Backtracking Support in Code Editing

YoungSeok Yoon

May 2015

CMU-ISR-15-103

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Brad A. Myers (Chair, CMU HCII)
Jonathan Aldrich (CMU ISR)
Christian Kästner (CMU ISR)
Emerson Murphy-Hill (North Carolina State University)

Submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy
in Software Engineering

Funding for this research comes in part from the Korea Foundation for Advanced Studies (KFAS) and in part from NSF grants CCF-0811610, IIS-1116724, and IIS-1314356. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect those of KFAS or the National Science Foundation.

Keywords: selective undo, backtracking, fine-grained edit history, logging, timeline visualization, history search, edit collapsing, Eclipse IDE, Java, exploratory programming

ABSTRACT

Programmers often need to *backtrack* while coding. Here, “backtracking” refers to when programmers go back at least partially to an earlier state of code, either by removing inserted code or by restoring removed code. For example, when some newly added feature does not work as imagined, the programmer might have to backtrack and try something else. When learning an unfamiliar API, programmers often need to try some sequence of object instantiation and method calls, run the program, and backtrack if the result is not as expected. I conducted a series of three empirical studies in order to better understand the backtracking behavior of programmers. The results indicated that backtracking is prevalent in programming, and programmers often face challenges when backtracking. For example, they had difficulties when trying to find all the relevant parts of code to be backtracked or when trying to restore some code they had deleted that later turned out to be needed.

However, programmers only have very limited support for backtracking in today’s tools. The linear undo command can only undo the most recent changes, and loses the undone changes as soon as the programmer makes a single new change after invoking the undo command. Version control systems such as Subversion and Git can also be used for backtracking, but only when the desired code is already committed in the repository. Furthermore, the results from the empirical studies showed that 38% of all the backtrackings are done manually without any tool support and 9.5% are selective, which means that they could not have been performed using the conventional undo command.

To help programmers backtrack more easily and accurately, I devised a novel selective undo mechanism for code editors, and implemented it in an IDE plug-in called AZURITE. The core idea is to combine the following mechanisms into a coherent programming tool: a *selective undo* mechanism for code editors, *novel visualizations* of the coding history, and a code change *history search*. AZURITE retains the full fine-grained code change history, and the selective undo mechanism allows users to select and undo one or more isolated edit operations, while appropriately detecting and handling conflicting operations. The visualizations and history search are the user interfaces that help users to select the desired edit operations to be backtracked and express what they remember about the code changes that they want to revert. In a controlled lab experiment, programmers using AZURITE performed twice as fast compared to the control group when completing typical backtracking tasks. My hope is that this selective undo tool will help programmers achieve their daily programming tasks more effectively.

ACKNOWLEDGEMENTS

Foremost, I would like to express my sincere gratitude to my advisor, Brad Myers. Throughout my entire career as a PhD student, he has constantly provided guidance and support to help me conduct better research, write better research papers, and prepare my presentations. I cannot express enough how fortunate I have been to have Brad Myers as my advisor. I would also like to thank all my thesis committee members, Jonathan Aldrich, Christian Kästner, and Emerson Murphy-Hill. Their detailed, constructive feedback greatly helped in making this dissertation better.

It has been a pleasure to work with all my co-authors, Joel Brandt, Andrew Faulring, Sebon Koo, Ashley Lai, Thomas LaToza, Tam Minh Le, and Cyrus Omar. I would not have been able to publish all the research papers without the hard work and insight of these individuals. My academic siblings, Kerry Chang, Michael Coblenz, and Stephen Oney, also deserve my thanks for all the interesting research discussions we had. I would also like to thank the faculty and friends in the Software Engineering PhD program. Their thoughtful feedback at the SSSG seminars was invaluable resource for improving my research and presentation skills.

I would like to thank the Korea Foundation for Advanced Studies for all the financial support for the last five years, and the National Science Foundation for providing research funding. In addition, I was fortunate to be part of the Exploratory Programming Group, which is an NSF-funded joint research project.

Getting through the PhD program was more than just doing research. I would like to give special thanks to my wife, Min Jeong Kim, who has always been with me and helped me get through all the difficult times. I thank my parents, Jang Ho Yoon and Sang Hee Lee, and my younger brother, Youngkwon Yoon, for being always on my side and praying for my soul.

TABLE OF CONTENTS

FIGURES

TABLES

1. INTRODUCTION

Since programmers are human, it is unrealistic to expect them to complete a whole task on the first attempt without making any mistakes. Besides, programmers may intentionally make temporary changes to the code, either as an experiment or to help with debugging. As a consequence, programmers need to *backtrack* while coding. Throughout this dissertation, the term “backtracking” is defined as “programmers going back at least partially to an earlier state of code, either by removing inserted code or by restoring removed code,” and does not refer to the algorithm for solving constraint satisfaction problems in the artificial intelligence area or similar (cf. [Cormen 2009]). For example, programmers fix typos and correct minor mistakes, and they try out different values for parameters to methods. When programmers try to learn an unfamiliar API, they might try writing some code and running it to see if the code works as expected, and if it does not, they backtrack and try something else.

In some situations, programmers 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 [Sandberg 1988][Sametinger 1992]. Often, problems are ill-defined, and there is no single correct solution for these problems. Rather, there are several alternative solutions with their own strengths and weaknesses [Reitman 1965][Simon 1973][Terry 2004]. In order to evaluate each solution, the programmer might implement one, backtrack, and implement another.

Also, backtracking plays an important role in situations where alternative solutions need to be managed for a given task. When programmers are unsure about which algorithm, library, or UI component to use in a given situation, then they might want to try out one of the alternatives to see how it works. If it does not work, then the first attempt might be reverted, which is an example of backtracking, and another attempt might be made. Moreover, when making another attempt after backtracking, it might turn out that the previous attempt was better, which leads to another backtracking situation. Several variation management tools have been developed [Hartmann 2008][Terry 2004], but these are limited in that users cannot easily backtrack and add a new alternative from there, if they did not plan ahead where they would need new alternatives.

Other researchers have shown that programmers do backtrack a significant amount while coding, much more than people do during the text editing of regular documents [Card 1980a][Card 1980b][MacKenzie 2002]. 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) kept track of the usage of commands executed by all the Eclipse users who have consented to provide their usage data. According to the UDC data collected from Jan. 2009 through Jan. 2010 (which is the latest data published), 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%). Murphy et al. also reported that delete was the most frequently executed command in their study [Murphy 2006]. Note that undo is not the same as backtracking because undo command can only revert the most recent changes but backtracking includes when programmers revert some changes that were made a while ago.

As part of this dissertation work, I conducted a series of empirical studies of backtracking, since little was known about the backtracking behaviors of programmers (Chapter 4). My results confirm that backtracking happens frequently. First, it was shown that the backspace keystrokes were 12.41% of all the keystrokes made in the code editor, which is a higher percentage for backspace compared to normal document editing (e.g., 7.10% in [MacKenzie 2002]). An exploratory lab study and a follow-up online survey confirmed that backtracking is quite common in programming, and programmers often reported having problems when they want to backtrack. In addition, a longitudinal study was conducted with 1,460 hours of actual code editor usage data from 21 programmers. The programmers in this study backtracked 10.3 times per hour on average, and 34% of all the detected backtracking instances were performed manually without using the undo command or any other tool support.

