2018-10

Scoped: Evaluating A Composite Visualisation Of The Scope Chain Hierarchy Within Source Code

Ivan Bacher
Technological University Dublin, ivan.bacher@tudublin.ie

Brian Mac Namee
University College Dublin, Ireland

John D. Kelleher
Technological University Dublin, john.d.kelleher@tudublin.ie

Recommended Citation

Bacher, I., Mac Namee, B. & Kelleher, J. (2018). Scoped: evaluating a composite visualisation of the schope chain hierarchy within source code. VISSOFT 2018:*6th. IEEE Working Conference on Software Visualization*, Madrid Spain, 24-25 September, 2018. 10.1109/VISSOFT.2018.00021

Scoped: Evaluating A Composite Visualisation Of The Scope Chain Hierarchy Within Source Code

Ivan Bacher
School of Computing
Dublin Institute of Technology
Dubin, Ireland
ivan.bacher@dit.ie

Brian Mac Namee
School of Computer Science
University College Dublin
Dublin, Ireland
brian.macnamee@ucd.ie

John D. Kelleher
School of Computing
Dublin Institute of Technology
Dublin, Ireland
john.d.kelleher@dit.ie

***Abstract*—This paper presents two studies that evaluate the effectiveness of a software visualisation tool which uses a composite visualisation to encode the scope chain and information related to the scope chain within source code. The first study evaluates the effectiveness of adding the composite visualisation to a source code editor to help programmers understand scope relationships within source code. The second study evaluates the effectiveness of each individual component within the composite visualisation. The composite visualisation is composed of a packed circle tree diagram (overview component) and a list view (detail view component). The packed circle tree functions as an abstract mini-map to provide viewers with a high-level overview of the scope chain hierarchy within a source code document. The list view provides additional information about identifiers (variables, functions, and parameters) that are accessible from the scope within which the cursor is located, in the source code document. Both studies utilise a between-subject design, in which groups of participants were presented with source code fragments and asked to answer a series of code understanding questions. The results of the studies indicate that adding a composite visualisation to a source code editor can have a positive effect on code understanding, especially when the textual representation of the code no longer corresponds to the actual behaviour of the code (as is the case, for example, in languages such as JavaScript that implement variable hoisting).**

***Index Terms*—Software Visualisation, Source Code Visualisation, Empirical Evaluation**

I. Introduction

Programmers can find it difficult to understand the static and dynamic aspects of source code, as well as the many types of relations and hierarchies that exist within code. In fact, previous studies [1], [2] have shown that understanding source code accounts for more than half of the software development effort. Hence, a major part of the effort invested in software development is dedicated to understanding source code [3].

The scope chain hierarchy, which is a fundamental structure implemented by most programming languages, can be described as a set of rules that control the visibility and lifetime of variables, functions, and parameters [4], [5]. Further, it can be seen as one of the hierarchies that programmers find difficult to understand within source code. Two aspects of the scope chain hierarchy can be especially confusing for programmers. First, each programming language has a slightly different implementation of scope. Second, since the scope chain can be seen as a hierarchy, it supports nesting. Therefore, scopes can be nested within each other. This means that if an identifier (variable, function, or parameter) cannot be found in the immediate scope, the corresponding scope chain is traversed, starting at the parent scope-level and continuing until the identifier is found or until the outermost (global) scope has been reached.

Bacher et al. [6] present a prototype visualisation tool called Scoped[1], which integrates a composite visualisation within a source code editor. The tool is designed to facilitate understanding of the scope chain hierarchy and information related to the scope chain hierarchy within a source code document (more information about Scoped can be found in Section II and [6]).

This work presents two studies. The first study evaluates the effectiveness of adding a composite visualisation to a source code editor to help programmers understand scope relationships within source code. The second study evaluates the effect of each individual component within the composite visualisation. The main contribution of this work is the design and implementation of both studies, as well as the presentation and discussion of the study results. The results of the studies show that combining visualisations that encode the scope chain hierarchy with existing software development tools can facilitate source code understanding, especially when the behaviour of the code no longer matches the textual representation of the code (as is the case, for example, in languages such as JavaScript that implement variable hoisting).

The remainder of this paper is structured as follows. Section II describes the Scoped tool. Section III presents related work, in regards to visual representations of source code in combination with source code editors, and an overview of some relevant previous software visualisation evaluation studies. Section IV describes both experiments, including research questions, the questions which the participants were asked, as well as independent, dependent, and controlled variables. Section V presents the results of both experiments and discusses their implications. Finally, Section VI summarises the main findings and presents directions for future work.

