the cases are scalable which is a necessary feature for a visualization system in the field of software engineering, because this field always deals with data in large size. A few cases have been integrated into an IDE and available through plugins.

In addition, some relations between categories can be viewed from the table. Task, data and data processing are conceptually related to each other. For example, we can see when the task is program execution, most of the time data is source code, events and sequences and relationships. Also, data processing method is dynamic analysis which is suitable for this type of task and data.

6 Conclusion

The present work gathered 23 different studies that had focused on visualization of software evolution and maintenance. This pool included a mixture of design study papers. Each paper was described, and categorized based on its respective visualization method. A categorization including task, data, data processing, representation and availability classes is provided as well. Final result is shown in an exhaustive table which makes it easier to compare various systems and gain an overall opinion about the role of visualization systems in the field of software evolution and maintenance. A lot of research has been done on visualizing program's structure and code changes is saturated while user feedback and tracking bug reports areas are neglected. Using NLP methods is quite new in this field and it's expected to become involved more in future. The main challenge is still the large amount of complex data that researchers are indeed trying to deal with that through various processing methods or visualization techniques.

References

[1] Roman, Gruia-Catalin and Cox, Kenneth C. Program Visualization: The Art of Mapping Programs to Pictures, In: International Conference on Software Engineering 14th International Conference on Software Engineering: Proceedings. New York: Association for Computing Machinery, 1992.

[2] Hendrix, T. & II, James & Maghsoodloo, Saeed & McKinney, Matthew. Do visualizations improve program comprehensibility? Experiments with control structure diagrams for Java. ACM Sigcse Bulletin. 32. 382-386, 2000.

[3] M.H. Brown, Algorithm Animation, MIT Press Cambridge, 1988.

[4] Green, Thomas R. G. and Marian Petre. "When Visual Programs are Harder to Read than Textual Programs." (1992).

[5] Fjeldstad, R. K. and W. T. Hamlen. "Application program maintenance study - reports to our respondents." (1982).

[6] PRICE, A., BAECKER, R., AND SMALL, I. 1993. A principled taxonomy of software visualization. Journal of Visual Languages and Computing 4(3):211-266.

[7] MALETIC, J., MARCUS, A., AND COLLARD, M. 2002. A task oriented view of software visualization. Proceedings of IEEE Workshop of Visualizing Software for Understanding and Analysis Paris, France,, 32–40.

[8] KOSCHKE, R. 2003. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. Journal of Software Maintainance. Evol.: Res. Pract 15, 87–109.

[9] STOREY, M.-A. D., AND GERMAN, D. M. 2005. On the use of visualization to support awareness of human activities in software development: a survey and a framework. In SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization, ACM, New York, NY, USA, 193–202.

[10] BAECKER, R., DIGIANO, C., AND MARCUS, A. 1997. Software visualization for debugging. Commun. ACM 40, 4, 44–54.

[11] SENSALIRE, M., AND OGAO, P. 2007. Tool users requirements classification:how software visualization tools measure up. AFRIGRAPH '07' Proceedings of the 5th International Conference on Computer graphics, virtual reality, visualization and interaction in Africa, Grahamstown, South Africa.

[12] Munzner, T., (2014). Visualization Analysis and Design, A K Peters Visualization Series. CRC Press.

[13] Sasso, Tommaso Dal and Michele Lanza. "A closer look at bugs." 2013 First IEEE Working Conference on Software Visualization (VISSOFT) (2013): 1-4.

[14] Hanjalic, Avdo. (2013). ClonEvol: Visualizing software evolution with code clones. 2013 1st IEEE Working Conference on Software Visualization - Proceedings of VISSOFT 2013.

[15] Voinea, Lucian & Telea, Alexandru & Wijk, Jarke. (2005). CVSscan: Visualization of Code Evolution. 47-56.

[16] Erdemir, Ural & Tekin, Umut & Buzluca, Feza. (2011). E-Quality: A graph based object oriented software quality visualization tool. Proceedings of the 6th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT). 1 - 8.

