An Exploratory Study of Backtracking Strategies Used by Developers

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*—Developers frequently backtrack while coding. They go back to an earlier state by removing inserted code or by restoring removed code for various reasons. However, little is known about when and how the developers backtrack, and modern IDEs do not provide much assistance for backtracking. As a first step towards gathering baseline knowledge about backtracking and designing more robust backtracking assistance tools in modern IDEs, we conducted an exploratory study with 12 professional developers and a follow-up online survey. Our study revealed several barriers they faced while backtracking. Subjects often manually commented and uncommented code, and often had difficulty finding relevant parts to backtrack. Backtracking was reported to be needed by 3/4 of the developers at least "sometimes".**

***Keywords*-component; undo; exploratory programming**

I. INTRODUCTION

When developers write source code, it is unrealistic to expect them to complete the whole task on the first attempt without making any mistakes. Instead, there are various reasons why developers often have to backtrack while coding. By "backtrack," we mean when users go back at least partially to an earlier state either by removing inserted code or by restoring removed code, and not an algorithm for solving constraint satisfaction problems in the artificial intelligence area. For example, developers fix typos and correct minor mistakes, and they try out different values for parameters to methods. At a higher level, when developers try to learn an unfamiliar API, they often try writing some code and running it to see if the code works as expected, and backtrack if it does not. In some situations, developers will program in an exploratory manner. They quickly build prototypes that meet the known requirements of the system. If the prototypes fail in some way or uncover any fundamental flaws of the requirements, they backtrack and refine the requirements [1, 2]. Often, the problems themselves are ill-defined [3, 4, 5]. For these problems, there is no single correct solution, but there are several alternative solutions with their own strengths and weaknesses. In order to evaluate each solution, the developer might implement one, backtrack, and implement another.

Prior research has shown that developers do backtrack a significant amount while coding, much more than people do during the text editing of regular documents [6, 7, 8]. One way to measure the frequency of backtracking is to count the text editing commands related to backtracking, such as delete, undo, and the toggle-comment commands executed in the code editor. The Eclipse Usage Data Collector (UDC) keeps track of the usage of commands executed by all the Eclipse users who have consented to provide their usage data [9]. According to the UDC data collected from Jan. 2009 through Jan. 2010, the delete command is the most frequently executed command among all the commands executed in the code editor (at 15.32% of all commands). The undo command was 7th (4.26%). Our own study data also supports this. We found that the backspace keystrokes were 12.41% of all the keystrokes made in the code editor [10]. Note that this is a much higher percentage for backspace compared to normal document editing (e.g., 7.10% in [8]). Murphy et al. also reported that delete was the most frequently executed command in their study [11].

Despite the frequency of backtracking in development contexts, modern IDEs do not provide much support. For example, there are no sophisticated undo mechanisms other than restricted linear undo model [12], which has several shortcomings. A developer cannot easily undo the changes that they made some time ago, but only can undo the most recent changes in the command history. Also, when the developer undoes several steps backwards and makes a new change from that point, all the previously undone commands are discarded and cannot be redone, because the undo model does not keep the command history tree but only keeps a linear list. Another way IDEs help with backtracking is to provide a version control system (VCS), but this is not always adequate for several reasons. It only works if the user thought to commit the desired version, which may not always be the case, and it is often too heavy-weight for small experiments. And neither Undo nor a VCS helps when the users undesired and desired changes are intermixed.

As a first step towards supporting more robust backtracking in modern IDEs, it would be helpful to first know more about when and how developers backtrack when they write source code. However, to the best of our knowledge, there has been no thorough study about backtracking in the software development context. To gather baseline knowledge about backtracking, we conducted an exploratory lab study with 12 developers to see when and how they backtracked. We observed several backtracking patterns that the developers used, and what types of barriers they faced while they backtracked. Then, to gather more feedback from professional developers in general, we also conducted a follow-up online survey about the backtracking

situations they face and what types of strategies they use in order to solve those problems. A total of 103 developers provided some information and 48 completed the entire survey. The survey results confirmed that developers backtrack frequently and further provided us with information on the problems they have while backtracking.

