Towards Better Code Snippets: Exploring How Code Snippet Recall Differs with Programming Experience

Michelle Ichinco and Caitlin Kelleher
Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, Missouri, United States
michelle.ichinco, ckelleher@wustl.edu

***Abstract*—Programmers of all experience levels attempt to leverage code snippets with varying success, often as reminders or to learn new skills. To date, little work has explored the specific elements within code snippets that are challenging for novices. Comparing how novices and experts recall code snippets may expose what code elements programmers focus on and inform new approaches for improving examples for inexperienced programmers. We conducted a study, inspired by past novice-expert studies, in which we asked everyday, occasional, and non-programmers to study and then recall code snippets. The key distinctions and similarities in the types and locations of recalled tokens provide insight for a set of recommendations that could improve the presentation of code snippets.**

I. Introduction

Many programmers attempt to use code examples on the web, within tutorials and documentation, and through auto-complete [1]. Code snippets can be essential learning resources, especially for the millions of end-user programmers who learn programming informally [2]. These less experienced programmers often struggle using code examples [3], [4]. One study found that a main hurdle for novices attempting to use code examples was isolating the critical elements of examples related to their problems [4].

Novices may have trouble identifying important elements of code snippets because their lack of knowledge forces them to process each element of the code individually [5]. In contrast, experts can automatically 'chunk' multiple elements and process them as one unit because they have schema. Schema are long-term memory knowledge structures that help experts to organize new information [5]–[7]. Schema likely also help experts identify essential elements of content because they can quickly align new content with their existing knowledge [8]–[10]. However, work has not addressed how the specific types and order of elements that novices and experts recall can inform code example design. Furthermore, existing research on this topic primarily occurred before blocks programming languages became a popular way for novices to begin programming. Understanding the differences in how experts and novices recall code snippets could provide insight into how to help novices focus their attention more effectively on critical elements of both text and block examples.

We ran an exploratory study comparing how everyday, occasional, and non-programmers recall snippets of code. To explore differences between text and block code, participants studied and recalled two text and two block snippets. This work seeks to answer the question: Which code snippet elements do different levels of programmers initially recall? We have two contributions: 1) the key similarities and differences in recall between everyday, occasional and non-programmers, and 2) recommendations for beginning to improve code snippet presentations.

II. Related Work

This study was inspired by past work in novice-expert recall and code comprehension and contributes to work exploring important elements in code examples.

Research has explored chunking through comparisons of novice and expert recall in a variety of fields. One famous study found that expert chess players could correctly recall more chess pieces than novices for a valid configuration, supporting the theory that the experts can chunk common chess configurations using schema [6], [7]. Researchers have also studied the differences between novices and experts in fields such as physics [11] and programming [12]–[16]. Studies looking at schema and chunking in programming have often involved novice and expert programmers recalling code. Some of these studies replicate the chess study for programming, finding that experts can accurately recall more correctly structured code than random code [8]–[10], [15], [17]. Studies have also looked at how recall correlated with skills like comprehension and debugging [17], [18]. While prior work considered the differences between novices' and experts' knowledge and processing, our study design and analysis enable us to recommend improvements for example code, based on the types of elements programmers focus on.

Researchers have also developed a variety of theoretical models explaining the processes of comprehending code, such as top-down and bottom-up [19], [20]. Empirical work on code comprehension has used a variety of methods in addition to recall, such as eye tracking [21], answering questions [22], code modification [23], and debugging [24]. We expected

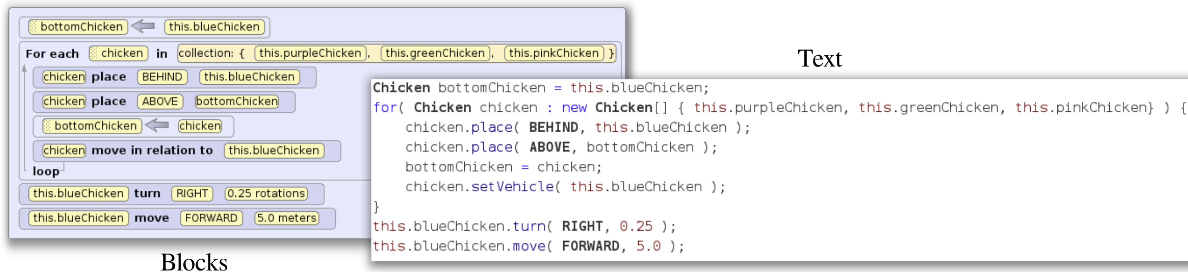

Fig. 1. An example of blocks and text versions for the same code snippet.

the differences in how novices and experts focus on and recall code in this study to complement findings in code comprehension.

