Capturing and Analyzing Low-Level Events from the Code Editor

YoungSeok Yoon
Institute for Software Research
Carnegie Mellon University
Pittsburgh, PA, USA
youngseok@cs.cmu.edu

Brad A. Myers
Human-Computer Interaction Institute
Carnegie Mellon University
Pittsburgh, PA, USA
bam@cs.cmu.edu

Abstract

In this paper, we present FLUORITE, a publicly available event logging plug-in for Eclipse which captures all of the low-level events when using the Eclipse code editor. FLUORITE captures not only what types of events occurred in the code editor, but also more detailed information such as the inserted and deleted text and the specific parameters for each command. This enables the detection of many usage patterns that could otherwise not be recognized, such as "typo correction" that requires knowing that the entered text is immediately deleted and replaced. Moreover, the snapshots of each source code file that has been opened during the session can be completely reproduced using the collected information. We also provide analysis and visualization tools which report various statistics about usage patterns, and we provide the logs in an XML format so others can write their own analyzers. FLUORITE can be used for not only evaluating existing tools, but also for discovering issues that motivate new tools.

Categories and Subject Descriptors D.2.6 [**Software Engineering**]: Programming Environments – integrated environments.

General Terms Human Factors

Keywords event logger; code editor; editing strategies

1. Introduction

To evaluate usability of certain programming language features or tools, it is important to know when and how the developers actually use them. There exist several different methods for gathering usage data, we have not found one that is suitable for analyzing fine-grained code editing history without requiring laborious manual inspection. In order to address this limitation, we built a publicly available event logging plug-in for Eclipse called FLUORITE[1]. FLUORITE keeps track of all of the events that occur in the code editor and saves the log files in XML format.

The granularity of events that FLUORITE logs is very fine since it logs character typing, moving the text cursor, changing the selected text, and all other Eclipse commands executed in the code editor. FLUORITE not only logs the command IDs but also the parameters to those commands. For example, the Find command has additional searchText and replaceText parameters. In the case of text editing events, the inserted or deleted text is also recorded.

What makes FLUORITE unique is that FLUORITE's time-stamped and detailed event logs enable us to analyze the developers' complex code editing strategies which are often composed of *sequences* of commands. For example, we saw in our logs that backspace was 12.41% of all the keystrokes in code editing, and it is often used in sequences of more than four backspaces in a row (4.35 on average) generally used to fix typos or rename variables. This type of analysis cannot be done using the types of usage data available from the Eclipse Usage Data Collector (UDC) [1] or other available logging tools (see section 2).

Using the snapshots of the initial source files and the deleted / inserted text from all the commands, it is possible to completely reproduce any file snapshot at any given time. This enables us to know in which situation a command was executed.

FLUORITE is useful for many different purposes. First, it can be used in lab studies or field studies for evaluating existing tools. FLUORITE logs can be used to detect and measure the time for various usage patterns or events of interest, without needing the experimenter to manually

[1] **F**ull of **L**ow-level **U**ser **O**perations **R**ecorded **I**n **T**he **E**ditor

annotate a videotape. FLUORITE can also be useful for motivating new tools. Ko *et al.* laboriously hand-analyzed videotapes of code editing in their study of Eclipse editing [13], and showed that people spend significant time scrolling, which motivated interesting new tools. FLUORITE will provide an easier way to get such data, and thus might help motivate other ideas for new tools that would help programmers in the future. In addition, we anticipate in the future that FLUORITE's logging and analysis may be used in real-time to support novel code editing operations that will depend on the history.

2. Related Work

There are many different sources of developers' usage data each with its own strengths and weaknesses. In many cases, FLUORITE can be used to complement the shortcomings of those methods. One way is to directly ask the developers who regularly use the target programming language or tool through interviews or surveys. Although these methods are effective and the investigators can get useful insights about the target feature, the responses from the subjects may not be reliable. For instance, many operations are performed quite automatically by the developers, so it is possible that they could report that they use a feature a lot but could not remember the specific occasions.

