An Empirical Evaluation of Models of Programmer Navigation

Alka Singh, Austin Z. Henley, Scott D. Fleming, Maria V. Luong
Department of Computer Science
University of Memphis
Memphis, Tennessee 38152
Email: {arsingh, azhenley, Scott.Fleming, mluong}@memphis.edu

Abstract—**In this paper, we report an evaluation study of predictive models of programmer navigation. In particular, we compared two operationalizations of navigation from the literature (click-based versus view-based) to see which more accurately records a developer's navigation behaviors. Moreover, we also compared the predictive accuracy of seven models of programmer navigation from the literature, including ones based on navigation history and code-structural relationships. To address our research goals, we performed a controlled laboratory study of the navigation behavior of 10 participants engaged in software evolution tasks. The study was a partial replication of a previous comprehensive evaluation of predictive models by Piorkowski et al., and also served to test the generalizability of their results. Key findings of the study included that the click-based navigations agreed closely with those reported by human observers, whereas view-based navigations diverged significantly. Furthermore, our data showed that the predictive model based on recency was significantly more accurate than the other models, suggesting the strong potential for tools that leverage recency-type models. Finally, our model-accuracy results had a strong correlation with the Piorkowski results; however, our results differed in several noteworthy ways, potentially caused by differences in task type and code familiarity.**

I. Introduction

A key cost during software evolution tasks is in the navigation of source code. Developers spend considerable time navigating code dependencies to understand how and where features are implemented. For example, one study of developers engaged in debugging found that the developers spent roughly half of their time foraging for information in code—a navigation heavy activity [23]. Moreover, two more studies of developers engaged in software maintenance tasks found that the developers navigated over 5 times per minute [4] and spent 35% of their time on the mechanics of navigation alone [12].

Many promising tools have been proposed to make navigation more efficient, and these tools are often implicitly based on predictive models of programmer navigation. That is, each tool leverages some underlying model to predict where the programmer wants to go, and based on the prediction, tries to help the programmer get there more efficiently. For example, some tools use models that make predictions based on past navigation behavior (e.g., [2], [9]) or past code modifications (e.g., [31], [32]). On the other hand, other tools use models based on other factors, such as structural relationships in code (e.g., [14]) and lexical properties of code components (e.g., [29]). Although preliminary evidence has shown that these tools help developers in efficient navigation, most of the tools have not been validated for a wide variety of development contexts and have not received widespread adoption in practice.

The success of these tools is closely tied to the predictive accuracy of their associated models; however, researchers have only recently begun to systematically compare the accuracy of different models. One early study by Parnin and Görg evaluated a set of four predictive models, all based on page-caching algorithms [20]. These models focused exclusively on the access history of code components (e.g., how recently or frequently accessed) to make their predictions. A subsequent model-evaluation study by Piorkowski et al. expanded the battery of models to include ones based on code structure and lexical properties [22]. In both the Parnin and Piorkowski studies, recency-based models, which favor more recently visited code components in their predictions, stood out for their high predictive accuracy.

Although these studies have begun to shed light on the relative effectiveness of various models, open questions remain. One open question is how best to operationalize navigation. That is, how should a system decide when a developer has navigated from one code component (e.g., method or subroutine) to another? The most widely used operationalization of navigation in tool research has been the *click-based* operationalization in which the position of the text caret in a code editor serves as a proxy for where a developer has placed his/her attention. However, other operationalizations of navigation are also possible. For example, Piorkowski et al. introduced a different operationalization of navigation called *view-based* that uses the line of code in middle of the code editor to approximate where a developer's attention is [22]. Although these operationalizations aim to measure the same phenomenon, they yielded substantial differences in the Piorkowski model-accuracy results, leaving the question as to which operationalization more accurately captures developer navigation.

Another open question is the extent to which these prior study results generalize. In particular, the most recent and comprehensive study to date, reported by Piorkowski et al. [21], had a very limited sample (one participant) and that participant worked only on two tasks that involved debugging unfamiliar code. However, numerous prior studies have confirmed order-of-magnitude individual differences in the productivity of developers [17]. Moreover, a developer's familiarity with the source code under development can have a considerable effect on his/her information needs during tasks, and thus, where he/she chooses to navigate in the code. Therefore, there is a question as to whether the Piorkowski results will generalize to other developers in other development contexts and tasks.

To answer these open questions, we conducted an empirical study of developers' navigation behavior during software evolution tasks. In particular, our study seeks to address the following research questions:

RQ1: How accurately do the prior operationalizations of navigation (click-based and view-based) approximate developers' actual navigations?

RQ2: (a) Which predictive models of programmer navigation most accurately predict where a developer will navigate next, and (b) to what extent do Piorkowski et al.'s prior evaluation results of such models generalize to multiple developers performing varied software evolution tasks on familiar software?

To address RQ1, we applied the prior click- and view-based operationalizations of navigation (detailed in Section II-A1) to our participant data, and compared the navigations those operationalizations recorded with the navigations discerned by human observers. We chose this comparison, because a human observer is able to take more information into account in deciding where the programmer's attention is than the more primitive view-based or click-based operationalizations. Although we do not necessarily expect a human observer to discern with perfect accuracy where a developer's attention is, an observer can see both what the models "see" as well as additional information such as developer gestures and utterances. Thus, it stands to reason that a human observer could provide a reasonable measure of accuracy upon which to compare the models.

To address RQ2, we designed our study as a partial replication of the Piorkowski study. Like the Piorkowski study, ours observed individual developers engaged in development tasks in a controlled laboratory environment, and evaluated the same battery of predictive models (minus one that required bug-report text, which was not present in our study). In contrast to the Piorkowski study's small sample of one developer, our study involved 10 participants. Instead of working on an unfamiliar code base, our participants worked on their own software projects with which they were already familiar. Instead of working on two researcher-prescribed debugging tasks, our participants worked on a wide variety of evolutionary tasks—chosen by the participants based on the particular needs of their projects at the time of the study.

The remainder of the paper is organized as follows. Section II provides background on the prior operationalizations of navigation and predictive models. Section III details our study method. Section IV reports the results of our study. Section V discusses implications of our findings with respect to related work as well as limitations of our work. Section VI concludes with a summary of our findings and the future research directions they point to.

II. Background

The two main research questions of this work center around two concepts from the literature: (1) operationalizations of navigation and (2) predictive models of navigation. The operationalizations of navigation provide different approximations for recording where a programmer places his/her attention (often trading off on certain strengths/weaknesses, such as automatability versus ability to detect certain types of navigation). Given a sequence of recorded navigations, the predictive models of navigation that serve as our focus aim to forecast where the programmer will navigate to next.

A. Operationalizations of Navigation

In addressing our research questions, we applied two operationalizations of programmer navigation from the literature [22]: the *click-based* operationalization and the *view-based* operationalization. These operationalizations each provide a method for approximating the sequence of code locations (Java methods in our case), where a programmer puts his/her attention.