In working towards helping programmers better comprehend examples, researchers have identified beneficial qualities of good code examples through: 1) analyzing effective code examples [25]–[29], and 2) determining how to create effective code summaries [30]–[34]. For the most part, this body of work has not addressed differences in programming expertise.

III. Methods

We ran a study using Mechanical Turk to explore how programming experience affects recall of code.

A. Participants

We recruited participants through Amazon's Mechanical Turk (MTurk), an online crowdsourcing platform, in order to recruit a diverse population of participants [35]. We recruited three populations based on self-reporting: 1) 21 non-programmers, 2) 21 occasional programmers, who program once in a while or used to program in the past, and 3) 18 everyday programmers, who program on an everyday basis. We grouped together participants who program once in a while and in the past because prior work has suggested that past programmers often forget many details [36]. We had 24 female, 35 male, and 1 unspecified gender participants. Participants ranged in age from 22 to 50 ($M = 33.3, SD = 7.3$).

B. Materials

Through pilot testing, we iteratively created four snippets of code in blocks and text for participants to recall. Each code snippet had 8-11 lines of code and included one of four control flow constructs: 1) a *while* loop, 2) an *if-else* block, 3) a *for* loop iterating three times, and 4) a *for* loop iterating through a list of objects. Fig. 1 shows block and text versions of the *for each* loop code snippet. We created code snippets in Looking Glass [37], a Java-based blocks programming environment for creating 3D animations. Looking Glass has a unique storytelling API for 3D animation, including methods like *walk*, *turn*, and *think* [38].

C. Study Protocol

Our study had: 1) an introduction phase, and 2) a study and recall phase, in which participants had three chances to memorize and recall four code snippets.

The first part of the study included an introduction, a demographic survey, and sample tasks. In order to determine participants' programming experience, participants filled out a survey that asked how often they program, how they learned to program, and which programming languages they know. To introduce participants to the mechanics of the study, participants first stepped through instructions and completed two sample recall tasks: one for text and one for blocks.

Participants then completed four study and recall tasks. In each of these tasks, participants saw a snippet of code (as in Fig. 1) and then attempted to recall it. In order to explore the differences in recall for text and blocks code, two consecutive tasks showed code in Java, while the other two tasks showed code in blocks. We randomized and balanced the ordering of the Java and blocks sets. Participants had three chances to memorize and recall each code snippet. Participants had 90 seconds in their first attempt and 30 seconds in the second and third attempts to recall each code snippet, based on pilot testing. We did not want to limit participants' recall by their typing speed, so participants did not have time limits for recall. After each pair of block and text code snippets, participants answered questions about their cognitive load for those tasks, using the validated difficulty and mental effort scales [39].

IV. Results

This section reports: A) the overall differences in programmers' recall, and B) differences in participants' recall of token types and positions in the first attempt. Because participants recalled most of the tokens they recalled in their first attempt, this paper only looks at participants' first attempt at recalling each code snippet. To compare recall of examples, we looked at token types, as shown in Table I and the position of tokens within examples. We compared token type recall using the Kruskal-Wallace test with the Dunn test for follow-up pairwise comparisons. To understand the position of tokens recalled, we analyzed the correlation between the line number and the percentage of tokens recalled in each line, using Spearman's R.