Another way of gathering usage data is performing contextual inquiries or experiments in lab settings. Often, the participants are required to think aloud while performing their tasks, and their screen and voice are recorded for further analyses. However, the experimenter must then manually inspect the videotape (as was done in [6, 13, 15, 16]) in order to analyze the results, which can be time-consuming and error-prone.

Usage data can also be obtained by mining software repositories and their revision histories. Many researchers have been used this method to gain insights about code clones [4, 5, 12] and how the developers refactor [11, 18, 21]. There is plenty of available data in the open source software repositories and from industry, and the data can be analyzed automatically. One problem with this method is that we still cannot know what events happened between two consecutive revisions. Instead, we can only infer what types of commands the developers might have used to change the code from one revision to the next.

Mylyn keeps track of the user interaction history internally in order to derive the task context [9, 19]. Using the Mylyn Monitor API [2], investigators can retrieve the user interaction data for their own analyses. FLUORITE differs from the Mylyn Monitor in that FLUORITE focuses more on the details of the user interaction in the code editor, whereas Mylyn Monitor collects more abstract user interaction data on the entire IDE. For example, when the developer selects a class from the package explorer, Mylyn Monitor logs that there was a selection event from the package explorer with the name of the selected class, whereas FLUORITE logs exactly which file was opened, and the offset and length of the highlighted text (i.e., the name of the class) in the file.

The Eclipse Usage Data Collector (UDC) is another useful source of developers' Eclipse usage data [1]. The UDC collects usage information from all the Eclipse users all over the world who consented to upload their usage data to the UDC. The UDC provides several usage reports including the commands report. These reports have been used by many researchers [18, 20]. However, the commands report from UDC can be misleading because it fails to capture some important commands executed in the code editor. It ignores many of the most frequent keyboard commands such as navigating source code with arrow keys and deleting the previous character with the backspace key because they are not explicitly bound as shortcuts by default. In contrast, FLUORITE collects all commands regardless of the use of their shortcut keys. Also, with UDC, you cannot find out anything about sequences of commands or specific parameters of commands, since UDC only reports the number of occurrences of each command.

Syde and Replay are tools for Eclipse that can record fine-grained change history of Java-based systems in multi-developer settings [8]. These tools are intended to help developers understand the code's evolution, but they could

```
<Command __id="2" _type="MoveCaretCommand" caretOffset="142" docOffset="142" timestamp="3977"/>
<Command __id="3" _type="EclipseCommand" commandID="eventLogger.styledTextCommand.SELECT_LINE_DOWN"
timestamp="5598"/>
<DocumentChange __id="4" _type="Delete" docASTNodeCount="22" docActiveCodeLength="125" docExpression-
Count="10" docLength="151" endLine="9" length="39" offset="142" startLine="8" timestamp="7186">
  <text>
    <![CDATA[     System.out.println("Hello World!");

]]>
  </text>
</DocumentChange>
<Command __id="5" _type="EclipseCommand" commandID="org.eclipse.ui.edit.delete" timestamp="7202"/>
<Command __id="6" _type="EclipseCommand" commandID="org.eclipse.ui.file.save" timestamp="8099"/>
```

Figure 1. Example log generated by FLUORITE. The developer (1) moved the cursor by clicking mouse button, (2) selected one line by SHIFT + DOWN, (3) deleted selected code using the DELETE key, and (4) saved the file. Each event has its own parameters, and the whole deleted text is listed in DocumentChange event.

Event Type	Name	Description
Command	MoveCaret	Move cursor using the mouse
	SelectText	Select (highlight) text
	Find	Find, & Find & Replace
	InsertString	Type new text
	Run	Run/Debug the application
	FileOpen	Open or activate a new file
	Assist	Quick fix/Content assist
	Eclipse	All other Eclipse commands
Document Change	Insert	Text insertion
	Delete	Text deletion
	Replace	Deletion & insertion in one step
Annotation	Annotate	Manual annotation by the user

Figure 2. Complete list of the different types of events

be used to track the editor usage as well. Syde differs from FLUORITE in that it records changes at the abstract syntax tree (AST) level, not the textual level. Also, it only records the operations which modify the AST, and so, for example, the `SelectText` command will not be recorded by Syde.

