Visually Analyzing the Structure and Code Quality of Component-based Web Applications

Hagen Tarner*, Daniel van den Bongard†, and Fabian Beck‡
paluno, University of Duisburg-Essen
Essen, Germany
Email: *hagen.tarner@paluno.uni-due.de, †daniel@vdbongard.com, ‡fabian.beck@paluno.uni-due.de

***Abstract*—Monitoring code quality and dependencies is an important task to keep software maintainable. While generally well researched, only little work on visually analyzing code quality of component-based front-end web applications exists that considers the specifics of such software systems. We propose an approach to visualize dependencies and code quality metrics of component-based JavaScript React applications. Our prototype implementation uses a node-link diagram for dependency visualization, tailored to the specific component structure of React applications and enriched with various visual cues. It is linked with different panels to show code quality and exact source code locations. Recommendations on how the quality of the system under analysis can be improved and refactored are provided. We evaluated our prototype in a small user study with four participants and found that it helped in program comprehension tasks and finding refactoring opportunities.**

***Index Terms*—JavaScript, React, software visualization, code quality metrics, dependency graph, refactorings**

I. Introduction

The Internet is not simple static hyperlinked texts anymore. More and more sophisticated JavaScript front-end applications run inside the web browser and become challenging to develop. The main programming language for realizing these applications of increasing complexity is JavaScript. To adapt to the needs of developers, the language increases its feature set constantly. One of the main features needed for being able to write complex software is the concept of modularization (or *components* in the context of web frameworks). We propose an interactive visualization for the analysis of component-based JavaScript front-end applications, that run in the web browser.

On the one hand, maintaining and further developing complex, component-based JavaScript front-ends is not different from other programming languages—it requires a solid understanding of the program's code and can be facilitated by software visualization. On the other hand, JavaScript front-end frameworks, specifically for usage in *single-page applications* (SPA), share some important characteristics that discern them from other code. Most of the popular web frameworks (e.g., *React*[1], *Angular*[2], or *Vue.js*[3]) follow the reactive programming paradigm. Using template languages to mix presentation and application logic, they can propagate changes in data to update the state of the front-end application.

[1]https://reactjs.org
[2]https://angular.io
[3]https://vuejs.org

In this work, we want to (I) visualize the structure of component-based JavaScript applications for interactive front-ends to facilitate understanding and (II) inspect their code quality to offer refactoring opportunities. We focus on one of the most popular [1] front-end frameworks, React, and its template language *JavaScript Syntax Extension* (JSX).

A common way of visualizing the structure of a software system is to model dependencies as a graph. To show dependencies between components, we use a tailored visualization based on a horizontally layered graph layout (see A in Figure 1). Our visualization of the dependency graph allows two levels of analysis: (I) an overview of the dependencies between components of the application (Figure 1) and (II) a detail-level perspective of the dependencies of functions of a single component (Figure 2).

Our prototype helps in understanding a software system by providing visual guidance on code quality metrics gathered by static analysis, which we show in a side panel for the current selection (see B in Figure 1). For the analysis of JSX template code, we added several specific metrics. Based on these metrics, our prototype further helps in identifying refactoring opportunities by providing visual cues in the node-link diagram and showing specific warnings for violating best practices.

