2017

Scoped: Visualising the scope chain within source code

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

Brian Mac Namee
Technological University Dublin, brian.macnamee@tudublin.ie

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

Eurographics Conference on Visualization (EuroVis) 2017
B. Kozlíková, T. Schreck, and T. Wischgoll
(Guest Editors)

Short Paper

Scoped: Visualising the scope chain within source code

I. Bacher[1], B. Mac Namee[2], J. D. Kelleher[1]

[1]Dublin Institute of Technology, Dublin, Ireland
[2]University College Dublin, Dublin, Ireland

Abstract

This paper presents an interactive visualisation tool that encodes the scope chain, and information related to the scope chain, within source code. The main goal of the tool is to support programmers when dealing with issues related to scope and to provide answers to questions such as to which scope does a specific variable or function belong to and can I access a specific variable from the scope I am currently located in. The design guidelines followed during the implementation of the tool, as well as the design rationale behind the main features of the tool are described. Finally, the results of a pilot user experience evaluation study are presented where an interesting observation was that the tool seemed to support programmers in verifying and correcting their assumptions when asked questions about specific scoping issues within a source code document.

Categories and Subject Descriptors (according to ACM CCS): I.3.8 [Computer Graphics]: Applications—

1. Introduction

A major part of the effort invested in software development is dedicated to understanding source code [Tel15]. Previous studies [Cor89, KMCA06] show that understanding source code accounts for more than half of the software development effort. This effort includes understanding the static and dynamic structure of the code, as well as the many types of relations and hierarchies that exist within the code. Using visualisations can facilitate source code understanding. For example, Hendrix et al. [HCM02], Miara et al. [MMNS83], and Wettle et al. [WLR10] illustrate the effectiveness of a software visualisation in regards to source code understanding activities through controlled experiments.

A fundamental paradigm that nearly all programming languages implement is a set of rules that control the visibility and lifetime of variables, functions, and parameters [Cro13, Sim14]. This set of rules is called *scope*. Figure 1 illustrates an example of scope and the *scope chain* within a source code fragment. The example contains three nested scopes, where *scope 1* encompasses the global scope and has one identifier in it: *foo*. *Scope 2* encompasses the scope of *foo*, which includes the three identifiers: *a*, *bar*, and *b*. Finally, *scope 3* encompasses the scope of *bar*, and it includes one identifier: *c*.

Scope is important to the programmer because it reduces naming collisions and provides automatic memory management [Cro13]. However, two aspects of scope can be a source of confusion for programmers. First, each programming language has a slightly different implementation of scope. Second, scope 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 next scope-level and continuing until the identifier is found or until the outermost (global) scope has been reached.

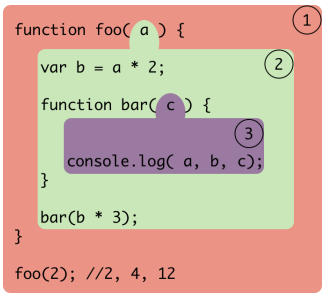

Figure 1: *Code example containing three nested scopes [Sim14]*

This work presents an interactive visualisation tool, which aims to support programmers in understanding the scope chain and information related to the scope chain during the activities of creating, understanding, and debugging code. A demo of Scoped, the interactive visualisation tool, can be found at: *http://tiny.cc/jsscope*. The main contributions of this work are threefold:

- An investigation into the issues programmers face in regards to the scope chain and information related to the scope chain within JavaScript source code.
- The design and implementation of Scoped, a novel interactive visualisation tool aimed at aiding programmers in understanding

the scope chain, as well as information related to the scope chain within source code documents written in JavaScript.
- A pilot user experience evaluation study which gathered further design requirements and investigated user behaviour when interacting with Scoped.

The remainder of this paper is structured as follows. Section 2 presents related work in regards to visual representations of source code in combination with source code editors. Section 3 discusses comprehension issues programmers face, with particular focus on issues specific to JavaScript and the scope chain within JavaScript. Section 4 describes Scoped and the design rationale behind various aspects of the tool. Section 5 presents findings from a pilot user experience evaluation study. Finally, section 6 presents implications for future work.

2. Related work

To the best of our knowledge, there are a limited number of visual tools aimed at helping programmers understand the scope chain and information related to the scope chain within source code.

