Towards Automated Coding of Program Comprehension Gaze Data

Michael Hansen
Indiana University
School of Informatics and Computing
2719 E. 10th Street
Bloomington, IN 47408 USA
mihansen@indiana.edu

Robert L. Goldstone
Indiana University
Dept. of Psychological and Brain Sciences
1101 E. 10th Street
Bloomington, IN 47405 USA
rgoldsto@indiana.edu

Andrew Lumsdaine
Indiana University
School of Informatics and Computing
2719 E. 10th Street
Bloomington, IN 47408 USA
lums@indiana.edu

ABSTRACT

Gaze data collected during program comprehension provides insight into programmers' thought processes. Manual coding of this data, however, can be tedious and subjective. We define and demonstrate an automated coding scheme for most categories in this workshop's coding scheme. We discuss potential sources of error when abstracting from fixations to areas of interest and patterns, and consider alternative definitions for some codes. For the high-level `Strategy` category, we inform coding decisions with metrics computed over a rolling time window.

Categories and Subject Descriptors

H.1.2 [**Information Systems**]: User/Machine Systems—*software psychology*

1. INTRODUCTION

Gaze data collected during program comprehension provides an insight into programmers' thought processes that is difficult to gain using common performance measures [1]. The process of interpreting and coding this gaze data, however, is tedious and highly subjective. To aid in the discovery of strategies for use in programming education, automated coding can be done with fixation data obtained directly from the eye-tracker. By building on the abstraction gained from lower-level automated coding – e.g., from fixations to blocks, lines, parameter lists, etc. – we demonstrate that codes from most categories in this workshop's coding scheme can be automatically and reasonably assigned.

Automated coding requires precise definitions of each category and code. At a low level, this means defining *areas of interest* (AOIs) based on syntax or semantics, and then deciding to which AOI (if any) each fixation belongs. Section 2 discusses the details of AOI creation and fixation assignment. These details must be explicit because the process of quantizing fixations introduces new potential sources of error. Section 3 defines all automatically-assigned codes in terms of AOI rectangles or lower-level codes. These definitions fit the authors' intuitions, but should not be taken as absolute or final. To aid in the manual assignment of `Strategy` codes, we make use of several fixation metrics computed over rolling time windows in each trial (Section 4).

2. QUANTIZING FIXATIONS

Fixations are quantized gaze positions over time. To abstract further, we draw rectangles around areas of interest (AOIs) and assign each fixation to zero or more AOIs. For simplicity, we assume the AOI rectangles in the `Block`, `SubBlock`, `Signature`, and `MethodCall` categories do not overlap. Codes in these categories, therefore, are mutually exclusive (not the case for `Pattern`).

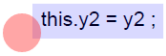

Figure 1: Example assignment of a fixation to an AOI. A circle is drawn around the fixation point, and the AOI with the largest overlap is assigned.

To determine whether or not a fixation belongs to an AOI, we do the following: (1) draw a circle around the fixation point with radius R, and (2) choose the AOI rectangle with the largest area of overlap (Figure 1). The choice of R depends on the size of the experiment screen and how far away the participant was sitting. Using $R = 20$ pixels, Figure 2 shows a timeline for subject 1's trial where each fixation has been quantized by line. Particular high-level patterns, such as `Scan` (highlighted), become readily apparent with such plots. Caution must be exercised, however, because noise at the lowest levels (raw gaze data) may result in a wrong AOI or code assignment.

3. CODING SCHEME DEFINITIONS

To facilitate automation of the coding process, we must precisely define each portion of the coding scheme. Even for very basic codes, such as `Body` from `SubBlock`, different reasonable definitions are possible. For example, should a fixation be coded as `Body` if it hits an opening curly brace (`{`)? For functions defined with K&R style braces, the opening brace is part of the signature line, and would likely not be considered part of the body:

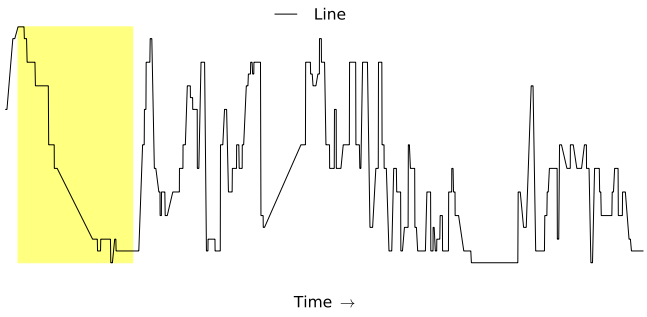

Figure 2: Timeline of line fixations for subject 1 (entire trial). The automatically identified `Pattern:Scan` portion is highlighted (2.034-18.642s).

```
public Rectangle(int x1, int y1, int x2, int y2) {
    // constructor body
}
```

With more compactly defined functions, such as `width()`, the separation between body and signature is not as clear:

```
public int width () { return this.x2 - this.x1 ; }
```

We suggest the following definitions for `SubBlock`. The opening brace is counted as part of the signature, whether or not the function is defined on a single line. To be consistent, the closing brace (}) is never considered part of the body. Figure 3 shows areas of interest overlaid on the `rectangle` program according to these definitions.