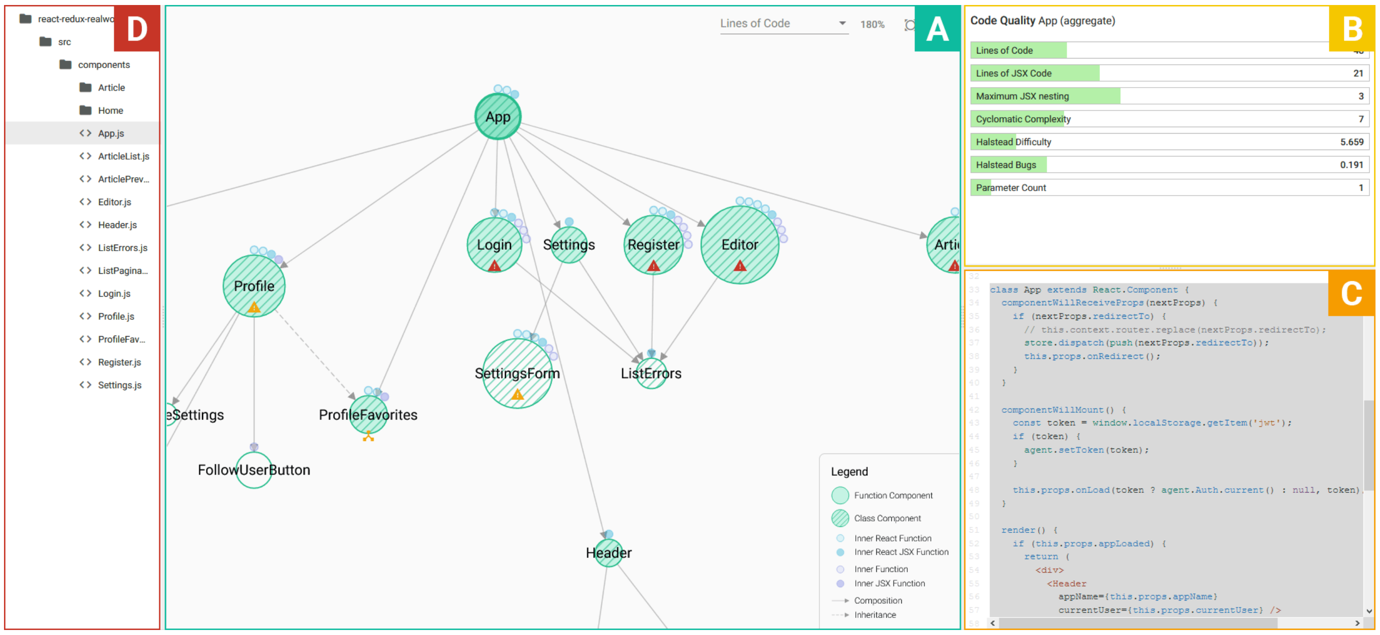
We evaluated our prototype in a user study with four participants, and were able to verify that it helps in comprehending an unknown codebase, as well as finding refactoring opportunities in an already known codebase.

The prototype is implemented as a web-based tool and available online at https://vis-tools.paluno.uni-due.de/component-graph/.

II. Related Work

Visualizing the code quality of a software system is a common topic in software engineering. There exist numerous tools to analyze and summarize code quality, often in dashboard-like interfaces (e.g., SonarQube[4]). Code quality has also been visualized as interactive reports using natural language generation [2], directly inside the code editor as in-situ text visualizations [3], or in a tabular format with embedded bar charts [4]. Dependencies, as part of the structure, are often modeled as graphs and visualized in node-link diagrams. These

[4]https://www.sonarqube.org



Code Quality App (aggregate)

Lines of Code	[illegible]
Lines of JSX Code	21
Maximum JSX nesting	3
Cyclomatic Complexity	7
Halstead Difficulty	5.659
Halstead Bugs	0.191
Parameter Count	1

```
class App extends React.Component {
  componentWillReceiveProps(nextProps) {
    if (nextProps.redirectTo) {
      // this.context.router.replace(nextProps.redirectTo);
      store.dispatch(push(nextProps.redirectTo));
      this.props.onRedirect();
    }
  }

  componentWillMount() {
    const token = window.localStorage.getItem('jwt');
    if (token) {
      agent.setToken(token);
    }

    this.props.onLoad(token ? agent.Auth.current() : null, token)
  }

  render() {
    if (this.props.appLoaded) {
      return (
        <div>
          <Header
            appName={this.props.appName}
            currentUser={this.props.currentUser} />
```

Fig. 1. Our tool consists of four panels: A the *Component Graph Panel* is a node-link diagram that visualizes the component-based structure with additional information, B the *Code Quality Panel* shows metrics of selected entities (the root component *App* is selected in this example), C the *Source Code Panel* provides the source code of the selected entity, and D the *File Tree Panel* displays the loaded directory and file structure of the application.

can be specifically tailored for one domain (e.g., OSGi [5]), to serve a specific purpose (e.g., finding code smells in Java projects [6] or extracting functionality into components [7]). One approach that applies node-link dependency graph visualization to JavaScript projects is Hunter [8]. Besides the dependency graph, Hunter also shows directory contents, file sizes, and the source code of a given JavaScript application. As Hunter's focus is program comprehension, it does not report any code quality metrics. An approach to simultaneously visualize structure and code quality is E-Quality [9]. Here, the shapes, sizes, and color of the nodes and links are used to encode different code quality metrics. This approach, however, is designed for object-oriented software systems in Java. Determining the code quality of JavaScript projects has already been addressed without the specific use of visualizations, for instance, detecting code [10] or dependency smells [11] and identifying refactoring opportunities [12], [13], [14]. However, we are not aware of any visualization approach for code quality of JavaScript projects or one that considers the specifics of component-based web applications.