Microprints [DLR05] is a visualisation technique which builds on the code-map metaphor [BMK17] and encodes the structure of source code. The technique can be used to show information related to state changes, method control flow, and method invocations. The authors propose three variations of the technique, specialised on each of these aspects. Each variation uses a different colour mapping to encode information. For example when dealing with state changes, assignments are displayed in red, method arguments in purple, the self variable in blue, instance variables in cyan, and global variables in yellow.

Bacher et al. [BMK16b, BMK16a] present two visualisation tools that use tree visualisation techniques to encode the various hierarchies within a source code. The first tool [BMK16b] uses an icicle tree [HBO10b] in combination with a source code editor to encode the hierarchical structure of the nested elements (tags) in HTML documents. The second tool [BMK16a] incorporates icicle trees, node-link trees, and circular tree maps in combination with a source code editor, to encode the control structure and scope chain within source code. Both of the visualisations are linked to the source code editor. Moving the cursors to a different element, control structure, or scope within the code will update the visualisations accordingly.

Cross et al. [CHM98] introduce the control structure diagram (CSD), which augments indented program text in order to make the nesting and scope of source code more explicit [Die07]. The authors state that many programmers consider the source code to be the only trusted specification of the software, therefore, the visualisation adds graphical constructs to the code, without disrupting its familiar appearance. For example, vertical lines are drawn over the code in order to show the extent of code blocks, and vertically stretched oval lines are used to show the extent of loops. Furthermore, diamond glyphs are used to indicate the alternatives of conditional statements. The primary purpose of the CSD is to reduce the time required to comprehend source code by clearly depicting the control constructs and control flow at all relevant levels of abstraction. In a controlled experiment, the authors show that CSD has a positive effect on program understanding in regards to shortening response times and increasing correctness [HCM02].

It is also worth mentioning that modern browsers such as Google Chrome and Mozilla Firefox provide developers with a set of web authoring and debugging tools. These tools can be used to set JavaScript breakpoints, which enable the user to pause the execution of a program on a specific line of code. While paused, a pane can be used to view the values of variables in the local, ancestor, and global scopes. However, the code must first be executed to achieve this and no visual overview of the scope chain is provided.

3. Code comprehension issues

It is difficult to consider visualisation without also considering the task it is meant to support. It is unlikely that any single visualisation tool can address all issues related to source code understanding simultaneously. In order to design and develop an interactive visualisation tool that facilitates understanding a scope chain and information related to it, we must first examine the comprehension difficulties associated with source code understanding, specifically those related to scope.

LaToza and Myers [LM10] surveyed 179 professional developers about hard-to-answer questions they asked during the process of creating, debugging, and understanding code, in order to get a better understanding of a developer's information needs. The authors were able to synthesise 94 distinct questions which could be split into 21 categories. 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. Fard and Mesbah [FM13] propose a list of *code smells* for applications written in JavaScript. Code smells are patterns in source code that indicate potential comprehension and maintenance issues. Once detected, code smells need to be re-factored to improve the design and overall quality of the code [FM13]. The list is composed of 13 code smells, where 7 are existing well-known smells adapted to JavaScript, and 6 are specific JavaScript code smells, collected from various JavaScript development resources. The smells related to scoping issues include: *this* keyword usage, variable naming conflicts, scope chaining, nested callbacks, and excessive global variables.

Examining the literature related to comprehension difficulties associated with source code understanding was a starting point for obtaining information requirements for the design and implementation of Scoped. To gather further information requirements, specifically ones related to scoping issues, an analysis of the 50 most popular stack overflow questions was conducted. The question retrieved were tagged with the keywords "*scope*" and "*JavaScript*" and were categorised into 4 broad categories: *Identifiers* (36%), *this context* (36%), *design issues* (17%), and *other* (17%). The identifiers category deals with issues regarding the declaration, accessibility, and state of an identifier (variable, function, and parameter). The *this* context category is concerned with issues relating to the *this* keyword associated with each individual scope, as the use of the keyword within source code has specific behaviour associated with it. The design issues category includes issues corresponding to design decisions programmers make during the process of writing

and modifying code. Finally, the other category includes the usage of specific methods and frameworks.

