Code Scanning Patterns in Program Comprehension

Christoph Aschwanden[1] and Martha Crosby[2]
Adaptive Multimodal Interaction Laboratory
Department of Information and Computer Sciences
University of Hawaii at Manoa
[1]caschwan@hawaii.edu [2]crosby@hawaii.edu

Abstract

Various publications have identified Beacons to play a key role in program comprehension. Beacons are code fragments that help developers comprehend programs. It has been shown that expert programmers pay more attention to Beacons than novices. Beacons are described as the link between source code and hypothesis verification. Beacons are sets of key features that typically indicate the presence of a particular data structure or operation in source code. However, only little research has been done trying to identify and explain them in greater detail. It has been demonstrated that good variable and procedure names help in program comprehension. Documentation is beneficial as well. The so-called swap operation for variables is a strong indicator for a sorting algorithm. We conducted an eye tracking study using the EventStream software framework as the instrument to investigate programmers' behavior during a code reading exercise. Preliminary results suggest Beacons to be present when the longest fixation duration is thousand milliseconds or higher. Comparing participants with correct understanding versus participants with wrong understanding showed differences in focus of attention. Based on the study conducted, we suggest to consider "int k=(a+b)/2" as Beacons during program comprehension as well as lines of code which exhibit very long fixations above 1000 milliseconds.

1. Introduction

Programming is considered a challenging endeavor to undertake. But what makes it so hard? The fact is people spend a long time to actually learn how to program. Programs cannot easily be written down. Rules and constraints have to be considered for the code to function properly. Wiedenbeck, Soloway, von Mayrhauser et al. suggest that people use different approaches to understand a program. Research by Brooks, Letovsky, Littman et al. shows comprehension to be top-down, bottom-up, knowledge based, as-needed, control flow based and integrated. Most studies have been conducted with paper and pencil tests. Eye movements have rarely been used to identify eye scanning patterns during software comprehension [19][21][22].

How does a programmer perceive code? What types of scanning patterns are used during the comprehension process? Do all programmers use the same techniques, i.e. scanning patterns to understand a program? Do programmers with varying experience levels show different traits?

Eye movement studies by Crosby and Stelovsky [21] have determined that people use a variety of scanning patterns. Programmers' strategies range from single scan to multiple scan, i.e. programmers scan through the code once or several times to understand it. Some developers focus more on numbers, while others focus more on text. Some people use comparative strategies during the comprehension process. Other studies suggest that programmers use Beacons.

It has been shown that good variable and procedure names help in program comprehension. Documentation is beneficial as well. The so-called swap operation for variables is a strong indicator for a sorting algorithm. It has been shown that expert programmers pay more attention to Beacons than novices. Various publications have identified that Beacons play a key role in program comprehension. Beacons are code fragments that help developers comprehend programs. Experienced programmers rely on Beacons to guide their comprehension process. Brooks describes Beacons as the link between source code and hypothesis verification [11]. Wiedenbeck claims that Beacons are a set of key features that typically indicate the presence of a particular data structure or operation in source code [87]. However, very little research has been done to try to identify and explain them in greater detail.

So, how can eye movement data be used to explain why some scanning patters yield better results than others? Can scanning patterns be classified in a meaningful way and how do they relate to other studies that focus on models of program comprehension such as top-down or bottom-up?

An empirical study was conducted to identify scanning patterns in program comprehension. Twelve algorithms were shown to participants who had to recognize and name them correctly. An eye Tracking System and the EventStream software framework [4][28] were used as the instruments to evaluate people's eye movements during comprehension. Scanning patterns were analyzed to support or deny the notion of Beacons.

2. Related Work

Several models have been proposed to explain how software developers understand programs. Software comprehension has been described as top-down by Brooks [11], bottom-up by Basili and Mills [6][73], knowledge based by Letovsky and Soloway [53], as-needed by Littman and Pinto [54][76], control-flow based by Green and Pennington [35][58][59] and integrated by von Mayrhauser [85]. While the integrated model of program comprehension has been published most recently, there is

no clear evidence on why people scan through code the way they do.

Domain level knowledge is important when programmers attempt to understand a program. Especially in object oriented languages Ramalingam and Wiedenbeck describe domain level knowledge as imperative [67]. Application domain knowledge has been shown beneficial for program comprehension. People that are familiar with a domain tend to understand programs better than people that are not familiar with the domain. Research by Shaft, Vessey and von Mayrhauser indicates the top-down approach is used to scan through source code. While bottom-up is used if people are unfamiliar with a particular application domain [16][51][72][84].

Experience level can be defined as the number of years programming [15]. However, other factors exist that influence experience level including intellectual capability, knowledge base, cognitive style, motivation level, personal characteristics and behavioral characteristics [23]. Cognitive factors have been found to play an important role in programming proficiency [9][27]. Research by Adelson shows expert developers rely on abstract problem descriptions to understand code. Experts use a semantical approach in the comprehension process. Novices on the other hand are driven by how a program works syntactically rather than what a program is doing semantically [2]. Davies, Green, Soloway and Ehrlich argue that experienced programmers use programming plans during program comprehension [24][34][36][69][75]. Experts use more advanced strategies such as the top-down model and Beacons when trying to understand a program. But results vary and are inconclusive [30][52]. Little is known on how people become experts. Evidence suggests that some people are more skilled than others, independent of the number of years programming. However, little proof is given on the reason why.