1.1. PROBLEM: LIMITED SUPPORT FOR BACKTRACKING

Despite the frequency of backtracking in development contexts, modern IDEs do not provide much support. For example, there are no sophisticated undo mechanisms used in IDEs other than the *restricted linear undo model* [Berlage 1994]. However, this restricted linear undo model, which is widely used in most text and code editors, is not suitable for all situations.

The most significant problem is that the users can only undo the most recently performed edits. This can be very inconvenient when users realize that they made a mistake after making some other changes that they want to retain. In addition, programmers may intentionally make changes to the code that they want to remove later on. For instance, a developer might insert many print statements in different places in the process of debugging, then fix the bug, and finally want to remove all those print statements. Since there would be some other changes (for actually fixing the bug) that the developer wants to retain after the insertions of the print statements, the conventional undo command cannot be used for removing the print statements. Also, when the programmer 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 complete command history tree but only keeps a linear list. Moreover, the undo implemented in code editors only works on one file at a time, whereas many edits to be backtracked span multiple files in the project.

Another popular way of backtracking is using a version control system (VCS) such as Subversion or Git. A VCS allows users to revert some code to a previous version. This is not the same as backtracking either, because backtracking can (and is actually quite likely to) happen between two version control snapshots. In fact, version control relies on the assumption that the desired code is already committed to the repository. This may not always be the case, especially in a backtracking situation, because it is likely that the programmer is experimenting and the code is unstable or there are many temporary code fragments that should not be committed to the repository.

1.2. MOTIVATING EXAMPLE

Imagine a scenario where a programmer is working on a graphical user interface (GUI) in Java Swing and wants to implement a simple panel with three vertically arranged buttons, as shown in Figure 1-1. There should be a fixed amount of padding inside the entire panel and between the buttons. First, she starts out with having a stub method that returns an empty panel. She then makes the following changes in order.

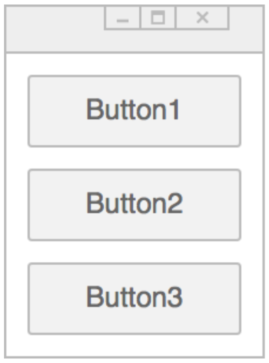


Figure 1-1. A sketch of the desired UI

1. She creates three button objects and adds them to the panel (Figure 1-2a).
2. Running the application shows horizontally laid out buttons, so she looks for some layout manager to use. She first tries out **`GridBagLayout`** (Figure 1-2b).
3. The intermediate code seems too complicated for just a simple vertical layout. She looks for a simpler layout manager, and discovers **`BoxLayout`**. She uses undo command multiple times to get rid of all the **`GridBagLayout`** code (*backtracking* to Figure 1-2a).
4. She writes some code with **`BoxLayout`**, resulting in much simpler code and vertically laid out buttons (Figure 1-2c).
5. She changes some properties of the buttons, such as the background color and button text (Figure 1-2d).

She now wants to finish up the layout and add some spacing between the buttons before moving further. However, she realizes that **`BoxLayout`** does not directly support spacing while **`GridBagLayout`** does. Therefore, she wants to restore the **`GridBagLayout`** code she wrote in step 2, while keeping the changes from step 5.

```java
private JPanel createButtons() {
  JPanel p = new JPanel();

  JButton button1 = new JButton("Button 1");
  JButton button2 = new JButton("Button 2");
  JButton button3 = new JButton("Button 3");
  p.add(button1);
  p.add(button2);
  p.add(button3);

  return p;
}
```

(a)

```java
private JPanel createButtons() {
  JPanel p = new JPanel();
  p.setLayout(new GridBagLayout());
  GridBagConstraints c = new GridBagConstraints();
  ... (omitted) multiple lines of code
  ... (omitted) for configuring c.
  JButton button1 = new JButton("Button 1");
  JButton button2 = new JButton("Button 2");
  JButton button3 = new JButton("Button 3");
  p.add(button1, c);
  p.add(button2);
  p.add(button3);

  return p;
}
```

(b)

```java
private JPanel createButtons() {
  JPanel p = new JPanel();
  p.setLayout(new BoxLayout(p,
      BoxLayout.Y_AXIS));

  JButton button1 = new JButton("Button 1");
  JButton button2 = new JButton("Button 2");
  JButton button3 = new JButton("Button 3");
  p.add(button1);
  p.add(button2);
  p.add(button3);

  return p;
}
```

(c)

```java
private JPanel createButtons() {
  JPanel p = new JPanel();
  p.setLayout(new BoxLayout(p, BoxLayout.Y_AXIS));

  JButton buttonOrange = new JButton("Orange");
  buttonOrange.setBackground(Color.orange);
  JButton button2 = new JButton("Button 2");
  JButton button3 = new JButton("Button 3");
  p.add(buttonOrange);
  p.add(button2);
  p.add(button3);

  return p;
}
```

(d)

Figure 1-2. The code changes for the motivating example. The green highlight shows newly inserted lines, and the grey highlight shows updates to the existing code.

This example illustrates the problems of existing backtracking mechanisms discussed above in Section 1.1. At this point, the regular undo command cannot be used because she had previously used the undo command to remove the **`GridBagLayout`** code and then she made some *new* changes from there, so the needed operations have been eliminated from the undo stack. Even if she had not used the undo command in step 3, the undo command would still be inappropriate for this situation, because it will necessarily revert the changes made in step 5, which is also not desired. Moreover, it would be very unlikely that the **`GridBagLayout`** code had been committed to a version control system, because the code was still an incomplete state. The only option she now has is to reproduce the **`GridBagLayout`** code from scratch, which is inefficient. It would be much more convenient for her if there was at least a semi-automatic way of restoring the desired code from Figure 1-2b while keeping the subsequent desired edits from Figure 1-2d.

1.3. An Approach: Selective Undo in Code Editors

These problems can be solved by having a *selective undo* feature in code editors. Users could select *specific* edit operations performed in the past, for example the insertions of the print statements for debugging, and invoke the selective undo command to revert only the code affected by the those operations. The results from my longitudinal backtracking study showed that 9.5% of all the backtrackings performed by the participants were selective, meaning that they could not have been handled by the conventional undo command (Section

4.3). Selective undo has been well researched in the area of graphical editors [Berlage 1994][Myers 1996][Myers 1998].