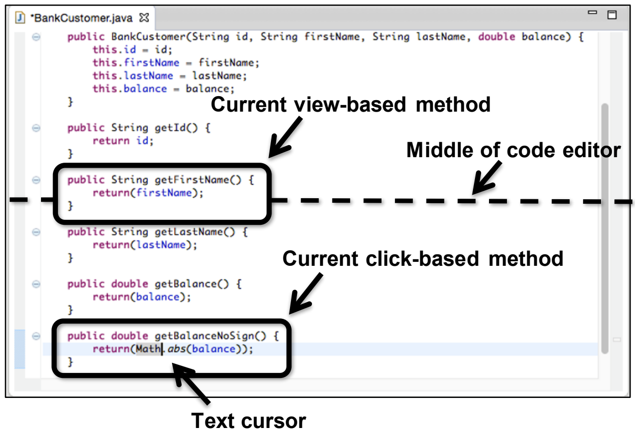
1) Click-Based Operationalization: A click-based navigation is operationalized as a change in the code editor's text-cursor position from one Java method to another. When a programmer clicks in a method other than the current method where the text cursor is, the action is counted as a click-based navigation. That is, if the current position of a developer's cursor is in some method *A*, and then the developer clicks in another method *B*, it is counted as a click-based navigation. However, if the current position of the developer's cursor is in method *A*, and the developer clicks somewhere else inside the body of the method *A*, it is not counted as a click-based navigation. In particular, the following actions were counted as click-based navigations:

1) clicking into a method other than the current one,
2) clicking on a tab to make the contents of another file visible, and
3) opening a file, for example, by clicking on it in the package explorer.

In the case of a newly opened file, the first method in the opened file is considered to be the next navigation.

For example, in Fig. 1, the current method under the click-based operationalization is `getBalanceNoSign` in which the text cursor is currently present. If the programmer clicks somewhere else inside the `getBalanceNoSign` method, it will not be counted as a click-based navigation. However, if programmer clicks in any other method, such as `getLastName`, the click-based operationalization records a navigation to that method.

Although the click-based operationalization is automatable and has been widely applied (e.g., in [20], [21], [22]), it has a couple disadvantages. It will fail to record navigations if a developer scrolls through a file, but does not actually



```
public BankCustomer(String id, String firstName, String lastName, double balance) {
    this.id = id;
    this.firstName = firstName;
    this.lastName = lastName;
    this.balance = balance;
}

public String getId() {
    return id;
}

public String getFirstName() {
    return(firstName);
}

public String getLastName() {
    return(lastName);
}

public double getBalance() {
    return(balance);
}

public double getBalanceNoSign() {
    return(Math.abs(balance));
}
```

Fig. 1. Example of navigations under the click-based versus view-based operationalizations. The editor depicted is the Eclipse Java editor (the same type used by our study participants).

click in any of the methods that scroll by. Also, just because a developer clicked in a method does not necessarily mean that developer has his/her attention on that method. In a recent study, these limitations contributed to significant differences in the navigations recorded by the Mylyn Eclipse plugin, which uses a click-based approach, and the navigations recorded by iTrace, a research-prototype system based on eye-tracking [10].

2) View-Based Operationalization: Unlike the click-based operationalization of navigation, the view-based operationalization does not take the position of the text cursor into account to operationalize a navigation; instead, it defines the current method as the one in the middle of the text-editor pane. For example, in Fig. 1, the current method (view-based) is `getFirstName`, which is present in the middle of the text-editor pane. Furthermore, there were two additional rules in the recording of view-based navigations. First, if in switching or opening editor tabs, a method A's definition becomes completely visible in the text editor and it is present above the middle of the screen while method B is in the middle of the screen, then navigations are recorded in the order of first a navigation to method A and then to method B. Second, if a programmer scrolls through a file, navigations to each method are recorded in the sequence they come to the middle of the screen.

The chief disadvantage of the view-based operationalization of navigation is that, if the programmer quickly scrolls through a file, it records all the methods that pass through the middle of the screen as navigations—even if the programmer did not actually place his/her attention to those methods. Another drawback is that the view-based operationalization has not been previously automated, and hence, recording view-based navigations is currently a labor-intensive manual process.

B. Predictive Models of Programmer Navigation

To address the question of which predictive models of programmer navigation are most accurate (RQ2), we applied the same battery of predictive models of navigation used by Piorkowski et al. [22]. These models can be grouped into three major categories: working set approximation models, structural similarity models, and lexical similarity models. In this work, we have evaluated all the models from the prior work, except for the lexical similarity model, *Bug Report Similarity*. That model assumes that the developer is working on a debugging task with a bug report, which was not the case in our study.

Following from Piorkowski et al. [22], each of the models takes as input a sequence of method-to-method navigations from a developer's programming session. Such a navigation history H is a sequence of method-to-method navigations $(m_1, m_2, \ldots, m_n)$ such that for every method m_i in H, $m_i \neq m_{i+1}$. If the navigation history H_j for a programming session is given up to a particular point where $H_j = (m_1, m_2, \ldots, m_j)$, each model tries to predict the method m_{j+1} to be visited next. At any given point, the developer may have opened multiple source files. The set of methods known to the developer at a particular point in time is M_j. M_j contains all the methods defined and referenced in the previously opened files irrespective of whether the file is currently open or closed. It holds that the navigation history H_j must contain a subset of the methods from M_j, because the developer must open a file before navigating to one of its methods.

In order for a model to predict the next method m_{j+1}, the model ranks the methods from the set of known methods $M_j - \{m_j\}$ from least likely to most likely. For a rank of a method to be calculated, the model creates an activation mapping A_j from each method in $M_j - \{m_j\}$ to an activation value. According to a model's predictions, the methods with higher activation values are more likely to be visited next than ones with lower values. The activation function is used to create a ranking function R_j such that R_j is actually just A_j with rank-transformed activation values. The higher the activation value of the method, the higher the rank number the method will get (with the rank of 1 being the "highest" rank). For example, a method with rank 1 is more likely to be visited next by a developer than a method with rank 3 according to a model. Also, in the event of tied activation values, if there are n methods with the same activation value, their ranks are computed by averaging over n. Every model uses a different approach for calculating the activation value for methods.

Some models incorporate the notion of a structural relatedness between methods such that higher activation is assigned to methods that are more closely related to the current method than to less closely related methods. Such models maintain a graph G_j such that every method in M_j corresponds to a different vertex in G_j. Given a particular model, for all m in $M_j - \{m_j\}$, $A_j(m)$ for that model is $|M_j|$ minus the length of the shortest path from m to the current method m_j.

1) Recency and Working Set Models: The *Recency* model ranks more-recently visited methods higher than less-recently visited methods. Formally, for every method m in the set of known methods M_j, if the developer already visited m, the activation function A_j will assign an activation value to the method m such that $A_j(m) =$ the max sequence number for m

in the programmer's navigation history H_j; otherwise, if the method m was not visited previously, the activation $A_j(m) = 0$.