III. Visualization Approach and Implementation

To understand the structure, monitor the code quality, and find refactoring opportunities in component-based JavaScript applications, we developed a prototype tool using *D3.js*[5] and Angular. The tool consists of four main views: (A) the Component Graph Panel, (B) the Code Quality Panel, (C) the Source Code Panel, and (D) the File Tree Panel (see Figure 1).

Our tool is specifically tailored to React-based web applications. React organizes a single-page application into individual components, which encapsulate the application logic and can output visual elements to the screen. For the latter, React uses *JavaScript Syntax Extension* (JSX) as a template language, that is based on a mixture of HTML and JavaScript. JSX allows writing JavaScript expressions (e.g., variables, loops, and conditional statements) directly inside HTML blocks.

It is crucial to consider these specifics in the visualization to provide a meaningful visualization of the structure of a React application. For instance, we enriched the main node-link diagram within the Component Graph Panel with separate representations of framework-specific functions and included metrics to specifically analyze JSX code in the Code Quality Panel.

Upon opening the application and loading source code, our tool computes the dependency graph and code quality metrics inside the browser. For the analysis to work on the client side, without the need for a back end, we use *Babel*[6] (for AST generation and analysis) and *TyphonJS-ESComplex*[7] (for code quality measurements). This architecture allows analyzing projects without any (potentially privacy- or copyright-sensitive) information being transferred to a server. Furthermore, it allows our tool to run completely offline, without the need of any network connection.

[5]https://d3js.org

[6]https://babeljs.io

[7]https://github.com/typhonjs-node-escomplex/typhonjs-escomplex

A. *Component Graph Panel*

The main view of our tool, the *Component Graph Panel*, is a node-link diagram in the center of the screen (see A in Figure 1). It supports two levels of analysis: an application-wide overview of all analyzed component and a component-wide detail level view.

1) Overview of the Application's Components: In its initial state, this panel gives an overview of all components in the analyzed codebase. This depiction of the dependency graph uses nodes for components and links for their relations. For the initial layout computation and to highlight the hierarchical aspect of component composition and inheritance, we used a constraint-based graph layout algorithm [15] provided by *WebCoLa*[8] to construct a layered graph layout. As with many computed node-link layouts, an optimal reading experience of the result is not always guaranteed. To mitigate this, our tool supports a force-directed graph layout[9] as an alternative to the layered graph layout—whereas it does not show the hierarchical structure as clearly, it often better reveals clusters of components. Furthermore, both layouts allow manual repositioning of nodes by dragging them with the mouse.

React components can be linked by either composition or inheritance. This is visualized as a solid or dashed line between the nodes, respectively. The direction of the relation is indicated by small arrow heads. React offers two ways of writing a component: either as a class or as a function. Knowing about these component types can be important for understanding and refactoring the application. To visualize this distinction, we use a striped filling for class components and a solid filling for function components. The size of the node encodes a user-selectable metric (for a list of supported metrics, see subsection III-B). This is set per default to the number of lines of code.

React components (both class and function types) often consist of a number of inner functions. The number and types of these function can give a first approximation of the component's complexity. To show the number and type of the inner functions, we arrange color-coded circles around each node, and highlight those functions that use framework-specific or JSX code in light blue while others are light purple.

2) Component Drill-Down: Double-clicking a node opens the second level of analysis of the Component Graph Panel (see A.2 in Figure 2). This detail-level view of a single component visualizes all inner functions. Again, we use light blue for framework-specific and light purple for generic JavaScript functions. The functions that return JSX code are marked with a double border. The layout of the node-link diagram in the Component Drill-Down is also controlled by a layered graph layout. On the top, there is a node for the component itself, in the first vertical level all framework-specific inner functions, and below all other functions. Similar to the Component Graph Panel, this can be switched to a force-directed layout.

Function calls inside a component are visualized as the links of the diagram. We include the component itself as a node to have a root node for the graph. This way, dangling nodes, which are not connected to any other node, indicate a function that is never called from inside the component.