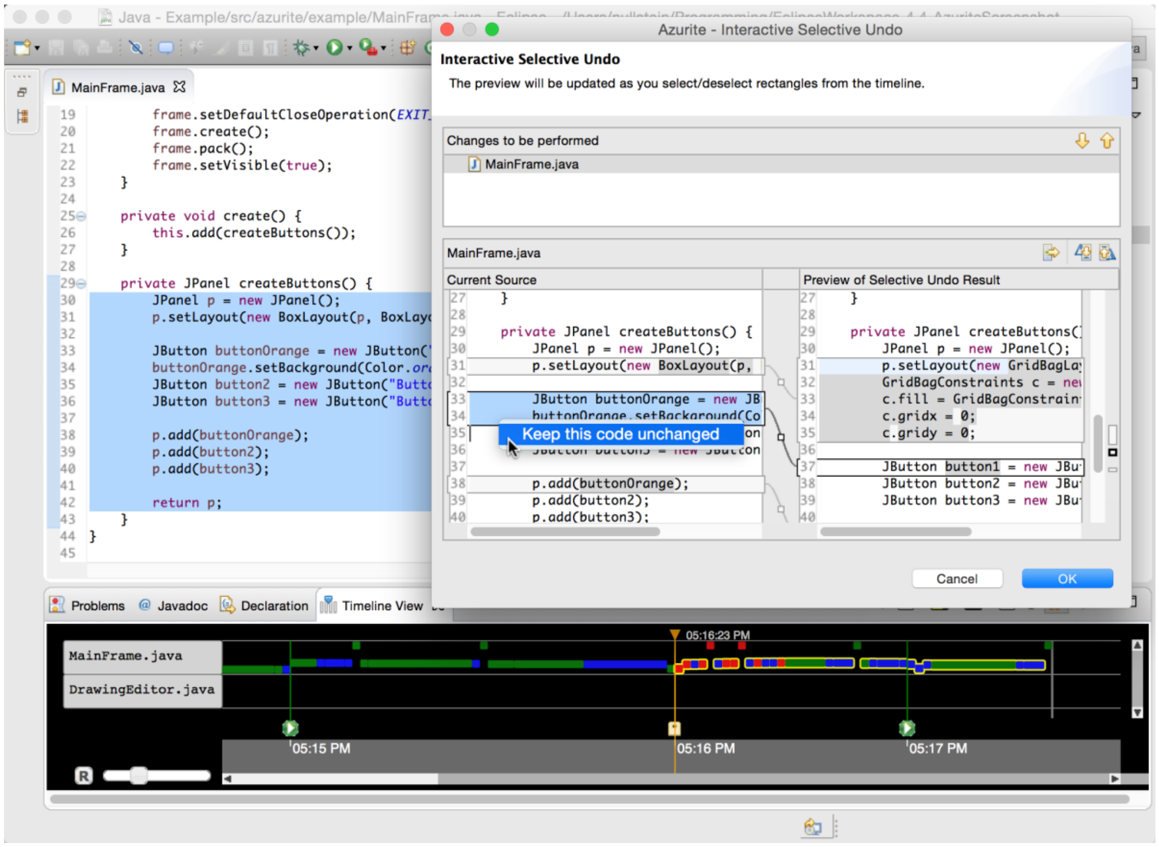
1.3.1. Challenges of Providing Selective Undo in Code Editors

However, selective undo has not been used with text or code editors due to the many text-specific challenges. First, as Berlage pointed out, existing selective undo mechanisms are designed to work best when the system has identifiable objects that are affected by operations, but text does not have the notion of objects but rather has a stream of characters [Berlage 1994]. Second, there can be many "regional conflicts" among edit operations. A *regional conflict* can occur when the region of a later edit overlaps the region of the earlier edit which the user wants to selectively undo. When there is a regional conflict among the edit operations, the result of a selective undo may not be well defined. To illustrate this point, consider the following example. An edit operation e_1 changes the code from "`myFontSize = 12;`" to "`myRectangleSize = 12;`" and sometime later, another operation e_2 changes it to "`myRegionArea = 12;`". This is an example of regional conflict because the affected ranges of the two operations are overlapping and the "`Rectangle`" text inserted by e_1 is only partially available in the current code. In this case, it is not clear what the result of selectively undoing operation e_1 alone should be. The system should be able to detect such cases and provide an appropriate approach to resolving them.

A final challenge of providing selective undo for code is that it is difficult to provide intuitive user interfaces for the user to *find* what to selective undo. Many existing selective undo user interfaces for graphics present a list of edit operations performed in the past along with human-readable descriptions of individual operations [Berlage 1994][Myers 1996][Myers 1998]. However, text editing operations are much more fine-grained than graphical editing, so it is hard for the users to interpret the high level edit intent just by looking at the individual text edits. In addition, graphical applications can use a thumbnail to represent a snapshot of the graphics at a certain point of time, which makes it easier to present the edit history to the user [Kurlander 1988][Klemmer 2002][Terry 2004][Kurlander 1990][Chii 1998]. In contrast, a thumbnail of a piece of a large text file does not give much information to the users.

1.4. Azurite: A Selective Undo Tool for Programmers

To solve the problem of limited support for backtracking while addressing the challenges mentioned above in Section 1.3 and complement the existing tools, I devised a novel selective undo mechanism for code editors, which is the main topic of this dissertation. The selective undo mechanism is implemented into a prototype tool called Azurite, as a plug-in for the Eclipse IDE (Figure 1-3). Azurite allows programmers to selectively undo fine-grained changes in the code editor. To provide this functionality, the system takes the stream of fine-grained code edits as input and maintains the mapping between the different segments of the current source file and the edit operations that introduced those segments. The system also keeps track of regional conflict relationships among edit operations (Section 5.1.2). The system makes use of this information to provide selective undo in code editors (Section 5.2).



```
        frame.setDefaultCloseOperation(EXIT_
        frame.create();
        frame.pack();
        frame.setVisible(true);
    }

    private void create() {
        this.add(createButtons());
    }

    private JPanel createButtons() {
        JPanel p = new JPanel();
        p.setLayout(new BoxLayout(p, BoxLayo

        JButton buttonOrange = new JButton("
        buttonOrange.setBackground(Color.or
        JButton button2 = new JButton("Butto
        JButton button3 = new JButton("Butto

        p.add(buttonOrange);
        p.add(button2);
        p.add(button3);

        return p;
    }
}
```

Interactive Selective Undo

The preview will be updated as you select/deselect rectangles from the timeline.

```
    }

    private JPanel createButtons() {
        JPanel p = new JPanel();
        p.setLayout(new BoxLayout(p,

        JButton buttonOrange = new JB
        buttonOrange.setBackground(Co
                                    on
        JButton button3 = new JButton

        p.add(buttonOrange);
        p.add(button2);
        p.add(button3);
```

```
    }

    private JPanel createButtons()
        JPanel p = new JPanel();
        p.setLayout(new GridBagLa
        GridBagConstraints c = ne
        c.fill = GridBagConstrain
        c.gridx = 0;
        c.gridy = 0;

        JButton button1 = new JBu
        JButton button2 = new JBu
        JButton button3 = new JBu
```

Figure 1-3. An example screenshot of AZURITE running in the Eclipse IDE. At the bottom, a timeline visualization of recent code changes is provided. The user is currently using the "Interactive Selective Undo" dialog in order to selectively undo the code and restore the `GridBagLayout` code without losing the desired code.