The *Working Set* model is similar to the Recency model, but the difference is that only a fixed number of recently visited methods are ranked, while all other methods are ranked zero. Formally, Working Set assumes a window-size Δ. If a method m is among last Δ visited methods in the navigation history H_j, the activation $A_j(m) = 1$; otherwise $A_j(m) = 0$.

2) Frequency Model: The *Frequency* model assigns higher ranks to methods visited more frequently than those visited less frequently. Formally, for every method m in the list of known methods M_j, the activation $A_j(m)$ = the number of occurrences of m in the developer's navigation history H_j.

3) Within-File Distance Model: The *Within-File Distance* model assigns higher ranks to methods textually closer to the current method in the file. The ranking function of this model is based on an adjacency factor. That is, this model assumes that the methods closer to the current method are more likely to be visited next. It creates a graph G_j such that there are links between method nodes, which are textually adjacent in a file. Formally, for every method m in M_j, there is an undirected edge from m to the methods adjacent to m in the file in which m is defined. The adjacent methods may come before or after method m in the file.

4) Forward Call Depth and Undirected Call Depth Models: The *Forward Call Depth* model ranks methods based on a call graph with unidirectional links. That is, the methods being called from the current method are ranked higher than the other methods. Similar to the Within-File Distance model, the Forward Call Depth model also creates a graph G_j; however, in this case, a directed edge connects each method m to every method called from within the definition of m. Formally, in the constructed graph G_j, there is a directed edge from method m_a to m_b, if and only if the definition of method m_a contains a call to method m_b.

The *Undirected Call Depth* model is similar to the Forward Call Depth model, with the only difference being that the methods on both ends of a call are considered in the ranking. That is, this model ranks both the methods which are being called from the current method as well as the methods which call the current method. Formally, in the constructed graph G_j, there is an edge between method m_a and m_b, if method m_a contains a call to method m_b or vice versa.

5) Source Topology Model: The *Source Topology* model ranks methods higher which share one or more of several structural relationships with the method where the developer's attention currently lies. The Source Topology model constructs a source topology graph which is similar to the graphs constructed by other models, except that in addition to method nodes, this graph contains nodes for classes, interfaces, variables, packages, and projects. If there is a *calls-a*, *has-a* or *within-file adjacency* relationship among these nodes, then there is an edge between these elements. Formally, for every element v in $M_j \bigcup C_j \bigcup V_j \bigcup P_j$ where C_j is the set of classes or interfaces referenced in the files the programmer has opened so far, V_j is a set of variables, and P_j is the set of packages, there is a vertex that maps to v. The source topology graph has an edge between elements v_a and v_b if v_a calls v_b, if v_a has v_b, or if v_a and v_a are adjacent methods in a file.

III. Study Method

To address our research questions, we conducted a laboratory study of developers engaged in software evolution tasks. To evaluate the accuracy of prior operationalizations of navigation (RQ1), we applied the prior operationalizations (i.e., click-based and view-based, detailed in Section II-A) to our participant data, and compared the navigations those operationalizations recorded with the navigations discerned by human observers. To address the question of model accuracy (RQ2), we used our navigation data as input to the battery of predictive models from the Piorkowski study [22] (detailed in Section II-B) and compared how accurately the models predicted the code locations to which our participants actually navigated during their tasks.

A. Participants

Our study participants consisted of 10 graduate students (8 male, 2 female) enrolled in a graduate-level software engineering course at the University of Memphis. They had an average of 6.75 years of programming experience (*SD* = 2.51). Seven of the ten participants also had experience programming professionally (*M* = 2.93 years, *SD* = 1.54).

B. Tasks and Environment

Each participant took part in an individual study session in which the participant worked on a team software project from the software engineering course. Each project team had 4–5 members, and worked collaboratively to develop the project. Although our study sessions involved only one participant per session, the participant was free to communicate with teammates via phone or internet. The software projects under development were Java EE based web applications to help university students and advisors track student progress through the degree program. The applications included features to manage course and grade information, and to monitor and visualize students' fulfillment of degree requirements. On average, the project code bases consisted of 9344 lines of Java code spread across 84 code files.

During the study sessions, the participants were free to work on the project tasks of their choice—mainly depending on the needs of their particular projects at the time of the session. The tasks on which participants worked could be loosely classified as adding new features to the system or fixing bugs in the source code; however, the specific features and bugs varied widely. Most of the participants worked on four or more distinct tasks during their sessions.

The programming environment used by participants consisted of a Windows PC with the Eclipse integrated development environment (IDE) and other software for web application development (e.g., web browsers, a version control system, and a database management system). The computer was also outfitted with several technologies for recording the participant sessions: screen-capture software, an HD webcam, and a headset with microphone.

C. Procedure

Each study session lasted roughly 135 minutes. For the first 15 minutes of the session, the participant completed a background questionnaire and took part in a short study-training exercise. For the remaining 120 minutes, the participant worked on his/her development tasks.

To better understand where participants placed their attention, we employed the *think-aloud method* [28]. The think-aloud method is a well-validated empirical method by which participants externalize their inner dialogue by continuously uttering their thoughts as they perform a task. It has been widely used in psychology and human–computer interaction to gain insight into a person's goals and intentions. Following the method, we asked our participants to "think aloud" while working on their tasks. If a participant fell silent for three minutes, an attending researcher asked the participant to "please keep talking." Using this method has been shown to be minimally disruptive to cognition during mentally intensive tasks, such as programming [3].

D. Qualitative Analysis Method

To understand what navigations a human would discern from observing a developer at work, we applied a qualitative coding method [25]. In particular, our human analysts applied a set of coding rules to identify when a programmer navigated to a Java method (i.e., subroutine). The analysts coded navigations to a method based on criteria, such as the programmer talking about the method, the programmer creating/editing the method, and the programmer copying/selecting/highlighting the contents of the method. In a few special situations, the analysts ignored certain programmer actions that might have otherwise been coded as navigations. These cases generally involved the programmer bringing code into view, but then not giving any indication of actually looking at the code. Some common scenarios included times when the programmer switched to an editor tab only to execute the code file using an IDE feature, times when he/she clicked on a tab clearly by mistake, and times when he/she closed a tab causing a random tab to gain focus.

To ensure our qualitative codes could be reproduced reliably by other researchers, we used a standard inter-rater reliability method [25]. Following the method, two researchers analyze 20% of the total data independently and then check their level of agreement. If their results agree 80% or more using the Jaccard index, they can divide the remaining data and analyze it separately. In our study, two researchers independently analyzed the same 4 hours of video data (20% of our 20 hours of video), and they achieved 86% agreement. Because this exceeded the 80% threshold, they divided up and separately coded the remaining 16 hours of video.