To find clusters in the graph of inner functions of a component, we use the Louvain community detection method [16] for directed graphs. In case any clusters are found, we replace the previously described color-coding and assign each cluster a unique color (see A.C in Figure 2). The coloring mode can be switched with a button at the top of the screen.

B. *Code Quality Panel*

To evaluate the code quality of a component, we use a metric-based static analysis approach. Selected metrics covering different quality aspects are visualized in the Code Quality Panel as bar charts (see B in Figure 1). The list of metrics contains: Lines of Code, Cyclomatic Complexity, Halstead Difficulty, Halstead Bugs, and Parameter Count. These are computed for components as well as inner functions. For components, we also compute two extra metrics that deal with the possibility of JSX output: Lines of JSX Code and Maximum JSX Nesting. The first one gives the number of lines of code of all JSX root nodes combined, while the second one returns the maximum number of JSX element nesting.

Each of the metrics has a specific threshold, which is based on values given by McCabe IQ[10] (for general-purpose metrics) and personal experience (for special React metrics). The values visualized as bar charts are colored according to the metric's threshold (less than 0.8 of the threshold: green; between 0.8 and 1.0: orange; above 1.0: red). Below the bar charts, we show messages and warnings from the React documentation[11] regarding the deprecation of lifecycle methods.

To easily spot components with metric violations, we also add small icons to the relevant nodes in the Component Graph Panel (more than 0.8 of the threshold: yellow warning sign; above 1.0: red warning sign; for example nodes *Editor* and *SettingsForm* in A in Figure 1).

C. *Source Code Panel*

To inspect the details of the actual implementation, the Source Code Panel (see C in Figure 1) shows the source code of the file that contains the currently selected component or function. It uses syntax highlighting and line numbering to mimic the look-and-feel of an IDE. A light gray background is added to indicate the source code that belongs to the currently selected component or function in the Component Graph Panel. Depending on the selection, this can be the whole content of a file or just parts of it. This is useful especially in the Component Drill-Down view, where the user can select only parts of a whole file by clicking for example a single function.

[8]https://marvl.infotech.monash.edu/webcola
[9]https://github.com/d3/d3-force

[10]http://mccabe.com/iq.htm
[11]https://reactjs.org/docs/react-component.html#legacy-lifecycle-methods

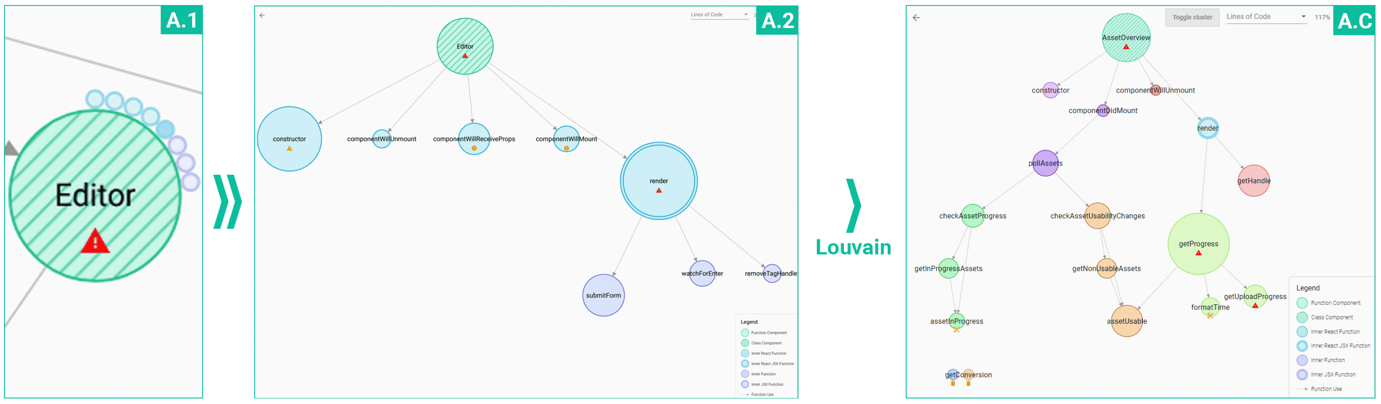


Fig. 2. Each node in the *Component Graph Panel* contains a preview of the inner structure of a component (small circles around the node in A.1). Double-clicking the node opens the *Component Drill-Down* (see A.2), that displays all inner functions inside a component and their relations. Nodes of framework-specific functions are colored light blue, components green, and regular JavaScript functions light purple. Inner functions, that return JSX Code have a double-stroked border. This coloring mode can be switched to instead show clusters, computed via the Louvain method (see A.C).