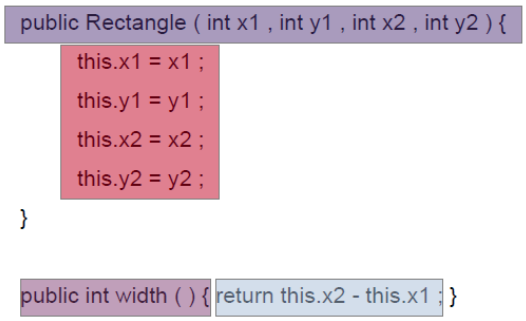

Figure 3: `SubBlock` areas of interest for constructor and `width` method. Signature and body are consistently separated.

3.1 Signature and MethodCall

Both `Signature` and `MethodCall` have `Name`, `Type`, and parameter list codes. For a signature like `main`'s:

```
public static void main (String[] args) {
    // ...
}
```

we consider `public static void` to be the type, `main` to be the name, and the arguments **plus** surrounding parentheses to be the formal parameter list. When coding method calls, however, we only consider `Name` and `ActualParameterList`. While the type and name of a method call are distinct linguistically (e.g., `System.out` and `println`), they are physically combined as a single "word" (`System.out.println`). Unlike signatures as well, the types and names of method calls are both in the same grammatical category (identifiers), as opposed to being in separate categories (keywords and identifiers). For these reasons, we do not separate type from name for `MethodCall` (Figure 4). Lastly, we do not code nested calls hierarchically (e.g., `foo(bar())`) because it would cause within-category overlap of the AOIs.

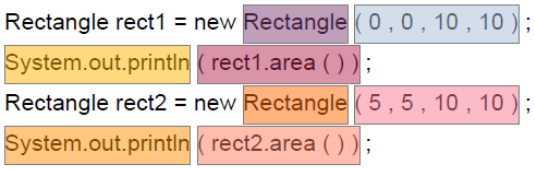

Figure 4: `MethodCall` areas of interest for `main` method. We do not distinguish between `Name` and `Type`.

3.2 Pattern

The most basic pattern, `Linear` is defined as the subject following at least 3 lines in text order. We follow this definition with one caveat: blank lines are not taken into account. For example, fixations on lines 1, 2, then 4 for the `rectangle` program are coded as `Linear` because line 3 is blank.

The `JumpControl` pattern, while seemingly simple, hides a great deal of complexity. Whether or not a transition between two lines follows execution order depends on where the subject is in evaluating the program! For example, a transition between line 11 (`width()` definition) and line 15 (`area()` definition) follows execution order only if the subject is currently evaluating the call to `this.width()` in the body of `area()`. For now, we code any line transition that *could* follow execution order as `JumpControl`. Future definitions of this code should take previous fixations into account in order to guess where the subject is in the call stack.

`LineScan` is defined in English as the subject reading the whole line in "rather equally distributed time." For simplicity, we operationalize this definition by splitting each line into a set of equally-sized rectangles (Figure 5). A `LineScan` is coded for any set of consecutive fixations that hit at least 3 distinct rectangles on a single line. While this does not explicitly address the "equally distributed time" portion of the English definition, it assigns codes that match the authors' intuitions for the sample data. Another option would be to use the rolling metrics discussed in Section 4 – e.g., fixation spatial density and duration.

Building on `LineScan`, we can simply define `Signatures` as a line scan of a signature line (`SubBlock:Signature`) immediately followed by a fixation inside the corresponding function/constructor body (`SubBlock:Body`). With this definition, we identify two instances of the pattern in subject 2's trial (`width` starting at 7 seconds and the constructor starting around 26 seconds).

The `Scan` pattern, inspired by results from Uwano et al. [4], can be operationalized using two sets of constraints. A `Scan`

starts the first time a fixation moves down the screen relative to the previous fixation, and stops when one of two conditions is met: either (1) more than 3 fixations move up the screen, or (2) more than 1.5 seconds are spent on the same line. The highlighted portion of Figure 2 has been identified using this definition, and matches well with the authors' intuitions.

Figure 5: A single line split into equally-sized rectangles. We code a `LineScan` if 3 or more distinct rectangles are fixated consecutively.

4. STRATEGIES & ROLLING METRICS

Codes from the categories described above can be assigned based (mostly) on observation. The `Strategy` category of codes, however, requires more interpretation. To aid in the identification and interpretation of strategies, we compute three fixation metrics over the course of each trial using a rolling window. Windows are 4 seconds in size and are shifted by 1 second during each step. On average, a single time window will contain about a dozen fixations.

