The Influence of the Task on Programmer Behaviour

Annie T.T. Ying and Martin P. Robillard
School of Computer Science, McGill University, Montréal, QC, Canada
{*annie.ying,martin*}*@cs.mcgill.ca*

***Abstract*—Programmers performing a change task must understand the existing software in addition to performing the actual change. This process is likely to be affected by characteristics of the task. We investigated whether the nature of a task has any relationship with *when* a programmer edits code during a programming session. We characterized differences in editing behaviour with three types of editing styles: EDIT-FIRST, EDIT-LAST, and EDIT-THROUGHOUT. We based our analysis on the interaction history of over 4000 programming sessions collected as part of the development history of open source projects. Our results showed that an enhancement task (as opposed to a bug fix) was less likely to be associated with a high fraction of source code edit events at the beginning of the programming session. To our surprise, we also found that the presence of a stack trace in a bug report did not significantly affect the editing style of the programming session.**

***Keywords*-Program comprehension; Development interaction history; Mining software archives.**

I. Introduction

Programmers performing a change task must understand the existing software in addition to performing the actual modification [1]. Many theories have been proposed to describe the program understanding process. For example, Littman et al. observed that a programmer using a systematic strategy studied code in detail before making changes to the code, and a programmer using an as-needed strategy minimized this studying stage [2].

Such theories hint at the significance of *timing* of coding activity in a programming task. Given that a major goal when designing a software development tool is to present relevant information when a programmer needs it, understanding *when* programming edit events (that represent coding) occur in a task is key. When do programmers make edits within a task performed in modern software development environments? How is this timing of edits affected by external factors such as the nature of a task?

In this paper, we describe our analysis of the relationship between programmer behaviour and task type. The aspect of programmer behaviour we analyzed was based on *when* a programmer changes the code. We characterized differences in editing behaviour with three types of editing styles: EDIT-FIRST, EDIT-LAST, and EDIT-THROUGHOUT. The study was based on data that has been collected as part of the development history of open source projects: the interaction history of over 4000 programming sessions by over 100 programmers using the Eclipse development environment,[1] collected by the monitoring facility which supports the Mylyn task-focused interface [3]. This history of programming sessions was linked with their corresponding tasks, more specifically, bug reports[2] in the Bugzilla[3] bug tracking system.

We found that different types of tasks were associated with different editing styles in terms of *when* the edit events to the source code occur. For example, an enhancement task (as opposed to a bug fix) was less likely to be associated with a high fraction of source code edit events early on in the programming session. To our surprise, we also found that the presence of a stack trace in a bug report did not have a significant impact on the editing style of the programming session.

For the rest of the paper, we first summarize related work (Section II) and our study (Section III). We then present how the nature of tasks relates to editing styles (Section IV) and how the presence of a stack trace in a bug report relates to editing style (Section V). We end the paper with discussion (Section VI) and conclusion (Section VII).

II. Related work

We present three areas of related work: studies which use recorded usage data similar to the type of data we used in our study; other tools and analyses based on interaction history; and the line of research in cognitive models in program comprehension and recent empirical studies on programmers performing change tasks in lab settings.

Empirical studies on tool usage using interaction history

Several studies have looked into recorded software environment usage data to better inform software engineering tool design. Murphy et al. reported on how programmers use a development environment [4], based on usage data collected by a version of the interaction history monitoring facility supporting the Mylyn task-focused interface [5]. Murphy-Hill et al. used four different data sets, including tool usage data from the study by Murphy et al., to understand how developers refactor [6]. Parnin and Rugaber used interaction history, also including the same set of usage data

[1]http://www.eclipse.org
[2]We follow the convention to use the word "bug report" to refer to both a software bug report and an enhancement request.
[3]http://www.bugzilla.org

from Murphy and colleagues, to understand how programmers resume their work after having been interrupted [7]. Our study differs in the intent, in that we are interested in revealing the association between different editing styles and task types, rather than on general usage, refactoring usage or resumption strategies. Although the type of interaction history we used was the same or similar to interaction history used in these studies, our data was mined from a software archive, rather than collected for the purpose of the study as with in these studies.

Tools and analyses based on interaction history

Interaction history has started to become a popular source of input for various tools and analyses. Several tools have used interaction history for recommending code of interest. Both Mylyn [3] and wear-based filtering [8] use interaction frequency to highlight the elements of interest in the user interface of a development environment and filter away uninteresting elements. The NavTracks tool provides recommendations of which files are related to the currently selected files based on an analysis on cycles in the navigation history [9]. Robillard and Murphy used navigation data for inferring code that belongs to the same software concerns [10]. Parnin and Görg experimented with several measures based on analyzing interaction history of inferring the context that are relevant to a task [11]. Robbes and Lanza proposed a code completion tool based on analyzing interaction history [12].

Researchers have extended the change coupling idea—in the context of analyzing files that tends to get checked in together to a Software Configuration Management system [13]—to interaction history. Zou et al. identified several types of interaction coupling, such as co-change, change-view, co-view, and other more specific patterns [14]. Robbes et al. augmented this work with three other measures based on interaction history [15]: changed-based coupling (entities that change many times during a session are more coupled than those which only changed occasionally); interaction coupling (the number of switches between elements); time-based coupling (the proximity in time two entities changed).

A challenge with analyzing interaction history was to divide the sequence of events in an interaction history into meaningful units. The SpyWare tool displays a visualization and identifies sessions of work based on several measures including number of edits per minute [16]. Coman and Sillitti proposed an approach to segment development sessions [17]. Safer and Murphy proposed a tool to help developers recall information about recent tasks by tracking the navigation history and capturing screen snapshots [18].

In summary, all this prior work focuses on different ways to use interaction history in software engineering tools. In contrast, the purpose of our study is to understand how interaction history relates with a factor *external* to the interaction history, the task type.

Empirical studies of programmers in lab settings

Another line of work relevant to ours is the long history of program comprehension research. Storey's survey provides a comprehensive overview [19]. Numerous models describe the cognitive processes used by programmers to form a mental representation of the program, based on observations on programmers in lab settings. According to the top-down theory, programmers start with a top-level hypothesis about the general nature of the program, refined by subsequent sub-hypotheses [20]. According to the bottom-up theory, programmers read individual statements in the code and mentally group those statements into higher-level abstraction, capturing control and data flow [21]. Littman et al. noted that programmers use either a systematic or as-needed strategy [2].