II. Related Work

A. Study of Source Code Editing

There have been general studies about programmers' code editing strategies, but not for backtracking specifically. Kim et al. studied copying and pasting in the programming context [13]. Ko et al. analyzed programmers' character level code-editing strategies [14]. In that study, comment edits were 3% of all edits, and 60% of the comment edits were for temporarily commenting out code. Empirical studies on software evolution (e.g. refactoring [15, 16]) also focus on how developers make changes to code over time, but they are often limited to revision-level changes.

B. Undo Mechanisms

One way to support backtracking is with undo. Selective undo has been well studied in the graphical user interface (GUI) context [12, 17, 18]. These research systems allow users to undo the last change on a specific object or to select any command in the command history list. The Emacs editor adds the undo commands themselves to the end of the command list, which allows users to backtrack to any previously visited state. Emacs also supports undo in region, which selectively undoes the most recent change in the currently highlighted region, and the command history can be displayed as a tree [19].

However, these sophisticated undo mechanisms have not proven popular for code editors. One reason is that it is difficult to provide a meaningful thumbnail view of source code so users can determine where to go back to. Also, the code editing commands are too numerous and complex to be easily displayed in a command history list box where the user can choose one of the commands on the list.

C. Variation Management

Backtracking becomes important when trying out multiple alternative solutions. There exist several tools that help with variation management. A version control system (VCS) can be seen as a temporal variation management system that helps developers to revert a file or a set of files to an older version whenever something goes wrong with an experiment. However, there are many cases where a VCS cannot directly help with backtracking. As mentioned above, the user must think to commit the desired version, which may not happen if the developer realizes that backtracking is needed later. It may not be even possible to use a VCS to commit a certain variation when that version contains unstable code, which is likely to be the case during an exploration. Distributed VCSs such as Git can mitigate this problem somewhat by allowing users to clone the repository locally and experiment within the local clone, although committing is still a fairly heavyweight process. Also, a VCS cannot help the developer to selectively undo a large body of code grouped not temporally but logically.

There are a few other systems such as Juxtapose [20] and Parallel Pies [5], which facilitate design exploration by providing ways of adding alternatives at any time and moving among the alternatives. However, users must know in advance when they want to add variations in Juxtapose, and Parallel Pies works only in the graphical editing context.

Other work has studied ways to manage source code variations. Choice calculus provides a generalized representation for software variations at the source code level and avoids redundancies as much as possible [21]. Barista [22] had an alternative expressions tool which allows developers to select an alternative by clicking on one of the listed choices, but it was restricted to the expression level.

III. Lab Study

In order to study when and how developers backtrack using today's tools, and to identify barriers that developers face when they backtrack, we conducted an exploratory study in a controlled lab environment. We recruited 12 graduate students from Computer Science at Carnegie Mellon University. The participants were required to 1) have professional development experience or at least two internships as a software developer, and 2) be comfortable programming in Java. Of the 12 participants, 11 were male and 1 was female. Their average age was 24.8 years, and they had been programming for 5.5 years on average.

Participants were asked to finish two pairs of feature adding tasks in about 2 hours and to think aloud. The editing screens and their voice were recorded for further analyses. In addition, we developed and used an event logging plug-in for Eclipse called Fluorite [10] to capture all the low-level editing events. Subjects used the Eclipse IDE version 3.6.2 (Helios) on a laptop running Windows 7. They were told that they could use any Internet resources they wanted, and all the subjects made heavy use of Google and JavaDoc.

After completing the tasks, the participants were asked to fill out a post-survey questionnaire about their demographics and some the backtracking situations and strategies. We used the responses when designing our online survey questions. The participants were paid $30 for their efforts.

A. The Paint Program

We used a *Paint* program as the code base of our study, which has been previously used by other researchers [23, 24]. This is a simple Java Swing based drawing application composed of 10 Java files and a total of 452 lines of code.

Using the *Paint* program as our code base had several advantages. First, GUI development tends to be exploratory (i.e. involves extensive experiments with code), which means that the developers would often need to backtrack. Second, it had been shown by the previous studies that the code size is small enough to be understood and modified in a fairly short amount of time.

B. Features

The participants were asked to add new features to the *Paint* program. Because we wanted to get as much

TABLE 1. Study settings. Each participant did the tasks from top to bottom in sequence.