A. Overall

As expected, everyday programmers recalled larger overall percentages of the code snippets and found the tasks easier. Everyday programmers recalled significantly higher percentages of overall tokens than occasional and non-programmers, and occasional programmers also recalled significantly more

TABLE I. TOKEN TYPES

Group	Token Type	Example of token type
control flow	constructs	`if`, `for`, `while`
	keywords	`new`, `final`
	variable types & identifiers	`Integer`, `index`
	conditionals	`isCollidingWith`, `true`
	operators	`!`, `not`, `++`
objects	scope	`this`
	subject	`blueChicken`, `purpleChicken`
	accessors	`getRightHip`, `getLeftHip`
separators	separators	`;`, `(`, `)` `{`, `}`
actions	action identifiers	`move`, `turn`, `say`
arguments	numerical literals	`5`, `0`
	string literals	`'Uh oh'`
	enum literals	`ABOVE`, `FORWARD`
	function literals	`getDistanceTo`
	unit literals (only in blocks)	`meters`, `rotations`

tokens than non-programmers (see Table III). Although we expected to see differences in how programmers memorized and recalled block and text code examples, we found only a few differences, as shown in Table II.

Everyday programmers found the tasks significantly less difficult ($p < .05$) and everyday programmers required less effort than programmers with less experience ($p < .05$). Occasional programmers also found the tasks significantly less difficult than non-programmers ($p < .05$), but they did not need significantly less mental effort than non-programmers.

B. Which elements do programmers initially recall?

Programming experience aided in early recall of: 1) structural tokens, and 2) meaning details. All programmers initially recalled: 3) natural language tokens, and 4) tokens in the beginning of the code snippet.

1) Structural tokens: Everyday and occasional programmers primarily focused on core structural tokens in the first recall attempt (see Table III-Structural). These tokens set up the overarching control flow, objects, and the actions objects complete, but do not include the specifics of the control flow or actions. Core structural tokens include: construct tokens (e.g., `for each in`), related keywords (e.g., `collection`), separators (e.g., `{, }, ;`), objects (e.g., `this.bluechicken`) and actions (e.g., `turn`). For the most part, both everyday and occasional programmers recalled the highest percentages of core structural tokens. Occasional programmers also recalled significantly higher percentages of these tokens than non-programmers. This suggests that that both everyday and occasional programmers likely use schema to chunk critical structural components, but that occasional programmers probably do so somewhat less successfully. Specifically, occasional programmers may need more assistance with separator tokens. Non-programmers do not have knowledge to support recall of structural tokens and fall furthest behind in recalling construct

TABLE II. COMPARING BLOCKS AND TEXT, * $p < .05$,

Token	Everyday		Occasional		Non	
	Text	Blocks	Text	Blocks	Text	Blocks
Separators	ns	ns	19%	33%*	ns	ns
Construct	68%	50%, p=0.05	64%	43%*	ns	ns
Conditional	ns	ns	10%	24%*	5%	21%*

† All other tokens were non-significant

TABLE III. AVERAGE % OF TOKENS RECALLED BY EVERYDAY PROGRAMMERS AND DIFFERENCES BETWEEN GROUPS. $*p < 0.05$

	Token Type	Everyday	$\%_{Everyday} - \%_{Occasional}$	$\%_{Everyday} - \%_{Non}$	$\%_{Occasional} - \%_{Non}$
	All	64%	12%*	26%*	14%*
	String	74%	16%	25%	9%
Structural	Keywords	71%	12%*	30%*	12%*
	Construct	61%	5%	34%*	29%*
	Scope	61%	16%*	24%*	9%*
	Object	52%	15%*	23%*	8%*
	Separators	52%	21%*	28%*	7%*
	Actions	46%	13%*	22%*	9%*
Meaning	Variables	50%	15%*	23%*	8%
	Operators	56%	30%*	35%*	5%
	Conditional	36%	19%*	23%*	4%
	Enum	39%	18%*	15%*	−4%
	Numeric	34%	21%*	21%*	0%
	Unit	21%	16%*	16%*	0%
API	Accessors	24%	2%	5%	3%
	Functions	13%	8%	9%	1%

tokens. While many informal occasional programmers likely have some exposure to constructs, non-programmers who do not know how the constructs work will not realize their importance to the code snippets.

Structural tokens, specifically construct and separator tokens, were two of the differences between blocks and text code. Both everyday and occasional programmers recalled significantly more correct construct tokens for text than block code, likely because they were more familiar with the terms used in the text language (see Table II). In the block code, some constructs have been modified to clarify meaning, such as `repeat 3 times`. Blocks code also had far fewer separators, which may have helped less experienced programmers remember them. Occasional programmers recalled significantly higher percentages of correct separator tokens for block snippets than text snippets (see Table II). Separators can be critical for indicating structure and scope, so it may be especially important to consider the role of separators in code snippets.