More recently, several researchers have studied how programmers perform change tasks in a lab setting. Ko et al. reported on programmers using the Eclipse development environment on small maintenance tasks [22]. They specifically discussed editing patterns in terms of when editing, searching, and navigation operations happen in the task. Robillard et al. characterized how programmers who are successful at maintenance tasks typically navigate code [23].

Our work complements this large body of knowledge from lab settings by looking at a large amount of interaction history collected in the field. There is some initial evidence that the behaviour seen in the lab translates to the trace data of how a programmer interacts with a development environment collected in the field [24]. In addition, our study differs in the intent, which is to understand the editing styles from different types of task.

III. Research questions, data, and variables

We were interested in determining the relationship between different types of editing behaviour and the associated tasks. We categorized the different types of editing behaviour based on *when* edit events happen: We categorized a trace—a sequence of events in the interaction history—with a high fraction of edit events in the first half of the trace as having an EDIT-FIRST style, a trace with a lower fraction of edit events as having an EDIT-LAST style, and otherwise a EDIT-THROUGHOUT style. We could roughly map the EDIT-LAST style to a systematic strategy (a programmer tends to study code in detail before making changes) in the program comprehension theory and the EDIT-FIRST and EDIT-THROUGHOUT as different degrees of as-needed strategy (a programmer tends to minimize the studying stage) [2].

The two research questions we were interested in were motivated by prior work. Prior research has shown that the program comprehension strategy chosen by a programmer depends on the task, whether a task requires recall and comprehension [21]. We wanted to investigate how the task type (an enhancement or a bug fix) relates to editing style:

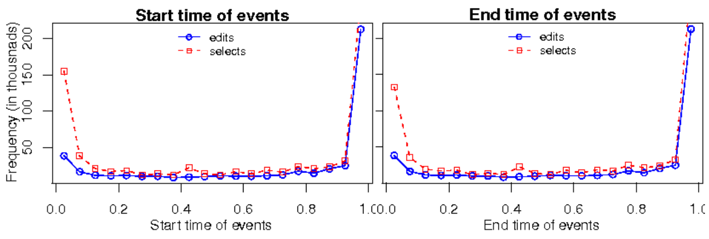

Figure 1. Distribution of edits and selections in a normalized trace timeline.

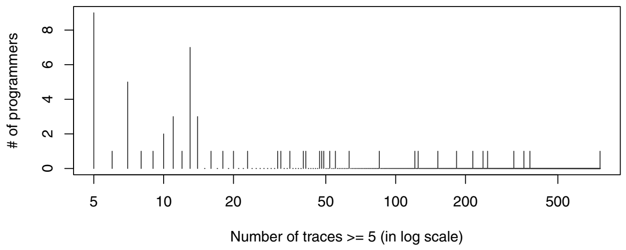

Figure 2. Distribution of the number of traces programmers contribute to

RQ1: How do different types of editing behaviour relate to task types?

To further understand the context in which the editing styles took place, we investigated whether the presence of a stack trace affects the editing style of a task. A stack trace in a bug report provides a concrete point for a developer to start the task. This information can impact how a developer approaches a task. Prior work has shown that the presence of a stack trace in a bug report affects how fast a task gets fixed [25]. Hence, our second research question is:

RQ2: Does the presence of a stack trace in a bug report affect editing behaviour?

A. *First source of data: interaction history*

Our data consists of traces of interaction history collected by the user action monitoring facility that supports Mylyn. This data was the basis for inferring editing styles. Each trace captured the interaction required to complete a task declared by the programmer using Mylyn. The monitor recorded events (such as edits and selections) a programmer performed on the Eclipse development environment.

Interaction history data was generated by the Mylyn monitor and was stored as an XML file containing events. Each event represents a user action on a program element, at a particular time recorded as a timestamp. The monitor captures two kinds of user actions: *selections* (editor and view selections via a mouse or a keyboard) and *edits*.[4] The monitor also records the signature of the program element involved in a user action. For example, the signature of a Java method contains the Java package and class the method was in, the name of the method, and the parameter type(s) of the method.

Conceptually, the interaction history is a sequence of ordered events. However, for scalability, the monitor does not record all user actions [26, p.43]. Most of the events involving the same program element through the same user action are aggregated. Understanding how these events are aggregated was hence a requirement for mining this data. When such an aggregation happens, the event was expanded to store two timestamps instead of one: the timestamp of the first event and the timestamp of the last event being aggregated. Figure 1 shows the effect of aggregate events on the data. The two graphs show the distribution of timestamps, each timestamp normalized between 0 and 1, where 0 represents the time a of the first event in the task, and 1 represents the time b of the last event in the task. Hence, each aggregated event start time or end time t is positioned at $\frac{t}{(b-a)}$ in the graphs. We used these timestamps for classifying traces to editing styles, described in Section III-C. The reason there were so many events close to the beginning and the end of traces in Figure 1 was because most of the aggregates capture similar events throughout the trace.

We analyzed traces from the development of Eclipse, which is written in Java. The interaction traces are archived in the Bugzilla issue-tracking system, each as an attachment to a bug report. 60% of the traces come from one Eclipse sub-project, the Mylyn development project itself, i.e., the interaction history of tasks involved to build the Mylyn tool. The Mylyn project demands that all code contributions to the project use Mylyn itself.[5] The intent is to make it easier to recover the program elements relevant to the implemented solution of a bug report when the solution needs to be revisited. This project rule is implemented by requiring each code contribution to be associated with a bug report and "task context" file, which contains the part of the interaction history associated with the solution of a bug report. For the rest of the sub-projects in Eclipse, there are no project rules that dictate the use of Mylyn. In other words, the attachment of traces is done on a voluntary basis. Table I shows the top five sub-projects in Eclipse with the most number of traces.