3. FLUORITE Implementation

FLUORITE is implemented as an Eclipse plug-in because Eclipse is one of the most widely used integrated development environments (IDEs). We based our code off of an open source Eclipse plug-in called Practically Macro [3], but it was not sufficient for our purposes because some important commands and parameters were missing (e.g. the FileOpen command), and it was not stable enough to record long sessions. Therefore, we augmented it to record all the commands and their parameters, increased its stability, and added the capability of capturing inserted and deleted text.

Once FLUORITE is installed on Eclipse, it begins to capture all the low-level events occurring in the code editor, and saves the transcript as an XML file when Eclipse is closing. An example transcript is shown in Figure 1.

3.1 Types of logged events

There are three different types of events that FLUORITE logs: *commands*, *document changes*, and *annotations*. The full list of different types of events is shown in Figure 2.

A *command* is an event directly invoked by a user's action. This includes typing new text, moving the cursor position or selecting text by keyboard or mouse, along with all editor commands such as copying, pasting, and undoing.

A *document change* event is logged whenever the active file is changed by any executed command. Each document change event contains the actual deleted or inserted text. This is needed because we cannot correctly reproduce the snapshots of the files by capturing only the commands. For example, when the developer copies a code fragment from a web browser and pastes it into the code editor, there is no way to find out what the pasted code was if we have only the command history. In addition, this simplifies the way we get the actual change results for each command: we can

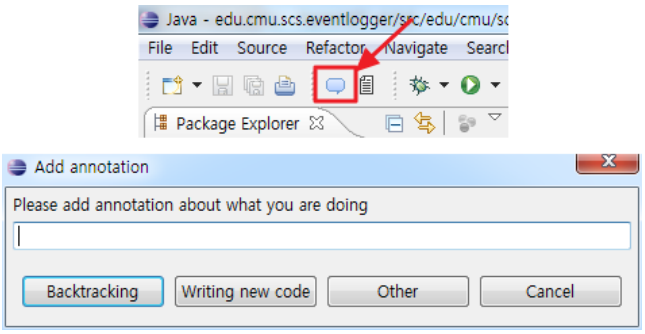

Figure 3. Annotation toolbar button and the dialog

just read the preceding[2] document change event for each command. There can be multiple document change events triggered by a single command (e.g., find and replace), and even no document changes if a command does not change any of the code content.

An *annotation* is logged when the developer wants to add an annotation at a given time to provide information to the investigator about the current activity. FLUORITE adds a toolbar button to Eclipse for adding annotations as shown in Figure 3, and a simple dialog box for inserting annotation pops up when the button is clicked. The buttons at the bottom of the window provide a quick way for users to identify certain events of interest.

3.2 Parameters

Each event is logged as an XML element, and the parameters for each event are logged as either attributes or sub-elements. There are a few parameters common to every event (Figure 4) and there are also event-specific parameters. For example, the MoveCaret command has the resulting cursor position as an offset from the beginning of the document, and the Find command has searchText and replaceText parameters. Also, every document change event has a few code size metrics (Figure 5), in order to

Parameter	Description
id	Unique ID (sequentially incremented)
type	Corresponds to the "Name" column
timestamp	Timestamp relative to the session start time
timestamp2	(*optional*) Timestamp of the last merged event
repeat	(*optional*) Number of events merged together

Figure 4. List of common parameters

Metric	Description
Code Length	Code length in # of characters
Active Code Length	[Code length] - [Comment length]
AST Node Count	# of all the AST nodes
Expression Node Count	# of all the expression nodes in AST

Figure 5. Currently logged code size metrics

[2] Currently, a document change event *precedes* the causing command rather than following it, due to the event handling order of the Eclipse code editor.

keep track of the code size changes.