Other research focuses on mental representations used by software developers during program comprehension. People build a mental image when trying to understand code [58][61]. Winner and Casey argue that non-verbal IQ is important for mental imagery in a field [88]. Also spatial rather than visual images are used when programmers build an abstract mental model of a problem [55]. Some studies address the rationale why programmers use a particular strategy to understand a program. The Information-Foraging theory by Pirolli and Card [63][64] has been successfully applied to anthropology [74], biology [77] or information retrieval in the World-Wide Web by explaining peoples' behavior as an evolutionary concept. However no success can be reported for other fields such as program comprehension in computer science.

Research by Tenny and Woodfield has shown that comments, documentation and meaningful variable or procedure names are beneficial to program comprehension. Good indentation correlates with code understanding as well. A study by Miara, et al. found three levels of indentation to be the optimal size [12][52][56][65][81][89]. Knuth uses Literate Programming [50] to improve the readability of software. However, Literate Programming has not been applied well in industry. It is argued the tradeoff between readability and time to write a program in literate style is too high to be beneficial for professional companies. Therefore, Literate Programming is not very well accepted outside of Academia.

Baecker, Bednarik, Hendrix and Storey focus more on tools, than on program comprehension itself [7][8][17][37][70][80]. Tools that enhance the readability of programs have been shown to be beneficial. In particular tools that try to improve the visualization of large data structures on the one hand and single lines of code on the other hand. Fisheye views are used to display large programs allowing browsing from a fish eye perspective. Fisheye views are used to replace scroll bars by magnifying selected areas while the rest of the text or imagery is displayed as tiny text or graphics. Fisheye views have been found superior compared to flat views [31]. Control structure diagrams [37] are used to enhance the readability of loop or conditional structures.

Several studies have been published that describe the various aspects of people trying to understand code. However, there is only limited evidence of how programmers perceive code. There is a lot of information about how but not why people read and comprehend programs the way they do.

2.1. Beacons and Chunks

Studies related to Beacons and eye tracking have been done sparsely. Beacons are described in numerous publications. Gellenbeck and Cook argue that it is not clear if Beacons really exist [33] and if they do, how they manifest themselves. Beacons are defined as a guide that programmers use during their code reading process. Brooks describes Beacons as providing the link between the process of verifying hypotheses and the actual source code [11]. Wiedenbeck describes Beacons as sets of key features that typically indicate the presence of a particular data structure or operation in source code [87]. Beacons are particularly useful during the top-down model of program comprehension. Meaningful variable and procedure names have been described as Beacons. The swap operation has been shown to be a Beacon and to be beneficial in comprehension as well [11][19][21][45][66][87].

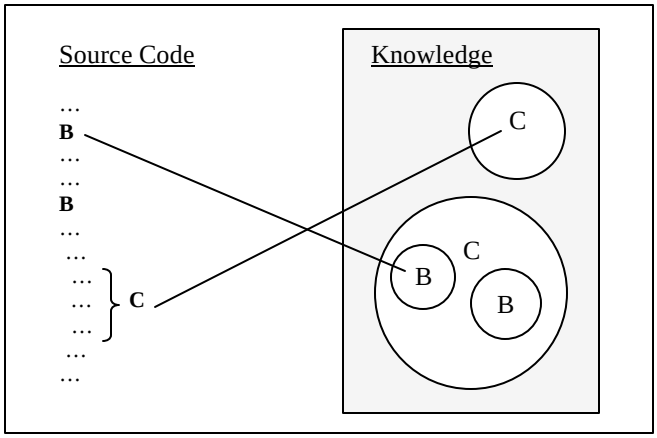

Figure 2.1 - Beacons and Chunks

Chunks are described as code fragments in programs. Available literature shows Chunks to be used during the bottom-up approach of software comprehension. Chunks vary in size. Several Chunks can be combined into larger Chunks [5][14][26][37][47][57][59][66][73][75][83].

Beacons and Chunks share similarities. Both are defined as code fragments. Figure 2.1 combines various theories describing Beacons and Chunks into one diagram. "B" represents a Beacon, while "C" represents a Chunk. How Beacons and Chunks exactly help programmers understand code has still to be defined.

Tracking the participant's eye movements can show their focus of attention. Numerous studies not related to computer science have been conducted. People were found to use fixations and saccades while they look at texts or imagery. Research by Zelinsky describes fixation on an object depends on the time to inspect the object and the time to comprehend it [90][91][92]. Does the same apply to reading code?

How does one set apart Beacons and normal code? Which lines of code qualify as Beacon; which lines of code don't? The question can't be answered with a simple yes or no. An answer based on a continuous scale is far more feasible. A swap operation could be rated as 95% Beacon likely, while a simple loop statement might for example get a 33% Beacon score. How does one define such a scale?

Eye movement research is well established in text reading and image recognition. However only a few studies have investigated programmers while they were reading code [19][21][22]. Programmers use various strategies to comprehend software. People's scanning techniques change from single to multiple scan eye movements. It has also been found that some people focus more on numbers, while others focus more on text. Scanning patterns range from top-to-bottom to left-to-right strategies. Rereading is a common practice as well. Bednarik found novices exhibit higher mean fixation durations than more experienced programmers [8]. There has no relationship been established between expertise and reading strategy.

Putting it all together, there is a lack of studies analyzing eye scanning patterns. Different models of comprehension such as top-down or bottom-up have been identified and could be verified by analyzing the participants' eye movements. Eye movement research gives more insight about what a programmer is looking at during the comprehension process.

3. Code Reading Experiment

An empirical study was conducted to identify scanning patterns and Beacons in code. Twelve algorithms were shown to participants who had to identify them and answer a number of questions. An ASL eye tracking system [3] was used to record eye movements during the experiment. The EventStream Framework was utilized as the instrument for setting up the experiments, data recording and data analysis.