In the motivating example above, the programmer can restore the deleted **`GridBagLayout`** code without losing the changes related to **`buttonOrange`** using AZURITE, with the following steps:

1. Find the point in time in the past where the text "**`GridBagLayout`**" existed in the **`createButtons`** method using AZURITE's history search.
2. Select all the edit operations within the **`createButtons`** method performed since the point found in step 1.
3. Launch the interactive selective undo dialog (Figure 1-3). Then, from the left panel, indicate the parts of the current code that should be kept unchanged.
4. After checking the preview of the selective undo result shown in the right panel, press the OK button to actually perform the selective undo.

AZURITE provides a rich set of user interfaces designed to help users complete various backtracking tasks. The list of steps described above is just one example, and there are several different ways to achieve the same result using AZURITE. Users can use AZURITE in the way that they feel the most comfortable.

To evaluate the effectiveness of AZURITE on completing backtracking tasks, an A vs. B evaluation study was conducted with 12 programmers. The study results showed that the group using AZURITE was twice as fast compared to the control group, when completing the provided backtracking tasks.

1.5. THESIS

This dissertation work seeks to evaluate the following thesis statement:

> Programmers will be able to perform backtracking tasks more easily and accurately by having a *selective undo mechanism* for code editors, *visualizations* of code change history designed for selective undo, and *history search* options to express what they remember about the previous edits that they want to backtrack.

1.6. CONTRIBUTIONS

This dissertation makes the following major contributions:

- A recording tool for capturing low-level events and fine-grained edits in the code editor, which is used for performing the empirical studies of this thesis work, and by several other research institutions. The recording tool is also used for providing selective undo feature in the code editor. (Chapter 3)
- Findings from three empirical studies to understand programmers' backtracking behaviors (Chapter 4)
- A novel selective undo mechanism for code editors that is capable of dealing with regional conflicts among edit operations (Chapter 5)
- A novel interactive timeline visualization of fine-grained code edit history (Chapter 6)
- A novel mechanism for summarizing fine-grained code edits in real time to provide "semantic zooming" (Chapter 7)
- Novel user interaction techniques for providing usable interfaces for selective undo (Chapter 8)
- Evidence from a user study that the prototype selective undo tool is usable and enables programmers to perform certain backtracking tasks about twice as fast compared to when not using the tool (Chapter 9)
- An exploration of applying the selective undo idea in a painting application using a script-model selective undo mechanism, which discovered many interesting design issues from the user studies (Chapter 10)

1.7. OUTLINE

The rest of this dissertation is organized as follows. Chapter 2 starts with discussing the related work, including the various undo mechanisms, variation management systems, and history visualization systems. Chapter 3 presents our tool called FLUORITE, which is a logging plug-in for Eclipse that captures all the fine-grained code edits and IDE interactions. FLUORITE was used for the empirical studies and the evaluation studies conducted in this dissertation work. FLUORITE is also used as the input source of our selective undo tool: it forwards all the captured coding events to the selective undo core component. Chapter 4 presents the results from a series of empirical studies of backtracking, which show that programmers frequently need to backtrack and the existing tool support is quite limited. Chapter 5 describes the core selective undo mechanisms, including the internal data structure maintained to support selective undo and the selective undo algorithm. Chapter 6 introduces the timeline visualization of code edits, the most basic user interface for selective undo, which displays all the fine-grained code edits and allows users to select one or more past edit operations and invoke selective undo command. Chapter 7 describes a real-time algorithm for collapsing related fine-grained edits and displaying higher-level edits in the timeline. Chapter 8 describes a set of additional user interfaces specifically designed for selective undo and their design rationale. The described user interfaces include code history diff view, history search dialog, and the interactive selective undo dialog presented above. Chapter 9 discusses the evaluation of our prototype tool AZURITE which implements the aforementioned selective undo mechanisms and user interfaces, in terms of its usability, usefulness, and performance. Chapter 10 summarizes our effort on applying the selective undo approach in painting applications, and presents interesting design issues not pertaining to selective undo in code editors. Chapter 11 discusses the limitations of this work and potential future work directions, and Chapter 12 concludes.

2.

RELATED WORK

This research is inspired by and was built upon previous work done in various areas, including undo models, version control and variation management systems, collecting and utilizing fine-grained interaction history, software visualizations, and empirical studies of code editing. This chapter summarizes related work in each of these areas.

2.1. UNDO MODELS

One way to support backtracking is with undo commands. The most widely adopted model of undo is called the *restricted linear undo model* [Berlage 1994]. The system keeps a list of all the executed commands and users can only undo the most recently performed commands. In this model, a redo command is also supported, and it is always performed in the opposite order of the undo, in order to make sure that the commands are re-executed the same state where they were originally executed. Although this model is very popular and well understood by the users, it has several major limitations as described in Section 1.1.

There are other more sophisticated undo models providing additional commands beyond undo and redo, which essentially enable selective undo in an indirect way. The US&R model [Vitter 1984] allows users to *skip* redoing an operation, using a tree-based data structure. Users can selectively undo an isolated operation, by undoing multiple steps until the target operation gets undone, skipping the redo command once, and then redoing the rest of the operations. The triadic model [Yang 1988] uses a simpler structure composed of a linear history list, and a circular redo list which can be *rotated* by users. Undoing an operation puts the operation at the beginning of the redo list, and rotating the redo list takes one operation at the beginning of the list and puts it at the end. Since the rotate command can be used to skip a redo command, users can selectively undo a certain operation in a similar way. However, both models require deep understanding of the underlying history structure to correctly perform selective undo. In addition, selective undo cannot be done in one step, which can be cumbersome for users.

2.1.1. SELECTIVE UNDO MODELS

Selective undo has been extensively studied for object-based graphical, interactive editors. With selective undo, users can select an operation (called the *target operation*, hereafter) from the command history and undo that operation, isolated from the rest of the operations in the history. There are three types of selective undo models in general: *script model*, *inverse model*, and *cascading selective undo*.

In the *script model*, the system tries to guarantee that the final result to be as if the target operation had never been performed [Archer 1984]. That is, the system rolls back all the operations in the command history to the point immediately before the target operation was performed, skips the target operation, and reruns all the following operations that were not previously undone. This model has not been widely used, but we adopted this model for a pixel-based *painting application* (as opposed to a drawing application having identifiable objects), which will be discussed in Chapter 10.

In the *inverse model* (or direct selective undo), as introduced in GINA system by Berlage, the system adds the *inverse* of the target operation to the current context [Berlage 1994]. Thus, the selective undo command itself is added to the end of the command history. The Amulet [Myers 1996] and Topaz [Myers 1998] systems had a similar selective undo feature, and these also allowed repeating a selected command on a new object. To support this undo model, the editor commands should be represented by command objects, each with its own **`undo`** function [Myers 1996][Gamma 1994]. The inverse model is simpler compared to the script model in that the rest of the command history is not affected by the selective undo command. The selective undo for code editors described in this dissertation (Chapters 5 through 9) uses the inverse model.

The result of selective undo may be different between these two models in the presence of conflicts (or dependencies), which refer to the situations where there are some later performed operations in the history which are dependent on the target operation that the user is trying to undo. The following example, taken from [Berlage 1994], illustrates this point. Suppose that a graphical object is recolored with operation A, and then later that object is duplicated with a copy operation B. What would be the result of selectively undoing A? In the script model, both objects will return to the original color, because it works exactly as if the operation A had never happened. In the inverse model, however, only the original object will return to its original color without affecting the copied object, because the undo operation is applied to the current context.

