The Effect of Richer Visualizations on Code Comprehension

Dimitar Asenov
Dept. of Computer Science
ETH Zurich
dimitar.asenov@inf.ethz.ch

Otmar Hilliges
Dept. of Computer Science
ETH Zurich
otmar.hilliges@inf.ethz.ch

Peter Müller
Dept. of Computer Science
ETH Zurich
peter.mueller@inf.ethz.ch

ABSTRACT
Researchers often introduce visual tools to programming environments in order to facilitate program comprehension, reduce navigation times, and help developers answer difficult questions. Syntax highlighting is the main visual lens through which developers perceive their code, and yet its effects and the effects of richer code presentations on code comprehension have not been evaluated systematically. We present a rigorous user study comparing mainstream syntax highlighting to two visually-enhanced presentations of code. Our results show that: **(1)** richer code visualizations reduce the time necessary to answer questions about code features, and **(2)** contrary to the subjective perception of developers, richer code visualizations do not lead to visual overload. Based on our results we outline practical recommendations for tool designers.

Author Keywords
syntax highlighting; programming; code editor; visual programming; code comprehension; user study

ACM Classification Keywords
H.1.2. User/Machine Systems: Human factors; H.5.2. User Interfaces: Evaluation/methodology; D.2.3. Coding Tools and Techniques: Program editors; D.2.3. Coding Tools and Techniques: Structured programming

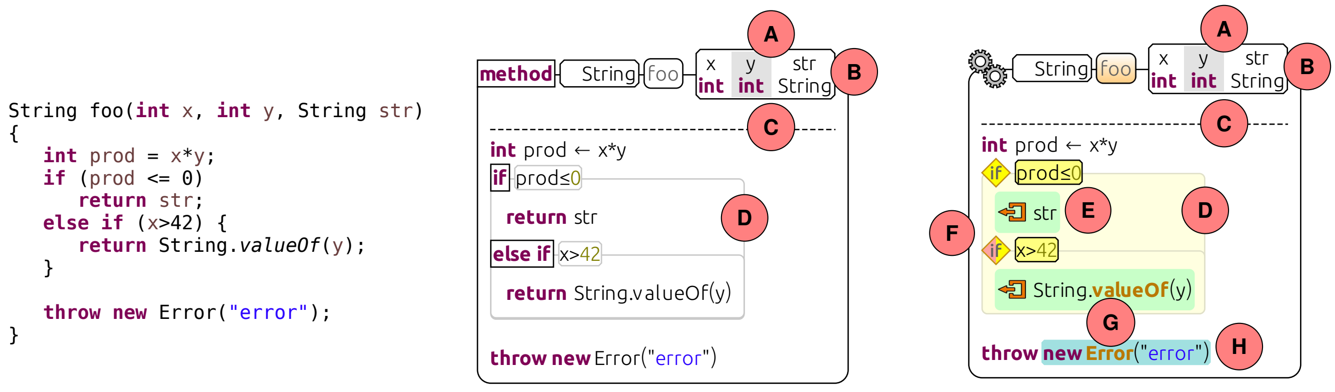
INTRODUCTION
In order to help developers be more efficient, recent research in programming environments [3, 6, 14, 16, 18, 19] has explored novel visualizations of code blocks and their relationships, spatial navigation between code fragments, and augmenting code with visual hints and abstractions. However, all of these enhancements leave the presentation of the actual code unchanged. Since source code is the core medium in which programmers work, improving its presentation is at least as important as enhancing the rest of the programming environment. Yet, syntax highlighting has rarely been empirically evaluated, and to our knowledge no study compared different visual code presentations of the same source code. To investigate improvements to code presentation as a way to complement other research on development environments we pose the following research questions:
RQ1: Do richer code visualizations affect the speed with which code features can be detected?
RQ2: Do richer code visualizations affect the ability to correctly answer questions about code structure?
RQ3: Do visually enhanced code constructs impair the readability of unenhanced ones?

