2018-10

# The Code Mini-Map Visualisation: Encoding Conceptual Structures Within Source Code

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

Brian Mac Namee
*University College Dublin, Ireland*, brian.macnamee@ucd.ie

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



## Recommended Citation

Bacher, I., Mac Namee, B., & Kelleher, J. (2018). The code mini-map visualisation: encoding conceptual structures within source code. *VISSOFT 2018: 6th. IEEE Working Conference on Software Visualization*Madrid, Spain, 24-25 September.

# The Code Mini-Map Visualisation: Encoding Conceptual Structures 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*—Modern source code editors typically include a code mini-map visualisation, which provides programmers with an overview of the currently open source code document. This paper proposes to add a layering mechanism to the code mini-map visualisation in order to provide programmers with visual answers to questions related to conceptual structures that are not manifested directly in the code. Details regarding the design and implementation of this scope information layer, which displays additional encodings that correspond to the scope chain and information related to the scope chain within a source code document, is presented. The scope information layer can be used by programmers to answer questions such as: to which scope does a specific variable belong, and in which scope is the cursor of the source code editor currently located in. Additionally, this paper presents a study that evaluates the effectiveness of adding the scope information layer to a code mini-map visualisation in order to help programmers understand scope relationships within source code. The results of the study show that the incorporating additional layers of information onto the code mini-map visualisation can have a positive effect on code understanding.**

***Index Terms*—Visualization, Code mini-map, Evaluation**

## I. Introduction

Source code editors are common tools that programmers use when writing, reading, or modifying code. All of these tasks involve understanding the microscopic and macroscopic details within the code [1]. The microscopic details include the mechanics of classes and methods which can be examined in the text of the code, while the macroscopic details include concrete high level concepts such as modules, system, and conceptual structures that are not manifested directly in the code. Many editors, including Atom[1] and Sublime Text[2], incorporate a visualisation which presents the viewer with a bird's eye view of the currently open source code document. This visualisation is, frequently, called a code mini-map.

The code mini-map visualisation is based on the code-map metaphor [2] and acts as an overview component mapping source code to a zoomed out representation, either by the use of pixels, pixel lines, or a scaled down representation of text, presenting viewers with a zoomed out view of the currently open source code document showing the layout of the code. This is important due to the fact that the layout of the code, as intended by the original developer, often conveys a great deal of information (e.g. associations or relations by physical proximity) [3]. Figure 1 shows an example of a code mini-map being used in a code editor.


Fig. 1. Code mini-map integrated into Atom

The code mini-map provides viewers with an overview of the currently open source code document. The source code editor provides the viewer with a view of the textual representation of the code, which corresponds to the microscopic details programmers need to understand when working with code. However, an open question remains on how to present programmers with information corresponding to the macroscopic details within the code (e.g conceptual structures that are not manifested directly in the code). In this work we introduce a layering concept to the code mini-map visualisation. This layering concept can be described as a mechanism for providing additional information to the viewer, while using the same underlying visualisation (the code mini-map). This approach is similar to the way a user is able to add additional information to an online map, for example traffic layers, street name layers, or elevation layers. The contributions of this work are the design and implementation of a visualisation approach which augments a code mini-map visualisation to include additional information corresponding to the macroscopic details within source code, as well as, a study that evaluates the augmented code mini-map

[1]https://atom.io/
[2]https://www.sublimetext.com/

visualisation and provides empirical data on the usefulness of this approach in the context of source code understanding

The remainder of this paper is structured as follows. Section II presents related work in the context of visualisations similar to the code mini-map. Section III presents details regarding code understanding issues. Section IV presents a description of the code mini-map as well as a description of the scope information layer that this work evaluates. Section V presents details corresponding to the evaluation study. Section VI presents the results of the controlled experiment and discusses their implications. Finally, Section VII summarises our findings and proposes directions for future work.

## II. Related Work

To the best of our knowledge, there are a limited number of visualisations similar to the code mini-map that are aimed at helping programmers understand the macroscopic details within a source code document. Bacher et al. [2] conducted a review of 21 software development tools, all of which employ a visualisation that is based on the code-map metaphor, which can be defined as the mapping of source code to a zoomed out representation to allow programmers to comprehend various statistics collected at the level of detail of individual lines of code.