3.1. Participants

The study was conducted during the last weeks of fall semester 2004 at the Adaptive Multimodal Interaction laboratory [1], University of Hawaii at Manoa. Participants were recruited from a third year computer science class and given extra credit for participation. Fifteen participants performed the experiment which was one hour or less in duration including the filling out the pre-questionnaire, the post-questionnaire and the consent form.

3.2. Materials

The experiment was geared towards elaborating scanning patterns and Beacons in code. The study consisted of three parts:

- **Pre-Questionnaire** to evaluate programming expertise and interest.
- **Experiment** to record eye movements during program comprehension.
- **Post-Questionnaire** to evaluate tasks and satisfaction with the experiment.

The pre-questionnaire asked questions about the experience level; the number of years programming; academic standing and programming languages known. Programming interest and motivation for participation were evaluated as well. The post-questionnaire asked about the difficulty level of the tasks performed. Furthermore problems encountered and satisfaction with the experimental setup was evaluated.

The experiment consisted of the recursive and non-recursive versions of six algorithms. Java is the primary language used to teach programming at the University of Hawaii. Therefore, all algorithms were shown in Java. These were the algorithms used:

- **Sum algorithm** - Sums up the items in an array.
- **Exponent algorithm** - Calculates a to the power of b.
- **Factorial algorithm** - Returns factorial of n.
- **Binary search algorithm** - Returns the index of an item in an array.
- **GCD algorithm** - Returns the greatest common divisor of two numbers.
- **Fibonacci algorithm** - Returns the nth element of the Fibonacci sequence.

Given the algorithms, two sequences of the tasks were created. The six algorithms were shown in both recursive and non-recursive form, following the Java style guidelines by Vermeulen et al. [82]. Recursive and non-recursive algorithms were displayed alternately. For each version, three algorithms were shown in the recursive form first, while the other three algorithms were shown in the non-recursive form first. Figure 3.1 depicts the two sequences created.

Version 1
1. Factorial (recursive)
2. Sum (non-recursive)
3. Binary Search (recursive)
4. Exponent (non-recursive)
5. GCD (recursive)
6. Fibonacci (non-recursive)
7. Exponent (recursive)
8. GCD (non-recursive)
9. Fibonacci (recursive)
10. Factorial (non-recursive)
11. Sum (recursive)
12. Binary Search (non-recursive)
Version 2
1. Factorial (non-recursive)
2. Sum (recursive)
3. Binary Search (non-recursive)
4. Exponent (recursive)
5. GCD (non-recursive)
6. Fibonacci (recursive)
7. Exponent (non-recursive)
8. GCD (recursive)
9. Fibonacci (non-recursive)
10. Factorial (recursive)
11. Sum (non-recursive)
12. Binary Search (recursive)

Figure 3.1 - Sequence of Tasks

The participants were divided into two groups, one group given task sequence version one, the other group given version two.

3.3. Procedure

The experimental session began by explaining the laboratory setup with the eye tracking system. The participants then filled out the consent form and the pre-questionnaire. The participants were seated in front of the computer monitor and the eye tracking system calibration was performed.

The experiment was started and twelve tasks were given to the participants. Each task required a participant to identify and name an algorithm correctly. Two oral questions were asked. The questions were about what the algorithm was doing and how the participants found out about it. The correctness of the answers was determined by (a) the participant naming the algorithm correctly or (b) the participant describing an algorithm in adequate detail. For example, answers by a participant not able to name the factorial algorithm, however describing it as "1*2*3*…*n" were counted as correct. Answers describing an algorithm line by line rather than by its purpose were counted as incorrect. After all the tasks were completed, participants were handed out the post-questionnaire.

3.4. Results

Post-questionnaire evaluation showed participants satisfied with the experiments overall. Fourteen out of fifteen participants responded they would be willing to sign up for the same study again. No complaints were recorded from participants.

For the twelve algorithms shown to the participants, a total of fifteen hours of data was collected during the experimental sessions. A repeated measure analysis shows a significant difference for correctness of answers between the six recursive and the six non-recursive algorithms, $F(1, 5) = 2.56$, $p = .036$. A significant effect is observed as well for subjects encountering the recursive algorithm first or vice-versa the non-recursive algorithm first, $F(1, 5) = 2.77$, $p = .025$. Participants exhibited learning as they performed better when they saw an algorithm the second time. No statistical differences were observed on the time spent by the participants to comprehend the various algorithms.

A separate analysis for each of the algorithms comparing participants who got the answers wrong with participants who got the answers right, shows no difference on what people look at or deem important for understanding. The exception is the non-recursive binary search algorithm. Subjects who got the answer right, focused more on line "int k= (a+b) / 2", $F(1, 11) = 5.36$, $p = .041$. The same however isn't true for the recursive counterpart, $F(1, 11) = 0.84$, $p = .38$. For most algorithms, the ratio of correct answers to wrong answers was fairly different. Distribution of correct and wrong answers was uneven. In regard to the non-recursive binary search algorithm, the distribution was six people correct and seven people wrong. For other algorithms, the unbalanced distribution prevented making any comparisons.

Tests of within subjects effects show that participants behaved as individuals. Analysis of time spent, $F(1, 5) = 10.0$, $p < .001$, and correct answers recorded, $F(1, 5) = 3.61$, $p < .001$, are highly significant. Also, eye movement speeds, $F(1, 5) = .49$, $p = .069$, suggest inherent differences between the way participants view programs.