Our first two metrics are simply fixation count and mean fixation duration [3]. Respectively, they are the total number of fixations in a time window and the mean duration of those fixations. Our third metric, fixation spatial density [2], is computed as follows: (1) divide the screen into a grid, and (2) calculate the proportion of cells in the grid which contain at least one fixation. We divide the portion of the screen containing code vertically into 10 equally-sized rectangles. A spatial density of 1, therefore, means that all 10 rectangles were fixated at least once in a time window.

Figure 6 shows our three rolling metrics computed for subject 1's trial (time windows with no fixations were dropped). Troughs in spatial density (solid blue line) correspond to windows in which subject 1 was concentrating on one or two lines. In some cases, this was correlated with an increase in fixation count (dashed green line), which may be useful for distinguishing between the `Debugging` and `TestHypothesis` strategies. The sharp increase in mean fixation duration just after the 70 second mark (dashed-dotted red line) corresponds with the subject focusing on the final line of the program:

```
System.out.println(rect2.area ());
```

The subject's task in this trial is to obtain the value of `rect2.area()`. Given the increased fixation duration and drop in both fixation count and spatial density at this point (at approximately 65-75 seconds), we hypothesize that the subject is performing the necessary mental calculation to compute the area of `rect2`. There are several off-screen fixations at 70-75 seconds in the video, supporting this hypothesis. While we may not be able to pinpoint shifts in strategy using this kind of visualization, we can quickly identify interesting time windows to investigate further.

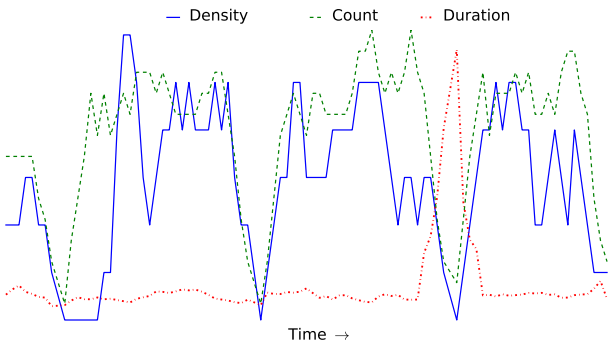

Figure 6: Rolling fixation metrics for subject 1 (entire trial) with a window size of 4 seconds and a step size of 1 second.

5. CONCLUSION & FUTURE WORK

We have defined and demonstrated an automated process for coding non-`Strategy` categories from the workshop's coding scheme. In most cases, this process assigns codes that match well with the authors' intuitions. In the context of programming education, automated coding helps researchers quantify differences between experienced and novice programmers. Such differences could inform the design of an automated tutor capable of providing highly-contextualized feedback to a student. For example, alternative strategies could be presented to students who fail to locate a bug in an exercise.

Automated coding also forces the coder to think precisely about areas of interest and how to define high-level codes, increasing confidence in subsequent analyses. Because the process is automated, it can be run with different, competing code definitions. Multiple quantitative cognitive models could also be used to inform coding (e.g., `JumpControl`), with deviations from expectations helping to refine the models.

For future work, we would like to achieve automated coding of the `Strategy` category in a way that agrees with human coders. This may not be possible without more precise definitions of `Debugging`, `DesignAtOnce`, etc. Previous psychology of programming research, combined with focused eye-tracking studies where only one strategy is likely to be used, will be crucial to achieving this goal.

6. ACKNOWLEDGMENTS

We would like to thank the workshop organizers for their efforts in constructing the coding scheme and providing the gaze data. All software will be made available online after the workshop. Grant R305A1100060 from the Institute of Education Sciences Department of Education and grant 0910218 from the National Science Foundation REESE supported this research.

7. REFERENCES

[1] R. Bednarik, N. Myller, E. Sutinen, and M. Tukiainen. Program visualization: Comparing eye-tracking patterns with comprehension summaries and performance. In *Proceedings of the 18th Annual Psychology of Programming Workshop*, pages 66–82, 2006.

[2] L. Cowen, L. J. Ball, and J. Delin. An eye movement analysis of web page usability. In *People and Computers XVI-Memorable Yet Invisible*, pages 317–335. Springer, 2002.

[3] A. Poole and L. J. Ball. Eye tracking in human-computer interaction and usability research: Current status and future. In *Prospects, Chapter in C. Ghaoui (Ed.): Encyclopedia of Human-Computer Interaction. Pennsylvania: Idea Group, Inc*, 2005.

[4] H. Uwano, M. Nakamura, A. Monden, and K.-i. Matsumoto. Analyzing individual performance of source code review using reviewers' eye movement. In *Proceedings of the 2006 symposium on Eye tracking research & applications*, pages 133–140. ACM, 2006.