[1]Interactive demo: http://tiny.cc/jsscope

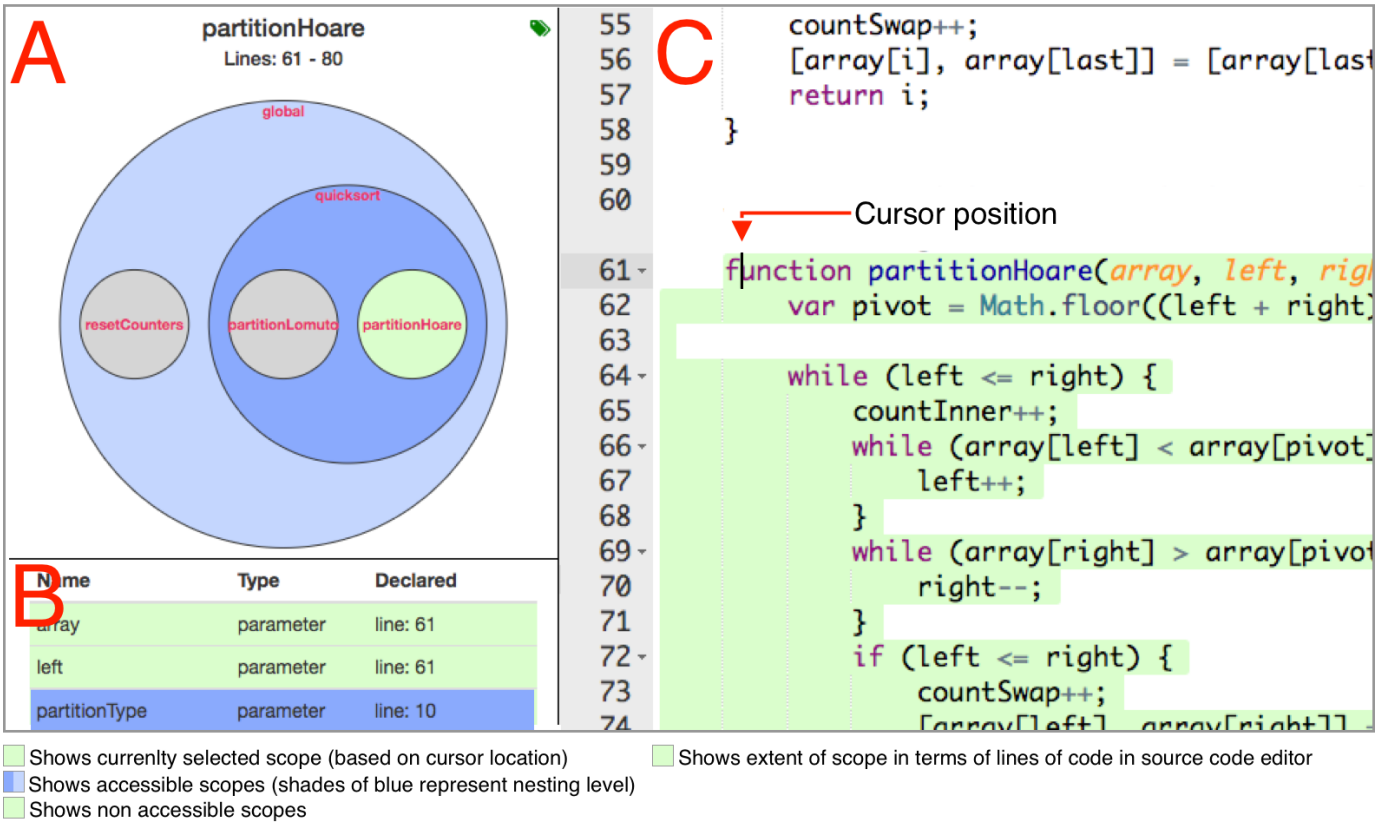

Fig. 1. The Scoped tool: A) Overview component, B) Detail view component, C) Source code editor

II. The Scoped Tool

Scoped [6] is the visualisation tool that this work evaluates. Figure 1 shows a screenshot of Scoped, which is designed to facilitate understanding of the scope chain and information related to the scope chain within a source code document. Scoped integrates a composite visualisation within a source code editor. The visualisation has two components. The first is a packed circle tree diagram [7] (Figure 1 component A) that functions as an abstract mini-map to provide viewers with a high level overview of the scope chain hierarchy within a source code document. The second is a list view (Figure 1 component B) that provides additional information corresponding to identifiers (variables, functions, and parameters) that are accessible from the scope within which the cursor is currently located, in the source code document.