B. *Second source of data: bug reports*

Our second data source was Bugzilla bug reports in the Eclipse project[6] that were associated with a trace. We derived *task type* as follows. Each bug report includes several types of fields: structured fields (for example, severity, priority, and the email address(es) of the programmer(s) assigned to the task), textual fields (for example, the title of the report, the description of the bug, and comments on the bug), attachments (for example, an image file containing a screen-shot), and relationships with other bug reports (for example, duplicates). A trace was associated with a bug report as an attachment of the report.

[4]The monitor can also capture *commands* (such as preference changes and saving a file). For this data set, however, commands were not recorded because the monitor was not configured to do so.

[5]http://wiki.eclipse.org/Mylyn_Contributor_Reference#Contributors

[6]https://bugs.eclipse.org/bugs/

Table I
TOP FIVE PROJECTS WITH THE MOST NUMBER OF TRACES

Projects	Description	# of traces
Mylyn	for task management	1994
PDE	for building Eclipse plug-ins	505
Platform	core framework in Eclipse	438
ECF	for building distributed servers & applications	142
MDT	for UML modeling	94

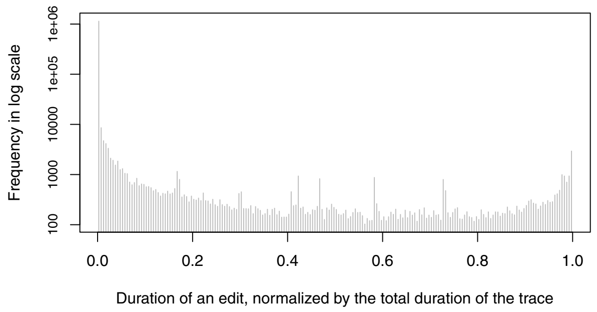

Figure 3. Distribution of the duration of edits

For our analysis, we constructed a data set that contained all the traces named "mylyn-context.zip" (the default name the Mylyn tool gives when exporting the interaction history to a trace attached to a bug report) that were attachments to bug reports, as well as the bug reports to which these traces were attached. The data set spanned roughly four years of development, from the first bug report on November 11, 2005, to December 29, 2009. In total, there were 3128 bug reports associated with at least one of the 4245 traces. The data set contained 153 programmers who contributed to at least one trace. Eighty-three of the programmers (54%) contributed fewer than five traces. The way we identified a programmer was by the email address indicated as the attacher of the trace on the bug report. Figure 2 shows the distribution of the number of traces of the 70 programmers (46%) who contributed 5 or more traces. The median number of traces for these 70 programmers was 14.5. Only 7 (4.6%) of them contributed more than 200 traces

C. *Editing style of a task*

For RQ1 (how do different types of editing behaviour relate to task types?) and RQ2 (does the presence of a stack trace in a bug report affect editing behaviour?), we needed the notion of editing behaviour for a task. We categorized a task as an editing style—EDIT-FIRST, EDIT-LAST, and EDIT-THROUGHOUT—based on the fraction of edit events at different points in a trace. We characterized *when* an edit event happened as follows. First, we normalized all the timestamps in a trace to [0,1]. An edit event e_t within a trace t was classified as the first half of the trace, if more percentage of the duration of e_t (the difference between normalized start and end times) resides in the first half, or more precisely:

$$e_t = \begin{cases} first & \text{if } e_{start} < 0.5 \text{ and } e_{end} < 0.5 \\ & \text{or } (0.5 - e_{start}) > e_{end} - 0.5) \\ second & \text{otherwise} \end{cases}$$

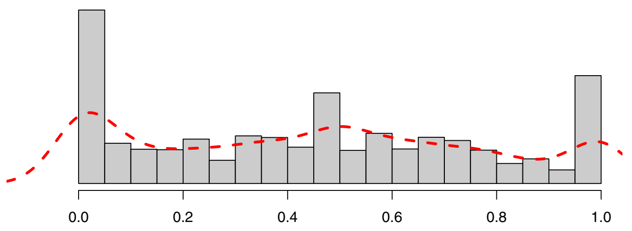

Figure 4. Density function of $Fr(e_t = first)$

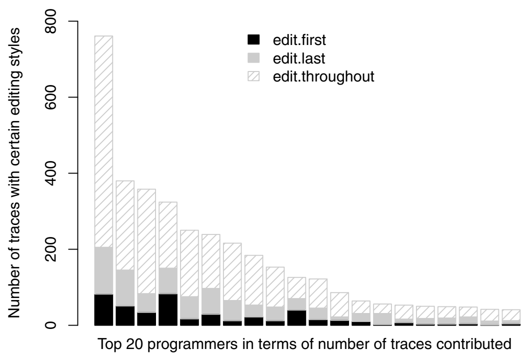

Figure 5. Editing styles for 20 programmers with the most contributions

Intuitively, we considered an edit event as a time range and classified it to the first or the second part of the trace based on whether the time range covered more of the first or the second part of the trace. The motivation for classifying an event this way was to deal with the fact that an edit event could be an aggregate of multiple edits on the same program element. We discuss further issues in the Discussion section. Figure 3 shows the distribution of the duration of edit events of the traces.

We characterized a trace using the fraction of edit events in the first half of all edit events in the trace: $Fr(e_t = first) = \frac{Nbr(e_t=first)}{Nbr(e_t)}$. Figure 4 shows the density function of $Fr(e_t = first)$. The figure reveals a tri-modal distribution. The distribution is outlined by the dotted line (in red) in Figure 4, with the three modes coinciding with the bar closest to $Fr(e_t = first)$ equal zero, the bar in the middle, and the bar closest to 1. Hence, three is the most natural number of styles. We used the two lowest points which divided the three modes in the density function as the thresholds to group the traces: 0.19 and 0.87. Consequently, we divided the traces with $Fr(e_t = first)$ close to 0 (between 0% to 19% of events in the first half of the trace), close to 1 (between 87% to 100%), or neither. We called these traces EDIT-LAST, EDIT-FIRST, and EDIT-THROUGHOUT, respectively. Of the 4245 traces, 539 (12.7%) traces were classified as EDIT-FIRST, 892 (21.9%) as EDIT-LAST, and 2698 (65.4%) as EDIT-THROUGHOUT. Figure 5 shows the proportion of editing styles of the 20 programmers with the most number of contributions.