D. File Tree Panel

The File Tree Panel (see D in Figure 1) uses a file-explorer metaphor to show the directory structure of the project. The files are displayed as icons inside folders, that can be expanded and collapsed. The list of files is filtered to show only files, that can be analyzed inside the tool. Clicking a file highlights the contained components in the Component Graph Panel. Double-clicking scopes the Component Graph Panel to include only components from files in the selected folder.

IV. Evaluation: User Study

To evaluate our tool, we conducted a small-scale user study with four participants. The sessions were designed as *think-aloud* sessions and were carried out remotely via video conferencing systems. All participants have a background in software development and experience with React-based applications to a varying degree. Some reported prior experience with static analysis tools and code quality metrics. As all participants work in the same company; internal projects could be used for the study.

The study was divided into two parts: (I) analyzing an unknown codebase to demonstrate the program comprehension aspect of our tool, and (II) identifying refactoring opportunities in an already known codebase. For the first part of the study, we used an open-source React sample application[12], and for the second part, we used a company-own React application all participants were familiar with. Both applications are mid-sized in terms of number of files and components to be analyzed.

Part I consisted of several tasks, including identifying nodes based on certain criteria (e.g., by name, by number and type of relations, or by metric), finding functions in a given component, and identifying all components of a given JavaScript file. As React components can output visual elements by returning JSX code, we also added a task of identifying the render path of a single component. The render path in this context is the shortest path in the dependency graph from the root component to the JSX function of the given component.

[12]https://github.com/gothinkster/react-redux-realworld-example-app

Part II consisted of an evaluation of the metric threshold visualization, the display of warning messages and icons, and the cluster visualization.

We asked participants to perform these tasks and rate their experiences and findings on a scale from 1 (worst) to 5 (best). Participants rated the statement *"The approach increases program comprehension of a component-based application."* with an average of 4.75 (4, 5, 5, 5) out of 5 points. Regarding the code quality visualization aspects of our tool we asked the participants to rate the statement *"The approach helps in evaluating the code quality of a component-based application."*, and they answered with an average of 4.75/5 (4, 5, 5, 5). The statement *"The approach helps in refactoring a component-based application."* was rated with 4.5/5 (4, 4, 5, 5) points on average. We also asked the participants for feedback on more general aspects of our tool, and participants rated the structure of the user interface with 4.5/5 (4, 4, 5, 5,) points on average, and the interaction possibilities with 4.25/5 (4, 4, 4, 5) points on average.

As part of the think-aloud character of the sessions, participants were asked to voice every thought and impression. We observed some participants having minor difficulties in finding specific features that were needed to complete the tasks. Most notably, multiple participants struggled to open the Component Drill-Down and inspect a single component. To mitigate this, we added small explanatory texts to the tool afterwards. Other suggested improvements include increased highlighting of selected components, keyboard shortcuts, text-based component search, and the feature of highlighting a path between two selected nodes.

With the participants' positive feedback and the overall high scores in the evaluation, we believe that our tool is easy to understand and provides relevant insights on the structure and code quality of component-based web applications.

Nevertheless, our study is only a first step towards evaluating the approach and, more generally, the support for understanding modern web applications. Some obvious shortcomings relate to the small sample size of participants and a missing control group. This and a potential social-desirability bias restrict the interpretability of any quantitative measures given above. Also, the qualitative part of the evaluation (i.e., interpreting the thinking-aloud protocols) is limited by the simple nature of tasks and the restricted extent of analysis.

V. Future Work

Our tool covers a number of core features already and can be considered a basis for developing an even more versatile analysis framework for React applications. The following ideas give an outlook on meaningful future extensions.

a) Dynamic Metrics: The Component Graph focuses on metrics recorded by static analysis. To support improvements of behavior and performance characteristics of an application, one idea would be to include dynamic metrics in the prototype. Possible metrics could be execution times of a component's function or the number of instantiated components. This information can then be used to debug performance bottlenecks and gain insight into runtime dependencies between components.

b) Visualizing the State Machine: Modern single-page applications are often managed as state machines that implement the Flux application architecture (e.g., *Redux*[13] or *MobX*[14]). For debugging purposes, it is already an established practice to step through changes done to the internal state of the application and observe how the application reacts to these changes. Offering a way to include a visualization of the state machine could be a possible extension of our tool. Showing state-based interactions between components could further aid program comprehension.

c) Adding External Dependencies: Currently, our prototype focuses on dependencies inside components and between components of a single application. An extension could be to also include components of external React-based dependencies (e.g., the dependencies usually defined in a `package.json` file of a JavaScript project). This could be realized by extending the central graph by a third level that depicts the project within the dependencies to the components of these external libraries and sources.