Step	Group 1 (7 subjects)	Group 2 (5 subjects)
Begin	Introduction	
Task1	F1-1	F2-1
Task2	F1-2	F2-2
Task3	Back to F1-1	Back to F2-1
Task4&5	F2-1 & F2-2	F1-1 & F1-2
End	Post-study questionnaire	

backtracking data as possible in 2-hour lab study, we designed the tasks so that they would lead the developers to backtrack regardless of any occurrences of their own exploration. To achieve this goal, we set up an imaginary scenario where a whimsical boss first asks the participants to implement a feature, changes his mind after testing the feature and asks them to implement the same functionality using a different user interface element. Because it does not make much sense to provide two different user interfaces for the same functionality, the participants were required to backtrack out of the first implementation to some extent. Starting over was not a good option however, because the first and second versions shared some code that the participants had to write, and only differed in the user interface part.

There were two different features to implement: *thickness control* (F1) and *x, y coordinates indicator* (F2). Each feature had two different user interfaces. The thickness control had to be implemented using a slider widget (F1-1) and then using a menu of buttons which preview the desired thicknesses (F1-2). The x, y coordinates indicator had to be located on a status bar at the bottom of the application window (F2-1) or in a modeless tool window which can be moved by the user (F2-2).

The study procedure and group settings are shown in Table 1. Another issue we wanted to study was whether the developers would behave differently if they knew they might need to backtrack later. Therefore, we first asked them to implement one of the features F_A-1, without telling them they might have to backtrack later. Then, they were asked to implement F_A-2 instead. Next, in order to see how they would restore the previous version, we asked them to go back to F_A-1 implementation. Finally, we gave them *both* F_B-1 and F_B-2 simultaneously and asked them to implement one at a time, using any strategy they wanted. We randomized whether participants used Feature 1 as F_A and Feature 2 as F_B (Group 1) or vice versa (Group 2).

C. FLUORITE Logger for Eclipse

To capture all the low-level code editing events, we developed and used FLUORITE[1], our event logging plug-in for Eclipse [10]. FLUORITE logs every command executed in the code editor including typing new text, copying and pasting, undoing and redoing, and logs all of the deleted and inserted text along with their timestamps. Using this data, we could detect several code editing patterns composed of sequences of commands which are closely related to backtracking.

[1] http://www.cs.cmu.edu/~fluorite/

Having this data has many advantages. Not only does it reduce the time to inspect the videotapes significantly [13], it also enables various automatic analyses. And in the future, we may be able to use the analysis in support of new tools in the IDE that will directly help the developers.

IV. RESULTS

A. Overview

The study took 96.6 minutes on average. The task accomplishment varied a great deal across the participants. Of the 12 participants, only 3 people completed all the five tasks. 3 people could only complete one task and had to give up all the others. Overall, the participants completed only 58.3% of all the tasks.

We hoped that the 4 different features would have the similar difficulties, but it turned out that F1-1 (thickness control using slider widget) was the easiest. 11 participants succeeded on F1-1, while each other feature was successfully completed by about 5 of the participants[2]. We speculate that F1-1 was the easiest because there was a working example of the slider widget right in the code base (the color slider).

Even though some participants were not very successful in completing the tasks, we did not exclude those data because the participants still backtracked to some extent while trying to figure out how to get the tasks completed.

B. Command Statistics & Keystroke Distribution

We counted how frequently each IDE command was executed and each keyboard key was pressed. Table 2 shows the top twenty commands executed, and separately, the top 20 keystrokes typed across all the participants. Except for typing and code navigation commands, the most frequent commands are the backtracking related commands (inverted rows). Considering that the navigation commands would be expected to be large since FLUORITE logs multiple instances of the same event when the user holds down a key and it auto-repeats, we can see that backtracking related commands are very frequently executed. The command statistics are somewhat different from those of Murphy et al. [11] because the two logging tools differ in what types of commands are logged. However, the rank orderings of commands are consistent if we only compare the main editor commands such as Delete, Save, Copy, Paste, and Assist.

Table 2 lists two different Assist commands. The first one counts all the content assist executed automatically (e.g., when the user types a dot following a variable name), and the second one only counts the manually executed content assist and quick fixes.