In this work we will concentrate on the first category (identifiers) and information related to this category. This choice is supported by the literature presented earlier in this section, as it shows that many maintenance and comprehension issues are due to the poor understanding of scope. Therefore, Scoped should aid programmers in answering questions such as: *in which scope is an identifier declared*, *can an identifier be accessed from the current scope*, and *is an identifier that has already been defined within the parent or global scope being overwritten?*

4. Design

Following [CFM05, Die07, HPS14, PQ06] we describe the system under the following principles: target users, language specific, natural mapping, and link to the code.

Target users: Over fifty thousand programmers participated in the stack overflow 2016 survey [Sta16], which asked questions related to preferred technology, experience, and occupation. Out of all of the participants, only 7% identify as "rockstars" (expert programmers). Furthermore, 59% are under the age of 29 and 50.3% have less than 5 years of programming experience. Using this information and the assumption that expert programmers do not have any difficulties in regards to scope, the presented tool is targeted towards non expert programmers. We define non expert programmers as programmers that do not use JavaScript on a day to day basis and have less than 5 years of experience using the language.

Language specific: Scope rules are implemented differently in different programming languages, and so some programming languages are more prone to cause scope comprehension issues than others [KMCA06]. For this work the programming language of choice is JavaScript, as the syntax of the language makes false promises that can lead to errors [Cro13, Sim14]. This choice is supported by the comprehension issues presented in Section 3, as it shows that many maintenance and comprehension issues are due to the poor understanding of scope within JavaScript source code.

Natural mapping: Scope can be described as a series of neatly nested "bubbles" that each act as a container, in which identifiers are declared [Sim14]. Building on this metaphor, we believe that a tree visualisation technique which shows parent child relations using containment is best suited for the task of visualising the scope chain hierarchy within a source code document. These tree visualisation techniques include treemaps [JS91], circular treemaps [WWDW06], icicle trees [HBO10a], and sunburst trees [SCGM00]. Each of these techniques have their pros and cons, however, we believe that the circular treemap is well suited for encoding the scope chain hierarchy of a source code document, as the hierarchical structure is explicitly shown and the technique strictly follows the "bubbles" metaphor used to explain the concept of scope. For the next version of Scoped, an interesting user study would be to investigate the effectiveness of each of these tree visualisation techniques, as to date, we do not have data on which technique is optimal for the visualisation of a scope chain within a source code document.

Link to code: Developers become 'code-centric' when working with code, therefore visual representations of source code need to clearly link the encoded information to the code, especially when tasks involve source code manipulation [CFM05]. Programmers want and need information to be provided in the context of their working environment, which is a source code editor. Cordy [Cor03] states that to achieve success, tools must present information in the context of source code and provide strong coupling between code and the visualisations.

Figure 2 shows a screenshot of Scoped. The tool consists of three main components: A) An overview visualisation encoding the scope chain within source code. B) The identifier (variable, function, and parameter) information panel. C) a source code editor. Colour is used to link each of the components. Within the overview visualisation, green is used to show in which scope the cursor is currently located in within the source code editor. Blue is used to represent the parent and ancestor scopes which the currently selected scope can access, and grey is used to represented the remaining scopes that cannot be accessed. Additionally, the type of scope (function or block) is encoded using solid or dashed lines. The identifier information panel presents variables, functions, and parameters that can be accessed in the currently selected scope. Colour is used to show local identifiers (green) and identifiers belonging to parent or ancestor scopes (blue). The globe icon represents identifiers that belong to the global scope. Within the source code editor the currently selected scope is highlighted in green (as shown in the Figure 2), parent and ancestor scopes are highlighted in blue, and scopes which cannot be accessed are highlighted in grey, when hovering over an identifier in component B) or a node within component A).

5. User experience evaluation

A pilot user experience evaluation study [LBI*12] was conducted to probe for further design requirements and to understand user behaviour when interacting with Scoped. Five participants took part in the evaluation study which lasted for approximately 45 minutes for each participant. The majority of the participants were postgraduate students and none used JavaScript on a day to day basis. In regards to using students, Kitchenham et al. [KPP*02] state that "*using students as subjects is not a major issue as long as you are interested in evaluating the use of a technique by non-expert users*".