D. Characterizing tasks

For RQ1, we used the severity field of the bug report to categorize tasks. The field can take one of the following categories: enhancement, blocker, critical, major, normal, minor, and trivial. The severity field was first provided when the report was created by a user or a programmer according to a guideline set by the project.[7] One problem with this process is that some users may not follow the guideline for assigning the field. Fortunately, the field does tend to eventually evolve to an agreed-upon one [27]. Therefore, we took the latest value of the severity field to define the nature of a task.

In addition, we addressed the imprecise nature of the severity field by grouping the seven severity categories into coarser ones: *enhancement tasks* (only consisting of the enhancement severity category). *minor bug fixes* (aggregating minor and trivial severity categories), and *major bug fixes* (aggregating blocker, critical, major, and normal severity categories). Of the 3134 bug reports, 674 (21.5%) were enhancement tasks, 444 (14.2%) were minor bug fixes, and 2016 (64.3%) were major bug fixes. The 4245 traces that were associated with these 3134 bug reports, 1180 (27.8%) were associated with enhancement tasks, 522 (12.3%) with minor bug fixes, and 2543 (60.0%) with major bug fixes.

For RQ2, we were interested in the presence of a stack trace in a bug report. We determined whether a bug report contained a stack trace using a set of regular expressions which captured the output of typical formats of stack traces.

To ensure that the bug severity and the presence of a stack trace field did represent a meaningful and reliable grouping of the bug reports, we applied a clustering algorithm on the bug reports using bug severity and the presence of a stack trace, and one additional feature, the number of comments. The motivation for using the number of comments as a feature was that the amount of comments quantified how much discussion took place to resolve the bug: if no long discussion existed, it was likely that the bug had a clear fix.

To find the clusters, we used an algorithm called PAM (Partitioning Around Medoids[8]), an algorithm that tends to be robust in the presence of outliers.[9] One challenge with using a typical clustering algorithm is to determine the number of clusters, *k*. To solve this problem, we used the silhouette width,[10] which can be used to find the optimal number of clusters [28], by running the clustering algorithm with different *k* values and seeing which one gives a larger silhouette width. Applying the clustering algorithm, we found that the optimal number of clusters is four, with a silhouette width of 0.94, meaning the tasks form extremely strong clusters. Table II shows the medoids (centers) of the clusters. This information is useful for understanding what is a typical member in a cluster. Note the correspondence between this grouping with the grouping defined by only the severity field. Roughly speaking, enhancement tasks correspond to Cluster 3, minor bug fixes correspond to Cluster 4, major bug fixes with the presence of a stack trace in the report correspond to Cluster 1, and major bug fixes without a stack trace correspond to Cluster 2.

Table II
MEDOIDS OF THE RESULTING CLUSTERS ON TASKS

	# of comments	severity	stack trace?
Cluster 1	5	major	yes
Cluster 2	6	major	no
Cluster 3	9	enhancement	no
Cluster 4	6	minor	no

IV. HOW DO DIFFERENT TYPES OF EDITING BEHAVIOUR RELATE TO TASK TYPES? (RQ1)

Having presented the data and the variables in this study, we now assess RQ1 from a statistical perspective and then explain some of the statistical findings more qualitatively.

Statistical assessment

For the statistical assessment, we first used a chi-squared test of independence to determine whether there existed an association between editing styles and task types, since both variables are nominal. This chi-squared test only informed us whether a statistically significant association between the two variables existed, not which of the nine possible associations—three types of task, each could be associated with three editing styles—are statistically significant nor the direction (positive or negative) of the association. Thus, if the chi-squared test yields a significant result, we would investigate which particular type of task was related to which editing style through nine post-hoc tests, one for each of the possible association. For all the test statistics presented in this paper, the significance level was at $\alpha = 0.05$. We performed these tests using various packages in the R statistics framework, except noted otherwise .[11]

[7]http://www.eclipse.org/tptp/home/documents/process/development/bugzilla.html

[8]The essence of the PAM algorithm is based on first searching for *k* (the specified number of clusters) representative objects or medoids, and then assigning each object to the nearest medoid [28]. The way the algorithm specifies whether two objects are "near" is by a distance function, which is the Euclidean distance of the three features in our case.

[9]We used an off-the-shelf implementation of this algorithm in the R cluster package: http://cran.r-project.org/web/packages/cluster/index.html

[10]Intuitively, the silhouette width measures how well points (in our case, points in the feature space) in a cluster are close to each other compared to their closest cluster. A value close to 1 means that the points are well-clustered and they were assigned to very appropriate clusters, close to 0 means that the points could be assigned to another closest cluster, and a value close to -1 means that the points have been misclassified. A clustering with a silhouette width above 0.7 is considered strong, reasonable between 0.5 and 0.7, weak or artificial between 0.25 and 0.5, and no substantial structure less than 0.25 [28].

[11]http://cran.r-project.org

Table III
BREAK-DOWN OF EDITING STYLE PER TASK GROUP. BOLD ENTRIES REPRESENT SIGNIFICANT ASSOCIATIONS

	EDIT-FIRST	EDIT-LAST	EDIT-THRUOUT	total
Enh.	**125 (-)**	341	714	1180
Minor	69	**124 (-)**	**329 (+)**	522
Major	**401 (+)**	714	**1428 (-)**	2543
total	595	1179	2471	

Chi-squared test - Is there a relationship between editing styles and task types: Table III shows the raw data, the contingency table used in the chi-squared test, with the break-down of editing styles for each task type used in the chi-squared test, expanded with the particular association found in the post-hoc test in the next stage of the analysis described later. A chi-squared test revealed a statistically significant relationship between task type and editing styles ($df = 4$, $\chi^2 = 28$, $p = 0.000069$).

Post-hoc tests - Which task types are associated with which editing styles: To investigate where the association might lie, we examined the standardized Pearson residuals for each of the possible association, i.e., each cell in the contingency table used in the chi-squared test [29].[12]

Our data in Table III with three editing styles and three types of tasks had nine possible associations, requiring nine tests using standardized Pearson residuals. Five of these tests resulted in a statistically significant result:

1) Enhancement tasks were negatively associated with EDIT-FIRST (standardized Pearson residual = -4.0, p = 0.000067). This association was indicated in Table III, as bold in the cell that pertains to the row denoting enhancement tasks and the EDIT-FIRST column.
2) Minor bug fixes were positively associated with EDIT-THROUGHOUT (standardized Pearson residual = 2.4, p = 0.017) and
3) negatively associated with EDIT-LAST (standardized Pearson residual = -2.2, p = 0.029).
4) Major bug fixes were positively associated with EDIT-FIRST (standardized Pearson residual = 4.0, p = 0.000058) and
5) negatively associated with EDIT-THROUGHOUT (standardized Pearson residual = -3.3, p = 0.00090).