Colour is used to link each of the components. Within the packed circle tree, green is used to show the scope in which the cursor is currently located. The colour blue is used to represent the parent and ancestor scopes which can be accessed from the currently selected scope. The colour grey is used to represent the remaining scopes that cannot be accessed for the currently selected one. The identifier information list view presents variables, functions, and parameters that can be accessed from the currently selected scope. Again, colour is used to identify local identifiers (green) and identifiers belonging to parent or ancestor scopes (blue). When a user hovers over a node in the packed circle tree diagram the corresponding program text within the source code editor is highlighted in green (as shown in the Figure 1).

Scoped has been designed with a number of common programming use cases in mind. For example, a common scenario in software development is that a programmer is tasked with re-factoring existing code. Before reading the code, the programmer can use Scoped to get an insight into the underlying complexity of a code fragment in terms of scope nesting. While reading code, a programmer typically has to scroll through it as the available screen real estate of a their display is limited. This can cause disorientation. The overview component of Scoped, however, can be used to limit disorientation by visually showing developers their current location within the scope chain hierarchy of a source code document. Furthermore, the overview component shows the scope in which the cursor is currently located, as well as which scopes are accessible and the list view shows which identifiers can be accessed from the current cursor position within the source code document. This provides the programmer with a visual way of answering common questions related to the understanding of scope during the process of reading, writing, or refactoring code.

III. Related Work

To facilitate source code understanding, it is important to maximise the readability of the textual representation of

the code [8]. Previous studies [9], [10] have shown that the typographic appearance of source code can influence the speed and accuracy of comprehension, by making the structural and syntactical composition of source code more visible (e.g. through the use of indentation and variations in font-face).

Cross et al. [11], [12] go even further and introduce the control structure diagram (CSD), which is a graphical representation that maps directly to source code and augments the program text in order to make the nesting and control flow of the code more explicit [13]. A controlled experiment was conducted to evaluate the effectiveness of CSDs [12]. The results of the experiment show that CSDs have a positive effect on program understanding in regards to shortening response times and increasing correctness of responses to questions posed about code [12]. This can be seen as initial evidence that integrating visualisations into existing software development tools can facilitate the process of understanding source code. However, while a CSD makes the nesting of code blocks within a source code document more visible, it does not explicitly encode information related to the scope chain.

Lommerse et al. [14] present code cushions, a technique that can be used to help programmers grasp the overall structure of one or more source code documents by making the extent of syntactic constructs, such as scopes and control structures, more visible. The technique maps directly to the code and augments the program text with geometric shapes whose outlines encompass a construct's text extent. However, in the context of facilitating a programmer's understanding of the scope chain within a source code document, the effectiveness of code cushions remains in question.

Wettel et al. [15] present a controlled experiment for the empirical evaluation of a software visualisation approach based on a city metaphor. They show that their approach leads to an improvement, in both correctness and completion time, over commonly used software exploration tools. Additionally, Wettel et al. present a list of experimental design guidelines which were distilled from an exhaustive survey of research dealing with experimental validation of software engineering, information visualisation, and software visualisation approaches. Important points, from Wettel et al.'s guidelines, that were considered during the implementation of this work include: (a) take into account the range of experience levels of the participants, and (b) report results on individual tasks.

In summary, the literature has shown that software visualisations can have a positive effect on facilitating the understanding of the structure, behaviour, and evolution of code. However, to the best of our knowledge, there are a limited number of software visualisations aimed at helping programmers understand the scope chain and information related to the scope chain within source code. Furthermore, Sensalire and Ogao [16] report that the current implementations of software visualisation tools lack many of the features desired by programmers, one being "*Integration into existing software development tools*". The authors state that this is a very important aspect, as even if a tool is able to generate amazing visualisations, the effort and time spent switching between the visualisation and code environment may have an effect on the knowledge preservation for programming, hence the desire for integration [16].

IV. Experiment Details

The studies presented in this work have been designed to evaluate the effectiveness of adding a composite visualisation to a source code editor, and to measure the effectiveness of each of the individual components within the composite visualisation in regards to facilitating the process of understanding scope relationships within source code. The main research question we aim to answer is: *Does combining a composite visualisation with a source code editor facilitate the process of understanding source code?*

We conducted two studies to evaluate the Scoped tool. The first study evaluated the effectiveness of adding a composite visualisation to a source code editor and the second study evaluated the effectiveness of each of the individual components within the composite visualisation. Both studies used a between-subject design. In the first study one group of participants (Group A) was presented with a standard source code editor with no visualisations present and the other group of participants (Group B) was presented with a source code editor that included the composite visualisation presented in Figure 1. In the second study one group of participants (Group C) was presented with a source code editor that included only the overview component (component A from Figure 1) and the other group of participants (Group D) was presented with a source code editor that included only the detail view component (component B from Figure 1).