V. OBSERVATIONS

A. Deleting vs. Commenting Out

7 of the 12 participants habitually commented out their code rather than deleting it, whether or not they thought the

[2] F1-2, F2-1, and F2-2 were successfully completed by 5, 6, and 4 out of 12 participants, respectively.

TABLE 2. Command and keystroke distributions. The top twenty entries are listed for each. Shaded entries are related to code navigation, and the inverted entries are related to backtracking.

Commands		Keystrokes	
Type char.	17092 (31.8%)	Down arrow	5797 (12.64%)
Line down	5795 (10.8%)	Backspace	5693 (12.41%)
Delete prev.	5692 (10.6%)	Up arrow	4495 (9.80%)
Move caret	4686 (8.7%)	Right arrow	3586 (7.82%)
Line up	4491 (8.4%)	Left arrow	2751 (6.00%)
Col. next	3544 (6.6%)	S	1873 (4.08%)
Col. prev.	2715 (5.1%)	Ctrl	1854 (4.04%)
Select text	1975 (3.7%)	Shift	1652 (3.60%)
Sel. col. next	1035 (1.9%)	Enter	1387 (3.02%)
File open	907 (1.7%)	T	1289 (2.81%)
Sel. col. prev.	857 (1.6%)	E	1250 (2.72%)
Save	852 (1.6%)	N	1003 (2.19%)
Delete	576 (1.1%)	I	882 (1.92%)
Paste	459 (0.9%)	C	871 (1.90%)
Assist(auto)	456 (0.8%)	Space	859 (1.87%)
Run	391 (0.7%)	A	800 (1.74%)
Copy	314 (0.6%)	O	750 (1.63%)
Undo	294 (0.5%)	V	619 (1.35%)
Assist(manual)	213 (0.4%)	L	610 (1.33%)
Sel. line down	212 (0.4%)	Delete	576 (1.26%)
Others	1113 (2.1%)	Others	7275 (15.86%)
Total	53669	Total	45872

code was going to be reused later. However, even the participants who explicitly said that they usually comment out code also deleted code during the study, because they said they did not like messing up the code with lots of comments. In some cases, those deleted code fragments turned out to be needed later on.

Some programming languages provide specific ways of activating and deactivating code. For example, C/C++ has preprocessor directives such as `#ifdef` [25], and the .NET Framework provides the `Conditional` attribute which allows developers to conditionally activate a certain method according to the current build configuration. However, since our study was based on Java, these language specific methods were not available, so they could only use conventional comments.

B. Common Reasons for Commenting Out

During the lab study, participants articulated three main reasons why they commented out the code instead of deleting it. First, the developers commented out code because they knew that the code being commented out was going to be used again. This includes the situation where the code was one of the variations and the developer wanted to be able to switch to another variation. Also, when the developer had implemented two different features simultaneously and wanted to test one at a time, they left the code for the feature under test and commented out the other. This was the most common reason given.

The second common reason for commenting out is to keep the code snippet as a good example. This situation differs from the previous one in that the code is not expected to be used at the moment, but the developer wants to keep the code anyway. This could happen when the developer thinks that the code could be used as a structural template for other similar code. For example, in our study, the participants had to add different types of listeners to the graphical widgets. When developers tried out one type of listener but it did not work, they often commented it out because the listener creating and adding structure is pretty much the same regardless of the type of the listener they would use. Also, when it turned out that an example code snippet they found from the Internet did not quite fit to the given situation, they often commented out the code rather than deleting it because they did not want to have to search for the example again in case it might be needed later on.

Finally, developers occasionally commented out code in order to remind themselves that the code was not good. They kept the code there because they wanted to avoid making the same mistakes afterwards.

C. Problems the Participants Faced while Backtracking

The study participants faced various problems when they were trying to backtrack. Participants often had problems finding the right code fragment to be reverted in the source file. For instance, when implementing F1-1 (thickness control feature using the slider widget), most participants copied and pasted the code for the color sliders and modified it. Because the original code and the pasted code looked very similar, participants were often confused and looked at or even edited the wrong code.