[17] Alexandru, Carol & Behnamghader, Pooyan & Proksch, Sebastian & Gall, Harald. (2019). Evo-Clocks: Software Evolution at a Glance. 10.1109/VISSOFT.

[18] Langelier, Guillaume & Sahraoui, Houari & Poulin, Pierre. (2008). Exploring the Evolution of Software Quality with Animated Visualization. Proceedings - 2008 IEEE Symposium on Visual Languages and Human-Centric Computing.

[19] Guzman, E., Bhuvanagiri, P., & Brügge, B. (2014). FAVe: Visualizing User Feedback for Software Evolution. 2014 Second IEEE Working Conference on Software Visualization, 167-171.

[20] Baum, David & Schilbach, Jan & Kovacs, Pascal & Eisenecker, Ulrich & Müller, Richard. (2017). GETAVIZ: Generating Structural, Behavioral, and Evolutionary Views of Software Systems for Empirical Evaluation. 10.1109/VISSOFT.2017.12.

[21] Voinea, Lucian and Alexandru Telea. "Multiscale and multivariate visualizations of software evolution." SoftVis '06 (2006).

[22] Sandoval Alcocer, Juan & Bergel, Alexandre & Ducasse, Stéphane & Denker, Marcus. (2013). Performance Evolution Blueprint: Understanding the Impact of Software Evolution on Performance. 2013 1st IEEE Working Conference on Software Visualization - Proceedings of VISSOFT 2013. 1-9. 10.1109/VISSOFT.2013.

[23] Orso, Alessandro & Jones, James & Harrold, Mary. (2003). Visualization of Program-Execution Data for Deployed Software. 67-76, 211. 10.1145/774833.774843.

[24] Ogawa, Michael & Ma, Kwan-Liu. (2010). Software evolution storylines. 35-42. 10.1145/1879211.1879219.

[25] Ball, Thomas & Eick, Stephen. (1999). Software Visualization in the Large. Computer. 29. 10.1109/2.488299.

[26] Therón, Roberto & González-Torres, Antonio & García-Peñalvo, Francisco. (2008). Supporting the understanding of the evolution of software items. 189-192. 10.1145/1409720.1409750.

[27] Lin, Shen & Taiani, François & Ormerod, Thomas & Ball, Linden. (2010). Towards anomaly comprehension: Using structural compression to navigate profiling call-trees. Proceedings of the ACM Conference on Computer and Communications Security. 103-112. 10.1145/1879211.1879228.

[28] Levin, Stanislav & Yehudai, Amiram. (2019). Visually Exploring Software Maintenance Activities.

[29] Gouveia, Carlos & Abreu, Rui. (2013). Using HTML5 visualizations in software fault localization. 2013 1st IEEE Working Conference on Software Visualization - Proceedings of VISSOFT 2013. 1-10. 10.1109/VISSOFT.2013.6650539.

[30] Czyz, Jeffrey & Jayaraman, Bharat. (2007). Declarative and visual debugging in Eclipse. Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology EXchange. 31-35. 10.1145/1328279.1328286.

[31] Jones, James & Harrold, Mary & Stasko, John. (2002). Visualization of Test Information to Assist Fault Localization. 10.1145/581396.581397.

[32] Petrillo, Fabio & Lacerda, Guilherme & Pimenta, Marcelo & Freitas, Carla. (2015). Visualizing interactive and shared debugging sessions. 140-144. 10.1109/VISSOFT.2015.7332425.

[33] Froehlich, Jon and Paul Dourish. "Unifying artifacts and activities in a visual tool for distributed software development teams." Proceedings. 26th International Conference on Software Engineering (2004): 387-396.

[34] Wettel, Richard & Lanza, Michele. (2007). Visualizing Software Systems as Cities. VISSOFT 2007 - Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis. 92-99. 10.1109/VISSOF.2007.4290706.

[35] Sasso, Tommaso & Minelli, Roberto & Mocci, Andrea & Lanza, Michele. (2015). Blended, not stirred: Multi-concern visualization of large software systems. 106-115. 10.1109/VISSOFT.2015.7332420.