Understanding code structure is part of more complex programming activities that developers perform regularly. Therefore, we asked 15 questions about code structure in a controlled study with 33 developers and compared their performance when using traditional syntax highlighting to two richer code visualizations, which go beyond changing font properties. The results strongly indicate that richer visualizations reduce response times on a wide range of questions about code structure and do not lead to visual overload, contrary to developers' feedback.

RELATED WORK
Green and Petre [10, 11] show that the usability of notations varies with the programmer's task, and neither textual nor visual notations are generally superior. However, they analyze notations for distinct programming models, and do not compare different notations for a single programming language. Hendrix et al. [13] show that a control structure diagram of the code, embedded in the indentation area to the left of the text, can improve comprehension. However, the code itself is presented as plain text without even syntax highlighting. Feigenspan et al. [9] investigate a specific use of color showing that different backgrounds for the components of a software product line help to identify which component some code belongs to. In the only study of syntax highlighting that we are aware of, Hakala et al. [12] investigate users' performance using the default coloring scheme of the Vim code editor, code without highlighting, and one other coloring scheme. Surprisingly, they found no overall difference between the three schemes. The surprising results and scarcity of such studies merit more empirical investigation.

Recent work on programming environments [3, 6, 7, 14, 19] shows that visual enhancements can be beneficial. However, all of these tools still use standard syntax highlighting for presenting code. Barista [15] and Envision [1, 2] are research prototypes of structured code editors that allow flexible visualization of code fragments. The effect of their code presentation has not been investigated before. For our study, we use Envision, available as an open-source project.

CHI'16, May 07 – 12, 2016, San Jose, CA, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3362-7/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2858036.2858372



```
String foo(int x, int y, String str)
{
   int prod = x*y;
   if (prod <= 0)
      return str;
   else if (x>42) {
      return String.valueOf(y);
   }

   throw new Error("error");
}
```

(a) v-low (Eclipse - default settings) **(b) v-med** (Envision - alternate settings) **(c) v-high** (Envision - default settings)

Figure 1: A Java method rendered using three different levels of visual enhancements. **v-low** shows Eclipse using default settings. **v-med** shows a variant of the Envision structured editor with the following enhancements over **v-low**: **(A)** all lists use alternating white and gray background instead of commas; **(B)** the names of formal parameters appear above the types; **(C)** a dashed line separates the method body from its signature; **(D)** blocks are visually outlined instead of showing curly braces. **v-high** also shows a variant of Envision, with the following additional enhancements over **v-med**: **(E)** some constructs, like compound statements have a background color; **(F)** many (but not all) keywords are replaced with icons; **(G)** method and constructor calls have orange text; **(H)** some expressions have a specific background color.

Conversy [4] proposed a framework based on the Semiology of Graphics (SoG) to model the visual perception of code. SoG recognizes seven visual variables: shape, luminosity, color, position, size, orientation, and linking marks. In our work, increasing levels of visual variety correspond to more extensive use of these visual variables, which Conversy suggests can improve the performance of readers.

VISUALIZATIONS AND EVALUATION METHOD

To determine if enhanced code presentations can help developers to more quickly comprehend code, we conducted a controlled experiment with three different levels of visual variety for code presentation: **v-low** corresponds to the default Java syntax highlighting in Eclipse, **v-med** adds additional visual enhancements, and **v-high** further increases visual variety. Both **v-med** and **v-high** are produced with the Envision code editor. Figure 1 illustrates the most important differences between the three levels. For each level, we took screenshots of 298 methods from the open-source Java text editor jEdit. The methods were selected so that they are complex (have at least 2 parameters and 3 block statements) but still fit within one 1920x1080 screen. All comments were removed and methods with features not supported by Envision were filtered out.