There is another class of selective undo model called *cascading selective undo* which takes care of the conflicts [Cass 2005]. In this model, all the subsequent operations dependent on the target operation are all undone together, which eliminates the ambiguity described above. Their user studies showed that people could predict and understand what the system will do [Cass 2006][Cass 2007].

There are other applications providing selective undo features. Selective undo was applied in spreadsheets [Kawasaki 2004] by allowing users to select a region in the spreadsheet and perform *regional undo*. Dwell-and-spring [Appert 2012] is a selective undo mechanism for direct manipulation. It provides an interface for undoing any press-drag-release interaction.

However, unlike in these graphical applications, 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.

Finally, all of these approaches assume that there is an object on which the operations can be performed: primitive graphical objects such as shapes in graphical editors, and individual cells in spreadsheets. In contrast, there is no clear notion of objects in text and code editors,[1] since edit operations typically affect ranges of text, and the text itself moves around and is changed. The same problem occurs in painting programs, since edit operations typically affect areas of pixels.

The issue of conflicts among operations is not limited to the object-based graphical editors. In fact, text and code editors face the exact same issue when the region of a later performed edit operation overlaps with the region of another earlier performed edit, which is referred to as a *regional conflict* (see Section 5.1.2). In AZURITE, when the user tries to selective undo some edit with one or more conflicts, AZURITE provides the user with several alternative results to choose from.

2.1.2. REGIONAL UNDO IN TEXT EDITORS

Some text editors such as Emacs[2] and DistEdit [Prakash 1994] support regional undo, where the user undoes the most recent operation that affected a specific selected region of text, which can be seen as a special case of selective undo. Regional undo is useful and also relatively easy to implement compared to the generic selective undo, because it always undoes the most recent operation performed in the selected region, which guarantees that there are no regional conflicts with the target operation. Regional undo is directly supported in AZURITE using a keyboard shortcut, by searching for all edits for the region of code and invoking selective undo on the last one, or by using code history diff view and using the revert button. In regional undo, however, there can be an ambiguity if the user selects a region which partially overlaps with an operation's effective region. Li and Li refer to this problem as region overlapping, and introduce the idea of *partial undo* as a solution, which undoes only overlapped part of the operation when an operation partly falls in the given undo region [Li 2003]. In this situation, AZURITE would do the same thing when using the code history diff view or the regional undo shortcut to revert a certain region of code to one of the previous versions.

[1] In fact, projectional editors (or structured editors) such as JetBrains MPS (https://www.jetbrains.com/mps/), in which users can directly edit the underlying document structure (e.g., abstract syntax tree), do have objects. However, they do not usually provide selective undo features, but in theory, it would be possible to apply the existing selective undo approaches to implement per-object selective undo.

[2] http://www.gnu.org/software/emacs/manual/html_node/emacs/Undo.html

2.1.3. *Tree-Based Undo Models*

As seen in the US&R model [Vitter 1984], one way to extend the conventional linear undo is to keep the edit history as a tree instead of as a linear list. When the user undoes multiple steps and makes a new edit from there, it creates a new branch in the history tree and puts the new operation in it, while keeping the previously undone operations in the previous branch. One of the problems of this approach is that it becomes difficult to provide useful and usable interfaces for users. Moreover, selective undo cannot be clearly presented in a history tree, because a selective undo command would create a new node which has never been visited before, thus making it not distinguishable from any other normal operations in the history.

Several text editors and plug-ins provide tree-structured visualizations which allow users to move around among the different nodes and make new changes from any of the existing nodes [Losh 2012][Cubitt 2010]. However, as the edit history gets bigger, it becomes more difficult to understand the history because the nodes do not provide sufficient useful information for the user to navigate the tree.

2.1.4. *Operational Transformation in Collaborative Editing*

Another use of selective undo is in collaborative editing, where multiple people can edit a document concurrently, which has been studied by various systems (e.g., [Ellis 1989][Berlage 1993][Choudhary 1995][Prakash 1994][Sun 2002]). For a real-time collaborative editing tool, maintaining consistency of a document across different sites is a major challenge, in the presence of multiple local copies of the shared document and network latency. To address this problem, a line of technology called *operational transformation* (OT) has evolved by the Computer-Supported Cooperative Work (CSCW) community [Sun 1998]. OT provides formal foundations for maintaining consistency properties.

Sun gives a good summary of how OT can be used to perform a selective undo operation correctly in a collaborative editing environment [Sun 2002]. The high-level idea is that when a selective undo of operation O is invoked, the system processes the undo command as if it was an inverse operation $\bar{O}$ generated immediately after O, that is concurrent with all the other operations afterwards. Then, the system can process the undo operation using the well-defined rules of OT.

This approach has a number of major differences from the selective undo approach described in this dissertation. First, because the main purpose of OT is to maintain document consistency, this approach is excessively complicated for the purpose of providing a single-user selective undo.[3] Second, the OT-based undo approach modifies the history buffer. After successfully undoing operation O, the inverse operation $\bar{O}$ would be added to the history buffer immediately after O, and all the rest of the operations in the history buffer would be transformed against $\bar{O}$. Essentially, this makes the $O \circ \bar{O}$ pair a no-op, and works similar to the script-based selective

[3] Joseph Gentle, author of ShareJS (http://sharejs.org/), a web library for OT, says "I am an ex Google Wave engineer. Wave took 2 years to write and if we rewrote it today, it would take almost as long to write a second time."

undo model. In contrast, AZURITE uses the inverse model and always adds the undo operations at the end of the history.

Finally, and most importantly, the issue of regional conflict is still not very well defined in OT-based undo approach. Due to the inherent complexity of the consistency issue, OT literature almost always considers only two primitive edit operations: single character insertion and single character deletion. When the edits are limited to single character, the regional conflict does not even occur, because the edits do not have the notion of *edit regions*. However, this may not be very practical, because users would not want to undo only individual character level edits. For example, most of the available text and code editors automatically group a series of character edits and make it possible to undo at a higher-level. Similarly, pasting text over a selected region causes a range of text to be replaced.