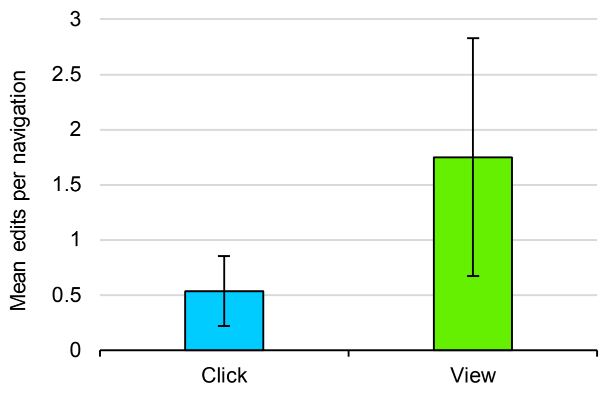


Fig. 2. On average, the navigation sequences produced by the click-based operationalization (blue bar) had significantly lower edit distances from the human-reported navigation sequences than did the sequences produced by the view-based operationalization (green bar). Shorter bars indicate fewer differences, with zero indicating no difference. Whiskers denote standard error.

IV. Results

A. RQ1 Results: Operationalizations of Navigation

To understand how accurately prior operationalizations of navigation (click-based and view-based) measure developers' navigations (RQ1), we applied the operationalizations to our study data, and we compared the resulting navigation sequences with the results of our qualitative analysis performed by human observers. As our main metric of comparison, we compared the *edit distance* between the human-reported navigation sequence and each of the click-based and view-based navigation sequences. Edit distance is a standard method to quantify the similarity between two strings or graphs by calculating the minimum number of edits (i.e., insertions and deletions) needed to transform one graph into the other [30]. In our edit-distance calculation, each navigation sequence represented a linear graph with the nodes mapping to the navigation destinations (i.e., Java methods). Using this approach, we were able to compare how closely the prior operationalizations' sequences of methods matched the sequence perceived by our human analysts.

As Fig. 2 shows, the click-based navigation sequences were considerably closer to the human-reported navigation sequences than were the view-based ones. In fact, the average edit distance for the view-based navigations was over three times the average distance of the click-based ones. Indeed, a paired *t*-test detected a significant difference between the click-based and view-based edit distances ($t(9) = 4.32$, $p = 0.002$). (A Shapiro-Wilk test for normality indicated that the *t*-test was appropriate in this case.)

To understand how the discrepancies reflected by the edit distances affected the other qualities of the navigation sequences, we analyzed several higher-level characteristics of navigations. In particular, we analyzed for each recorded navigation sequence the rate of navigation (navigations per minute; Fig. 3), the number of different places visited (different methods per hour; Fig. 4), and proportion of revisits (percentage of navigations that revisited previously visited places; Fig. 5).

As Fig. 3 shows, both operationalizations of navigation tended to record more navigations than the humans reported; however, the difference for view-based was considerably more than for click-based. In fact, on average, the view-based operationalization produced more than twice the number of navigations reported by the human observers. Not surprisingly, there was a statistically significant difference in rate of navigation between view-based and the human observer (Wilcoxon signed-rank test: $W = 54$, $p < 0.01$). The difference in navigation counts recorded by the click-based operationalization versus by the human observers also rose to the level of statistical significance (Wilcoxon signed-rank test: $W = 48$, $p < 0.05$); however, the magnitude of the difference was comparatively small (only 15% larger).

As Fig. 4 shows, the number of different methods that the operationalizations recorded participants navigating to followed similar trends as total counts of navigations. Again, both operationalizations recorded more different methods navigated to than the human analysts reported, with the view-based operationalization producing significantly more (over 1.5 times as many on average; Wilcoxon signed-rank test: $W = 55$, $p < 0.01$). However, in this case, the difference in counts reported by the click-based operationalization and those reported by the human analysts did not rise to the level of statistical significance.

Interestingly, as Fig. 5 shows, the discrepancies in the navigation sequences recorded by the operationalizations and human observers had little effect on the proportion of navigations that participants made to methods already visited. In particular, both operationalizations agreed with the human observers that a substantial proportion (over 80%) of participant navigations were revisits. Moreover, no statistically significant difference could be detected between the percentages recorded by the operationalizations of navigation and those reported by the human analysts.

B. RQ2 Results: Model Accuracy

To evaluate our battery of predictive models of navigation (Section II-B) and test the generalizability of Piorkowski et al.'s findings [22] (RQ2), we ran each of the models on our navigation data, and compared the accuracy of their predictions both with respect to each other as well as to the Piorkowski results. We report only predictive-model results using our human-reported and click-based navigation data, because our results for RQ1 showed that the view-based operationalization diverged considerably from the observations of human analysts.

Following the method from the Piorkowski study, we used a *hit-rate* metric to compare the accuracy of the various predictive models. The predictive accuracy of a model is determined by the model's ability, at a given moment in time, to predict a programmer's next navigation. A model is said to get a *hit* if the programmer's actual navigation is among the model's top-W predictions, where W is a variable window

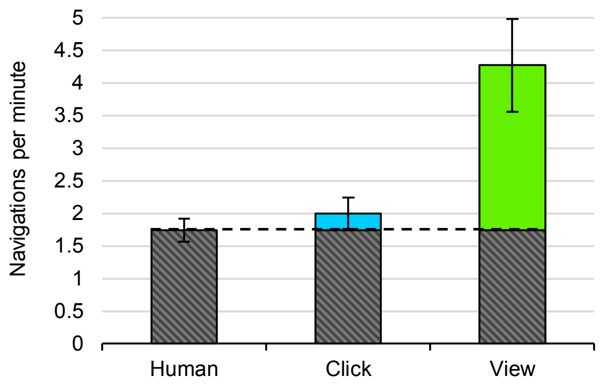


Fig. 3. On average, the view-based operationalization (green bar) recorded significantly more navigations than the human observers, whereas the difference for click-based (blue bar), although also statistically significant, was substantially less. Inner bars denote the overlap in count with the human-reported navigations. Whiskers denote standard error.

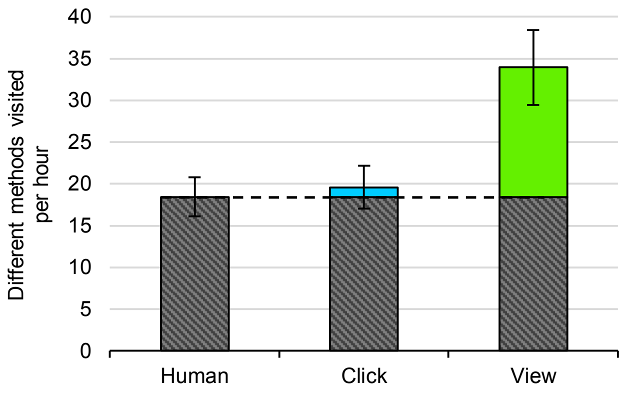


Fig. 4. On average, the view-based operationalization (green bar) recorded significantly more different methods visited than did the human observers, whereas no significant difference was found between the click-based navigations (blue bar) and human-reported ones. Inner bars denote the overlap in count with the human-reported navigations. Whiskers denote standard error.