2) Meaning detail tokens: Everyday programmers recalled many of the specific details that completed the construct and action statements in their first recall attempt. However, occasional programmers often recalled at rates similar to non-programmers (see Table III-Meaning). The variables, operators, and conditionals specify details, such as the iteration in a loop like `Integer index = 0; index<3; index++` or the condition for a while loop `!this.woodenBoat.isCollidingWith(this.iceberg)`. The enums, numbers, and units specify the details of actions, such as in `move(FORWARD, 5.0 meters)`, in which `FORWARD` is the enum. These tokens require a deeper understanding of the code snippet functionality in order to be memorable. Many of these tokens are also similar, making them somewhat difficult to recall correctly through direct memorization. For conditionals, one meaning detail token type, the blocks format may have helped occasional and non-programmers focus on or remember the details, possibly due to the natural language conditionals in block snippets. For example, text snippets use `isCollidingWith`, while the

TABLE IV. CORRELATION BETWEEN TOKEN TYPES AND LINE NUMBER

Token	Everyday	Occasional	Non-programmer
All	$-.26^{***}$	$-.37^{***}$	$-.37^{***}$
String	$-.55^{***}$	$-.67^{***}$	$-.57^{***}$
Construct	$-.19^{*}$	$-.42^{***}$	$-.32$
Separators	$-.25^{***}$	$-.33^{***}$	$-.32^{***}$
Action	$-.21^{***}$	$-.28^{***}$	$-.32^{***}$
Scope	$-.25^{***}$	$-.27^{*}$	$-.29^{***}$
Object	$-.24^{***}$	$-.25^{***}$	$-.25^{***}$
Variable	$-.32^{***}$	$-.42^{***}$	$-.30^{***}$
Operators	$-.26^{*}$	$-.33^{***}$	$-.29^{***}$
Conditional	$-.37^{*}$	$-.48^{*}$	$-.44^{*}$
Numerical	ns	$-.13$	ns
Unit	$-.23^{*}$	ns	ns
Accessors	ns	$-.19^{*}$	ns

†Keywords, Enum, and Function: no significant correlations.
† $*p < .05$ $***p < .001$

block snippet uses `is near`. This aligns with the recall rates for string tokens, as discussed next.

3) Natural language tokens: All of the programmer groups recalled surprisingly high percentages of string tokens. In this context, string tokens are the natural language arguments for `say` and `think` actions, like a penguin that says `"uh oh"`. We might expect occasional and non-programmers to recall these with ease, due to their simplicity, natural language, and uniqueness amongst the other code tokens. However, we did not expect everyday programmers to focus on string tokens, which are often less critical to the overall animation than structure or meaning details. Due to the storytelling nature of these code snippets, string tokens provide contextual information related to the overall animation, possibly making them more important to everyday programmers as well. The position of string tokens may have also been a factor, as many of these tokens were located at the beginning of the code examples.

4) Beginning of the code snippet: Participants recalled tokens closer to the beginning of the code snippet more than lines further down, with a few exceptions. All participants had significant negative correlations between the line number and the percentage of token types recalled (see Table IV). This indicates a relationship between the line number and recall. Everyday programmers often had weak correlations ($< .3$), while occasional and non-programmers often had more moderate correlations ($.3 < r < .5$). Some token types had no correlations, either because they only occurred at the beginning of examples, like keywords, or because participants rarely recalled them. Only occasional programmers recalled more numbers and accessors, such as `getRightHip`, earlier in the example than later. This may indicate that occasional programmers focused on less important tokens due to their location.

V. LESSONS LEARNED: RECOMMENDATIONS FOR IMPROVING CODE EXAMPLES

The results of this study indicate three main parts of an example that likely affect focus and recall: 1) the elements in the code themselves, 2) the overall presentation, and 3) the emphasis or de-emphasis of specific elements.