For our within-subjects study, we recruited 33 non-color blind participants with at least 1 year of Java experience (*average*=5.5, *SD*=4.2, *max*=20). Each participant was presented with the 15 yes/no questions shown in Table 1 in random order. Within each question, we showed participants 15 method screenshots for each of the three visual variety levels. The order of the levels was randomized, but all screenshots from one level were shown in succession. Thus participants saw 45 screenshots per question, all of which were of randomly chosen methods from our pool of 298. In total we recorded $15\times3\times15$ = 675 answers per participant. For each combination of question and visual variety level, we drop the first three answers and average the rest for each participant in order to account for the learning curve. We iteratively designed the 15 questions so that they match the following criteria: (i) **sim-**

Id	Question
Q1	Does the method throw exceptions directly in its body using a throw statement?
Q2	Are all local variables immediately initialized (assigned) as part of their declaration?
Q3	Is subtraction (-) used in an expression? Any use counts, for example: a-b, -1, –i, x -= 3.
Q4	Is the 'this' identifier used in the method?
Q5	Is there a local variable (not a method parameter) of type String?
Q6	Is there an if statement with an else branch? Both else, and if else count.
Q7	Is the type of the second method parameter 'int'?
Q8	Is there a loop nested inside another loop?
Q9	Does the method have exactly 3 parameters?
Q10	Is there a top-level loop (not nested inside any other statement) that appears after some code containing an if-statement? The if-statement may be nested.
Q11	Does the method explicitly create new objects of any type, including arrays or exceptions.
Q12	Does the method catch any exceptions?
Q13	Is there a loop that contains two or more method/constructor calls? The calls can be anywhere inside the loop, including in nested statements, or arguments.
Q14	Are there multiple points from which the method can return (the end of the body is usually one such point)? Throwing exceptions does not count.
Q15	Does the method use an explicit type cast?

Table 1: The questions about code structure that we asked in our study. These are divided in three categories based on which visual variety levels enhance the fragments of code relevant for answering: **Q1-Q5**: all visual variety levels use the same textual (unenhanced) presentation; **Q6-Q10**: both **v-med** and **v-high** provide substantial enhancements; **Q11-Q15**: only **v-high** provides a substantial enhancement.