3.3 Merging events

In order to prevent the log files from being unnecessarily large, FLUORITE merges multiple events of the same type in a row whenever possible. For instance, when the developer moves the cursor to ten lines by holding down the up arrow key, the ten events are merged together as one XML element and the repeat parameter is set to 10. In some cases, some of the parameters must be merged as well. For example, when merging multiple InsertString commands which represent typing new text, the data parameter must be merged so as not to lose important information. Two consecutive events are merged only if their time difference is no greater than the specified threshold (2 sec. by default).

4. Example Analyses & Results

We also built a FLUORITE analyzer which can produce several types of analysis reports and visualize them. Since the FLUORITE log files are written in XML format, people can implement their own analyzers as well.

We used FLUORITE during a lab study of 12 students performing some small editing tasks for about 2 hours each. These tasks used the Paint program from [7, 13], and had users add four new features. Some of the more interesting patterns and results are summarized next.

4.1 Code editing pattern detection

It is possible to detect various code editing patterns which are composed of sequences of commands. As an example, our analyzer can detect *fixing typo patterns* from the logs. Some fixing typo patterns can be detected by looking at three consecutive document change events as follows: 1) Any Insert event, 2) a Delete event whose deletion range is somewhere inside the previous Insert event, 3) an Insert event whose starting position is the same as that of the previous Delete event. Figure 6 shows a few sample fixing-typo patterns detected by this algorithm. Some of the detected patterns are not merely typo corrections. For example, the pattern starting from ID 1061 in Figure 6, we can see that the developer decided to declare an array instead of declaring multiple variables.

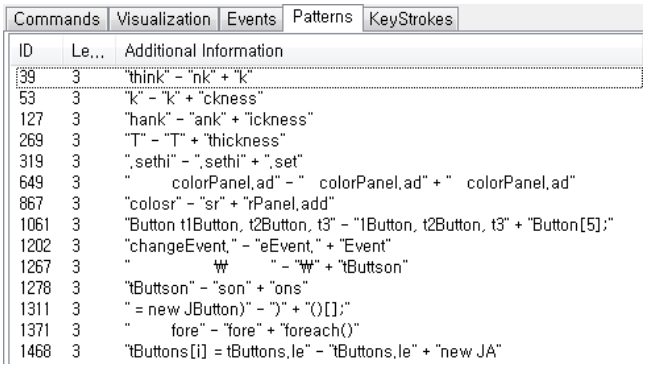

Figure 6. Example of detected fixing typo patterns. A pattern is represented in the form of "originally typed text" - "deleted text" + "newly typed text". The ID column indicates the ID of the event where the pattern starts so the investigator can jump to the events list and see what was happening around that time.

It is important to note that this kind of fine-grained editing pattern detection cannot be easily done with the data that comes from other types of tools. Generalizing this pattern detection algorithm so that it can detect more complicated patterns remains as part of our current work.

4.2 Code length graph

Since several code size metrics are logged whenever a document change event occurs, it is possible to plot the code size over time either for each file or as a whole. Currently supported metrics are listed in Figure 5. From the logs we collected, we have noted that the *code length* graph and the *active code length* graphs differ significantly, which indicates that developers often comment out or uncomment code. We can also see some interesting editing trends in the graphs. In Figure 7, the steadily increasing part indicates that the developer was typing new code, small fluctuations mean the developer was doing a small experiment or fixing minor mistakes, and a big, sudden change means commenting / uncommenting a block of code or cutting and pasting.

If there is an interesting place on the graph and we want to investigate more thoroughly to see what was happening, we can jump to an events-list view by double clicking the point on the graph. This brings up the event-list viewer that shows the full list of events and their parameters. The event whose timestamp is closest to the selected point on the graph is highlighted. In the events list view, we can filter the events and see only the types of events we are interested in, and also see all the detailed parameters.