To examine the degree of these associations, we looked at odds ratios, comparing the odds of two editing styles on two groups of tasks [29, p.55] and reporting the ones related to the five significant associations revealed from the residual analysis:

[12]The standardized Pearson residual for a cell (in Table III, for example) would indicate how much different the cell count was from the count if no associations were present. These residual values were standardized, in the sense that the probability distribution of these values can be approximated by the standard normal distribution. The cells with standardized Pearson residuals exceeding an absolute value of 1.96 (from the distribution table) at the $\alpha = 0.05$ significance level would be the cells with significant associations. A positive standardized Pearson residual would indicate a positive association and a negative value would indicate a negative association.

- For the two kinds of bug fixes (that is, data from the second and third rows of Table III), the odds ratio of EDIT-THROUGHOUT style instead of EDIT-LAST for minor bug fixes relative to major bug fixes was $\frac{329/124}{1428/714} = 1.32$. This ratio compares the odds of a task being EDIT-THROUGHOUT over EDIT-LAST for minor bug fixes (the numerator $329/124$) compared to that of major bug fixes (the denominator $1428/714$)). This ratio quantified the size of the effect of two relationships we found significant: #2 (minor bug fixes were positively associated with EDIT-THROUGHOUT) and #5 (major bug fixes were positively associated with EDIT-FIRST) from the residual analysis.
- For enhancements and minor bug fixes, the odds ratio of EDIT-FIRST style instead of EDIT-LAST for enhancements relative to minor bug fixes was 0.658. This ratio quantified the size of the effect of two relationships we found significant: #1 (enhancement tasks were negatively associated with EDIT-FIRST) and #3 (minor tasks were negatively associated with EDIT-LAST).
- For major bug fixes and enhancements, the odds ratio of EDIT-FIRST style instead of EDIT-THROUGHOUT for major bug fixes relative to enhancements was 1.60, and the odds ratio of EDIT-FIRST style instead of EDIT-LAST for major bug fixes relative to enhancements was 0.835. These ratios quantified two relationships we found significant: #1 (enhancement tasks were negatively associated with EDIT-FIRST) and #4 (major bug fixes were positively associated with EDIT-FIRST).

Eliminating effect of project experience on RQ1: In RQ1, we investigated the effect of task type on editing style. However, the investigation did not take into account the effect of individual programmers on the analysis. Programmers who contributed to more traces were being sampled more than programmers who contributed to fewer traces. To eliminate the effect on the number of bugs a programmer contributed to, we used a stratified analysis, the Cochran-Mantel-Haenszel (CMH) test [29], where each strata was a group of programmers with the similar number of bug contributions. In the CMH test, the data are arranged in a *set* of two-dimensional tables, each table representing data from a particular strata.

A chi-squared test and a CMH test both address the question of associations between two variables, but the difference between a CMH test and a chi-squared test is that the CMH test investigates the overall association among the *set* of tables instead of the association in just one table as in a chi-squared test. In the chi-squared test we performed on the data from Table III, the data consisted of a table with nine cells, combination between the three task types (minor, major, and enhancement) and the three editing styles (EDIT-FIRST, EDIT-LAST, and EDIT-THROUGHOUT). For the CMH test, we divided the data into four stratas—programmers with the number of bug contributions in the first quartile, second quartile, third quartile, and the fourth quartile—and the data from each strata corresponding to a table with nine cells from combining the three task types and the three editing styles.

The CMH test revealed a statistically significant relationship between task type and editing styles, adjusting for the effect of programmers with different number of contributions of bugs ($df = 4$, $CMHstatistics = 13$, $p = 0.010$). Note that the CMH test result could change if we defined the stratas differently. For example, in the extreme case where all programmers are in one strata, this case is the same as the chi-squared test we performed on Table III. To investigate how sensitive was the definition of stratas on the test statistic, we experimented with the other extreme case of having each individual programmer as one strata. The CMH test[13] revealed a nearly statistically significant relationship with $p = 0.062$.[14] We interpreted this result as the data did reveal a significant association between task type and editing style even after adjusting for individual programmers.

In-depth analysis

For a deeper understanding of the statistical relationships, we analyzed a sample of the traces and the corresponding tasks more qualitatively. For selected statistical relationships, we chose three traces involved in the relationship.

Understanding a trace with hundreds or thousands of events was a major challenge. To assist in this process, we built a timeline based event visualizer. We also examined the bug report the trace was attached to and the actual code changes, from the patch attached to the bug report and/or the version of the code checked into the version history (based on the bug ID included in the check-in comment).

The factors we used in this in-depth analysis were motivated by prior work (see the respective citations). Three concerned the traces: the number of programming sessions defined as continuous activity in the trace separated by interruption of more than a day (this aspect was important as prior work has noted the prevalence of interruptions and the effort to resume [30]); duration of the trace; and the ways program elements were being selected (as studied in prior work [31]). Two concerned the code changes: size of the code changes; and how do the program elements touched in the trace compared to the elements that are actually changed in the patch or check-in (prior work has used elements in an interaction trace and the actual check-in [32]). *"Equal"* means that the trace contains all the events that can explain the change and no additional events; *"incomplete"* means that the trace misses to capture events that can explain part of the change; *"related extras"* means that the trace contains events that can explain the change, plus other events on related code; and *"unrelated extra"* means that the trace contains events that can explain the change, plus other events on unrelated code; this can be interpreted as a trace that contains parts of another unrelated programming session. These five factors are the last five columns of Table IV. The first six columns are descriptive information about the traces, the associated bug reports and the variables in this study.