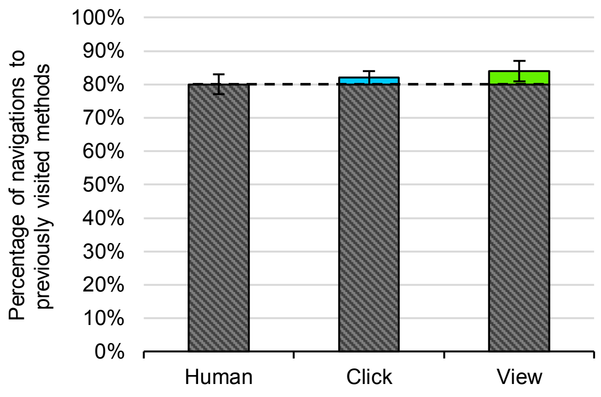


Fig. 5. On average, the percentage of navigations to previously visited methods recorded by both the click-based (blue) and view-based (green) operationalizations were similar to the percentage reported by human observers (no significant differences). Inner bars denote the overlap in percentage with the human-reported navigations. Whiskers denote standard error.

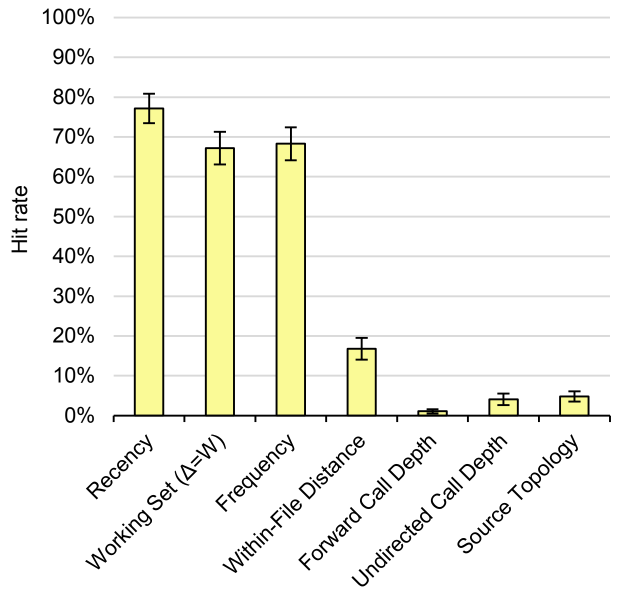


Fig. 6. Predictive accuracy of each model for our human-reported navigation data. Following from [22], the hit-rate computation used a window size W = 10. Whiskers denote standard error.

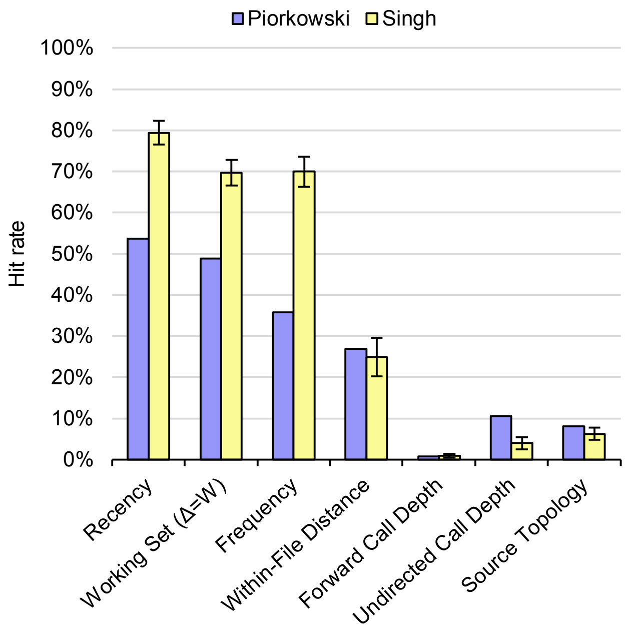


Fig. 7. Predictive accuracy of each model for our click-based data (yellow bars) juxtaposed with the predictive accuracy the click-based data of Piorkowski et al. [22] (purple bars). Following from [22], the hit-rate computation used a window size W = 10. Whiskers denote standard error. The Piorkowski bars lack whiskers because their sample size was 1 (variance undefined).

size. For example, consider a model that predicts five methods, m_1, m_2, m_3, m_4 and m_5, as its top-five predictions, where m_1 is the top most prediction with rank 1 and m_5 is ranked 5 as its lowest prediction. If the next method that the programmer navigates to is m_4, then the model would get a hit for window size $W \geq 4$, but a miss for $W \leq 3$. An additional complication that the metric addresses is that some of the models' rankings were only partially ordered. To address the issue of ties, we refined the definition of a hit as follows: if R is the rank of the navigated-to method and T is the number of ties at that rank, then a hit is recorded if $\lfloor T/2 \rfloor + \lfloor R \rfloor < W$. Following the Piorkowski study, we used window-size W = 10 for purposes of our comparisons.

As the bars in Fig. 6 show, the predictive model, Recency, achieved the greatest accuracy (hit rate) on our human-reported navigation data. Its mean hit rate of 77% was 9–10 percentage points above the next greatest models, Frequency and Working Set, and towered over the other models, the closest of which, Within-File Distance, was 60 percentage points behind. A Kruskal–Wallis test confirmed this difference, detecting a statistically significant difference between the model hit rates ($\chi^2(6) = 58.96$, $p < 0.0001$).

Similar to our human-reported data, Recency also achieved the greatest accuracy on our click-based data. As the yellow bars in Fig. 7 show, the predictive accuracies of the models on our click-based navigation data were almost identical to those on our human-reported navigation data (reported in Fig. 6). A Spearman's rank correlation coefficient calculation confirmed this correlation between the click-based and human-reported model results, reporting a "very strong" correlation ($r_s(5)$ = 1.00, p = 0.0004).

Comparing our model hit rates with those of the Piorkowski study (purple bars in Fig. 7), a strong correlation between the Piorkowski results and ours is apparent. Indeed, the rankings of the models by performance are almost identical between the Piorkowski hit rates and ours, with only Working Set and Frequency being swapped. A Spearman's rank correlation coefficient calculation further confirmed this correlation, reporting a "very strong" correlation ($r_s(5)$ = 0.93, p = 0.007).

However, a key difference between the Piorkowski results and ours was also apparent for the three most accurate models. For Recency, Working Set, and Frequency, the hit rates for our navigation data were noticeably greater than those for the Piorkowski data. In fact, our hit rates for those models were greater than the Piorkowski hit rates by 21–34 percentage points. (No statistical test could be performed to confirm these differences, because the Piorkowski sample size of 1 made the data inappropriate for such tests.)

To shed light on the reason for these differences between the Piorkowski results and ours, we also compared several characteristics of their navigation data with ours. In particular, we compared how much participants navigated (Fig. 8), how many different methods they navigated to (Fig. 9), and what proportion of their navigations were revisits (Fig. 10).