4.3 Keystroke distribution report

The keystroke distribution report gathers all the keystroke data from the logs and draws a pie chart showing the frequency of various types of keystrokes. In our study, there were a total of 45,872 keystrokes, and the five most frequent keystrokes were down arrow (12.64%), backspace (12.41%), up arrow (9.80%), right arrow (7.82%), and left arrow (6.00%), respectively. Although this data may be exaggerated because FLUORITE logs multiple instances of the same event when the developer holds down a key so it auto-repeats, it is still interesting to see that developers navigate a lot within a file using arrow keys. This result is consistent with Ko *et al.*'s observation [13] that developers spend about 16% of their time navigating dependencies.

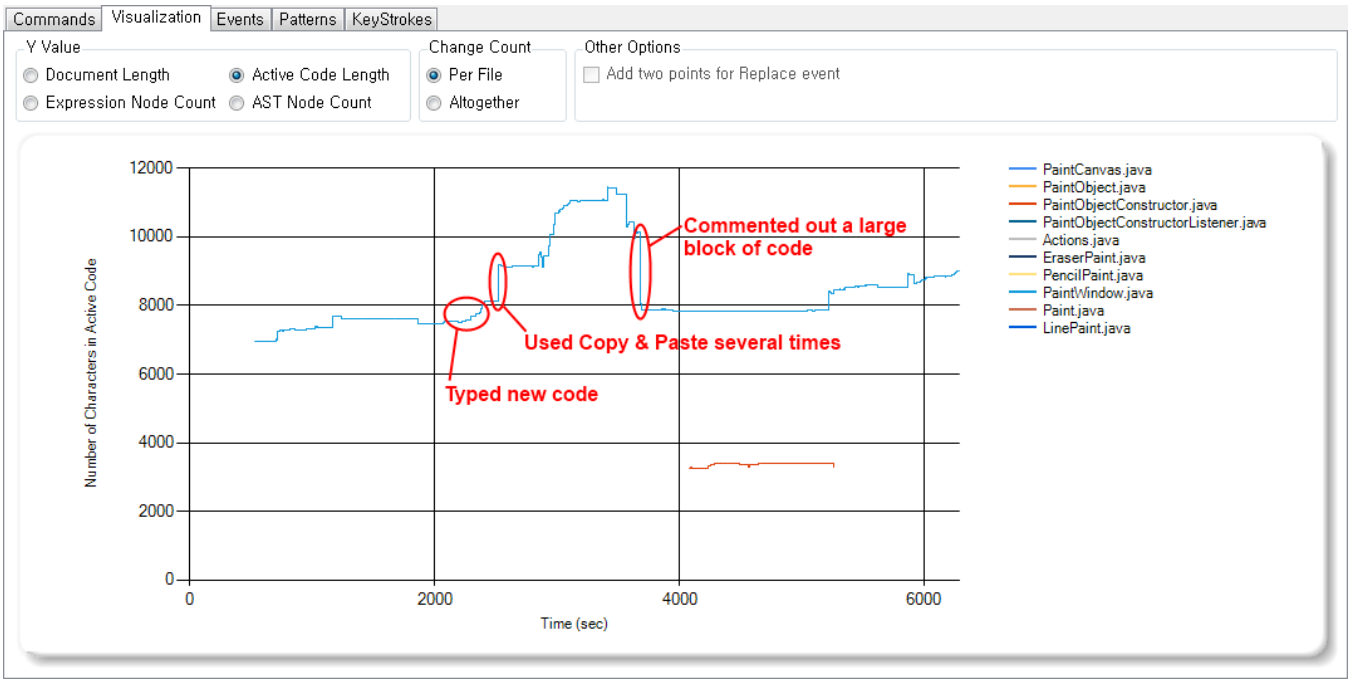

Figure 7. Example active code length graph drawn from one of our logs by the FLUORITE analyzer. We marked some interesting points using ellipses and the corresponding code editing strategies are described. Y-axis value can be one of the metrics described in Figure 5. Only line graphs of the files that have been changed during the session are drawn. The graph can be zoomed with the mouse scroll wheel, and the user can double click on a point to jump to the events-list view.