The evaluation followed a think aloud protocol [Nie94], where participants were encouraged to verbalise their thoughts as they moved through the user interface to complete the given tasks. Each participant was first given a short introduction to the visualisation tool, where the main features and functionality of the tool were explained. Next, each participant worked through a 15 minute tutorial together with the observer. After the tutorial, the participants were given 2 tasks to complete. The first task involved answering a series of questions related to scope issues in the context of the identifier category presented in Section 3. The questions included asking which identifiers (variables and functions) belonged to a specific scope, moving the cursor to a specific line in the code and answering in which scope the cursor is currently located in, as well as moving the cursor to a specific line in the code and answering whether a certain identifier was accessible from that line. The second tasks involved finding a bug within a source code fragment,

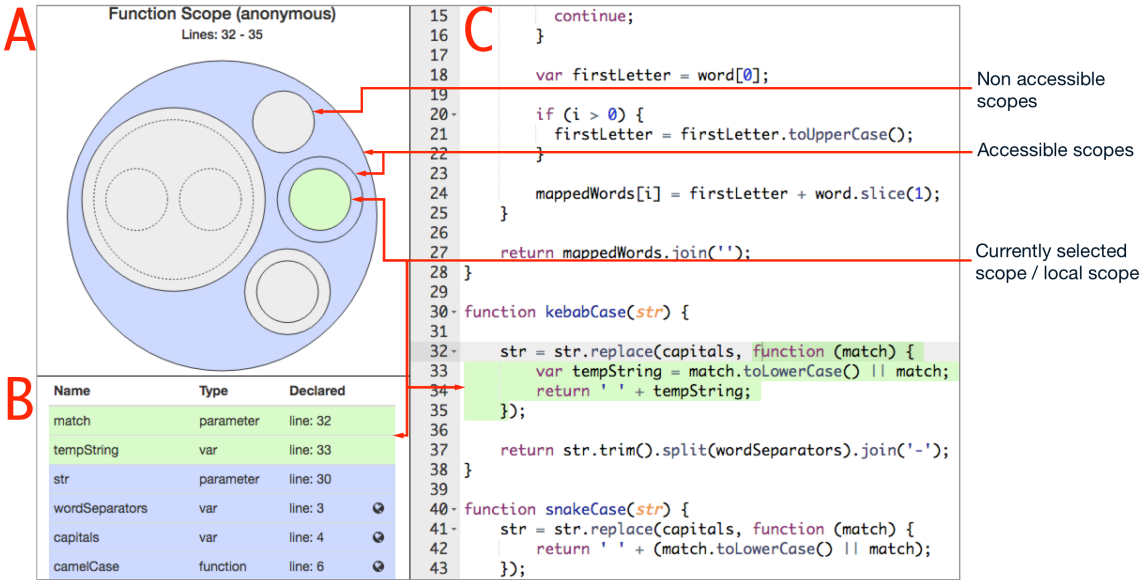

Figure 2: *Scoped: A) Scope chain visualisation, B) Identifier information view, C) Code editor. Colour highlighting is used to link the three components, where green represents the currently selected/local scope, blue represents scopes that can be accessed from the currently selected scope, and grey represents scopes that cannot be accessed from the currenlty selcted scope.*

where the bug was composed of a local variable using the same name as a variable defined within the parent scope. After the participants completed both of the tasks, they were asked to fill out a short survey, which included questions relating to their experience with the tool.

All of the participants frequently interacted with the visualisations within Scoped when attempting to answer the questions during the first task. When answering the question of which identifiers belong to the global scope, none of the participants investigated the source code. Instead they moved the cursor to the global scope within the source code editor or clicked on the circle representing the global scope within the scope chain visualisation and then investigated the identifier information panel in order to answer the question. A participant commented "*This is easier than looking through the code, all I have to do is click on a circle or move the cursor to a function and then I can find the information I am looking for within the list view*". When asked to which scope a specific variable belonged, the majority of the participants used the visualisation first, and then looked at the code. An interesting observation was that the participants seemed to use the visualisations to verify their assumptions before answering the questions.

During the second task the participants mainly focused on the code in order to understand its functionality. After the participants understood how most of the code worked, they attempted to look at the code in order to find any lines that might seem suspicious. When a suspicious line was found, the participants glanced at the visualisations within the tool in order to investigate which identifier belonged to which scope and if an identifier could be accessed from the currently selected scope.