1) Major and EDIT-FIRST *style:* One of the results from the statistical assessment—more specifically, the residual analysis—was that major bug fixes were positively associated with EDIT-FIRST style (result #4) while minor bug fixes with the EDIT-THROUGHOUT style (result #2). We found this a bit surprising because major bug fixes are more complex than minor bug fixes, so we would expect minor bug fixes, not major bug fixes, to be associated with EDIT-FIRST style.

To shed some light on an answer, we randomly selected and analyzed three EDIT-FIRST traces that are associated with major bug fixes, #261136, #267399, and #215156. The additional dimensions we documented for these three bugs are summarized in the first three rows of Table IV.

We found that two of the traces associated with major bug fixes were EDIT-FIRST because the trace contained selection events after all the required editing was completed. In the fix of #261136 (which involved removing a file that broke the unit tests), the selection events after the single edit solely responsible for the code change touched several files related to running the tests. For #267399 (which fixed a synchronization problem with two UI views), the selections involved after the edit events touched the same methods that were being edited.

A possible explanation was that the selections performed after the edit events were part of a code review. We further speculated that as these two bugs were marked as having major severity, it was important enough to deserve a code review. Even if the process did not prescribe it, developers might look over the code to ensure everything was as it should be.

2) Minor bug fixes and EDIT-THROUGHOUT *style:* The residual analysis revealed that minor bug fixes were associated with EDIT-THROUGHOUT style (result #2). Again, this seemed a bit surprising because minor bug fixes tend to not require a code exploration, more of a EDIT-FIRST scenario.

To investigate why, we randomly selected and analyzed three EDIT-THROUGHOUT traces that were associated with minor bug fixes (#275884, #280811, and #282445). The three traces actually looked as expected, with selection events interleaving with edit events that contributed to the final patch. All three involved small changes (all within one file as shown in the second last column in Table IV) and were short (within a minute, 15 minutes, and 35 minutes, respectively, again shown in Table IV).

We hypothesized that minor bug fixes could be associated with EDIT-THROUGHOUT style because minor bug fixes

[13]the version of CMH test with Monte Carlo sampling approximation of an exact test due to the high percentage (25%) of cells with zero

[14]We performed the test using the StatXact software, http://www.cytel.com/software/StatXact.aspx.

Table IV
DIMENSIONS OF THE IN-DEPTH ANALYSIS

Bug ID	Trace attchmt. ID	Change summary	Editing style	Bug severity	# sessions	Duration of trace	Navigation method(s)	Size of patch/ check-in	Trace compared to patch/check-in
261136	122682	Removed an obsolete file which broke the tests	EDIT-FIRST	major	1	2 mins	Search View, Pkg Explorer	deleted 1 file	related extras
267399	127829	Fixed a synchronization problem with 2 UI views	EDIT-FIRST	major	1	39 mins	editor, Content Outline view	edited 1 file	related extras
215156	86777	Optimized code dealing with Web requests	EDIT-FIRST	major	1	18 mins	CVS Synch view	edited 5 files	incomplete
275884	135865	Changed a build file to distribute the code online	EDIT-THROUGHOUT	minor	1	within 1 min	Pkg Explorer	edited 1 file	equal
280811	143967	Improved a minor appearance issue in the UI	EDIT-THROUGHOUT	minor	1	15 mins	editor	edited 1 file	related extras
282445	141960	Improved a minor appearance issue in the UI	EDIT-THROUGHOUT	minor	1	35 mins	Pkg Explorer	edited 1 file	equal
174413	136079	Added of a pane in an existing view in the UI	EDIT-LAST	enh.	3	15 days	editor	added 1, edited 2	equal
277179	136568	Improved a minor appearance issue in the UI	EDIT-LAST	minor	1	within 1 min	editor	edited 1 file	incomplete
256774	119208	Fixed a bug in rendering non-UTF8 characters	EDIT-LAST	enh.	1	24 mins	Search View	edited 1 file	unrelated extras

Table V
BREAK-DOWN OF STYLE FOR REPORTS WITH A STACKTRACE OR NOT

	EDIT-FIRST	EDIT-LAST	EDIT-THROUGHOUT	total
Contain a stack trace	68	118	250	436
Does not contain	572	1136	2509	4217
total	640	1254	2759	

were typically simple and small changes which just needed to be typed up, rather than requiring exploration beforehand (a possible EDIT-LAST scenario) or code review after the edit (a possible EDIT-FIRST scenario). To verify this hypothesis, we used the Kruskal-Wallis ANOVA test [33] to determine whether the durations of tasks from the three groups—minor, major bug fixes, and enhancement tasks—were the same. We used this non-parametric test because the durations of the three groups did not follow a normal distribution. The test revealed that the three groups of durations were significantly different ($p = 2.2 \times 10^{-16}$). Post-hoc Wilcoxon tests [33] showed that the durations of enhancement tasks were significantly greater than minor bug fixes ($p = 2.2 \times 10^{-16}$) and also greater than major bug fixes ($p = 2.2 \times 10^{-16}$), but the difference between the durations of minor and major bug fixes were not significant ($p = 0.32$). Thus, our speculation that programmers took shorter time to fix minor bug fixes was partially right, as minor bug fixes indeed took shorter time than enhancement tasks but not significantly shorter than major bug fixes.

3) Enhancement tasks and EDIT-FIRST *style:* As we learnt from the residual analysis, enhancement tasks were negatively with EDIT-FIRST (results #1). This could be explained by the fact that enhancement tasks might require exploration throughout the trace. We noticed in one of the enhancement tasks (#174413) that spans a longer time-frame (15 days) that the timing of the edit events depended on a reply in the bug report.

V. DOES THE PRESENCE OF A STACK TRACE IN A BUG REPORT AFFECT EDITING BEHAVIOUR? (RQ2)

We explored relationships between editing styles of a task and the presence of a stack trace through chi-squared test in a similar way to our investigation of RQ1. Table V shows the contingency table. A chi-squared test revealed that the presence of a stack trace was independent from editing styles ($\chi^2 = 2.5$, $df = 2$, $p = 0.28$).

VI. DISCUSSION

We explore some of the implications in the software engineering context, threats to validity, and future work.