Several key differences in the navigation characteristics of our participants and the Piorkowski participant stand out. Our participants, on average, navigated nearly twice as much as the Piorkowski participant (Fig. 8), but despite navigating more, our participants visited noticeably fewer different methods than the Piorkowski participant (Fig. 9). While these differ-

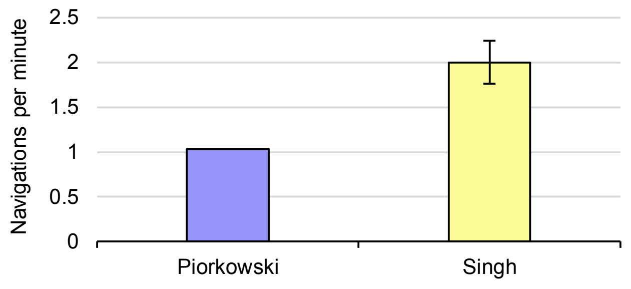


Fig. 8. On average, our participants navigated at a substantially higher rate than the Piorkowski participant. Whiskers denote standard error. The Piorkowski bar lacks whiskers because their sample size was 1 (variance undefined).

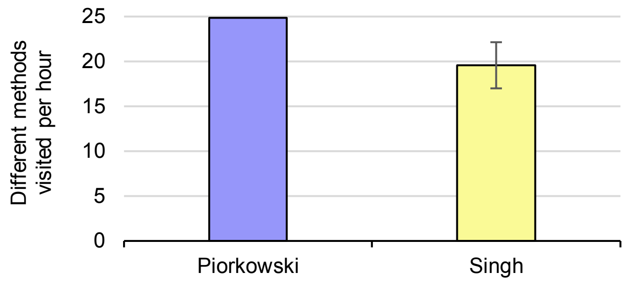


Fig. 9. On average, our participants navigated to noticeably fewer different methods than the Piorkowski participant. Whiskers denote standard error. The Piorkowski bar lacks whiskers because their sample size was 1 (variance undefined).

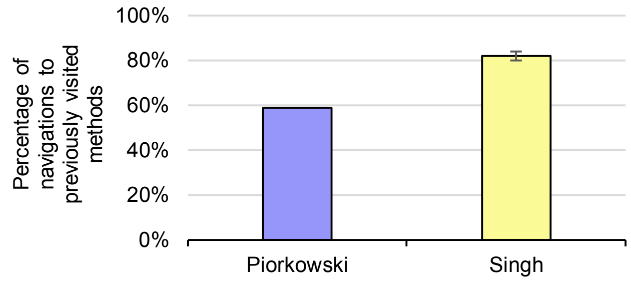


Fig. 10. On average, a greater proportion of our participants' navigations were revisits than were the Piorkowski participant's. Whiskers denote standard error. The Piorkowski bar lacks whiskers because their sample size was 1 (variance undefined).

ences are telling, perhaps most noteworthy is the difference of over 20 percentage points between the Piorkowski participant's proportion of revisits and our participants' proportion (Fig. 10). Our participants' greater proportion of revisits directly explains the greater accuracy of the Recency model for our participants' data. (Similar to above, no statistical tests could be run to confirm these differences because of the Piorkowski study's sample size of 1.)

V. Discussion

A. Click-Based Navigations Agree Closely with Humans

As our data showed, the click-based operationalization of navigation recorded navigations very similar to the ones reported by our human observers. For instance, the edit distance between the click-based sequences of navigations and the human-reported ones was comparatively low—roughly one edit for every two human-reported navigations (Fig. 2). Furthermore, the click-based navigations had very similar high-level characteristics to the human-reported ones in terms of frequency of navigation (Fig. 3), variety of different methods visited (Fig. 4), and proportion navigations that revisited recently visited methods (Fig. 5).

These click-based results hold great promise for the designers of software engineering tools that aim to track developers' navigations. The click-based operationalization is relatively easy to implement, with many code editors already providing text cursor location via a plug-in API (e.g., as in Eclipse and Visual Studio). In fact, a number of tools already use the click-based operationalization to track navigations. For example, recommender systems, such as Mylyn [9], NavTracks [26], Piorkowski et. al's recommender based on information foraging theory [21], and Team Tracks [2], have leveraged click-based logs of navigations to provide recommendations to the programmer. Furthermore, code-visualization tools, such as Stacksplorer [8] and Blaze [13], have also used click-based navigation tracking to provide visualizations relevant to the current code element.

B. View-Based Navigations Poorly Approximate Humans

In contrast to the click-based navigations, the view-based navigations differed considerably from the ones reported by our human analysts. On average, the edit distance between the click-based navigations and the human-reported ones was substantial—for every 2 human-reported navigations, over 3 edits were required to make the click-based navigations agree (Fig. 2). These differences also carried over into higher-level characteristics of the navigations: the view-based operationalization recorded more than double the number of navigations reported by human observers (Fig. 3), as well as a significantly greater variety of different methods visited (Fig. 4).

A key reason that the view-based navigations differed so greatly from the human-reported ones was that rapid scrolling through a file resulted in many spurious view-based navigations. For example, P2's navigation behavior yielded 766 view-based navigations, whereas human observers reported only 194 for the same session. In one representative 45-second episode, P2 quickly scrolled through two code files, *Publications.java* and *PublicationsDAO.java*, both of which contained numerous short methods. For this episode, the view-based operationalization recorded 31 navigations versus only 4 reported by the human observers. The view-based operationalization's tendency to overestimate navigations may be especially problematic in practice. For example, previous studies have shown that developers often resorted to scrolling through code during maintenance tasks [11] and that developers frequently performed excessive back-and-forth scrolling while trying to find a particular method of interest in files [6].

It remains an open question as to whether the view-based operationalization would perform better if it was modified to account for rapid scrolling. For example, if the method body had to remain in the center of the screen for more than 3 seconds in order to be counted as a navigation, the number of spurious navigations might be reduced considerably. However, such a change would further increase the complexity of the view-based operationalization, which is already more difficult to implement in most code editors than, for example, the click-based operationalization. In any case, the high level of disagreement we observed between the view-based operationalization and our human analysts' reports leaves considerable doubt as to the practical utility of the view-based operationalization as it is currently defined.

C. Recency Makes the Best Predictor

Of the seven predictive models of navigation evaluated, the Recency model stood out as producing significantly more accurate predictions than the others. For both our human-reported and click-based navigation data sets, the Recency model had a hit rate approaching 80%, whereas no other model achieved a hit rate greater than 70%, and the majority had hit rates below 30% (Figs. 6 and 7). A key reason for Recency's high accuracy was that our participants did a substantial amount of revisiting, of which the Recency model takes considerable advantage. For example, on average, over 80% of our participants' click-based navigations revisited recently accessed methods (Fig. 10).