Our results indicate that several ways of selecting or designing example code could help draw programmers' attention to important elements: 1) position important elements early, as this is where programmers paid the most attention, 2) minimize similar identifier names to make them easier to distinguish, and 3) limit natural language string use within code, as they can draw undue attention. An expert could likely make these decisions, but example code could also be selected algorithmically using these constraints. Even with well-selected code, programmers will likely still need further support.

Occasional and non-programmers had the most difficulty with structural tokens and tokens that filled in the meaning of the core code structure. Helping less experienced programmers discover, focus on, and understand these critical elements of code snippets will be essential in enabling non-experts to effectively use example code. For instance, less experienced programmers had significant trouble remembering separators and their correct location. Reducing the focus on separators unrelated to the critical elements of the example might help everyday and non-programmers notice and remember the important separators.

In addition to the high-level structure and meaning, some examples may utilize specific elements that programmers should or should not pay attention to. When these critical details occur later in an example, programmers would likely benefit from help noticing those elements. Furthermore, some examples may involve many unique details which could distract a non-expert. Newer programmers would likely benefit from having these either removed or de-emphasized.

VI. THREATS TO VALIDITY

Since we wanted to derive possible directions for further exploration and design, we chose to risk a possible higher false positive rate, rather than a higher false negative rate in our statistical analysis. While we did not correct for multiple comparisons, the small number of comparisons and the fact that we chose the comparisons in advance does reduce our risk of a high false positive rate.

VII. CONCLUSION

This study explored what everyday, occasional, and non-programmers focused on and recalled for code snippets. Programmers ranging from experts to complete novices rely on code snippets to learn new programming skills and to attempt to accomplish programming tasks outside of formal education contexts. By exploring what programmers focus on and recall in code snippets, we can recommend ways to design code snippets to better support the growing number of programmers learning through examples. We provide recommendations for improving example code, but leave the application and evaluation of these recommendations to future work.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant Nos. 1054587 and 1440996. We would like to thank Kyle Harms and Randy Brachman for their valuable feedback.

References

[1] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two studies of opportunistic programming: interleaving web foraging, learning, and writing code," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2009, pp. 1589–1598.

[2] C. Scaffidi, M. Shaw, and B. Myers, "Estimating the numbers of end users and end user programmers," in *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on*, 2005, pp. 207–214.

[3] M. B. Rosson, J. Ballin, and H. Nash, "Everyday Programming: Challenges and Opportunities for Informal Web Development," in *2004 IEEE Symposium on Visual Languages and Human Centric Computing*, Sep. 2004, pp. 123–130.

[4] M. Ichinco and C. Kelleher, "Exploring novice programmer example use," in *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*. IEEE, 2015, pp. 63–71.

[5] F. Gobet, P. C. Lane, S. Croker, P. C. Cheng, G. Jones, I. Oliver, and J. M. Pine, "Chunking mechanisms in human learning," *Trends in cognitive sciences*, vol. 5, no. 6, pp. 236–243, 2001.

[6] A. D. De Groot and A. D. de Groot, *Thought and choice in chess*. Walter de Gruyter, 1978, vol. 4.

[7] W. G. Chase and H. A. Simon, "Perception in chess," *Cognitive psychology*, vol. 4, no. 1, pp. 55–81, 1973.

[8] K. B. McKeithen, J. S. Reitman, H. H. Rueter, and S. C. Hirtle, "Knowledge organization and skill differences in computer programmers," *Cognitive Psychology*, vol. 13, no. 3, pp. 307–325, 1981.

[9] B. Adelson, "Problem solving and the development of abstract categories in programming languages," *Memory & cognition*, vol. 9, no. 4, pp. 422–433, 1981.

[10] R. S. Rist and others, "Plans in programming: definition, demonstration, and development," in *first workshop on empirical studies of programmers on Empirical studies of programmers*, 1986, pp. 28–47.

[11] M. T. Chi, P. J. Feltovich, and R. Glaser, "Categorization and representation of physics problems by experts and novices," *Cognitive science*, vol. 5, no. 2, pp. 121–152, 1981.

[12] M. T. Chi, R. Glaser, and E. Rees, "Expertise in problem solving." DTIC Document, Tech. Rep., 1981.