Implications for tool design

Our study provided some initial evidence that different types of tasks were associated with different editing styles (for example, major bug fixes are more likely to be associated with EDIT-FIRST styles). Since a large part of editing happens in a software development environment, our results have implications on the design of such environments. If we know the editing style of a task, we can dynamically configure the software development environment to present only the most relevant parts to the particular editing style. More concretely, for an EDIT-FIRST programming session, the development environment can dynamically show more of the editing related features, rather than the navigation related features. The general idea of adaptive capabilities of the development environment was inspired by several positions [19], [31], [34]. To enable this vision to work, we would need to build a *model* to predict editing styles, in contrast to finding statistical associations in our study. Building such a model would likely need factors other than the task type.

Threats to validity

Our study follows a quantitative design and exhibits the usual threats: construct validity, where our measure only approximates the phenomenon under study (programmer behaviour) and external validity. The main concern with external validity was that not all the bug reports in Eclipse contained a trace and not all developers who contributed to Eclipse submitted a trace. The only exception was the Mylyn sub-project. Strictly speaking, our data is representative of the Mylyn project. As an initial analysis, we believe that our data provides a good picture of the development habits in a subset of Eclipse projects. We explain additional threats that stem from our specific study environment:

Generalizability of interaction data collected by the Mylyn monitor: Since the purpose of Mylyn is to reduce the need of unnecessary navigation to program elements relevant to a task, we expect that the programming behaviour with using Mylyn has less navigation than the programming behaviour without using Mylyn [5]. The impact of using data collected by the Mylyn monitor, as opposed to similar data monitors, is that Mylyn forces programmers define tasks, and the storage of the code elements relevant to the task. From the point of view from a researcher mining the traces, these task-based data is great because there is no need to infer which part of the interaction history belongs to which tasks. This step is necessary when a monitor does not collect data based on tasks and there is prior work which has attempted to address this step [17]. Another feature of Mylyn that affects the data is that a programmer can copy a previous and similar task context to the current task the programmer is working on. In the trace, the events are copied without any meta-information that the copy operation was performed. We believe that this operation was not used enough to affect our results: we compare the content of all traces with each other and found only 1.4% (68) of the traces are contained in more than one other trace.

A trace might not have aligned with a task: A trace may not be an accurate representation of the programming associated with a change. For example, the trace of bug #215156 missed a key part of the change since changes to three methods that were in the patch were not edited according to the trace. This information is rendered as "incomplete" in the last column of Table IV. The other type of an inaccurate trace is one that contained unrelated program elements in the trace. The trace of bug #256774 contains such unrelated program elements in the beginning of the trace (marked as "unrelated extras" in the last column of Table IV). This could happen when a programmer inadvertently forgot to switch from a previous task, to fixing #256774. In addition, programmers who may seem to be inspecting the code might be taking a break. From our in-depth analysis, we see that a majority of the traces seem to be accurate.

Event aggregation: One of the goals of our study was to make use of interaction data already available. The particular type of data we looked at was collected by the Mylyn monitoring facility, which is lightweight and already used by many users as Mylyn is part of the standard Eclipse download. However, a major challenge in using this data was that an edit event in the trace could be an aggregate of multiple edits on the same program element. We attempted to address this issue when we defined the editing style: we considered an edit event as a time range and classified to the first or the second part of the trace based on whether the time range covered more of the first or the second part of the trace. In other words, we have implicitly assumed that the rate edit was uniform within the time range. However, this might not be a valid assumption.

A possible way to deal with this problem is to use the degree-of-interest value associated with a program element recorded in each event in the trace. The degree-of-interest value reflects how frequently and recently a particular program element has been interacted with in the duration of a task. If an element has a higher degree-of-interest value, we can assign a higher weight of the associated event in the definition of editing style. There are two issues with this approach. First, the degree-of-interest value captures both the number of interactions and the timing of the events. Thus, we cannot infer the number of interactions and the exact timing of an interaction just from the degree-of-interest value alone. Second, the degree-of-interest can also capture explicit interest declaration by a programmer: when a program element is deemed to be of relevance to the task by the programmer, or when deemed otherwise, through the Mylyn user interface.

An edit might not be a "real" edit: In our study, we have assumed, implicitly, that an edit event corresponded to some meaningful change in the code whereas a selection event corresponded to a step in the navigation or understanding of the code. However, an edit could serve as a means of understanding the code. For example, Davies observed that some programmers inserted debug statements to understand a program's execution [35]. We believe this is not a problem. Of the sample of traces we have examined in the in-depth analysis, none of which indicated an excessive number of unrelated files having been edited. Therefore, there does not seem to be evidence of this debugging practice in our data.

Future work

One promising area to study on the interaction history is how programmers navigate, which was studied in a prior study [23]. In the in-depth analysis of RQ1, we observed different ways programmers arrive at a file in Eclipse (the third last column in Table IV): through the editor, Package Explorer (which displays the Java package, class, and method structure), Content Outline View (which displays program elements such as methods and fields in a Java class), Search View (which displays search result through a textual search), and CVS Synchronize View (which displays the differences between the files in the workspace and those in CVS.[15]) In addition, not surprisingly, we observed differences in the program elements leading to a particular element: from the superclass, the structural parent, or a referenced element. It would be interesting to analyze this in a more quantitative manner and large scale using the traces.

[15]http://www.nongnu.org/cvs/

VII. Conclusion

We found significant differences among enhancement tasks, minor, and major bug fixes in terms of when edit events on program elements happen. Knowing such editing patterns could help software development tool designers in various contexts. For example, knowing a programming session being EDIT-FIRST, the development environment could show more of the editing related features, rather than the navigation related features. The data we used in our analysis were a recent form of software archive, over 4000 interaction traces that are attachments of bug reports. To our knowledge, this is the largest set of public editing traces of programmers. This data is a rich form of data for other researchers to use for empirical study purposes and to evaluate interaction history analysis.

For others wishing to replicate our study, we have made all the data and R scripts available.[16]

Acknowledgement

We thank Barthélémy Dagenais, Ekwa Duala-Ekoko, Thomas Fritz, and anonymous reviewers for their comments on earlier drafts of this paper, Prof. Jose Correa for his advice on the CMH statistic. This work was funded by fellowships from McGill and NSERC.