2.1.5. OTHER UNDO MODELS

Similar to operational transformation approach, Hayashi et al. proposed the idea of edit history refactoring, which is a restructuring of an edit history without affecting the final result of the code, and implemented it in their system called Historef [Hayashi 2012][Hayashi 2015]. Historef also provides a selective undo feature using history refactoring. The selected operations are first moved to the end of the history using *swap* refactoring, the changes are *merged* into a

Undo Model	Structure	Return to Any Previous State?	Selective Undo Support	Selecting Target Operation(s)	Reference
Restricted Linear Undo	Linear list		Not Supported	N/A	[Berlage 1994]
History-Tree Visualizations	Tree	Yes	Not Supported	N/A	[Losh 2012][Cubitt 2010]
Photoshop Non-linear Undo	Non-linear list	Yes	Not Supported	N/A	
Revision Control for Images	Directed acyclic graph	Yes	Not Supported	N/A	[Chen 2011]
US&R	Tree		Undo, skip, then redo (manual)	Indirect	[Vitter 1984]
Triadic Model	Undo list + Redo list (rotatable)		Undo, rotate, then redo (manual)	Indirect	[Yang 1988]
Script-Model Selective Undo	Linear list		Pretend that the target operation never happened	Direct	[Archer 1984] Aquamarine (Chapter 10)
Inverse-Model Selective Undo	Linear list	Yes	Add the inverse operation at the end	Direct	[Berlage 1994] [Myers 1996] AZURITE (Chapters 5-9)
Cascading Selective Undo	Linear list		Undo all the conflicting operations together	Direct	[Cass 2005]
Regional Undo	Linear list		Filter the operations in the region, and undo them	By region	[Li 2003] [Kawasaki 2004]
History Refactoring	Linear list	Yes	Move the target operations to the end	Direct	[Hayashi 2012]

Table 2-1. Feature table of the existing single-user undo models.

single operation, and then the inverse operation of it is executed. This approach, however, cannot address situations where the operations conflict, which our selective undo can handle. Historef also does not provide any visualizations or history search mechanisms that would help users to find and select the operations to be undone.

Adobe Photoshop provides a history window with a mode for “non-linear undo”, but this is different from selective undo—when the user undoes operations and then does new ones, Photoshop’s non-linear undo retains the undone operations on the undo stack rather than remove them. However, future undos still start at the last operation and continue backwards through all previous operations in order.

Chen et al. presented another system that provides a revision control system for images based on a directed acyclic graph (DAG), which enables users to make forks and joins and then move around in the history and see various versions of images [Chen 2011]. However, it does not support selective undo or the script model.

As a summary, a feature table of the existing undo models for single-user environments described above is provided in Table 2-1.

2.2. Version Control and Variation Management Systems

A version control system (VCS) can be seen as a variation management system [Conradi 1998]. Traditional centralized version control systems such as Subversion help programmers 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 in Section 1.1 above, the user must think to commit the desired version, which may not happen if the programmer only later realizes that backtracking is needed. 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.

2.2.1. Features of Git Related to Selective Undo

In recent years, a distributed version control system (DVCS) called Git[4] became one of the most popular version control systems in the software development community. The 2014 version of the Eclipse community survey[5] reports that Git is the first most used VCS (33.3%), surpassing Subversion (30.7%), which used to be the dominant tool. In a DVCS environment like Git, a programmer normally works with a local clone of the public repository, and *pushes* the local changes to the shared repository only when the local version seems to be stable. This style of workflow can mitigate the problem of committing an unstable piece of code because the local clone does not affect the repository of other colleagues, although committing is still a fairly heavyweight process.

[4] http://git-scm.com/

[5] https://ianskerrett.wordpress.com/2014/06/23/eclipse-community-survey-2014-results/

Git provides a large set of powerful features, some of which are closely related to selective undo. When the changes to the code that the programmer wants to undo (called *target changes*, hereafter) are already grouped as a single commit in the history, there are multiple ways to revert that particular commit in the current context, which is essentially selectively undoing the changes at a coarse-grained level. The "**`git revert`**" command can be used to create an inverse commit of the target commit and it adds the new commit at the end of the commit history. Git also provides the "**`git cherry-pick`**" command, which is essentially a *selective redo* command. Using the cherry-pick command, users can apply some of the changes from one branch to another. This command can be used to mimic selective undo between branches, because users can cherry-pick all the commits except for the commit containing the target changes.

One limitation of this approach is that the target changes must have already be isolated as a single commit, separated from the other changes. In other words, if the target changes are intermixed with other changes in a single commit, it can be very tedious to selectively undo only the target changes. Another limitation is that the revert and the cherry-pick commands can cause regional conflicts, in which case the user has to manually fix all the conflicts and then commit again. Although regional conflicts can also occur in AZURITE, there are a few important differences. First, because Git keeps track of the line-level changes, the regional conflict may occur even if the changes are not overlapping, when the changes are made in the same line. For example, imagine there is a variable declaration statement with an initial value, such as "**`int foo = 1;`**" (v1). The user changes this code to "**`int bar = 1;`**" (v2), and then to "**`int bar = 2;`**" (v3). Assuming these three versions are separately committed to Git, the user should be able to use the **`git revert`** command to undo the variable name change. However, because these two changes were made in the same line, Git will produce an error message indicating that there is a conflict that needs to be resolved by the user (Figure 2-1), and when the user opens the file, the conflicting part is marked as shown in Figure 2-2, and the user needs to manually fix the code to the desired state and then invoke the **`git commit`** command to finish the revert operation.

```
error: could not revert 6a7f7ad... Foo to Bar
hint: after resolving the conflicts, mark the corrected paths
hint: with 'git add <paths>' or 'git rm <paths>'
hint: and commit the result with 'git commit'
```

Figure 2-1. Error message generated by Git showing there is a conflict that should be resolved.

```
<<<<<<< HEAD
int bar = 2;
=======
int foo = 1;
>>>>>>> parent of 6a7f7ad... Foo to Bar
```

Figure 2-2. The content of the file that has the conflict. The user need to manually resolve this conflict and then make another commit to finish the revert operation.

In contrast, selectively undoing the variable name change using AZURITE would not result in a conflict, because AZURITE keeps the accurate regions of the individual edits, even within a single line. Even when there is a conflict, the conflict resolution could be done by simply clicking one of the options provided by AZURITE, which is much easier compared to the manual conflict resolution process of Git. Granted, there are numerous third-party visual merge tools that can be used in conjunction with Git which helps resolving conflicts (e.g., DiffMerge,[6] Araxis Merge[7]), but they still require some manual work from the user.

There is another situation where Git can help achieve selective undo. Suppose that there are local code changes which are not yet committed and some of those changes should be selectively undone. In other words, only some of the local changes should be selectively committed. This can be achieved using the "**`git add --patch`**" command, which presents an interactive command line interface where the user can review each *hunk* (Git terminology referring to a contiguous lines of changes) in the local changes and decide whether to include the hunk in the commit or not. The closest feature provided in AZURITE is the interactive selective undo dialog (Figure 1-3, Section 8.4), which allows users to review the selective undo result and dynamically add or remove edit operations. The "**`git add --patch`**" command, however, cannot be used to restore code that is neither in the committed code nor in the local changes (i.e., some code that was produced after the last commit but removed in the current local version), whereas AZURITE can be used to restore such code.

2.2.2. *LOCAL HISTORY FEATURES OF INTEGRATED DEVELOPMENT ENVIRONMENTS*

Most of the popular IDEs support local history keeping features, where the snapshots of each source file is automatically kept in a history upon file save (e.g., Eclipse, Visual Studio[8]) or as the code changes (e.g., NetBeans). Xcode 4 has a feature called Version Editor, where the history of a file is displayed in a code compare view with two panels, and users can move through the history using the vertical timeline located between those two panels. However, these features are limited compared to AZURITE in that (1) the history is shown in a linear list without any human-readable descriptions or cues, (2) changes can only be seen at the file level, (3) history search is not supported, and (4) selective undo is not directly supported, so users must compare the local and the desired older versions and merge the wanted changes manually. Similar to these IDEs, cloud-based text editors such as Google Docs support linear revision history, but with the same limitations.