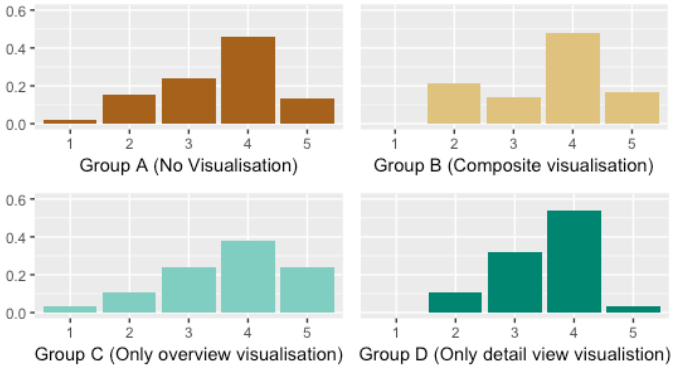

Fig. 2. Participant distribution of JS experience

All participants (from both experiments) were presented with the same source code documents, which were obtained from public code repositories[2]. The programming language used in the studies was JavaScript. Javascript is currently the most popular programming language according to the stackoverflow developer survey results 2016[3], 2017[4], and 2018[5].

[2]Experiment data: https://www.tiny.cc/scoped-exp-data
[3]https://insights.stackoverflow.com/survey/2016
[4]https://insights.stackoverflow.com/survey/2017
[5]https://insights.stackoverflow.com/survey/2018

In both experiments participants were presented with source code documents and asked to answer a series of questions[2] designed to interrogate their understanding of the scope chains within those documents. The questions were designed using an analysis of popular stack overflow questions related to understanding scope in source code [6]. Participants from both experiments were presented with the same set of questions. The ability of participants to correctly answer questions was used as a proxy measure for their understanding of scope within source code documents. By measuring the differences in the ability of participants to correctly answer questions under different conditions we explore the effectiveness of the visualisations included in Scoped.

The main compounding factor that could influence a participant's performance during the studies is their JavaScript programming experience level. In order to control for this factor, participants were asked to enter their self estimated programming experience level [17] using a 5 point Likert scale, where 1 indicated very inexperienced and 5 indicated very experienced. Figure 2 shows the distribution of the participants' self estimated experience level with the JavaScript programming language. The figure shows that all four groups have a similar distribution of novice to expert programmers. The results of a Kruskal-Wallis [18] test show that there is no statistically significant difference in JavaScript experience between the four groups (p-value >0.05).

V. Results & Discussion

For the first study, 88 participants were recruited. Each of these participants was randomly assigned to one of two conditions. Participants in Group A were presented with a source code editor with no visualisation (this was the baseline group), participants in Group B were presented with a source-code editor with the composite visualisation. 46 participants were allocated to Group A and 42 to Group B. All participants answered 6 code understanding questions related to the concept of scope. To measure potential differences between the groups on an aggregated level, the number of correct answers given by each participant was counted. Figure 3 presents the distribution of participant correctness scores for Group A and Group B. The plots show that the groups have different distributions and that a greater number of participants in Group B answered all of the questions correctly compared to Group A. Participants from Group A had an average correctness score of 4 and participants from Group B had an average correctness score of 5. A Kruskal-Wallis [18] test shows that there is a statistically significant difference between both groups (p-value <0.05). We believe that this is an interesting finding as it shows that participants presented with the composite visualisation were able to answer the code understanding questions with higher accuracy compared to participants not using the visualisation.

For the second study, an additional 57 participants were recruited, where 28 participants were allocated to Group C (overview visualisation) and 29 participants were allocated to Group D (detail view visualisation). The participants were given the same code documents and code understanding questions as in the first experiment. Figure 3 presents the distribution of participant correctness scores for Group C and Group D. The plots show that correctness scores for Group C and D were quite similar. Participants from Group C and Group D had an average correctness score of 5. Hence, we believe this shows that the type of visualisation does not have a big impact on participant correctness scores.