References

[1] B.W. Boehm, "Software Engineering," *IEEE Trans. on Computers*, no. 25, pp. 1226–1241, 1976.

[2] D.C. Littman, J. Pinto, S. Letovsky, and E. Soloway, "Mental models and software maintenance," *Journal of Systems and Software*, vol. 7, no. 4, pp. 341–355, 1987.

[3] M. Kersten and G. Murphy, "Mylar: a degree-of-interest model for IDEs," in *Proc. of Int'l Conf. on Aspect-Oriented Soft. Dev.*, 2005, pp. 159–168.

[4] G.C. Murphy, M. Kersten, and L. Findlater, "How are Java software developers using the Eclipse IDE?" *IEEE Software*, pp. 76–83, 2006.

[5] M. Kersten, "Focusing knowledge work with task context," Ph.D. dissertation, The University of British Columbia, 2007.

[6] E. Murphy-Hill, C. Parnin, and A.P. Black, "How we refactor, and how we know it," in *Proc. of the Int'l Conf. on Soft. Eng.*, 2009, pp. 287–297.

[7] C. Parnin and S. Rugaber, "Resumption strategies for interrupted programming tasks," in *Proc. of the Int'l Conf. on Program Comprehension*, 2009, pp. 80–89.

[8] R. DeLine, A. Khella, M. Czerwinski, and G. Robertson, "Towards understanding programs through wear-based filtering," in *Proc. of the Symposium on Software Visualization*, 2005, pp. 183–192.

[9] J. Singer, R. Elves, and M.A. Storey, "Navtracks: Supporting navigation in software maintenance," in *Proc. of the Int'l Conf. on Software Maintenance*, 2005, pp. 325–334.

[10] M.P. Robillard and G.C. Murphy, "Automatically inferring concern code from program investigation activities," in *Proc. of the Int'l Conf. on Automated Soft. Eng.*, 2003, pp. 225–234.

[11] C. Parnin and C. Görg, "Building usage contexts during program comprehension," in *Proc. of the Int'l Conf. on Program Comprehension*, 2006, pp. 13–22.

[12] R. Robbes and M. Lanza, "How program history can improve code completion," in *Proc. of the Int'l Conf. on Automated Soft. Eng.*, 2008, pp. 317–326.

[16]http://www.cs.mcgill.ca/ aying1/2011-icpc-editing-style-data.zip

[13] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Proc. of the Int'l Conf. on Software Maintenance*, 2003, pp. 23–32.

[14] L. Zou, M.W. Godfrey, and A.E. Hassan, "Detecting interaction coupling from task interaction histories," in *Proc. of the Int'l Conf. on Program Comprehension*, 2007, pp. 135–144.

[15] R. Robbes, D. Pollet, and M. Lanza, "Logical coupling based on fine-grained change information," in *Proc. of the Working Conf. on Reverse Engineering*, 2008, pp. 42–46.

[16] R. Robbes and M. Lanza, "Characterizing and Understanding Development Sessions," in *Proc. of the Int'l Conf. on Program Comprehension*, 2007, pp. 155–166.

[17] I.D. Coman and A. Sillitti, "Automated identification of tasks in development sessions," in *Proc. of the Int'l Conf. on Program Comprehension*, 2008, pp. 212–217.

[18] I. Safer and G.C. Murphy, "Comparing episodic and semantic interfaces for task boundary identification," in *Proc. of the Conf. of the Center for Advanced Studies on Collaborative Research*, 2007, pp. 229–243.

[19] M.A. Storey, "Theories, tools and research methods in program comprehension: past, present and future," *Software Quality Journal*, vol. 14, no. 3, pp. 187–208, 2006.

[20] R. Brooks, "Towards a theory of the comprehension of computer programs," *Int'l Journal of Man-Machine Studies*, vol. 18, no. 6, pp. 543–554, 1983.

[21] N. Pennington, "Stimulus structures and mental representations in expert comprehension of computer programs," *Cognitive Psychology*, vol. 19, no. 3, pp. 295–341, 1987.

[22] 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. on Soft. Eng.*, pp. 971–987, 2006.

[23] M.P. Robillard, W. Coelho, and G.C. Murphy, "How effective developers investigate source code: An exploratory study," *IEEE Trans. on Soft. Eng.*, vol. 30, no. 12, pp. 889–903, 2004.

[24] G.C. Murphy, P. Viriyakattiyaporn, and D. Shepherd, "Using activity traces to characterize programming behaviour beyond the lab," in *Proc. of the Int'l Conf. on Program Comprehension*, 2009, pp. 90–94.

[25] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proc. of the Int'l Symposium on Foundations of Soft. Eng.*, 2008, pp. 308–318.

[26] M. Kersten and G.C. Murphy, "Using task context to improve programmer productivity," in *Proc. of the Int'l Symposium on Foundations of Soft. Eng.*, 2006, pp. 1–11.

[27] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *Proc. of the Int'l Conf. on Automated Soft. Eng.*, 2007, pp. 34–43.

[28] L. Kaufman and P.J. Rousseeuw, *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons, 2005.

[29] A. Agresti, *Categorical data analysis*. John Wiley & Sons, 1990.

[30] A.J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *Proc. of the Int'l Conf. on Soft. Eng.*, 2007, pp. 344–353.

[31] M.P. Robillard and G.C. Murphy, "Program navigation analysis to support task-aware software development environments," in *Proc. of the Workshop on Directions in Soft. Eng. Env.*, 2004, pp. 83–88.

[32] T. Fritz, J. Ou, G.C. Murphy, and E. Murphy-Hill, "A degree-of-knowledge model to capture source code familiarity," in *Proc. of the Int'l Conf. on Soft. Eng.*, 2010, pp. 385–394.

[33] D. Sheskin, *Handbook of parametric and nonparametric statistical procedures*. Chapman & Hill/CRC, 2004.

[34] G.C. Murphy, M. Kersten, M.P. Robillard, and D. Čubranić, "The emergent structure of development tasks," in *Proc. of the European Conf. on Object-Oriented Programming*, 2005, pp. 33–48.

[35] S.P. Davies, "Models and theories of programming strategy," *Int'l Journal of Man-Machine Studies*, vol. 39, no. 2, pp. 237–267, 1993.