When they were trying to backtrack all the code fragments related to a certain source code level element such as a variable, method, or class, it took some effort to find all the relevant code fragments. Although participants rarely made mistakes at this, occasionally they did miss a few statements that should have been reverted. Often, this happened because two or more elements were involved in a single feature. For example, when restoring the commented-out slider widget, they often forgot to restore the associated change-listener code. One participant made this mistake even though he labeled the related code fragments using comments. We speculate that it would be even more difficult for the developers to find all the relevant code fragments when they are distributed across multiple files, but this did not happen in our lab study because mostly the participants implemented all the features in a single file.

The participants often added and removed debug outputs. Especially when they were implementing F2 (x, y coordinates indicator) – pretty much all of the participants added debug outputs using either a console output method (`System.out.println`) or a simple message box (`JOptionPane.showMessageDialog`) in order to check if the mouse listeners they had just added was called when the mouse cursor was moved, and if the x,y values were correct. However, after they had finished implementing the feature, they sometimes forgot to remove the debug outputs. All the participants who used the message dialog did remove it since the message box was continuously interfering, while many of the ones who used console output did not.

D. Do the Developers Behave Differently when They Know In Advance that They Might Need to Backtrack?

Not surprisingly, even the participants who usually just deleted the code did comment out the code when they believed that the code was likely to be reused soon. For example, when they were doing task 3 (getting back to F_A-1

after completing F_A-2), pretty much all of the participants commented out the code for F_A-2 because they thought we might ask them to go back to F_A-2 again.

Only a few participants behaved differently when they were doing task 4 (implementing F_B-1 & F_B-2 simultaneously). One participant used a flag variable so that he could select either of the two user interface variations dynamically. Four other participants marked each code fragment using comments, and only one of the variations would be activated (uncommented) at a time. When we asked the participants to switch to a different variation, they manually searched for all the currently-activated variation code fragments using the labels and commented them out, and then searched for all the code fragments to be activated and uncommented them. This worked, but it was a tedious process. Also, when only one of the variations gets accepted and the others are rejected, one would need to manually search for all of the rejected variations and delete them.

E. Backtracking as "Restoring" Code

So far, we mostly discussed backtracking as removing / deactivating code that was recently added. However, restoring previously removed code is also an important aspect of backtracking. We observed several problems with restoring code during our lab study.

For example, one participant had a serious problem with restoring code. After copying and pasting some code and testing the program, he meant to delete only the pasted code, but he accidentally selected the copied and pasted code together and deleted them, because they happened to be adjacent and looked very similar. He realized that something went wrong about 2 minutes later when he tested the program, and then spent 1 more minute to figure out what was wrong, and then spent 30 seconds to locate where the deleted code should be put back. However, he could not remember how the deleted code looked nor could he restore the code even after he correctly found where it went. He had tried to restore the deleted code[3] from memory for 6 minutes, but eventually failed to produce correct code and gave up.

Another participant faced a similar problem, but in this case, he *did* remember what the code looked like, and he knew that he had deleted the code quite recently. He therefore could restore the code by taking advantage of the linear undo feature of the code editor. He first executed the undo command multiple times until the desired code fragment was restored, copied the code fragment, executed redo command to the end to remove all the other extra commands that still should be redone, and then pasted the code into the desired position. This takes advantage of the feature that undo/redo does not affect the contents of the clipboard.

For the cases where the developer wanted to restore a specific code fragment that was recently deleted, they often remembered one or more features of the deleted code such as the original location from where the code was deleted (or the surrounding code), the names of one or more code elements in the deleted code, or how the code looked. Thus, we speculate that in general, even when they could not easily reproduce the code from scratch, they probably could recognize the code if it was able to be displayed somehow.

We also observed that participants reproduced the same code fragments repeatedly from memory. For example, when implementing F2 (x, y coordinates indicator), participants wrote complex expressions which would result in the desired output string[4]. They used these expressions with the debug outputs to check if they were getting the correct values, and then retyped the whole expression when trying to display it in the desired graphical widget. Reproducing such expressions was not difficult, but it was very tedious and inefficient.

VI. Online Survey

A. Methodology

In order to get more feedback from general software developers besides just graduate students at Carnegie Mellon, we conducted an online survey, which took about 15~20 minutes to complete. The survey was posted on several online developer forums including reddit.com [5] and dzone.com[6] and others. A total of 103 developers answered at least some of the questions, and 48 of them completed the whole survey. Of the 48 people who finished, 31 were from dzone.com, 15 were from reddit.com, and the other 2 were from the Eclipse developer forum. Our analyses of this survey are based on the responses from these 48 people who completed the survey so all of the questions would have the same number of answers.