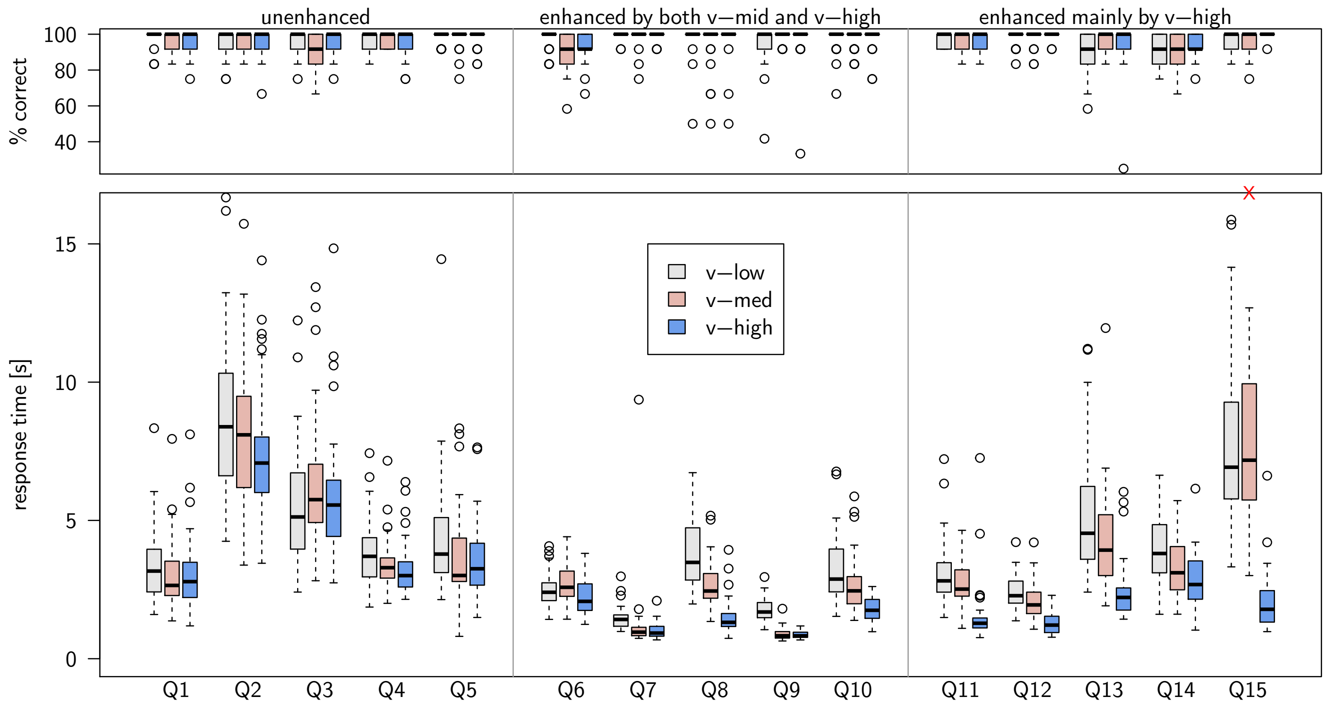


Figure 2: Box plots (n = 33) of response correctness and time per question and tool. The × indicates two out-of-graph outliers at 19.4s and 17.3s. We observe that increasing the level of visual variety does not affect correctness and lowers response times. The whiskers represent the lowest/highest data points still within 1.5 × interquartile range (IQR) of the lower/upper quartile.

ple: we ask only questions about method structure that can be answered within several seconds without complex reasoning. This enables us to attribute any differences to the speed of comprehension even when developers have very different experience; (ii) **matching visual enhancements**: to test our hypotheses we need questions pertaining to constructs that are either enhanced differently or unenhanced by **v-mid** and **v-high**; (iii) **practical** and **representative**: Each question is a component of a practical programming activity and similar low-level questions are typical when reading code. They appear for example as parts of more complex questions that developers frequently ask during maintenance tasks as observed by Sillito et al. [20]. For instance, imagine a developer who is inspecting a method in order to improve its performance. The developer is looking for time-consuming operations, such as loops, especially nested loops (**Q8**), and calls to other methods, especially within loops (**Q13**). Generally, questions about code structure occur frequently as subtasks of many programming activities, such as looking up APIs (**Q7**, **Q9**), searching for errors or exceptions (**Q1**, **Q2**, **Q3**, **Q15**), tracing local definitions (**Q2**, **Q5**), understanding method structure and control flow (**Q6**, **Q8**, **Q10**, **Q12**, **Q13**, **Q14**), optimizing code (**Q8**, **Q13**), and tracking object lifetimes and state (**Q4**, **Q11**).

Before the study, participants received a brief introduction to the three code presentations and were given a visual legend that they could use during the entire study. The study itself is implemented as an OpenSesame [17] script. To enable replication or additional analysis, all data and scripts are available at `www.pm.inf.ethz.ch/research/envision.html`.

We make the following hypotheses:
H1: Response times are lower in questions pertaining to visually enhanced constructs (Q6-Q15).
H2: Response times in questions pertaining to unenhanced constructs (Q1-Q5) are unaffected by richer visualizations of other constructs.
H3: Correctness is unaffected by richer visualizations.