Bacher et al. state that in general, the main motivation for producing visualisations based on the code-map metaphor is the need for a direct mapping from a visual representation to the source code and back. This direct mapping is needed in order to support the encoding of specific properties and relationships, to make programmers aware of promising locations within the code to motivate further exploration. The main goal of using visualisations based on the code-map metaphor is to provide programmers with a "big picture" view of a code base or a code file, while still being able to understand information collected at the source code level of detail. This information can include execution traces, search query results, code ownership, and code age. A main finding of the review was that although the literature indicates that visualisations based on the code-map metaphor are perceived by the research community to be helpful across all aspects of the software development process, there is a lack of quantitative evidence to support this perception. Therefore, the effectiveness of visualisations incorporating the code-map metaphor are still unclear, especially in the context of facilitating source code understanding.

DeLine et al. [4] present the Code Thumbnail Scrollbar, which supplements a code document's vertical scrollbar with a thumbnail image of the entire document. The Code Thumbnail Scrollbar is intended to allow programmers to form spatial memory of the code and provides a stable, one-dimensional space per document, with visual landmarks to help the viewer distinguish different parts at a glance (e.g. the code shape). The authors provide quantitative evidence that if present, the visualisation will be used by programmers for the tasks of code exploration, navigation, and selection. This is an interesting and important finding as it provides initial evidence that a visualisation similar to the code mini-map can be useful for exploring and navigating a source code document. Furthermore, the authors present possible feature suggestions that were obtained from participant feedback. A relevant feature suggestion for this current paper was the use of colour highlighting to show specific details which correspond to method definitions (callers, callees, and field uses). In the context of the code mini-map layering approach introduced in this paper, it is possible to add the method definition highlighting feature to the code mini-map as an additional layer, allowing programmers to activate this layer when attempting to answer questions related to specific methods within a source code document.

Robbes et al. present Microprints [5], [6] which is a visualisation technique used to show information to programmers when faced with the task of reading object-oriented code. The authors state that the main information that programmers are looking for can be classified into three categories: 1) state changes and accesses, 2) method control flow, and 3) method invocations or object interactions. Encoding all of this information into a single visualisation would lead to an unreadable picture, as far too much information would be displayed. Hence, the authors propose three variations of Microprints, where each variation is specialised on one of the above mentioned categories. Each variation also uses a different colour mapping to encode the corresponding information. The main difference between the visualisations presented by Robbes et al. [5] and the code mini-map visualisation is that the code mini-map shows the viewer a zoomed out representation of a source code document, whereas Microprints show the viewer a zoomed out representation of an individual method from three different perspectives. It would be possible to incorporate additional layers to the code mini-map which also show the same information as the Microprints.

## III. Source code understanding

Source code has several distinctive properties [7]. For example, it is written in programming languages that have strictly defined grammars with non-ambiguous semantics and it contains many types of relations and hierarchies. Cherubini et al. [1] state that programmers need to understand both the microscopic and macroscopic details within the code, where the microscopic details include the mechanics of classes and methods which can be examined in the text of the code and the macroscopic details include concrete high level concepts such as module, system, and conceptual structures that are not manifested directly in the code. An example of a macroscopic detail within source code is the scope chain hierarchy.

The scope chain hierarchy is a fundamental conceptual structure implemented by almost every programming language, and can be seen as a set of rules that control the visibility and lifetime of variables, functions, and parameters [8], [9]. This hierarchy is important to the programmer because it can be used to reduce identifier naming collisions and also provides automatic memory management [8]. Two aspects of the scope chain hierarchy can be a cause of confusion for

programmers. First, each programming language has a slightly different implementation of scope. Second, because the scope chain can be seen as a type of hierarchy, it supports nesting. Therefore, scopes can be nested within each other, meaning 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.

In order to come up with real world tasks which can be used as a proxy to measure code understanding, the information needs of programmers should be taken into consideration. LaToza and Myers [10] surveyed 179 professional developers about hard-to-answer questions they asked during the process of creating, debugging, and understanding code. Interesting questions in the context of scope issues include where (in which scope) was a specific variable defined and where in the code can a global variable be changed. Bacher et al. [11] conducted an analysis of the 50 most popular stack overflow questions in the context of scope understanding issues within source code. The authors found that programmers have difficultly understanding issues regarding the declaration, accessibility, and state of an identifier (variable, function, or parameter).

As many source code maintenance and comprehension issues are due to the poor understanding of scope [11], we believe that adding a layer to the code mini-map visualisation which encodes information related to the scope chain hierarchy within a source code document can facilitate source code understanding. Hence, we will be focusing on tasks that evaluate a programmer's ability to answer questions such as in which scope is an identifier declared, and can an identifier be accessed from a specific line of code? We believe that these questions relate to real world code understanding and debugging issues that programmers face on a daily basis.