B. Demographics / Traits of Their Work

The survey was composed of three parts. First, we asked demographic information including their gender, age, and prior experience in software development. We also asked if they do their development work alone or as part of a group, and to what degree their specifications are flexible and to what extent they experiment, iterate, and/or explore while they develop. The demographics of the respondents are summarized in Table 3. 72.9% of all the respondents had been programming for more than 5 years and the overall average was 13 years. This indicates that the respondents were mostly professional programmers.

The respondents were asked to express if they worked alone or as part of a group, using a 5-point Likert scale. Each of the 5 choices received a rating from 12.5% to 27.1%, which means the respondents had diverse situations. The next question asked how flexible the developers' work was for different activities, and the results are summarized in Figure 1. We can see that they can experiment, iterate, and/or explore a lot for the coding details but not much for the user interface specifications or the desired behaviors.

We speculated that there might be more flexibility if the developer works alone or as part of relatively small groups. So, we investigated if there is any correlation between the

[3] 6 lines of code, excluding the blank lines and the lines only containing "}".

[4] similar to the following expression: "(X, Y) = " + x + ", " + y + ")"

[5] http://www.reddit.com/r/programming

[6] http://www.dzone.com/

TABLE 3. Demographics of the online survey respondents.

	value	number	percentage
Total respondents		48	
Gender	Male	44	91.7%
	Female	4	8.3%
Age	20-30	22	45.8%
	30-40	15	31.3%
	40-50	6	12.5%
	50-60	5	10.4%
	Average	32.5	σ = 9.4
Programming Experience (years)	< 1	1	2.1%
	1-3	4	8.3%
	3-5	8	16.7%
	>= 5	35	72.9%
	Average	13.0	σ = 9.9

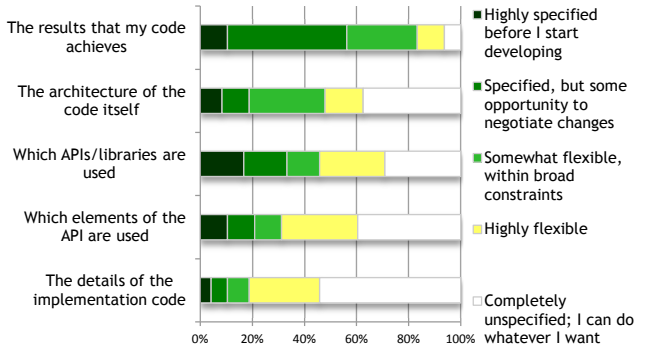

Figure 1. The responses for the question "For each of the following, please specify how often you need to experiment, iterate, and/or explore *while you are developing*." The lighter color represents more flexibility.

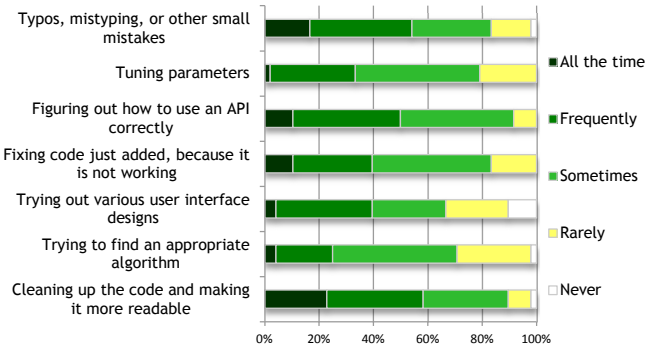

Figure 2. The backtracking situations shown to the survey respondents.

sizes of the groups in which the developers worked, and the flexibility of their work, but could not find any statistically significant correlation. Even when the developer worked alone, often the work was assigned by the boss or the customers and we did not find that the developer had much freedom. Only one of the respondents who always worked alone expressed that everything is completely unspecified and he can do whatever he wants.

C. Backtracking Situations and Their Strategies

In the second part of the survey, we presented seven different situations where the developers might need to backtrack. For each situation, we first asked how often the respondents faced the given situation. Figure 2 shows the responses for these questions. We can see that developers face these backtracking situations quite often. Roughly $^{3}/_{4}$ of the developers face these situations at least "sometimes".