RESULTS

Figure 2 shows a plot of the raw data we collected, whereas Figure 3 presents an estimation analysis of response time. This analysis avoids null-hypothesis significance testing, following Cumming [5] and Dragicevic [8]. As the data is not normally distributed we use Wilcox' robust bootstrapped estimation with trimmed means (B=2000, γ=.2) [21]. Response times across all visualizations are similar for questions **Q1-Q5**, as expected, but also for **Q6**, even though it pertains to visually enhanced code. This supports **H2**. For **Q7-Q10**, we observe that both **v-med** and **v-high** outperform **v-low**. The reduction in mean response time is substantial and varies between 21% and 63%. For **Q11-Q15**, we observe that **v-high** outperforms **v-low** and, except for **Q14**, also **v-med**. Again, the reduction in mean response time is substantial: between 29% and 75%. Except for **Q6** the data supports **H1**.

Due to a clear ceiling effect, the correctness data is inconclusive. **H3** seems to hold for the simple questions that we asked, but this cannot be generalized for more complex ones.

Participant Feedback

Overall, participants preferred **v-med**, which received the best average rank (2.4) on a 1 to 3 scale, followed by **v-high**

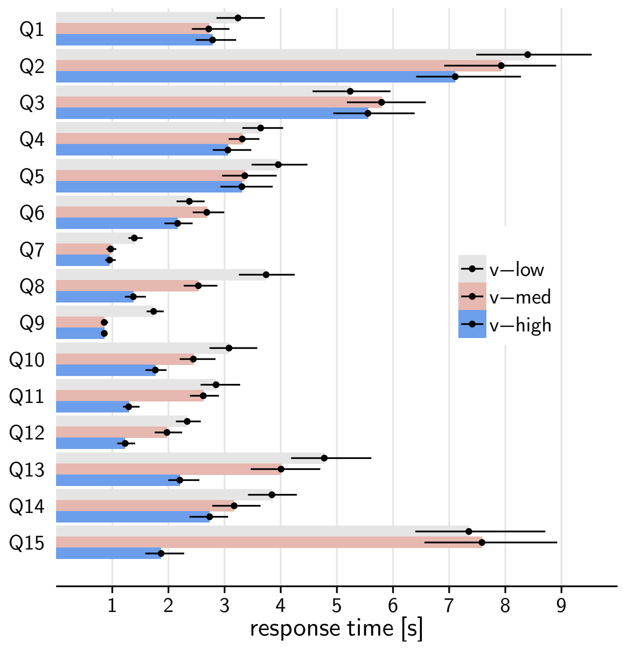


Figure 3: Estimated mean response times and 95% confidence intervals. A bootstrapped, trimmed means approach was used (B=2000, γ=.2).

(2.0) and **v-low** (1.7). The participants commented that **v-med** was helpful while still feeling more familiar than **v-high**. 30 out of the 33 participants provided textual feedback. 20 users found some aspects of **v-med** and **v-high** helpful. A clear theme from 23 responses is that **v-high** can sometimes feel overwhelming. 9 participants indicated that they would prefer a version that is a mix between **v-med** and **v-high**. 3 participants said they imagine using enhanced visualization levels but only for viewing code (e.g., code review) and they would rather write code using a traditional notation.

DISCUSSION

RQ1: In every question we investigated, increasing visual variety over **v-low** either had no effect, or substantially reduced the time to detect structural features of methods. The same effect was also observed when switching from **v-med** to **v-high**. The results show a medium to large reduction in response time (21%-75% or 0.5 - 5 seconds) in **Q7-Q15**. We believe that developers will ask similar low-level questions often, resulting in a tangible benefit. Richer visualizations can help to more quickly detect method features and we speculate that this might help developers maintain a state of flow and improve productivity.

RQ2: For the simple questions that we asked, participants almost always provided a correct answer, which resulted in a strong ceiling effect in the correctness data. This suggests that for such simple questions, traditional and richer visualizations are equally able to guide developers to the correct answer. It remains to be further investigated, what effect richer visualizations might have for more complex questions.