## IV. Code mini-map scope layer

This work proposes to add additional encodings to the existing code mini-map visualisation present in many modern source code editors and integrated development environments. The mechanism for providing additional information to the viewer can be described as a layering approach, where additional encodings are superimposed onto the code mini-map to help programmers answer specific questions related to the source code.

Figure 1 shows an example of a code mini-map. The code mini-map reduces each line of code to one pixel line, keeping the same line layout. This is an important aspect of the code mini-map as the layout of the code often conveys a great deal of information, such as associations or relations by physical proximity [3]. Additionally, a grey highlighted area is also added to the code mini-map which shows the current dimension of the source code editors viewport so viewers can see which section of the code document is currently visible on screen. The syntax highlighting present on the textual

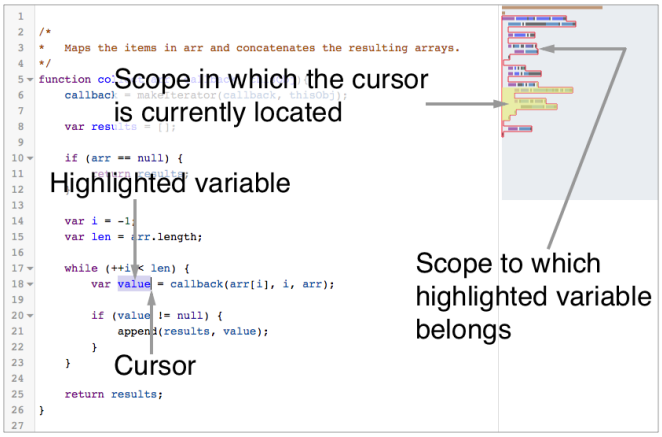


Fig. 2. Example of code mini-map with additional of encodings superimposed

representation of the code can also be transferred to the code mini-map.

Figure 2 shows a code editor combined with a code mini-map which includes the scope information layer. The yellow highlighted area on the code mini-map encodes the extent of the scope in which the cursor is currently located. The red border on the code mini-map shows to which scope a highlighted variable belongs. Each time the cursor changes location within the code editor the code mini-map is updated. If a programmer is interested in seeing to which scope a specific variable belongs, then the programmer must highlight that specific variable within the code editor. Figure 2 shows an example of this behaviour. In the example, the variable *value* is highlighted within the code editor. The red border on the code mini-map now shows the scope to which the variable *value* belongs. By looking at the code mini-map we can see that the variable *value* does not belong to the scope in which the cursor is currently located. Additionally, by placing the mouse pointer either in the yellow highlighted area or within the red border, the corresponding lines of code are highlighted within the code editor. Thus, the viewer can see which lines of code belong to either the scope in which the cursor is located in or the scope to which a specific variable belongs to. An interactive demo of the code map can be found here: http://tiny.cc/cmm-scope.

## V. Experiment details

The purpose of the experiment presented in this work is to evaluate the effectiveness of using the code mini-map to encode additional information corresponding to the macroscopic details within a source code document. In this case the macroscopic details align with facilitating a programmers understanding of the scope chain and information related to the scope chain within a source code document. Previous research [12]–[14] was used to guide the design, implementation, and organisation of the experiment.

The main research questions we aim to answer is: Does adding a scope information layer to a code mini-map visualisation increase the ability of programmers to understand the impact of scope on the correctness of their code?

To measure the effectiveness of adding additional layers of information to a code mini-map visualisation, 60 participants were randomly split into two groups where Group A was presented with a standard code mini-map visualisation and Group B was presented with a code mini-map visualisation that included additional encodings (the scope information layer). All participants were shown the same source code[3] and given a set of questions[3] that corresponded to scope understanding issues. The questions given to the participants were used as a proxy to measure the effect of both versions of the code mini-map on source code understanding (if a question is answered correctly).

The programming language used in this experiment is JavaScript, due to the fact that it is the most popular programming language according to the stack-overflow developer survey results 2016[4], 2017[5], and 2018[6]. Hence, the main factor that could influence a participant's performance during the experiment is their JavaScript programming experience level. To control for this confounding factor, each participant was randomly placed into one of two groups (Group A or Group B). Additionally, each participant was also asked to fill out a short questionnaire at the beginning of the experiment and the results of this questionnaire were used in a post-hoc analysis of the experiment to ensure that the randomisation process worked correctly. The goal of this questionnaire was to gather information corresponding to each participant's programming experience level. Participants were asked to enter their self estimated experience level using a 5 point Likert scale where each number within the scale was replaced with a level of Dreyfus' model of skill acquisition [15, p.162]: 1) Novice, 2) Advanced beginner, 3) Competent, 4) Proficient, and 5) Expert.