Another interesting observation is that developers heavily use the backspace key in the code editor. This seems to be a lot higher than the percent of backspacing in regular typing, for example, as in MacKenzie and Soukoreff's report that 7.10% of keystrokes were backspaces [17]. This provides further evidence, as mentioned in [14], that editing code is different than editing documents.

4.4 Command distribution report

The command distribution report differs from the keystrokes in that it focuses on Eclipse commands rather than just keystrokes. It is pretty close to the commands report of the Eclipse Usage Data Collector (UDC) [1], but FLUORITE also includes the commands missing from UDC. Consistent with the keystroke report, the five most frequent commands were InsertString (31.48%), down arrow (10.67%), backspace (10.48%), MoveCaret (using the mouse) (8.63%), and up arrow (8.27%), respectively. The proportion of backspaces is very large here as well, but backspace is not included in the UDC command report at all since backspace commands are ignored in the UDC logs.

5. Other Issues

During our exploratory study, there was no measurable performance loss caused by FLUORITE. This could be an important issue since FLUORITE would be inappropriate for field studies if it significantly slowed down the IDE.

The average log file size growth rate calculated from our study is 236.8KB/hour. Assuming that a developer works for 40 hours a week, that comes out to be 9.25MB/week, which is not terribly large given that the typical hard disks have hundreds of gigabytes. If FLUORITE compressed the log files as the Mylyn Monitor does [2], it could be reduced to approximately 1MB/week[3].

Since the FLUORITE log files contain the actual source code in them, it should not be used in a situation where the source code is confidential. FLUORITE does not upload log files automatically however, so in a field study the investigator can ask the study participant to send the log files whenever the code being edited is not confidential.

6. Future Work

Though FLUORITE is started from a macro plug-in, it does not currently support log replay functionality, due to the significant changes of the internal architecture. With a replayer as in [8], FLUORITE could be used as an alternative for video recording in the case where the subjects' voice or the usage data outside the code editor is not needed. Moreover, as Kim *et al*. claimed [10], a replayer might significantly reduce the effort for analyzing coding behavior.

[3] Estimated by concatenating all the log files and compressing it to a zip.

FLUORITE enables us to detect complex code editing patterns from the log files as described in Section 4.1. It is also possible to implement customized detection algorithms for each pattern in which an experimenter is interested, but it would be even better if there would be an easier way of detecting such patterns rather than writing code.

One of the limitations of the current version of FLUORITE, is that it cannot tell how the commands were executed (e.g., to distinguish whether a command was invoked using a keyboard key, a menu item, or an icon). We speculate that FLUORITE might be useful for analyzing usability problems if it were possible to distinguish the commands executed by keyboard shortcut from the ones executed by clicking menu items for example.

Finally, providing FLUORITE for other popular IDEs is a possible direction of future work. This would enable the collection of usage data from a larger developer pool and looking at issues that cross IDEs and languages. Since most the code editors share common functionalities, it would probably be possible to standardize the log format.

7. Conclusion

We are developing the FLUORITE logger and analyzer in the hopes that it will be useful to the community for when detailed analyses of programmers' edits are required. In our use of the current version, we have already uncovered a number of interesting results, and we plan to continue to develop and refine the tools, and hope that by providing FLUORITE as an open-source plug-in, available at http://www.cs.cmu.edu/~fluorite/, that it will also be useful for the community.

Acknowledgments

Funding for this research comes in part from the Korea Foundation for Advanced Studies (KFAS) and in part from NSF grant CCF-0811610. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of KFAS or the National Science Foundation.

References

[1] Eclipse Usage Data Collector (UDC), http://www.eclipse.org/org/usagedata/.

[2] Mylyn/Integrator Reference, http://wiki.eclipse.org/Mylyn/Integrator_Reference.

[3] Practically Macro Eclipse plug-in, http://sourceforge.net/projects/practicalmacro/.

[4] Aversano, L., Cerulo, L. and Di Penta, M. 2007. How Clones are Maintained: An Empirical Study. In *Proc. 11th European Conf. on Soft. Maint. and Reengineering* (CSMR'07). 81-90.