RQ3: The most interesting finding is that richer visualizations did not cause visual overload, even in **Q1**-**Q5**, where answers pertain to unenhanced constructs. This finding goes against the participants' overall preference for **v-med** and against the feedback of 23 participants, who reported some sort of subjective visual overload or confusion. This suggests that users are reluctant to adopt richer visualizations, even if they can be helpful. It is worth investigating whether a visualization that is more aesthetically appealing than **v-high**, but with a similar visual variety, will be more popular.

Limitations

Our study has several limitations. First, we tested the participants' responses on a limited number of questions in a controlled setting. To increase the applicability of our findings we picked questions that occur as components of more high-level regular programming tasks. Second, we draw all our sample methods from a particular Java code base, and results might not generalize to other code or other languages. To increase ecological validity we used an established, large, and actively maintained open-source project. Third, we have used only two particular code presentations that go beyond syntax highlighting. We argue that our findings are generalizable, because our questions predominantly test individual building blocks (e.g., outlines or colors), which can be combined to form other, more complex visualizations. Fourth, we measure only code comprehension. Nevertheless, reading code is an inherent part in most programming activities including writing, debugging, and testing, which suggests that improved visualizations could have an overall productivity benefit.

Recommendations for tool designers

Our results show that **v-high** outperformed **v-med**. This is in line with the SoG theory because SoG suggests that increased color variety can improve perception, and extensive use of color is the major difference between **v-med** and **v-high**. Based on these findings we recommend that tool designers boost the syntax highlighting capabilities of their tools in two ways: (i) use a wider variety of colors by default and (ii) enable the highlighting of more constructs. Our recommendation is practical since syntax highlighting is universal and improving it requires only marginal effort while being risk-free: one could simply revert to a classical coloring theme.

CONCLUSION AND FUTURE WORK

We have presented a user study that provides insight into enhancing syntax highlighting with richer visualizations. The results show that using more visual variety when rendering methods substantially reduces comprehension time of code features. A further interesting result is that even with the richest visualization that we evaluated, developers did not experience visual overload, despite expressing that they found the visualization overwhelming at times. Going forward, it is worth evaluating the effects of richer visualizations on programming activities other than comprehension, for example writing and debugging code. Additionally, in order to address concerns that many participants expressed with the aesthetics of the enhanced visualizations we used and to gain further understanding of the role of aesthetics, it is worth experimenting with systems that have a similar level of visual variety but different levels of visual appeal.

REFERENCES

1. D. Asenov and P. Müller. 2013. Customizing the visualization and interaction for embedded domain-specific languages in a structured editor. In *Visual Languages and Human-Centric Computing (VL/HCC), 2013 IEEE Symposium on*. 127–130. DOI: http://dx.doi.org/10.1109/VLHCC.2013.6645255

2. D. Asenov and P. Müller. 2014. Envision: A fast and flexible visual code editor with fluid interactions (Overview). In *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*. 9–12. DOI: http://dx.doi.org/10.1109/VLHCC.2014.6883014

3. A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and Joseph J. LaViola, Jr. 2010. Code Bubbles: a working set-based interface for code understanding and maintenance. In *Proceedings of the 28th international conference on Human factors in computing systems (CHI '10)*. ACM, New York, NY, USA, 2503–2512. DOI: http://dx.doi.org/10.1145/1753326.1753706

4. S. Conversy. 2014. Unifying Textual and Visual: A Theoretical Account of the Visual Perception of Programming Languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014)*. ACM, New York, NY, USA, 201–212. DOI: http://dx.doi.org/10.1145/2661136.2661138

5. G. Cumming. 2014. The New Statistics: Why and How. *Psychological Science* 25, 1 (2014), 7–29. DOI: http://dx.doi.org/10.1177/0956797613504966

6. R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. P. Reiss. 2012. Debugger Canvas: industrial experience with the Code Bubbles paradigm. In *Proceedings of the 2012 International Conference on Software Engineering (ICSE 2012)*. IEEE Press, Piscataway, NJ, USA, 1064–1073. http://dl.acm.org/citation.cfm?id=2337223.2337362

7. R. DeLine and K. Rowan. 2010. Code canvas: zooming towards better development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2 (ICSE '10)*. ACM, New York, NY, USA, 207–210. DOI: http://dx.doi.org/10.1145/1810295.1810331

8. P. Dragicevic. 2016. Fair Statistical Communication in HCI. In *Modern Statistical Methods for HCI*, J. Robertson and M.C. Kaptein (Eds.). Springer. In press.