Interesting findings can be reported from evaluating means. One sample t-test returned highly significant results for participants focusing on the many lines of code, $t > 5$, $df = 12$, $p < 0.01$. Some lines appear to draw higher interest (fixation duration average, longest fixation) from participants than others. Comparing recursive algorithms with their non-recursive counterparts showed similar traits in interests for similar code fragments. The average longest fixation duration for "int k = (a+b) / 2" is 1062ms for the recursive binary search, 1039ms for the non-recursive version. Neither correlation, nor ANOVA confirmed or disproved the findings.

Correlation of subjects on what line numbers they looked at the most and also said they deemed important for understanding returned no significant results. Some participants knew which line number they looked at the most and were also able to name them. However, even though some of the participants knew what they were doing, a paired samples t-test didn't yield any conclusive results. Knowing where you look and knowing what an algorithm does showed no relation at all, $t = -1.84$, $df = 12$, $p = .091$. Asking people what they deem important appears not to be a reliable way of determining Beacons or important lines in code.

Comparing correctness of answers versus programming interest exhibits an increase in correct answers for participants with higher interest in programming. A linear regression analysis returned no significant difference, $R^2 = .13$, $F(1, 13) = 1.97$, $p = .18$. Further studies are needed to verify these results. No correlations were observed between number of years programming, correctness of answers and programming interest. No relations were established for pupil size and eye blinks. Pupil size and eye blinks appear to be very individualistic and different for each of the participants.

Overall Correctness

Table 3.1 depicts the correctness of answers recorded for the algorithms in the study. Comprehension for factorial and sum algorithms is very high. On the other hand, GCD and Fibonacci were hard to understand.

Algorithms	Recursive	Non-Recursive	Both
Factorial	100%	79%	90%
Sum	79%	87%	83%
Binary Search	66%	45%	55%
Exponent	52%	66%	59%
GCD	7%	7%	7%
Fibonacci	6%	0%	3%
Total	**52%**	**47%**	**49%**

Table 3.1 - Correct Answers for Algorithms

Correct vs. Wrong

Comparing participants with correct answers to participants with wrong answers returned no significant results for most algorithms. Differences were observed only for the non-recursive binary search algorithm. For all the algorithms, with the exception stated above, participants, correct or incorrect, focused on the same lines of code or named the same areas as important.

No relation was established between lines of code participants looked at and line numbers they said were important for understanding. What participants looked at didn't correlate with what participants said. Subjects didn't spend more time on line numbers that they said were important. Neither did they spend less time.

Considering the binary search algorithm, participants who identified the algorithm correctly focused more on line numbers four and five (+). Subjects, who got it wrong, mentioned line numbers six and seven more frequently (-). They were also fixating more on line number one (-). See Figure 3.2.

```
1- public int do(int list[], int value) {
2    int a = 0;
3    int b = list.length - 1;
4+   while (a != b) {
5+     int k = (a + b) / 2;
6-     if (value > list[k]) {
7-       a = k + 1;
8      }
9      else {
10       b = k;
11     }
12   }
13   return a;
14 }
```

Figure 3.2 - Non-Recursive Binary Search Algorithm

The differences are significant for line number five, $F(1, 11) = 5.36$, $p = 0.041$. The same doesn't hold true for the recursive binary search algorithm, $F(1, 11) = 0.84$, $p = .38$. It appears for the non-recursive version, "int k=(a+b)/2" is a Beacon for program comprehension. More tests are needed to verify or dismiss these results.

Preliminary data indicates people better at programming focus more on recursions and loops, people less skilled more on conditional statements and line number one. None of these results were statistically significant.

Lines of Interest

Interesting findings can be reported from evaluating means. One sample t-test returned highly significant results for participants focusing on the many lines of code, $t > 5$, $df = 12$, $p < 0.01$. Some lines appear to draw higher interest (fixation duration average, longest fixation) from participants than others.

Fixation duration average refers to the portion of fixation time to total time per algorithm spent on a particular line of code. Longest fixation refers to the longest fixation duration that occurred on a particular line of code. Values were averaged for all participants, correct and incorrect. It appears participants used the same lines of code for comprehension.

The longest fixation duration appears also to be related for similar statements. Considering the binary search algorithm, the average longest fixation duration for "int k=(a+b)/2" is 1062ms for the non-recursive binary search, 1039ms for the recursive version. "if (value > list[k])" exhibits a 1191ms longest fixation in the non-recursive version, 1017ms in the recursive counterpart.

Considering the exponent algorithm, "k = k * a" exhibits a 829ms longest fixation (non-recursive), "a * do(a, b - 1)" exhibits 842ms (recursive). Both statements are related, however modified to fit into the recursive and non-recursive version of the algorithm. The same was found for the factorial algorithm with 947ms and 894ms. The sum algorithm exhibits 786ms and 712ms.

Similar statements seem to take similar time for comprehension. Similar traits were observed for GCD and Fibonacci. ANOVA didn't return any significant results regarding similar statements. However, similar statements didn't differ more than 200ms for the longest fixation duration.

The fixation duration average and longest fixation for the first line of code were found to be almost always larger for the recursive algorithms. Although that seems interesting, the complexity of the first line of code for the recursive algorithms was higher than for their non-recursive counterpart.

3.5. Discussion

Using the EventStream software framework as the instrument for the program comprehension study, each line of code was analyzed based on its longest fixation average. Some lines were found to have much higher fixation duration than others.

The sum, factorial and exponent algorithms didn't have any longest fixation average above 1000ms. This can be explained by the simplicity of these algorithms. Code statements were of rather elementary nature. For the other algorithms, statements in Table 3.2 were found to be over 1000ms.