[5] Bettenburg, N., Weyi, S., Ibrahim, W., Adams, B., Ying, Z. and Hassan, A. E. 2009. An Empirical Study on Inconsistent Changes to Code Clones at Release Level. In *Proc. 16th Working Conf. on Reverse Eng.* (WCRE'09). 85-94.

[6] Coman, I. D. and Sillitti, A. 2008. Automated Identification of Tasks in Development Sessions. In *Proc. 16th IEEE Int'l Conf. on Program Comprehension* (ICPC'08). 212-217.

[7] Fogarty, J., Ko, A. J., Aung, H. H., Golden, E., Tang, K. P. and Hudson, S. E. 2005. Examining task engagement in sensor-based statistical models of human interruptibility. In *Proc. CHI'2005*. 331-340.

[8] Hattori, L., D'Ambros, M., Lanza, M. and Lungu, M. 2011. Software Evolution Comprehension: Replay to the Rescue. In *Proc. 19th IEEE Int'l Conf. on Program Comprehension* (ICPC'11). 161-170.

[9] Kersten, M. and Murphy, G. C. 2006. Using task context to improve programmer productivity. In *Proc. 14th ACM SIGSOFT Int'l Symp. on Foundations of Soft. Eng.* (FSE'06). 1-11.

[10] Kim, M., Bergman, L., Lau, T. and Notkin, D. 2004. An ethnographic study of copy and paste programming practices in OOPL. In *Proc. Int'l Symp. on Empirical Soft. Eng.* (ISESE'04). 83-92.

[11] Kim, M., Cai, D. and Kim, S. 2011. An empirical investigation into the role of API-level refactorings during software evolution. In *Proc. 33rd Int'l Conf. on Soft. Eng.* (ICSE'11). 151-160.

[12] Kim, M., Sazawal, V., Notkin, D. and Murphy, G. 2005. An empirical study of code clone genealogies. In *Proc. 10th Euro. Soft. Eng. Conf. & 13th ACM SIGSOFT Int'l Symp. on Foundations of Soft. Eng.* (ESEC/FSE'05). 187-196.

[13] Ko, A. J., Aung, H. and Myers, B. A. 2005. Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks. In *Proc. 27th Int'l Conf. on Soft. Eng.* (ICSE'05). 126-135.

[14] Ko, A. J., Aung, H. H. and Myers, B. A. 2005. Design requirements for more flexible structured editors from a study of programmers' text editing. In *Proc. Extended Abstracts of* CHI'2005. 1557-1560.

[15] Ko, A. J. and Myers, B. A. 2003. Development and evaluation of a model of programming errors. In *Proc. IEEE Symp. on Human Centric Computing Languages and Environments* (HCC'03). 7-14.

[16] Ko, A. J. and Myers, B. A. 2004. A framework and methodology for studying the causes of software errors in programming systems. *J. Visual Languages & Computing*, 16, 1-2, 41-84.

[17] MacKenzie, I. S. and Soukoreff, R. W. 2002. Text entry for mobile computing: Models and methods, theory and practice. *Human-Computer Interaction*, 17, 2, 147-198.

[18] Murphy-Hill, E., Parnin, C. and Black, A. P. 2009. How we refactor, and how we know it. In *Proc. 31st Int'l Conf. on Soft. Eng.* (ICSE'09). 287-297.

[19] Murphy, G. C., Kersten, M. and Findlater, L. 2006. How are Java software developers using the Elipse IDE? *IEEE Soft.*, 23, 4, 76-83.

[20] Parnin, C. and Rugaber, S. 2009. Resumption strategies for interrupted programming tasks. In *Proc. 17th IEEE Int'l Conf. on Program Comprehension* (ICPC'09). 80-89.

[21] Xing, Z. and Stroulia, E. 2006. Refactoring Practice: How it is and How it Should be Supported - An Eclipse Case Study. In *Proc. 22nd IEEE Int'l Conf. on Soft. Maintenance* (ICSM'06). 458-468.