9. J. Feigenspan, C. Kstner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake. 2013. Do background colors improve program comprehension in the #ifdef hell? *Empirical Software Engineering* 18, 4 (2013), 699–745. DOI: http://dx.doi.org/10.1007/s10664-012-9208-x

10. T.R.G. Green and M. Petre. 1992. When visual programs are harder to read than textual programs. In *Human-Computer Interaction: Tasks and Organisation, Proceedings ECCE-6 (6th European Conference Cognitive Ergonomics)*. 57.

11. T.R.G. Green and M. Petre. 1996. Usability Analysis of Visual Programming Environments: A "Cognitive Dimensions" Framework. *JVLC* 7, 2 (1996), 131 – 174. DOI: http://dx.doi.org/10.1006/jvlc.1996.0009

12. T. Hakala, P. Nykyri, and J. Sajaniemi. 2006. An Experiment on the Effects of Program Code Highlighting on Visual Search for Local Patterns. *Psychology of Programming Interest Group* (2006), 38–52.

13. D. Hendrix, II Cross, J.H., and S. Maghsoodloo. 2002. The effectiveness of control structure diagrams in source code comprehension activities. *Software Engineering, IEEE Transactions on* 28, 5 (May 2002), 463–477. DOI: http://dx.doi.org/10.1109/TSE.2002.1000450

14. A. Z. Henley and S. D. Fleming. 2014. The Patchworks Code Editor: Toward Faster Navigation with Less Code Arranging and Fewer Navigation Mistakes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 2511–2520. DOI: http://dx.doi.org/10.1145/2556288.2557073

15. A. J. Ko and B. A. Myers. 2006. Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In *Proceedings of the SIGCHI conference on Human Factors in computing systems (CHI '06)*. ACM, New York, NY, USA, 387–396. DOI: http://dx.doi.org/10.1145/1124772.1124831

16. T. Lieber, J. R. Brandt, and R. C. Miller. 2014. Addressing Misconceptions About Code with Always-on Programming Visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 2481–2490. DOI: http://dx.doi.org/10.1145/2556288.2557409

17. S. Matht, D. Schreij, and J. Theeuwes. 2012. OpenSesame: An open-source, graphical experiment builder for the social sciences. *Behavior Research Methods* 44, 2 (2012), 314–324. DOI: http://dx.doi.org/10.3758/s13428-011-0168-7

18. F. Olivero, M. Lanza, M. D'Ambros, and R. Robbes. 2011. Enabling program comprehension through a visual object-focused development environment. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*. 127 –134. DOI: http://dx.doi.org/10.1109/VLHCC.2011.6070389

19. J. Ou, M. Vechev, and O. Hilliges. 2015. An Interactive System for Data Structure Development. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 3053–3062. DOI: http://dx.doi.org/10.1145/2702123.2702319

20. J. Sillito, G. C. Murphy, and K. De Volder. 2006. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering (SIGSOFT '06/FSE-14)*. ACM, New York, NY, USA, 23–34. DOI: http://dx.doi.org/10.1145/1181775.1181779

21. R. Wilcox. 2012. Chapter 4 - Confidence Intervals in the One-Sample Case. In *Introduction to Robust Estimation and Hypothesis Testing (Third Edition)* (third edition ed.), R. Wilcox (Ed.). Academic Press, Boston, 103 – 136. DOI: http://dx.doi.org/10.1016/B978-0-12-386983-8.00004-4