	Recursive	Non-Recursive

Binary Search	`int k = (a + b)/2;`
	? 1062ms
	`if (list[k]<value) {`
	? 1191ms
		`int k = (a + b)/2;`
		? 1039ms
		`if (value>list[k]) {`
		? 1117ms

GCD	`return do(b%a,a);`
	? 1371ms
		`int k = a;`
		? 1209ms
		`a = b % a;`
		? 1216ms

Fibonacci	`return do(n-1,b,a+b);`
	? *916ms*
		`b = a + b;`
		? 1062ms
		`a = b - a;`
		? 1191ms

Table 3.2 - Lines of Code with a Longest Fixation Duration greater than 1000ms

It appears lines crucial to the comprehension process have a higher longest fixation average. "int k = a" and "a = b % a" are rather simple statements by themselves, but are important for the comprehension process and draw higher interest from programmers than other lines. Statistical analysis didn't yield any significant results regarding Beacons, but it can be hypothesized that these lines of code are crucial to comprehension and could thus be considered as important during program comprehension.

For reading, fixations are typically 200-250 milliseconds in length. Fixation on an object depends on the time to inspect the object and the time to comprehend it [29][68][90][91][92]. For fixations far above 200-250, this time difference can be assumed to be used for information processing. For fixations 1000 milliseconds or above, at least 750-800ms are used for information processing. This number indicates that at least part of code reading can be categorized as problem solving rather than searching. However, not all fixations are long. Program comprehension appears to be a combination of searching and problem solving.

4. Contributions and Future Directions

So far, very little effort has been made to assess people's eye movements during code reading. Eye movements show where programmers focus their attention. Eye movements can be used to evaluate areas of interest for programmers that help their comprehension process. This knowledge in turn can be used to help software developers in the following ways:

- Create programming languages that better satisfy their needs:
  - Identify areas that are hard to comprehend and revise them.
- Create development environments that increase programmer's efficiency and quality of code:
  - Code/Beacon Highlighting
- Define better teaching methods:
  - What do experts look at compared to novices?
  - What to look at during debugging / error search.
- Integrate eye-tracking into the code reading process:
  - Does a programmer focus on the correct area in code?

A comparison of people with correct comprehension versus people with wrong comprehension didn't yield any conclusive results regarding their scanning behaviors. The only exception was the non-recursive binary search algorithm. Subjects who got the answer right, focused more on line "int k=(a+b)/2". This result corresponds to previous research [19]. There are code fragments that set apart less and more experienced programmers.

Evaluating fixation mean times shows some lines of code draw higher interests from programmers than others. It is hypothesized that the longest fixation average, as identified by the EventStream software framework, indicates Beacons in code. The longest fixation average is similar for related statements. Results are highly significant regarding the fixation time. Combining these two results we come up with two types of code fragments that help during comprehension:

1) Lines which are used by experts during the comprehension process: int k=(a+b)/2

2) Lines which are used by both experts and novices equally during the comprehension process

Statement **1)** appears to follow Brooks' description of Beacons: Beacons provide the link between the process of verifying hypotheses and the actual source code [11]. Expert developers rely on abstract problem descriptions to understand code. Experts use a semantical approach in the comprehension process [2][24][34][36][69][75]. Thus, it

can be assumed that experts use hypotheses and follow Brooks' description of Beacons. This explains Statement 1): experts focus more on certain lines than novices.

Statement **2)** appears to follow Wiedenbeck's description: Beacons are sets of key features that typically indicate the presence of a particular data structure or operation in source code [87]. Both experts and novices focus on certain lines equally, which would explain Statement 2).

Therefore, the author of this document suggests adding the following code fragments to the list of Beacons, which follow the theorems by both Brooks and Wiedenbeck:

- Statement **1)**: "int k=(a+b)/2"
- Statement **2)**: Content of Table 3.2
- Statement **2)**: Code Fragments with an average longest fixation of 1000ms or higher

References

[1] Adaptive Multimodal Interaction Laboratory (AMI), Information and Computer Sciences Department, University of Hawaii at Manoa. http://ami.ics.hawaii.edu

[2] Adelson, Beth. (1983). *Structure and Strategy in the Semantically-Rich Domains.* Ph.D. Thesis 1983, Harvard University.

[3] Applied Science Laboratories, Technology and Systems for Eye Tracking. http://www.a-s-l.com

[4] Aschwanden, C., Stelovsky, J. (2003). Measuring Cognitive Load with EventStream Software Framework. *HICSS Conference*, IEEE, 2003.

[5] Badre, A. (1982). Designing Chunks for Sequentially Displayed Information. In Badre, A. and Shneiderman, B. (eds.), Directions in Human Computer Interaction, Ablex Publishing, 179-193.

[6] Basili, V. R., Mills, H. D. (1982). Understanding and Documenting Programs. *IEEE Trans. Software Eng.* SE-8, 3 (May 1982), 270-283.

[7] Baecker, R. (1988). Enhancing program readability and comprehensibility with tools for program visualization. *Proceedings of the 10th international conference on Software engineering*, Singapore, Pages: 356 - 366.

[8] Bednarik, R., Myller, N., Sutinen, E., Tukiainen, M. (2005). Effects of Experience on Gaze Behavior during Program Animation. In P. Romero, J. Good, E. Acosta Chaparro & S. Bryant (Eds). Proc. PPIG 17, Pages 49-61.

[9] Bergin, S., Reilly, R. (2005). Programming: factors that influence success. *Technical Symposium on Computer Science Education*, Proceedings of the 36th SIGCSE technical symposium on Computer science education, St. Louis, Missouri, Pages: 411 - 415.