Next, we asked what types of strategies they use to backtrack. We showed eight different strategies related to backtracking, and asked them how often they think they use each strategy to solve the given situation using a 5-point Likert scale ranging from "Never (0)" to "Pretty much every time (4)". The result showed that only a few strategies are primarily used for each situation. When fixing typos and small mistakes, the most frequently used strategies were using backspace / delete keys, using undo command, and selecting and overtyping. When tuning parameters, they usually select the old parameter and overtype the new parameter. They look up the method list using the code completion list when they are trying to figure out how to use an API, or they manually replace one method with another. For debugging or trying out different solutions, they mostly comment out code, which is consistent with our observations from the lab study. Finally, when cleaning up code, they manually select the unnecessary code and delete it or use refactoring commands to better structure the code.

This information further hints at how we can detect each backtracking situation. For example, when the developer invokes the code completion menu and spends a significantly long time navigating the menu's items, this situation may indicate that the developer is learning an API.

D. Open-Ended Responses

The respondents were also asked to provide other strategies they use for each given situation, if any. We collected a total of 34 responses for these questions. Two of the strategies that our participants mentioned were also observed in our lab study. 2 people mentioned that they use Boolean flag variables to temporarily turn on or off code fragments. Another 2 participants said they would write a small code snippet separated from the main project in order to try out something.

Two other strategies that were mentioned were not observed in our lab study. 2 people mentioned that they move the parameters out of the code and put them in external configuration files or in databases so that they can change the values without rebuilding the software, even at runtime. 5 responses mentioned writing unit tests using mock objects to see how the API works. We speculate that our lab study participants did not use these strategies because the code base was fairly small and they had time limitations.

Next, we asked what other backtracking situations they faced. One respondent mentioned backtracking while writing a new interface file from scratch, maybe because it is often not clear what would be the correct interface to be exposed. 2 responses were about reorganizing and simplifying code structure. Another 2 responses said that they mainly backtrack because they find new corner cases or missed input values during testing.

Finally, we solicited ideas on what types of new features or commands for an IDE could help with experimenting and backtracking. 2 people wanted a tool where the developers

can type in small code snippets and run them, just as they can with scripting languages. 2 people wanted an IDE feature that allows the developers to take snapshots across multiple files at any point, and switch among those snapshots. 2 other people wanted a lighter version control system that can keep multiple versions of a method or class and allow users to select one of them easily. 1 person wanted an undo tree model instead of the conventional linear undo model.

VII. Future Work

We plan to conduct a more extensive field study of backtracking. We will use the FLUORITE tool to monitor the developers' coding behaviors and see what types of backtracking problems they face during their own regular development. We will also perform a retrospective contextual inquiry by interviewing selected participants after analyzing the log data. Ultimately, we plan to provide new development tools that will help developers backtrack more easily and accurately.

VIII. Conclusion

It is clear that backtracking is prevalent in coding and our study data provides additional evidence and information about when and how this happens. Although our study revealed that there are many different backtracking situations, none has previously been studied in depth. In both of our studies, it has been shown that the developers use several strategies when they face a backtracking situation, and many of them are still manual and error-prone. From our lab study, we identified several problems with backtracking which are common to many developers. There is still much room for improvement in modern development environments, and the research reported here provides evidence that more robust backtracking assistance tools would help developers write code more correctly and efficiently.

Acknowledgment

We thank all the developers who participated in our lab study and online survey. We also thank Andrew Ko for providing us the Paint program that we used in our lab study.

Funding for this research comes in part from the Korea Foundation for Advanced Studies (KFAS) and in part from NSF grants CCF-0811610 and IIS-1116724. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of KFAS or the National Science Foundation.

References

[1] D. W. Sandberg, "Smalltalk and exploratory programming," *SIGPLAN Notices,* vol. 23, 1988, pp. 85-92.

[2] J. Sametinger and A. Stritzinger, "Exploratory software development with class libraries," *Proc. 7th Joint Conference of the Austrian Computer Society*, 1992.

[3] W. R. Reitman, *Cognition and Thought*, John Wiley & Sons, 1965.