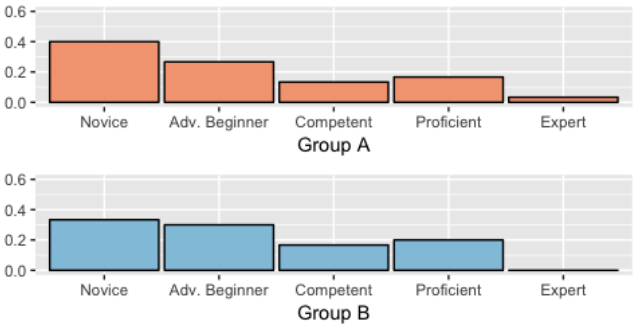


Fig. 3. Participant JavaScript programming experience

Figure 3 shows the participants' self estimated experience level with the JavaScript programming language. The images show that the both groups have a similar distribution of novice to expert programmers. The results of a Kruskal-Wallis test [16] show that there is a no statistically significant difference between the groups when comparing JavaScript programming experience (p-value $>0.834$).

[3]http://tiny.cc/cmm-scope-data
[4]https://insights.stackoverflow.com/survey/2016
[5]https://insights.stackoverflow.com/survey/2017
[6]https://insights.stackoverflow.com/survey/2018

## VI. Results and Discussion

All participants answered 20 code understanding questions related to scope understanding issues. Group A (code mini-map) consisted of 30 participants and Group B (code mini-map + scope info layer) consisted of 30 participants. In order to measure potential differences between the groups on an aggregated level, the number of correct answers given by each participant was counted.

Participants in Group A answered a total of 600 questions, where 356 (59.3%) were answered correctly and 244 (40.7%) incorrectly. Participants in Group B also answered a total of 600 questions, where 421 (71.2%) were answered correctly and 244 (29.8%) incorrectly. A Pearson's Chi-squared test [17] shows that there is a statistically significant difference between the groups when comparing the percent of correctly answered questions (p-value = $<0.001$).

Wettel et al. [14] suggest also reporting results on individual tasks/questions as this allows for a more precise and in-depth analysis of the strengths and weaknesses of an approach. Hence, to get a more precise and in-depth understanding of the number of correctly answered questions per group, results corresponding to each individual question were also analysed. Specifically, the number of correct answers for each individual question. 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. The colour of each bar represents to which group that bar corresponds.

By examining Figure 4, we notice two interesting things. The first is that for most of the questions (16/20), participants from Group B answered a greater number of questions correctly compared to participants from Group A. The second is that for 5 questions (Q3, Q5, Q11, Q12, Q14, Q15) the percent of correct answers is below 50% for both groups. This is interesting, as upon closer inspection, these questions were the only questions which corresponded to a specific, and tricky, feature of scope within JavaScript. This feature can be described as variable hoisting, e.g. variables declared in a block scope (if, while, for, and switch statements) are hoisted to the upper function scope rather than belonging to the block scope. Hence, the textual representation of the code no longer corresponds to the actual behaviour of the code

These results confirm that a specific feature of a programming language, such as variable hoisting, can be difficult to understand for programmers. Using a visualisation to encode this behaviour, in order to facilitate code understanding, has provided initial positive results. Therefore, we believe that visualisations can be particularly useful 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) and that these visualisations in turn, can be used to facilitate code understanding.

In summary, our current findings indicate that adding an additional layer of information to the code mini-map visual-