[10] Bertholf, C. F., Scholtz, J. (1993). Program Comprehension of Literate Programs by Novice Programmers. In *Empirical Studies of Programmers*: Fifth Workshop. Norwood, NJ: Ablex Publishing. p. 222.

[11] Brooks, Ruven. (1983). Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man-Machine Studies*, 18, 543-554.

[12] Brooks, Ruven. (1978). Using a Behavioral Theory of Program Comprehension in Software Engineering. *Proceedings of the 3rd international conference on Software engineering*, 196-201.

[13] Canas, J. J., Antoli, A., and Quesada, J. F. (2001). The role of working memory on measuring mental models of physical systems. *International Journal of Methodology and Experimental Psychology*, Vol. 22.

[14] Cant, S. N., Jeffery, D. R., Henderson-Sellers, B. (1995). A Conceptual Model of Cognitive Complexity of Elements of the Programming Process. *Information and Software Technology*, 37(7), 351-362.

[15] Chrysler E. (1978). Some basic determinants of computer programming productivity. *Communications of the ACM*, Volume 21, Issue 6 (June 1978), Pages: 472 - 483, ISSN:0001-0782.

[16] Clayton, Richard, Rugaber, Spencer, Wills, Linda. (1998). On the Knowledge Required to Understand a Program. *Working Conference on Reverse Engineering*.

[17] Clements, Paul, Krut, Robert, Morris, Ed, Wallnau, Kurt. (1996). The Gadfly: An Approach to Architectural-Level System Comprehension. *4th IEEE Workshop on Program Comprehension*.

[18] Coventry, Lynne. (1989). Some Effects of Cognitive Style on Learning UNIX. *International Journal of Man-Machine Studies*, 31, 349-365.

[19] Crosby, Martha E., Scholtz, Jean, Wiedenbeck, Susan. (2002). The Roles Beacons Play in Comprehension for Novice and Expert Programmers. *Proceedings of the 14th Annual Workshop of the Psychology of Programming Interest Group*, London, UK. June 18-21, pp. 58-73.

[20] Crosby, M., Auernheimer, B., Aschwanden, C., Ikehara, C. (2001). Physiological Data Feedback for Application in Distance Education. *PUI Conference*, September 2001.

[21] Crosby, Martha E., Stelovsky, Jan. (1989). The Influence of User Experience and Presentation Medium on Strategies of Viewing Algorithms, *IEEE*.

[22] Crosby, Martha E., Stelovsky, J. (1989) Subject Differences in the Reading of Computer Algorithms. In Designing and Using Human-Computer Interfaces and Knowledge Based Systems, G. Salvendy and M. Smith Eds., Elsevier Science, Amsterdam 137-144.

[23] Curtis, B. (1984). Fifteen years of psychology in software engineering: Individual differences and cognitive science. Proceedings of the 7th international conference on Software engineering, Orlando, Florida, United States, Pages: 97 - 106.

[24] Davies, Simon P. (1990). The Nature and Development of Programming Plans. *International Journal of Man-Machine Studies*, 32, 461-481.

[25] Davies, Simon P. (1993). Models and Theories of Programming Strategy. *International Journal of Man-Machine Studies*, 39, 237-267.