[13] J. Larkin, J. McDermott, D. P. Simon, and H. A. Simon, "Expert and novice performance in solving physics problems," *Science*, vol. 208, no. 4450, pp. 1335–1342, 1980.

[14] A. H. Schoenfeld and D. J. Herrmann, "Problem perception and knowledge structure in expert and novice mathematical problem solvers." *Journal of Experimental Psychology: Learning, Memory, and Cognition*, vol. 8, no. 5, p. 484, 1982.

[15] E. Soloway and K. Ehrlich, "Empirical studies of programming knowledge," *IEEE Transactions on Software Engineering*, no. 5, pp. 595–609, 1984.

[16] D. J. Gilmore and T. R. G. Green, "Programming plans and programming expertise," *The Quarterly Journal of Experimental Psychology*, vol. 40, no. 3, pp. 423–442, 1988.

[17] B. Shneiderman, "Exploratory experiments in programmer behavior," *International Journal of Computer & Information Sciences*, vol. 5, no. 2, pp. 123–143, 1976.

[18] I. Vessey, "Expert-novice knowledge organization: An empirical investigation using computer program recall," *Behaviour & Information Technology*, vol. 7, no. 2, pp. 153–171, 1988.

[19] A. Von Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, 1995.

[20] M.-A. Storey, "Theories, methods and tools in program comprehension: Past, present and future," in *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*. IEEE, 2005, pp. 181–191.

[21] T. Busjahn, C. Schulte, and E. Kropp, "Developing Coding Schemes for Program Comprehension using Eye Movements," *PPIG, University of Sussex*, 2014.

[22] S. Wiedenbeck, "Novice/expert differences in programming skills," *International Journal of Man-Machine Studies*, vol. 23, no. 4, pp. 383–390, Oct. 1985.

[23] J. Koenemann and S. P. Robertson, "Expert problem solving strategies for program comprehension," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1991, pp. 125–130.

[24] I. Vessey, "Expertise in debugging computer programs: A process analysis," *International Journal of Man-Machine Studies*, vol. 23, no. 5, pp. 459–494, 1985.

[25] M. Tavakoli, A. Heydarnoori, and M. Ghafari, "Improving the quality of code snippets in stack overflow," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. ACM, 2016, pp. 1492–1497.

[26] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns, "What makes a good code example?: A study of programming Q&A in StackOverflow," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 25–34.

[27] R. P. Buse and W. Weimer, "Synthesizing API usage examples," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 782–792.

[28] K. S.-P. Chang and B. A. Myers, "WebCrystal: understanding and reusing examples in web authoring," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2012, pp. 3205–3214.

[29] A. Head, C. Appachu, M. A. Hearst, and B. Hartmann, "Tutorons: Generating context-relevant, on-demand explanations and demonstrations of online code," in *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*. IEEE, 2015, pp. 3–12.

[30] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE, 2013, pp. 23–32.

[31] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Reverse Engineering (WCRE), 2010 17th Working Conference on*. IEEE, 2010, pp. 35–44.

[32] A. T. Ying and M. P. Robillard, "Code fragment summarization," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 655–658.

[33] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello, "Improving Automated Source Code Summarization via an Eye-tracking Study of Programmers," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 390–401.

[34] A. T. Ying and M. P. Robillard, "Selection and presentation practices for code example summarization," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 460–471.

[35] "Amazon Mechanical Turk - Welcome." [Online]. Available: https://www.mturk.com/mturk/welcome

[36] M. Ichinco, A. Zemach, and C. Kelleher, "Towards generalizing expert programmers' suggestions for novice programmers," in *Visual Languages and Human-Centric Computing (VL/HCC), 2013 IEEE Symposium on*. IEEE, 2013, pp. 143–150.

[37] "Looking Glass Community." [Online]. Available: https://lookingglass.wustl.edu/

[38] C. Kelleher, R. Pausch, and S. Kiesler, "Storytelling alice motivates middle school girls to learn computer programming," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2007, pp. 1455–1464.

[39] F. G. Paas, J. J. Van Merriënboer, and J. J. Adam, "Measurement of cognitive load in instructional research," *Perceptual and motor skills*, vol. 79, no. 1, pp. 419–430, 1994.