Triangulating with prior research, there is mounting evidence as to the high accuracy of the Recency model. The head-to-head model evaluations conducted by Parnin and Görg [20] and by Piorkowski et al. [22] both found that recency-based predictions yielded the highest accuracy. (The Parnin study's most accurate predictor, the Least Recently Used algorithm (LRU), was essentially equivalent to our Recency model.) Furthermore, the most accurate versions of the PFIS [15] and PFIS2 [16], [22] multi-factor models of programmer navigation relied heavily on recency to make their predictions. Still other studies of developers, although not direct evaluations of predictive models, reported high rates of revisiting (e.g., [4], [11]), which would strongly favor the accuracy of the Recency model.

These strong results in favor of Recency suggest a considerable opportunity for contemporary development environments to do more to support the revisiting of recent methods. In most code editors, tabs are the main feature that supports revisiting recently visited code (generally at the granularity of files). However, researchers have pointed out numerous weaknesses with the design of tabbed editors, especially in terms of usability. For example, tabs have been found to undermine spatial memory by becoming hidden when too many are open and by repositioning themselves in ways that are difficult for developers to predict [19]. Moreover, they do not provide sufficient information to help developers accurately recall their contents [19], and as a result, developers often forget what code tabs contain and have to search through their contents to find the desired code [26].

Indeed, a number of promising tool designs have been proposed that leverage Recency-type models. Of these designs, the Mylyn [9] tool (formerly Mylar) has perhaps had the greatest success, having become a standard feature in the Eclipse IDE. Mylyn tracks a developer's activity and uses a degree-of-interest model to both identify code relevant to the developer's task and to make revisiting that code more convenient and efficient for the developer. Following a similar approach, the Team Tracks tool tracks and shares the navigation patterns of team members to facilitate efficient revisiting of code previously visited by members of the team [2]. Other tools have aimed to directly address the problems with tabs. The Autumn Leaves tool addressed these problems using a Recency-based model that automatically grays out or closes tabs that are unlikely to be used in the future [24]. On the other hand, the Patchworks code editor replaces tabs entirely with a timeline-like interface that facilitates revisiting of open code fragments while overcoming the usability problems of tabs [6]. Still other tools have integrated Recency-type models into complex recommender systems. For example, one recommender system based on information foraging theory made the highest quality recommendations when it incorporated recency into its recommendation algorithm [21]. Although these tool designs have all shown considerable promise in studies, they have yet to achieve widespread adoption in practice.

D. Piorkowski Results Correlate, but with Key Differences

Comparing our model-accuracy results with those of Piorkowski et al.'s evaluation study [22], there was clearly a strong correlation (Fig. 7). For instance, a ranking of the models by hit rate produced exactly the same ranks for both our data and the Piorkowski data, except in one case (swapped Working Set and Frequency models). Indeed, the statistical correlation between our results and those of the Piorkowski study was deemed "very strong".

However, despite this strong correlation, our three most accurate models (Recency, Working Set, and Frequency) produced substantially better hit rates than they did in the Piorkowski study. In fact, these models' hit rates for our data were 21–34 percentage points higher than they were for the Piorkowski data (Fig. 7). Clearly implicated in these differences was the fact that our participants revisited code substantially more than the Piorkowski participant (Fig. 10). Such high revisiting behavior strongly favors these three models. However, the question remains as to why our participants revisited more than the Piorkowski participant.

One possible reason that our participants revisited code more than the Piorkowski participant was that they were familiar with the code bases they worked on, as opposed to the Piorkowski participant, who was unfamiliar with the code base he worked on. As one indicator of this difference in familiarity, our participants navigated at nearly double the rate that the Piorkowski participant did (Fig. 8); however, the Piorkowski participant visited a substantially wider variety of methods than did our participants (Fig. 9). These patterns suggest that the Piorkowski participant may have had to survey more code in order to gain an understanding of the project, whereas our participants were more able to focus in on the code relevant to their tasks. Moreover, the Piorkowski participant may have had to spend longer in methods in order to understand them, which slowed his rate of navigation.

Another possible reason for the differences between our model hit rates and those from the Piorkowski study had to do with the types of tasks performed. Our participants worked on their own software-evolution tasks, which mainly consisted of feature enhancements, whereas the Piorkowski participant worked on debugging tasks. For our participants' feature-enhancement tasks, they may have tended to navigate back and forth between methods while they implemented, for example, method calls and returns. On the other hand, the Piorkowski participant's debugging work involved considerable time spent just reading different parts of the code without actually modifying anything. Indeed, the Piorkowski participant made very few changes to the code and spent the majority of his time navigating. Thus, the differences in revisiting between our participants and the Piorkowski participant may have stemmed from the differences in the types of tasks they worked on.

E. Limitations

Our study had several limitations that are common in lab studies and that should be addressed in future work. Our participants were graduate students, and thus, may not be representative of professional programmers. However, as graduate students, 70% of them had some professional development experience. Additionally, the tasks and code bases were related to a course project, and thus, may not be representative of large software projects. However, our participants' tasks were less artificial than those in most lab studies of programmers (e.g., the Piorkowski study), because our participants were performing their own tasks on their own projects. Thus, we reduced some threats to generalizability and ecological validity arising from unfamiliarity with a code base or unrealistic tasks.

VI. Conclusion

In this paper, we have presented an empirical evaluation of predictive models of programmer navigation. In particular, our study aimed to understand the extent to which existing operationalizations of navigation (click-based versus view-based) accurately measure programmer navigation behavior, and to reveal which previously proposed predictive models of programmer navigation produce the most accurate forecasts. Additionally, our study evaluated the generalizability of findings from the most comprehensive prior evaluation of predictive models by Piorkowski et al. [22]. Key findings of our study included the following:

- The click-based operationalization of navigation recorded navigations that were highly similar to those reported by human observers, suggesting the operationalization's appropriateness for use in tools.
- In contrast, the view-based operationalization diverged significantly from our human evaluators' perceptions, suggesting that the operationalization may be poorly suited for tools.
- The predictive model based on recency stood out as the most accurate model, producing a significantly higher hit rate than all the other models, and motivating the potential for tool features based on the model.
- The model-accuracy results from the Piorkowski study correlated very strongly with our results, providing a strong confirmation of the models' relative accuracies.
- However, our data yielded considerably higher hit rates than the Piorkowski data for the three most accurate models, suggesting potential effects of code familiarity and task on navigation behavior.

The findings of our study hold important implications for future work. They advance the efficacy of using the click-based operationalization to automatically detect developer navigations. However, since the click-based operationalization was not a perfect match with our human observers' perspectives, they also suggest the potential for improved operationalizations. For example, researchers outside of software engineering have found that mouse-cursor location correlates with eye-tracking data [1], [5], [7], [27], thus potentially providing an even better estimator of a programmers' attention. Furthermore, software engineering researchers have been utilizing psychophysiological sensors to better understand the physical and mental state of programmers (e.g., [10], [18], [33]), which could lead to exciting new means of understanding programmers. Additionally, our findings, along with triangulated findings from the literature, show the importance of supporting the revisiting behavior of programmers. As we discussed (Section V-C), researchers have proposed numerous tools that aim to provide such benefits; however, few or none have yet to receive widespread adoption in practice. Our findings further make the case that such innovations can lead to a qualitative difference in the work of software developers, and we hope that they will spur the field's evolution beyond the increasingly inadequate revisiting support offered by current development environments.