[26] Davis, J. S. (1984). Chunks: A Basis for Complexity Measurement. *Information Processing and Management*, 20(1), 119-127.
[27] Evans, G. E., Simkin, M. K. (1989). What best predicts computer proficiency? *Communications of the ACM*, Volume 32, Issue 11, Pages: 1322 - 1327.
[28] EventStream Software Framework on the Web: http://www.dataexplorer.net
[29] Eyetrack III: Consumer Behavior in the Age of Multimedia (Eye Tracking) http://www.poynterextra.org/eyetrack2004/
[30] Fix, Vikki, Wiedenbeck, Susan, Scholtz, Jean. (1993). Mental Representations of Programs by Novices and Experts. *Conference Proceedings on INTERCHI'93*, 74-79.
[31] Furnas, G. W. Generalized Fisheye Views. *Human Factors in Computing Systems CHI '86 Conference Proceedings*, 16-23.
[32] Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994). Design Patterns. *Addison-Wesley*, ISBN 0-201-63361-2.
[33] Gellenbeck, Edward M., Cook, Curtis R. (1991). *An Investigation of Procedure and Variable Names as Beacons during Program Comprehension*. Tech Report No. 91-60-2, Corvallis: Oregon State University, Computer Science Department.
[34] Gilmore, D. J., Green, T. R. G. (1988). Programming Plans and Programming Expertise. *The Quarterly Journal of Experimental Psychology*, 40A(3), 423-442.
[35] Green, T. R. G. (1997). Cognitive approaches to software comprehension: results, gaps and limitations. *Extended abstract of talk at workshop on Experimental Psychology in Software Comprehension Studies 97*, University of Limerick, Ireland.
[36] Green, T. R. G., Navarro, R. (1995). Programming Plans, Imagery, and Visual Programming. In Nordby, K., Helmersen, P. H., Gilmore, D. J., Arnesen, S. (Eds.) *INTERACT-95*. London: Chapman and Hall (pp. 139-144).
[37] Hendrix, T. Dean, Cross, James H. II, Maghsoodloo, Saeed. (2002). The Effectiveness of Control Structure Diagrams in Source Code Comprehension Activities. *IEEE Transactions on Software Engineering*, Vol. 28/5, pp. 463-477.
[38] Hoc, Jean-Michel. (1983). Analysis of Beginners' Problem-Solving Strategies in Programming. *Psychology of Computer Use*, ISBN 0-12-297420-4.
[39] Holt, Ric. (2002). Software Architecture as a Shared Mental Model. *Proceedings of 2002*.
[40] Hornof, A. J., Halverson, T. (2003). Cognitive strategies and eye movements for searching hierarchical computer displays. *CHI '03: Proceedings of the SIGCHI conference on Human factors in computing systems*, ISBN 1-58113-630-7, 249-256.
[41] Howard, Richard A., Carver, Curtis A., Lane, William D. (1996). Felder's Learning Styles, Bloom's Taxonomy, and the Kolb Learning Cycle: Tying it all together in the CS2 Course. *SIGCSE Bulletin*, vol. 28, no. 1, March 1996, pp. 227-231.
[42] Jacob, J. J. K., & Karn, K. S. (2003). Eye Tracking in Human-Computer Interaction and Usability Research: Ready to Deliver the Promises. In J. Hyona, R. Radach, & H. Deubel (Eds.), The Mind's Eyes: Cognitive and Applied Aspects of Eye Movements (pp. 573-605). Oxford: Elsevier Science.
[43] Kahnei, J. H. (1983). Problem Solving by Novice Programmers. In T. R. G. Green, S. J. Payne and G. C. van der Veer (Eds.), *The Psychology of Computer Use*, London: Academic Press.
[44] Karn, K. S., Ellis, S., and Juliano, C. (1999). The hunt for usability: tracking eye movements. *In CHI '99 Extended Abstracts on Human Factors in Computing Systems* (Pittsburgh, Pennsylvania, May 15 - 20, 1999). CHI '99. ACM Press, New York, NY, 173-173. DOI= http://doi.acm.org/10.1145/632716.632823
[45] Khazaei, B., Jackson, M. (2002). Is there any Difference in Novice Comprehension of a Small Program Written in the Event-Driven and Object-Oriented Styles? *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, 0-7695-1644-0/02.
[46] King, L. (2002). The Relationship between Scene and Eye Movements. *Proceedings of the 35th Hawaii International Conference on System Sciences*, 2002.
[47] Kintsch, W. (1977). *Memory and Cognition*. John Wiley.
[48] Kitchenham, B. A., Pfleeger, S. L., Pickard, L. M., Jones, P. W., Hoaglin, D. C., El-Emam, K., Rosenberg, J. (2001). Preliminary Guidelines for Empirical Research in Software Engineering. *IEEE Transactions on Software Engineering*, 28(8), 721-734
[49] Klemola, T. (2000). A cognitive model for complexity metrics. *4th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, Sophia Antipolis and Cannes, France, June 12-16, 2000.
[50] Knuth, D. (1984). Literate Programming. *The Computer Journal*, 27(2), 97-112.
[51] Ko, Andrew Jensen, Uttl, Bob. (2003). Individual Differences in Program Comprehension Strategies in Unfamiliar Programming Systems. *11th IEEE International Workshop on Program Comprehension (IWPC'03)*.
[52] Koenemann, Jürgen, Robertson, Scott P. (1991). Expert Problem Solving Strategies for Program Comprehension. *In Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems*, New Orleans, LA, April 27-May 2.
[53] Letovsky, S., Soloway, E. (1986). Delocalized Plans and Program Comprehension. *IEEE Software*, pages 41-49, May 1986.
[54] Littman, D. C., Pinto, Je., Letovsky, S., Soloway, E. (1986). Mental Models and Software Maintenance. *In Empirical Studies of Programmers*: Papers Presented at the First Workshop on Empirical Studies of Programmers, June 5-6, 1986, Washington, D.C., E. Soloway and S Iyengar, eds. Norwood, N.J.: Ablex,

1986, 80-98. Reprinted in J. Syst. and Software 7, 4 (Dec. 1987), 341-355.

[55] Logie, R. H., Marchetti, C. (1991). Visuo-Spatial Working Memory: Visual, Spatial or Central Executive? In R. H. Logie and M. Denis, Eds. *Mental Images in Human Cognition*, pp. 105-115. Amsterdam: Elsevier.

[56] Miara, J. R., J. A. Musselman, J. A. Navarro, and B. Shneiderman. Program Indentation and Comprehensibility. Comm. ACM 26, 11 (Nov. 1983), 861-867.

[57] Miller, G. A. (1956). The Magical Number Seven, Plus or Minus Two: Some Limits of our Capacity for Processing Information. *Psychological Review*, 63, 81-97.

[58] Navarro-Prieto, Raquel. (1999). Mental Representation and Imagery in Program Comprehension. *Psychology of Programming Interest Group*, 11th Annual Workshop.

[59] Pennington, N. (1987). Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*, 19, 295-341.

[60] Perry, Dewayne E., Porter, Adam A., Votta, Lawrence G. (2000). Empirical Studies of Software Engineering: A Roadmap. *ICSE - Future of SE Track*, 345-355.

[61] Petre, Marian, Blackwell, Alan F. (1999). Mental Imagery in Program Design and Visual Programming. *International Journal of Human-Computer Studies*, 51, 7-30.

[62] Pirolli, P., Card, S. K., Van Der Wege, M. M. (2001). Visual Information Foraging in a Focus + Context Visualization. *ACM Conference on Human Factors in Computing Systems, CHI Letters*, Vol. 3.1, 506-513.

[63] Pirolli, Peter, Card, Stuart K. (1999). Information Foraging. Psychological Review, 106, 643-675.

[64] Pirolli, Peter, Pitkow, James, Rao, Ramana. (1996). Silk from a Sow's Ear: Extracting Usable Structures from the Web. *Proc. ACM Conf. Human Factors in Computing Systems, CHI.*

[65] Rajlich, V., Doran, J., Gudla, R. T. S. (1994). Layered Explanations of Software: A Methodology for Program Comprehension. *3rd IEEE Workshop on Program Comprehension*, Nov. 1415, 1994, Washington, D.C.