The participants brought several issues to our attention. For example, one participant stated that the identifier list view should support changing lines within the source code editor (e.g. navigating to a specific line of code where a specific variable is declared or accessed). While another participant stated that the link between the source code editor and the identifier information panel could be improved by adding a mechanism to highlight which variable is currently of interest to the user. Most participants also stated that it would have been helpful if labels were present within the scope chain visualisation. We plan to use the feedback provided by the participants in the next design iteration of the tool.

6. Conclusion

This work presented Scoped, an interactive visualisation tool to support a programmer's understanding of the scope chain and information related to the scope chain within a source code document. Additionally, the design rationale behind the main features of the tool and the issues the tool aims to address were presented. In the future we plan to integrate the feedback from the user experience evaluation study in order to improve the design of Scoped. We also plan on conducting further empirical evaluations to gather qualitative and quantitative data on the usability and usefulness of combining source code editors with compact visual representations encoding the complex properties of source code. Due to the positive feedback received from the user experience study, we believe that programmers are open to adopting visual tools during the process of working with source code, given that the visualisations within the tools clearly link the source code and that the source code clearly links to the visual representations. A demo of the tool can be found at: *http://tiny.cc/jsscope*.

References

[BMK16a] BACHER I., MAC NAMEE B., KELLEHER J. D.: On using Tree Visualisation Techniques to support Source Code comprehension. In *2016 IEEE 4rd Working Conference on Software Visualization, VISSOFT 2016 - Proceeding* (2016), no. Figure 2. doi:10.1109/VISSOFT.2016.8. 2

[BMK16b] BACHER I., MAC NAMEE B., KELLEHER J. D.: Using icicle trees to encode the hierarchical structure of source code. In *Proceedings of EG/VGTC Conference on Visualization (EuroVis 2016)* (2016), no. 2. doi:10.2312/eurovisshort.20161168. 2

[BMK17] BACHER I., MAC NAMEE B., KELLEHER J. D.: The codemap metaphor - a review of its use within software visualisations. In *Proceedings of the 12th Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications* (2017). 2

[CFM05] COX A., FISHER M., MUZZERALL J.: User perspectives on a visual aid to program comprehension. In *Proceedings - VISSOFT 2005: 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis* (2005), pp. 70–75. doi:10.1109/VISSOF.2005.1684308. 3

[CHM98] CROSS J. H., HENDRIX T. D., MAGHSOODLOO S.: Control Structure Diagram: An overview and initial evaluation. *Empirical Software Engineering 3*, 2 (1998), 131–158. doi:10.1023/A:1008085415145. 2

[Cor89] CORBI T. A.: Program understanding: Challenge for the 1990s. *IBM Systems Journal 28*, 2 (1989), 294–306. doi:10.1147/sj.282.0294. 1

[Cor03] CORDY J. R.: Comprehending reality - Practical barriers to industrial adoption of software maintenance automation. *Proceedings - IEEE Workshop on Program Comprehension 2003-May* (2003), 196–205. doi:10.1109/WPC.2003.1199203. 3

[Cro13] CROCKFORD D.: *JavaScript: The Good Parts*, vol. 53. 2013. arXiv:arXiv:1011.1669v3, doi:10.1017/CBO9781107415324.004. 1, 3

[Die07] DIEHL S.: *Software visualization: Visualizing the structure, behaviour, and evolution of software*. 2007. doi:10.1007/978-3-540-46505-8. 2, 3

[DLR05] DUCASSE S., LANZA M., ROBBES R.: Multi-level method understanding using microprints. In *Proceedings - VISSOFT 2005: 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis* (2005), pp. 33–38. doi:10.1109/VISSOF.2005.1684300. 2

[FM13] FARD A. M., MESBAH A.: JSNOSE: Detecting javascript code smells. *IEEE 13th International Working Conference on Source Code Analysis and Manipulation, SCAM 2013* (2013), 116–125. doi:10.1109/SCAM.2013.6648192. 2

[HBO10a] HEER J., BOSTOCK M., OGIEVETSKY V.: A Tour through the Visualization Zoo A survey of powerful visualization techniques , from the obvious to the obscure. *Communications of the ACM 53*, 5 (2010), 59–67. URL: http://cat.inist.fr/?aModele=afficheN{&}cpsidt=22906879, doi:10.1145/1743546. 3

[HBO10b] HEER J., BOSTOCK M., OGIEVETSKY V.: A tour through the visualization zoo. *Commun. Acm 53*, 6 (2010), 59–67. 2