VI. Conclusion

In this paper, we presented a visualization approach for understanding the structure and analyzing the code quality of component-based React applications. It visualizes dependencies and hints at refactoring opportunities. We evaluated our prototype in a small-scale user study. We showed that our prototype can help developers understand unknown codebases, as well as finding code locations in known code that violate best practices. The latter helps in identifying refactoring opportunities.

VII. Acknowledgments

This work has been funded by Deutsche Forschungsgemeinschaft (DFG) as part of research grant 288909335.

[13]https://redux.js.org
[14]https://mobx.js.org

References

[1] S. Greif and R. Benitte, "State of JS 2020," https://2020.stateofjs.com/en-US/technologies/front-end-frameworks/, 2020 (accessed July 12, 2021).

[2] H. Mumtaz, S. Latif, F. Beck, and D. Weiskopf, "Exploranative code quality documents," *IEEE Transactions on Visualization and Computer Graphics*, vol. 26, no. 1, pp. 1129–1139, 2020.

[3] E. R. Murphy-Hill and A. P. Black, "An interactive ambient visualization for code smells," in *Proceedings of the 5th ACM Symposium on Software Visualization*. ACM, 2010, pp. 5–14.

[4] H. Tarner, V. Frick, M. Pinzger, and F. Beck, "Visualizing evolution and performance metrics on method level as multivariate data," in *Proceedings of the 13th Seminar Series on Advanced Techniques & Tools for Software Evolution*. CEUR-WS.org, 2020.

[5] D. Seider, A. Schreiber, T. Marquardt, and M. Bruggemann, "Visualizing modules and dependencies of OSGi-based applications," in *Proceedings of the 4th IEEE Working Conference on Software Visualization*. IEEE Computer Society, 2016, pp. 96–100.

[6] E. van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Proceedings of the 9th Working Conference on Reverse Engineering*. IEEE Computer Society, 2002, pp. 97–107.

[7] A. Marx, F. Beck, and S. Diehl, "Computer-aided extraction of software components," in *Proceedings of the 17th Working Conference on Reverse Engineering*. IEEE Computer Society, 2010, pp. 183–192.

[8] M. Dias, D. Orellana, S. Vidal, L. Merino, and A. Bergel, "Evaluating a visual approach for understanding Javascript source code," in *Proceedings of the 28th International Conference on Program Comprehension*. ACM, 2020, pp. 128–138.

[9] U. Erdemir, U. Tekin, and F. Buzluca, "E-Quality: A graph based object oriented software quality visualization tool," in *Proceedings of the 6th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE Computer Society, 2011, pp. 1–8.

[10] A. M. Fard and A. Mesbah, "JSNOSE: Detecting Javascript code smells," in *Proceedings of the 13th IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE Computer Society, 2013, pp. 116–125.

[11] A. J. Jafari, D. E. Costa, R. Abdalkareem, E. Shihab, and N. Tsantalis, "Dependency smells in Javascript projects," *CoRR*, vol. abs/2010.14573, 2020.

[12] K. Gallaba, Q. Hanam, A. Mesbah, and I. Beschastnikh, "Refactoring asynchrony in Javascript," in *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution*. IEEE Computer Society, 2017, pp. 353–363.

[13] A. Feldthaus, T. D. Millstein, A. Møller, M. Schäfer, and F. Tip, "Tool-supported refactoring for Javascript," in *Proceedings of the 26th Annual ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. ACM, 2011, pp. 119–138.

[14] A. Feldthaus and A. Møller, "Semi-automatic rename refactoring for Javascript," in *Proceedings of the 28th ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. ACM, 2013, pp. 323–338.

[15] T. Dwyer, Y. Koren, and K. Marriott, "IPSep-CoLa: An incremental procedure for separation constraint layout of graphs," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 821–828, 2006.

[16] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, p. P10008, 2008.