[4] H. A. Simon, "The structure of ill structured problems," *Artificial Intelligence,* vol. 4, 1973, pp. 181-201.

[5] M. Terry, E. D. Mynatt, K. Nakakoji, and Y. Yamamoto, "Variation in element and action: supporting simultaneous development of alternative solutions," *CHI'04*, Vienna, Austria, 2004, pp. 711-718.

[6] S. K. Card, T. P. Moran, and A. Newell, "Computer text-editing: An information-processing analysis of a routine cognitive skill," *Cognitive Psychology,* vol. 12, 1980, pp. 32-74.

[7] S. K. Card, T. P. Moran, and A. Newell, "The keystroke-level model for user performance time with interactive systems," *Commun. ACM,* vol. 23, 1980, pp. 396-410.

[8] I. S. MacKenzie and R. W. Soukoreff, "Text entry for mobile computing: Models and methods, theory and practice," *Human-computer interaction,* vol. 17, 2002, pp. 147-198.

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

[10] Y. Yoon and B. A. Myers, "Capturing and analyzing low-level events from the code editor," *Proc. 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools* (PLATEAU'11), Portland, Oregon, USA, 2011, pp. 25-30.

[11] G. C. Murphy, M. Kersten, and L. Findlater, "How are Java software developers using the Elipse IDE?," *IEEE Software,* vol. 23, 2006, pp. 76-83.

[12] T. Berlage, "A selective undo mechanism for graphical user interfaces based on command objects," *ACM Transactions on Computer-Human Interaction* (TOCHI)*,* vol. 1, 1994, pp. 269-294.

[13] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in OOPL," *Proc. International Symposium on Empirical Software Engineering* (ISESE'04), 2004, pp. 83-92.

[14] A. J. Ko, H. H. Aung, and B. A. Myers, "Design requirements for more flexible structured editors from a study of programmers' text editing," *Extended abstracts CHI'05*, Portland, OR, 2005, 1557-1560.

[15] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *Proc. 31st International Conference on Software Engineering* (ICSE'09), 2009, pp. 287-297.

[16] M. Kim, D. Cai, and S. Kim, "An empirical investigation into the role of API-level refactorings during software evolution," *Proc. 33rd international conference on Software engineering* (ICSE'11), Waikiki, Honolulu, HI, USA, 2011, pp. 151-160.

[17] B. A. Myers and D. S. Kosbie, "Reusable hierarchical command objects," *Proc. SIGCHI conference on Human factors in computing systems: common ground* (CHI'96), Vancouver, British Columbia, Canada, 1996, pp. 260-267.

[18] B. A. Myers, "Scripting graphical applications by demonstration," *CHI'98*, Los Angeles, CA, 1998, pp. 534-541.

[19] T. Cubitt, "undo-tree.el version 0.3.1 --- Treat undo history as a tree," 2010; http://www.dr-qubit.org/emacs.php.

[20] B. Hartmann, L. Yu, A. Allison, Y. Yang, and S. R. Klemmer, "Design as exploration: creating interface alternatives through parallel authoring and runtime tuning," *Proc. 21st ACM Symp. on user interface software and technology* (UIST'08), Monterey, CA, 2008, pp. 91-100.

[21] M. Erwig and E. Walkingshaw, "The Choice Calculus: A Representation for Software Variation," *ACM Transactions on Software Engineering and Methodology* (TOSEM)*,* vol. 21, 2011.

[22] A. J. Ko and B. A. Myers, "Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors," *CHI'06*, Montréal, Québec, Canada, 2006, pp. 387-396.

[23] J. Fogarty, A. J. Ko, H. H. Aung, E. Golden, K. P. Tang, and S. E. Hudson, "Examining task engagement in sensor-based statistical models of human interruptibility," *CHI'05*, Portland, OR, 2005, pp. 331-340.

[24] 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," *Proc. 27th international conference on Software engineering* (ICSE'05), St. Louis, MO, 2005, pp. 126-135.

[25] D. Le, E. Walkingshaw, and M. Erwig, "#ifdef Considered Harmful: Promoting Understandable Software Variation," *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing* (VL/HCC'11), Pittsburgh, PA, 2011, pp. 143-150.