[HCM02] HENDRIX D., CROSS J. H., MAGHSOODLOO S.: The effectiveness of control structure diagrams in source code comprehension activities. *IEEE Transactions on Software Engineering 28*, 5 (2002), 463–477. doi:10.1109/TSE.2002.1000450. 1, 2

[HPS14] HUANG W., PARSONS P., SEDIG K.: *Handbook of Human Centric Visualization*. 2014. URL: http://link.springer.com/10.1007/978-1-4614-7485-2, doi:10.1007/978-1-4614-7485-2. 3

[JS91] JOHNSON B., SHNEIDERMAN B.: Tree-maps: a space-filling approach to the visualization of hierarchical information structures. *Proceeding Visualization '91* (1991), 284–291. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=175815, doi:10.1109/VISUAL.1991.175815. 3

[KMCA06] KO A., MYERS B., COBLENZ M., AUNG H.: An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering 32*, 12 (2006), 971–987. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4016573, doi:10.1109/TSE.2006.116. 1, 3

[KPP*02] KITCHENHAM B. A., PFLEEGER S. L., PICKARD L. M., JONES P. W., HOAGLIN D. C., EL EMAM K., ROSENBERG J.: Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering 28*, 8 (2002), 721–734. URL: http://ieeexplore.ieee.org/xpl/freeabs{_}all.jsp?arnumber=1027796{%}5Cnhttp://dl.acm.org/citation.cfm?id=636196.636197, doi:10.1109/TSE.2002.1027796. 3

[LBI*12] LAM H., BERTINI E., ISENBERG P., PLAISANT C., LAM H., BERTINI E., ISENBERG P., PLAISANT C., EMPIRICAL S. C., LAM H.: Empirical Studies in Information Visualization : Seven Scenarios. *IEEE Transactions on Visualization and Computer Graphics (2012)* (2012). 3

[LM10] LATOZA T. D., MYERS B. A.: Hard-to-answer questions about code. *Evaluation and Usability of Programming Languages and Tools on - PLATEAU '10* (2010), 1–6. URL: http://portal.acm.org/citation.cfm?doid=1937117.1937125, doi:10.1145/1937117.1937125. 2

[MMNS83] MIARA R. J., MUSSELMAN J. A., NAVARRO J. A., SHNEIDERMAN B.: Program Indentation and Comprehensibility. *Communications of the ACM 26*, 11 (1983), 861–867. URL: http://portal.acm.org/citation.cfm?doid=182.358437, doi:10.1145/182.358437. 1

[Nie94] NIELSEN J.: *Usability engineering*. Elsevier, 1994. 3

[PQ06] PETRE M., QUINCEY E. D.: A gentle overview of software visualisation. *PPIG News Letter*, September (2006), 1 – 10. URL: http://www.labri.fr/perso/fleury/courses/PdP/SoftwareVisualization/1-overview-swviz.pdf, doi:10.1.1.127.6350. 3

[SCGM00] STASKO J., CATRAMBONE R., GUZDIAL M., MCDONALD K.: An evaluation of space-filling information visualizations for depicting hierarchical structures. *International Journal of Human-Computer Studies 53*, 5 (2000), 663–694. doi:10.1006/ijhc.2000.0420. 3

[Sim14] SIMPSON K.: *You Don't Know JS: Scope & Closures*. 2014. 1, 3

[Sta16] [online]2016. URL: http://stackoverflow.com/research/developer-survey-2016 [cited 02.02.17]. 3

[Tel15] TELEA A. C.: *Data Visualization: Principles and Practice*. 2015. doi:10.1002/9780470417409.ch2. 1

[WLR10] WETTEL R., LANZA M., ROBBES R.: Empirical validation of CodeCity: A controlled experiment. *USI Technical Report Series in Informatics* (2010). URL: http://doc.rero.ch/lm.php?url=1000,42,6,20110309110626-OX/ITR1005.pdf. 1

[WWDW06] WANG W., WANG H., DAI G., WANG H.: Visualization of large hierarchical data by circle packing. *Proceedings of the SIGCHI conference on Human Factors in computing systems* (2006), 517–520. URL: http://doi.acm.org/10.1145/1124772.1124851, doi:http://doi.acm.org/10.1145/1124772.1124851. 3