When comparing the results of both studies, participants in Group A (no visualisation) did worst and participants in Group B (composite visualisation) did best. A series of Kruskal-Wallis tests show that there is no statistical difference between any of the groups, except between Group A and Group B. Hence, the full composite visualisation is necessary for statistically significant positive effect on code understanding compared with the baseline.

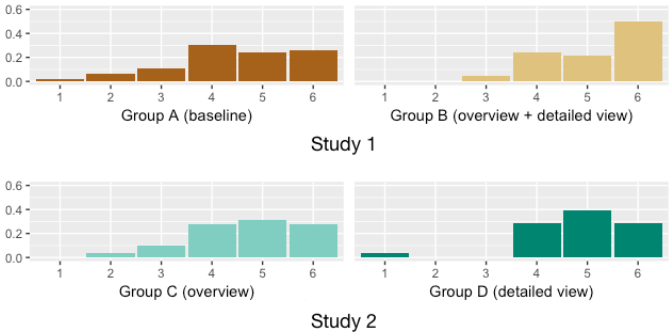

Fig. 3. Distribution of participant correctness scores between groups in the two studies

To get a more precise and in-depth understanding of participant correctness scores, results corresponding to each individual question were also analysed. Figure 4 displays a grouped bar chart, where each group of bars corresponds to an individual question and the length of each bar indicates the percentage of correct answers for that question. By examining Figure 4, we notice that for most of the questions, participants in Group B (composite visualisation), Group C (overview component), and Group D (detail view component) answered a greater number of questions correctly compared to Group A (no visualisation). Further, it seems that for three out of the six questions (Q2, Q3, and Q6) the percent of correct answers differs by a large amount for Group B (composite visualisation). After examining the source code associated with these questions more closely, we noticed that it incorporated a specific feature of JavaScript. The source code used for Q2, Q3, Q6 relied on hoisting [5, p.41], i.e. variables declared in a block scope (if, while, for, and switch statements) are hoisted to the upper function scope. This can be a great source of confusion for novice and experienced programmers alike, especially for programmers that are mainly familiar with languages such as C and Java that do not implement hoisting.

In summary, the results of the experiments show that combining a composite visualisation with a source code editor can have a positive effect on code understanding, especially

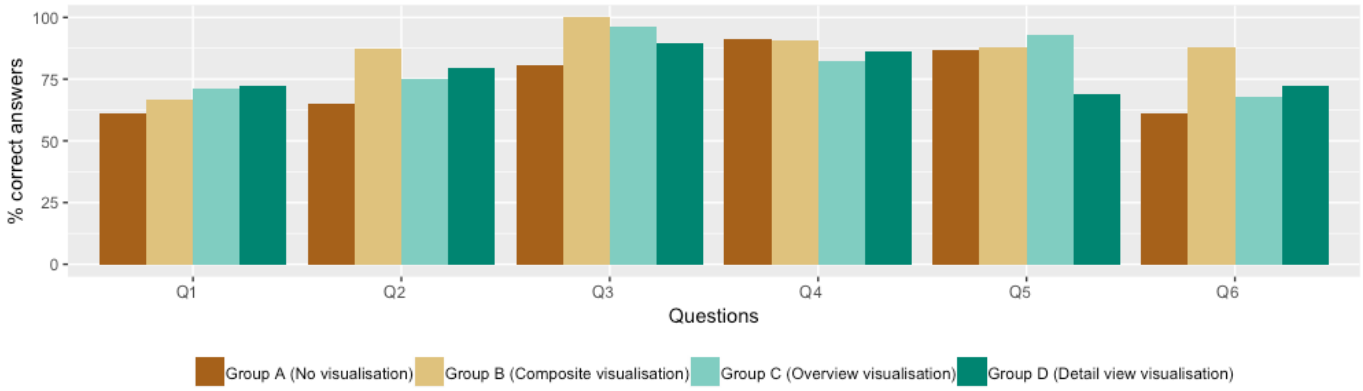

Fig. 4. Combined results for both experiments showing the percent of correct answers for each question.

in situations where the textual representation of the code no longer corresponds to the actual behaviour of the code. Additionally, the results show that both the overview and detail view components of the composite visualisation are important.

VI. Conclusion

This work presents two studies that evaluate the effectiveness of adding a composite visualisation to a source code editor in to help programmers understand scope relationships within source code, as well as the effectiveness of each individual component within the composite visualisation. The results of the experiments indicate that adding a composite visualisation to a source code editor can have a positive effect on code understanding and that both the overview and detail view components of the composite visualisation are important.