[66] Rajlich, V., Wilde, N. (2002). The Role of Concepts in Program Comprehension. *IWPC 2002*, 271-278.

[67] Ramalingam, Vennila, Wiedenbeck, Susan. (1997). An Empirical Study of Novice Program Comprehension in the Imperative and Object-Oriented Styles. *7th Workshop on Empirical Studies of Programmers*.

[68] Reichle, E. D., Rayner, K., Pollatsek, A. (2003). The E-Z Reader Model of Eye-Movement Control in Reading: Comparisons to Other Models. *Behavior and Brain Sciences* (2003) 26, 445-526.

[69] Rist, R. S. (1986). Plans in Programming: Definition, Demonstration and Development. In E. Soloway and S. Iyengar (Eds.), *Empirical Studies of Programmers*. Norwood, NJ: Ablex.

[70] Romero, P., Cox, R., du Boulay, B., and Lutz, R. (2002) Visual attention and representation switching during java program debugging: A study using the restricted focus viewer. *In Diagrams 2002: Second International Conference on Theory and Application of Diagrams*, number 2317 in Lecture Notes in Artificial Intelligence, pages 221-235. Springer-Verlag.

[71] Shaft, Teresa M. (1995). Helping Programmers Understand Computer Programs: The Use of Metacognition. DATA BASE Advances, 26(4), 25-46.

[72] Shaft, Teresa M., Vessey, Iris. (1995). The Relevance of Application Domain Knowledge: The Case of Computer Program Comprehension. *Information Systems Research*, 6(3), 286-299.

[73] Shneiderman, B., Mayer, R. (1979). Syntactic Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *Intl. J. Comp. & Info. Sciences* 8, 3 (June 1979), 219-238.

[74] Smith, E. A., Winterhalder, B. (1992). Evolutionary Ecology and Human Behavior. *De Gruyter*, New York.

[75] Soloway, E., Ehrlich, K. (1984). Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, SE-10(5), 595-609, Sept. 1984.

[76] Soloway, E., Pinto, J., Letovsky, S., Littman, D., Lampert, R. (1988). Designing Documentation to compensate for Delocalized Plans. *Communications of the ACM*, 31(11), 1259-1267, 1988.

[77] Stephens, D. W., Krebs, J. R. (1986). Foraging Theory. *Princeton University Press*, Princeton, NJ.

[78] Storey, M.-A.D., Fracchia, F.D., Müller, H.A. (1999). Cognitive Design Elements to Support the Construction of a Mental Model during Software Exploration. *Journal of Software Systems*, 44:171-185.

[79] Storey, M.-A.D., Wong K., Müller, H.A. (1998). How Do Program Understanding Tools Affect How Programmers Understand Programs? *Science of Computer Programming*, 36(2), Mar. 2000, pp. 183-207.

[80] Storey, M.-A. D., Wong, K., Fracchia, F.D., Müller, H. A. (1997). On Integrating Visualization Techniques for Effective Software Exploration. Proceedings of IEEE Symposium on Information Visualization, 38-45.

[81] Tenny, Ted. (1988). Program Readability: Procedures Versus Comments. *IEEE Transactions on Software Engineering*, Vol. 14, No. 9.

[82] Vermeulen, A., Ambler, S. W., Bumgardner, G., Metz, E., Misfeldt, T., Shur, J., Thompson, P. The Elements of Java Style. *Cambridge University Press*, ISBN 0-521-77768-2.

[83] Vessey, I. (1987). On Matching Programmers' Chunks with Program Structures: An Empirical Investigation. *International Journal of Man-Machine Studies*, 27, 65-89.

[84] von Mayrhauser, A., Vans, A.M. (1994). Comprehension Processes During Large Scale Maintenance. *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, May 16-21, 1994.

[85] von Mayrhauser, A., Vans, A.M. (1995). Program Understanding: Models and Experiments. *In Advances in Computers*, Volume 40, M. C. Yovits and M. V.

Zelkowitz, Eds. Academic Press Limited, 1995, pp. 1-38.
[86] von Mayrhauser, A., Vans, A. M (1994). *Program Understanding: A Survey*. Technical Report CS-94-120, Colorado State University, August 1994.
[87] Wiedenbeck, S. (1986). Beacons in Computer Program Comprehension. *International Journal of Man-Machine Studies*, 25, 697-709.
[88] Winner, E., Casey, M. B. (1992). Cognitive Profiles of Artists. In G. C. Cupchik and J. Laszlo (eds.), *Emerging Visions of the Aesthetic Process: Psychology, Semiology and Philosophy*. Cambridge: Cambridge University Press.
[89] Woodfield, S. N., Dunsmore, H. E., Shen, V. Y. (1981). The Effect of Modularization and Comments on Program Comprehension. *Proceedings of the 5th international conference on Software engineering*, 215-223.
[90] Zelinsky. Gregory J., Murphy, Gregory L. (2000). Synchronizing visual and language processing: An effect of object name length on eye movements. *Psychological Science*, 11, 125-131.
[91] Zelinsky, G. J., Rao, R. P. N., Hayhoe, M. M., & Ballard, D. H. (1997). Eye movements reveal the spatiotemporal dynamics of visual search. *Psychological Science*, 8(6), 448-453.
[92] Zelinsky, Gregory J. (1996). Using Eye Saccades to Assess the Selectivity of Search Movements. *Vision Research*, 36(14), 2177-2187.