2.2.3. *OTHER VARIATION MANAGEMENT SYSTEMS*

Backtracking becomes important when trying out multiple alternative solutions. There exist several tools that help with variation management. Juxtapose [Hartmann 2008] enables developers to add an alternative at any time, and allows them to move among alternative source

6 http://www.sourcegear.com/diffmerge/

7 http://www.araxis.com/merge/index.en

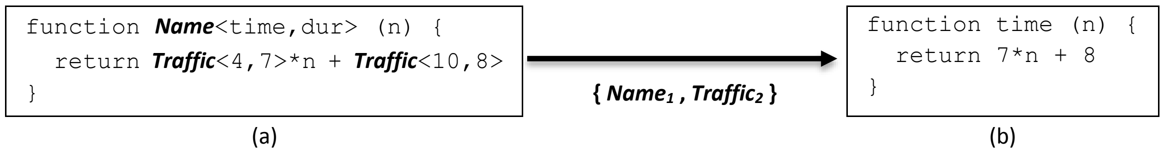
8 http://blogs.msdn.com/b/visualstudio/archive/2014/01/23/auto-history-extension-in-visual-studio-2013.aspx

files. When testing the application, multiple alternatives can be juxtaposed and the developer can compare the results directly. Also, Juxtapose automatically generates widgets for tunable application parameters so that the developer may change the values at run time and see the results without recompiling the whole application. Terry et al. proposed the Parallel Paths model [Terry 2004], which allows users to create a new variation at any time around a single command invocation, see the variations simultaneously in a single workspace, and edit them individually or as a whole. However, users must know in advance when they want to add variations in Juxtapose, and Parallel Pies works only in the graphical editing context. Barista [Ko 2006] had an alternative expressions tool which allows selecting an alternative by clicking on one of the listed choices, but it was restricted to the expression level.

2.2.4. Formal Representations of Variations

Other work has studied ways to formally represent and manipulate source code variations. Choice calculus provides a generalized representation for software variations at the code level and provides theoretical foundations of variation management [Erwig 2011][Walkingshaw 2013]. Choice calculus provides a syntax to represent variations within a variational program, semantics for the representation, and semantic preserving transformation laws which can be used by tools implementing choice calculus.

Consider the following example[9] *variational program* using choice calculus (Figure 2-3a).



```
function Name<time,dur> (n) {
  return Traffic<4,7>*n + Traffic<10,8>
}
```

```
function time (n) {
  return 7*n + 8
}
```

Figure 2-3. An example variational program annotated with choice calculus (a), and one of the variants obtained by selecting the first alternative from Name dimension, and the second alternative from the Traffic dimension (b).

In this variational program, there are two *dimensions*: **`Name`** and **`Traffic`**. Each of these dimensions has two *alternatives*. For example, the **`Name`** dimension has two alternatives: **`time`** and **`dur`**. The named tuple of alternatives (i.e., dimension + alternatives) such as **`Name<time,dur>`** is called a *choice*. A program *variant* can be obtained from a variational program by *selecting* the index of alternatives for each dimension. Figure 2-3b is an example variant obtained by selecting the first alternative of the **`Name`** dimension, and the second alternative of the **`Traffic`** dimension. Note that the **`Traffic`** dimension appears in multiple locations in the variational program. In this case, all the choices with the same name (dimension) should have the exact same number of alternatives, and only the alternatives at the same index can be selected together. For example, the alternatives **`4`** and **`8`** cannot be selected together in the example.

Although the original motivation for choice calculus is to support developing, maintaining, and analyzing variations in software, for example in software product lines, it is also closely related

[9] This example was originally created by Martin Erwig.

to selective undo. In theory, each code edit can be translated into a new choice, which is called the *choice edit model*. In this model, an Insert of **`foo`** can be can be represented as a choice **`X<ϕ,foo>`**, a Delete of **`foo`** as **`Y<foo,ϕ>`**, and a Replacement of **`foo`** to **`bar`** as **`Z<foo,bar>`**. Given a variational program annotated with this choice edit model, selective undo can be performed by selecting a different alternative for a particular dimension, while leaving the rest of the selections unchanged.

Using this choice edit model for selective undo is interesting in several aspects. First, regional conflict can be modeled as nested choices. Users could either undo a certain edit along with all the other edits depending on the target edit (i.e. *conflictees* as defined in Section 5.1.2) or leave the code unchanged. To compare this model with the conflict resolution interface of AZURITE, this model would only provide the options A2 and A3 without providing A1 (Section 5.2.2.2).

Another interesting aspect is that semantic dependencies (e.g., renaming method in the definition and all its call-sites) can be represented as a same-named dimension appearing in multiple locations of the variational program. Note, however, the notation does not provide anything about *how* to determine these semantic relationships among code edits.

A fundamental limitation of this model of selective undo is that it is difficult to translate all the fine-grained code edits into the choice edit model in practice. Since the amount of fine-grained edits generated in the code editor is fairly large, the variational program using the choice edit model would get more and more complicated very quickly with lots of dimensions and nested choices, which is likely to be unmanageable for the users.

2.3. COLLECTING AND UTILIZING FINE-GRAINED INTERACTION DATA

As part of this dissertation work, a longitudinal study of backtracking was conducted by analyzing fine-grained code edit logs captured by our FLUORITE tool (Section 4.3). This study can be seen as a software evolution study performed at a fine-grained level. While mining software repositories [Kagdi 2007], a popular software evolution research methodology, works at the commit level, our analysis was performed at the individual code edit level. For the backtracking study, it was necessary to use the fine-grained history, because programmers would often backtrack while experimenting, and the intermediate versions are very unlikely to be captured in version control system histories, which motivated the development of FLUORITE (Chapter 3).

2.3.1. FINE-GRAINED INTERACTION DATA COLLECTION TOOLS

There exist other tools that capture fine-grained code edits and/or user interactions with the IDEs. Mylyn keeps track of the user interaction history internally in order to derive the task context [Kersten 2006][Murphy 2006]. Using the Mylyn Monitor API,[10] 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

[10] http://wiki.eclipse.org/Mylyn/Integrator_Reference

the Mylyn Monitor collects more abstract user interaction data on the entire IDE. For example, when the programmer 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)[11] was another useful source of programmers' Eclipse usage data.[12] The UDC collected usage information from all the Eclipse users all over the world who consented to upload their usage data to the UDC. The UDC publicized several usage reports including the commands report. These reports have been used by many researchers (e.g., [Parnin 2009][Murphy-Hill 2009]). However, the command usage report from UDC was not suitable for my backtracking study because it did not capture some important commands executed in the code editor. It ignored 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 Eclipse commands or keyboard shortcuts. In contrast, FLUORITE collects all commands regardless of how they are invoked.