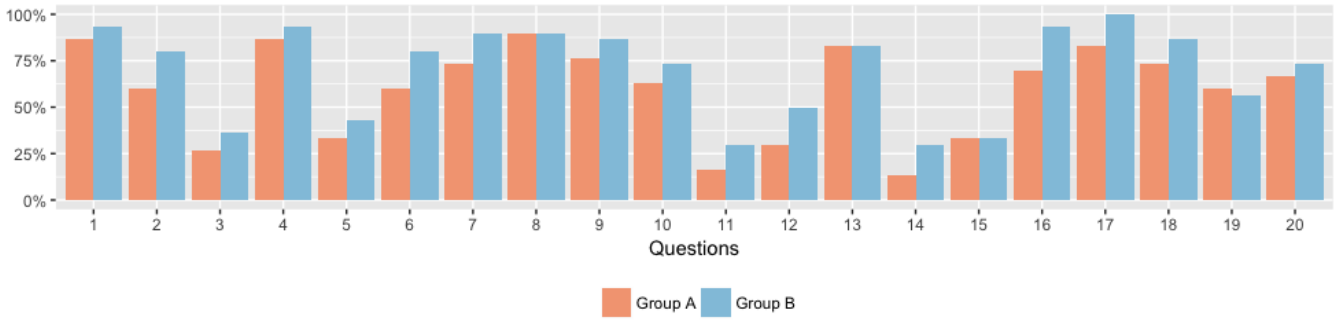


Fig. 4. Percent of correct answers for each individual question

isation can have a positive effect on code understanding. We have shown that this is the case for scope understanding issues in the context of JavaScript source code. However, we also believe that these results can be generalised in terms of using the code mini-map visualisation to encode other macroscopic details within a source code document.

## VII. Conclusion

This work introduces a layering concept to the code mini-map visualisation, which can be described as a mechanism for providing additional information to the viewer, while using the same underlying visualisation. The layering concept is similar to the way a user is able to add additional information to an online map.

As an example of this approach, this work presents a scope information layer, which is a layer of encodings that are superimposed into the code mini-map visualisation, in order to facilitate a programmer's understanding of the scope chain and information related to the scope chain within a source code document. The scope information layer can be used to show a viewer in which scope the cursor of a source code editor is currently located and to which scope a specific variable belongs. Additionally, this work presents a study that evaluates the scope information layer superimposed onto the code mini-map visualisation. The results of the study show that superimposing an additional layer of encodings onto a code mini-map has a positive effect on code understanding, in the context of scope understanding issues in source code written in JavaScript.

This experiment is a step towards providing quantitative data on the effectiveness of the code mini-map visualisation in the context of facilitating source code understanding. For future work, we believe that it would be interesting to conduct similar experiments using a series of debugging tasks, rather than code understanding questions. Another direction for future work entails the design, implementation, and evaluation of additional layers that can be added to the code mini-map visualisation. These layers can encode either the microscopic details within the code, the macroscopic details within the code, or a combination of the two. For example, code execution hotspots, code ownership, search query results, function call hierarchies, or control structure hierarchies.

## References


[1] M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko, "Let's go to the whiteboard: how and why software developers use drawings," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2007, pp. 557–566.

[2] I. Bacher, B. Mac Namee, and J. D. Kelleher, "The code-map metaphor-a review of its use within software visualisations." in *VISIGRAPP (3: IVAPP)*, 2017, pp. 17–28.

[3] J. I. Maletic, M. L. Collard, and A. Marcus, "Source code files as structured documents," in *Program comprehension, 2002. proceedings. 10th international workshop on*. IEEE, 2002, pp. 289–292.

[4] R. DeLine, M. Czerwinski, B. Meyers, G. Venolia, S. Drucker, and G. Robertson, "Code thumbnails: Using spatial memory to navigate source code," in *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*. IEEE, 2006, pp. 11–18.

[5] R. Robbes, S. Ducasse, and M. Lanza, "Microprints: A pixel-based semantically rich visualization of methods," in *Proceedings of 13th International Smalltalk Conference, ISC*, vol. 5, 2005, pp. 131–157.

[6] S. Ducasse, M. Lanza, and R. Robbes, "Multi-level method understanding using microprints," in *VISSOFT 2005*. IEEE, 2005, pp. 1–6.

[7] A. C. Telea, *Data visualization: principles and practice*. CRC Press, 2014.

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

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

[10] T. D. LaToza and B. a. Myers, "Hard-to-answer questions about code," *Evaluation and Usability of Programming Languages and Tools on - PLATEAU '10*, pp. 1–6, 2010. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1937117.1937125

[11] 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.

[12] H. Lam, E. Bertini, P. Isenberg, C. Plaisant, H. Lam, E. Bertini, P. Isenberg, C. Plaisant, S. C. Empirical, and H. Lam, "Empirical Studies in Information Visualization : Seven Scenarios," *IEEE Transactions on Visualization and Computer Graphics (2012)*, 2012.

[13] A. J. Ko, T. D. LaToza, and M. M. Burnett, "A practical guide to controlled experiments of software engineering tools with human participants," *Empirical Software Engineering*, vol. 20, no. 1, pp. 110–141, 2015.

[14] 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.

[15] G. Cheetham and G. E. Chivers, *Professions, competence and informal learning*. Edward Elgar Publishing, 2005.

[16] 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.

[17] R. L. Plackett, "Karl pearson and the chi-squared test," *International Statistical Review/Revue Internationale de Statistique*, pp. 59–72, 1983.