We believe that these studies are a first step toward gathering data on the effectiveness of combining source code editors with visual encodings of the scope chain and information related to the scope chain within a source code document. For future work, it would be interesting to conduct similar studies using debugging tasks, where participants are presented with a source code fragment that includes a scoping problem. Moreover, the visual encodings could also be used to encode other aspects within source code relating to the structure, behaviour, and evolution of the code.

The design and development of different approaches which can be used to encode the scope chain hierarchy within a source code document could also be a promising direction for future work. Interesting visualisation techniques to consider include the use of other tree visualisation techniques, such as tree-maps [19] and icicle trees [20]. Additionally, techniques which employ the code-map metaphor [21] are worth investigating, as these could provide a more natural mapping.

References

[1] A. Ko, B. Myers, M. Coblenz, and H. Aung, "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, 2006.

[2] T. A. Corbi, "Program understanding: Challenge for the 1990s," *IBM Systems Journal*, vol. 28, no. 2, pp. 294–306, 1989.

[3] A. C. Telea, *Data Visualization: Principles and Practice*, 2015.

[4] D. Crockford, *JavaScript: The Good Parts*, 2013, vol. 53, no. 9.

[5] K. Simpson, *You Don't Know JS: Scope & Closures*. O'Reilly Media, 2014.

[6] I. Bacher, B. Mac Namee, and J. D. Kelleher, "Scoped : Visualising the scope chain within source code," in *Proceedings of EG/VGTC Conference on Visualization (EuroVis 2017)*, 2017.

[7] W. Wang, H. Wang, G. Dai, and H. Wang, "Visualization of large hierarchical data by circle packing," *SIGCHI*, pp. 517–520, 2006.

[8] M. H. Clifton, "A technique for making structured programs more readable," *Newsletter ACM SIGPLAN Notices*, vol. 13, no. 4, pp. 58 – 63, April 1978.

[9] R. M. Baecker and A. Marcus, *Human Factors and Typography for More Readable Programs*, 1989.

[10] R. J. Miara, J. A. Musselman, J. A. Navarro, and B. Shneiderman, "Program Indentation and Comprehensibility," *Communications of the ACM*, vol. 26, no. 11, pp. 861–867, 1983.

[11] J. H. Cross, T. D. Hendrix, and S. Maghsoodloo, "Control Structure Diagram: An overview and initial evaluation," *Empirical Software Engineering*, vol. 3, no. 2, pp. 131–158, 1998.

[12] D. Hendrix, J. H. Cross, and S. Maghsoodloo, "The effectiveness of control structure diagrams in source code comprehension activities," *IEEE Transactions on Software Engineering*, vol. 28, no. 5, pp. 463–477, 2002.

[13] S. Diehl, *Software visualization: Visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media, 2007.

[14] G. Lommerse, F. Nossin, L. Voinea, and A. Telea, "The Visual Code Navigator: An interactive toolset for source code investigation," in *INFO VIS*. IEEE, Dec. 2005, pp. 25–32.

[15] R. Wettel, M. Lanza, and R. Robbes, "Software systems as cities: a controlled experiment," *2011 33rd International Conference on Software Engineering (ICSE)*, pp. 551–560, 2011.

[16] M. Sensalire and P. Ogao, "Visualizing object oriented software: Towards a point of reference for developing tools for industry," in *VisSoft 2007*, 2007, pp. 26–29.

[17] J. Feigenspan, C. Kastner, J. Liebig, S. Apel, S. Hanenberg, and K. Christian, "Measuring Programming Experience," *IEEE 20th International Conference on Program Comprehension (ICPC), 2012*, vol. 2005, no. of 2161, pp. 73–82, 2012.

[18] W. H. Kruskal and W. A. Wallis, "Use of ranks in one-criterion variance analysis," *Journal of the American statistical Association*, vol. 47, no. 260, pp. 583–621, 1952.

[19] B. Johnson and B. Shneiderman, "Tree-maps: A space-filling approach to the visualization of hierarchical information structures," in *Proceedings of the 2nd conference on Visualization'91*. IEEE Computer Society Press, 1991, pp. 284–291.

[20] J. Heer, M. Bostock, and V. Ogievetsky, "A tour through the visualization zoo," *Communications of the ACM*, vol. 53, no. 6, p. 59, 2010.

[21] I. Bacher, B. Mac Namee, and J. D. Kelleher, "The code-map metaphor: A review of its use within software visualisations," in *VISIGRAPP/IVAPP 2017*, Jan. 2017, pp. 17–28.