Acknowledgment

This material is based upon work supported by the National Science Foundation (NSF) under Grant No. 1302117. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] M. C. Chen, J. R. Anderson, and M. H. Sohn, "What can a mouse cursor tell us more?: Correlation of eye/mouse movements on web browsing," in *CHI '01 Extended Abstracts on Human Factors in Computing Systems (CHI EA '01)*. ACM, 2001, pp. 281–282.

[2] R. DeLine, M. Czerwinski, and G. Robertson, "Easing program comprehension by sharing navigation data," in *Proc. 2005 IEEE Symp. Visual Languages and Human-Centric Computing (VL/HCC '05)*, 2005, pp. 241–248.

[3] K. A. Ericsson, "Valid and non-reactive verbalization of thoughts during performance of tasks," *J. Consciousness Stud.*, vol. 10, no. 9–10, pp. 1–19, 2003.

[4] T. Fritz, D. C. Shepherd, K. Kevic, W. Snipes, and C. Bräunlich, "Developers' code context models for change tasks," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*. ACM, 2014, pp. 7–18.

[5] Q. Guo and E. Agichtein, "Towards predicting web searcher gaze position from mouse movements," in *CHI '10 Extended Abstracts on Human Factors in Computing Systems (CHI EA '10)*. ACM, 2010, pp. 3601–3606.

[6] A. Z. Henley and S. D. Fleming, "The Patchworks code editor: Toward faster navigation with less code arranging and fewer navigation mistakes," in *Proc. 32nd Annu. ACM Conf. Human Factors in Computing Systems (CHI '14)*. ACM, 2014, pp. 2511–2520.

[7] J. Huang, R. W. White, and S. Dumais, "No clicks, no problem: Using cursor movements to understand and improve search," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*. ACM, 2011, pp. 1225–1234.

[8] T. Karrer, J.-P. Krämer, J. Diehl, B. Hartmann, and J. Borchers, "Stacksplorer: Call graph navigation helps increasing code maintenance efficiency," in *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, 2011, pp. 217–224.

[9] M. Kersten and G. C. Murphy, "Mylar: A degree-of-interest model for IDEs," in *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)*. ACM, 2005, pp. 159–168.

[10] K. Kevic, B. M. Walters, T. R. Shaffer, B. Sharif, D. C. Shepherd, and T. Fritz, "Tracing software developers' eyes and interactions for change tasks," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '15)*. ACM, 2015, pp. 202–213.

[11] A. J. Ko, H. Aung, and B. A. Myers, "Eliciting design requirements for maintenance-oriented IDEs: A detailed study of corrective and perfective maintenance tasks," in *Proc. 27th Int'l Conf. Software Engineering (ICSE '05)*, 2005, pp. 126–135.

[12] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Trans. Softw. Eng.*, vol. 32, no. 12, pp. 971–987, Dec. 2006.

[13] J.-P. Krämer, T. Karrer, J. Kurz, M. Wittenhagen, and J. Borchers, "How tools in IDEs shape developers' navigation behavior," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, 2013, pp. 3073–3082.

[14] T. LaToza and B. Myers, "Visualizing call graphs," in *Proc. 2011 IEEE Symp. Visual Languages and Human-Centric Computing (VL/HCC '11)*. IEEE, 2011, pp. 117–124.

[15] J. Lawrance, R. Bellamy, M. Burnett, and K. Rector, "Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*. ACM, 2008, pp. 1323–1332.

[16] J. Lawrance, M. Burnett, R. Bellamy, C. Bogart, and C. Swart, "Reactive information foraging for evolving goals," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, 2010, pp. 25–34.

[17] S. McConnell, "What does 10x mean? Measuring variations in programmer productivity," in *Making Software: What Really Works, and Why We Believe It*. O'Reilly Media, Inc., 2010, ch. 30.

[18] S. C. Müller and T. Fritz, "Stuck and frustrated or in flow and happy: Sensing developers' emotions and progress," in *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*. IEEE, 2015, pp. 688–699.

[19] C. Parnin and S. Rugaber, "Programmer information needs after memory failure," in *Proc. 2012 IEEE 20th International Conference on Program Comprehension (ICPC '12)*. IEEE, 2012, pp. 123–132.

[20] C. Parnin and C. Görg, "Building usage contexts during program comprehension," in *Proc. 14th International Conference on Program Comprehension (ICPC '06)*. IEEE, 2006, pp. 13–22.

[21] D. Piorkowski, S. Fleming, C. Scaffidi, C. Bogart, M. Burnett, B. John, R. Bellamy, and C. Swart, "Reactive information foraging: An empirical investigation of theory-based recommender systems for programmers," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12)*. ACM, 2012, pp. 1471–1480.

[22] D. Piorkowski, S. D. Fleming, C. Scaffidi, L. John, C. Bogart, B. E. John, M. Burnett, and R. Bellamy, "Modeling programmer navigation: A head-to-head empirical evaluation of predictive models," in *Proc. 2011 IEEE Symp. Visual Languages and Human-Centric Computing (VL/HCC '11)*. IEEE, 2011, pp. 109–116.

[23] D. J. Piorkowski, S. D. Fleming, I. Kwan, M. M. Burnett, C. Scaffidi, R. K. Bellamy, and J. Jordahl, "The whats and hows of programmers' foraging diets," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*, 2013, pp. 3063–3072.

[24] D. Roethlisberger, O. Nierstrasz, and S. Ducasse, "Autumn Leaves: Curing the window plague in IDEs," in *Proc. 16th Working Conference on Reverse Engineering (WCRE '09)*. IEEE, 2009, pp. 237–246.

[25] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Trans. Softw. Eng.*, vol. 25, no. 4, pp. 557–572, Jul. 1999.

[26] J. Singer, R. Elves, and M.-A. Storey, "NavTracks: Supporting navigation in software maintenance," in *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05)*. IEEE, 2005, pp. 325–334.

[27] M. J. Spivey, M. Grosjean, and G. Knoblich, "Continuous attraction toward phonological competitors," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 102, no. 29, pp. 10 393–10 398, 2005.

[28] M. W. van Someren, Y. F. Barnard, and J. A. Sandberg, *The Think Aloud Method: A Practical Guide to Modelling Cognitive Processes*. Academic P., 1994.

[29] D. Čubranić and G. C. Murphy, "Hipikat: Recommending pertinent software development artifacts," in *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*. IEEE, 2003, pp. 408–418.

[30] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *J. ACM*, vol. 21, no. 1, pp. 168–173, Jan. 1974.

[31] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Trans. Softw. Eng.*, vol. 30, no. 9, pp. 574–586, Sep. 2004.

[32] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. IEEE, 2004, pp. 563–572.

[33] M. Züger and T. Fritz, "Interruptibility of software developers and its prediction using psycho-physiological sensors," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, 2015, pp. 2981–2990.