There exist other research tools that capture fine-grained code changes as FLUORITE does. OperationRecorder [Omori 2008] and CODINGTRACKER [Vakilian 2012][Negara 2012][Negara 2014] both take the raw text changes as inputs and turns them into AST-level change operations, whereas FLUORITE logs all the textual changes as-is. IDE++ [Zhongxian 2012] is a system that captures all types of IDE interactions, which are not limited to code edits. The data can be used in various ways, and there are other researchers who have analyzed their own fine-grained code change data to extract different information. Vakilian et al. collected detailed usage data of Eclipse refactoring tools using their CODINGSPECTATOR tool, and analyzed the data to discover usability problems of the refactoring tools [Vakilian 2014]. In their analysis, they detected the situations where the users used the refactoring tools in a way that is not ideal, indicated, for example, by cancellations or undoing of the refactoring commands. As another example, CODINGTRACKER logs were analyzed by adapting existing data mining techniques [Negara 2014], which is different from our per-node history keeping approach. They identified 10 previously unknown program transformation patterns. This shows that analyzing fine-grained code change history can be useful in many different ways. This line of empirical research is being continued by a team of researchers (in the COPE project: Change-Oriented Programming Environment[13]), and they are studying developers' test-driven development practices using the fine-grained logs they are collecting. Although CODINGTRACKER and IDE++ are similar to FLUORITE in that they also capture the fine-grained code edits from the code editors, I could not use them, unfortunately, because they were independently developed by different research groups in parallel with FLUORITE and could not be integrated with AZURITE.

[11] https://eclipse.org/epp/usagedata/

[12] Unfortunately, the Eclipse Usage Data Collector (UDC) project has been discontinued since 2010.

[13] http://cope.eecs.oregonstate.edu/

2.3.2. Replaying Fine-Grained Interaction Data

Syde is a tool for Eclipse that can record fine-grained change history of Java-based systems in multi-programmer settings [Hattori 2010a][Hattori 2010b]. This tool is intended to increase team awareness and help programmers understand the code evolution, but it could 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. As a follow-up tool, they developed Replay, a tool that can be used to replay the changes recorded with the Syde tool [Hattori 2011]. Their empirical study showed that programmers can answer the software evolution related comprehension questions in a significantly shorter time, when compared to using a traditional version control system.

Fine-grained code edit scripts can be used for creating coding tutorials with examples. JTutor [Kojouharov 2004] is a coding tutorial creator / replayer tool suite for Eclipse for students who are learning Java programming. Similar to FLUORITE, JTutor uses an XML-based data structure, with the initial snapshot and all the subsequent changes represented as individual steps. Similarly, SmartTutor [Zhang 2009] is a tutorial recorder / replayer tool that works in Eclipse, but it focuses on teaching how to use the IDE features, while JTutor focuses more on teaching how to program. Ginosar et al. created a coding tutorial editor tool for the Processing language, but the main focus was to make it easier for the tutorial creators to edit the existing tutorial scripts with tool support [Ginosar 2013].

2.3.3. Use of Fine-Grained Code Edits in Refactoring

BeneFactor [Ge 2012] is a refactoring tool that detects ongoing, incomplete manual refactoring while the programmer is editing the code and offers a command to finish the rest of the refactoring activities automatically. The refactoring detection process involves monitoring the fine-grained code edits and checking if a series of code edits match one of the pre-defined refactoring patterns, which are defined as state machines. Once the programmer asks BeneFactor to finish the manual refactoring, BeneFactor rolls back the manual refactoring (to revert the code context to the state where the automatic refactoring command can be correctly executed), and then invokes the automatic refactoring command to finish the desired refactoring. If the programmer made some interleaving edits with the manual refactoring that are independent from the refactoring, BeneFactor preserves those independent edits while rolling back. Interestingly, the paper refers to this process as *selective undo*, because only the edits that are part of the refactoring are being undone selectively. The selective undo here is an internal algorithm used to achieve the refactoring, not an explicit command that can be invoked by the users. The actual selective undo algorithm used in BeneFactor is similar to the script-based selective undo model: all the edits are undone first, and then only the non-refactoring edits are re-executed, skipping all the refactoring related edits.

2.3.4. Analyzing Fine-Grained Interaction Data to Discover Usability Problems

Detailed tool usage data can also be used to identify usability problems of specific tools. Akers et al. devised a study method called *backtracking analysis*, which is designed to capture usability problems of graphical creation-oriented programs such as Google SketchUp [Akers 2012]. To capture richer contextual information, their system automatically captured both the screens of participants and the backtracking events such as undo or erase. In their backtracking analysis, backtracking events are assumed to be indicators of usability problems of the creation-oriented programs. In contrast, our work aims to support programmers to backtrack more easily and effectively, with the premise that backtracking events in code editing are natural in exploratory programming activities.

2.4. Edit History Visualizations & Search Tools

One of the most important user interfaces provided in AZURITE is the timeline visualization of code edits (Chapter 6). There are other edit history visualizations using timelines. Chronos [Servant 2012] shows the results of history searches in a zoomable timeline. Since Chronos is designed to work with coarse-grained version control history, however, it is not adequate for visualizing a large amount of small edits. CodeTimeline [Kuhn 2012] is a visualization for presenting the *social history* of a software project, similar to Facebook's Timeline. Programmers can manually add sticky notes or photos to recall the social events associated with the project. It also visualizes some level of edit history information such as the lifecycle of all files and the code ownership. Automark[14] is a plug-in for Visual Studio, which generates a HTML or Markdown formatted coding history including the actual code edits, visited Stack Overflow questions or documentation pages, which is designed to help recover programmers' episodic memory after an interruption [Parnin 2012] or facilitate sharing a coding history with other people. The software evolution Storyline [Ogawa 2010] is another timeline visualization which focuses on who contributed to the project over time. These history visualization tools are primarily designed for helping people recall and share memories, not for providing editor commands as provided by AZURITE.

Aquamarine, our prototype painting application providing selective undo features (Chapter 10), displays the past interactions in a graphical History pane (Figure 10-1). There has been significant research on such displays. Chimera provided graphical histories as thumbnail snapshots which could be edited and re-used, and past actions could be modified [Kurlander 1988], but conflicts among operations were not specifically identified. The Designer's Outpost shows snapshots of the history of states of a web editing session with multiple users and keeps track of forks among versions [Klemmer 2002]. Systems have also used graphical histories to foster learning [Chi 2012][Grossman 2010] and creating macros for later reuse [Kurlander 1988][Lieberman 1992].

[14] https://visualstudiogallery.msdn.microsoft.com/078d00b7-dfbd-4cfa-97f9-8be08bb510ee

There are systems that provide history search, which has also been called "history slicing." OperationSliceReplayer [Maruyama 2012] uses the AST data kept by OperationRecorder [Omori 2008] to filter the changes that affected a certain class member. Chronos [Servant 2012] uses the version control snapshots to trace back to find which commits affected a certain area of code. The search scope of Chronos can be as small as a single line. These history search features are limited to region-based search, whereas the history search of AZURITE can find the target text in the history, even when the text does not even exist in the current code.

2.5. EMPIRICAL STUDIES 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 [Kim 2004]. Ko et al. analyzed programmers' character level code-editing strategies [Ko 2005b]. 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 [Murphy-Hill 2009][Kim 2011]) also focus on how programmers make changes to code over time, but they are often limited to revision-level changes.

2.6. CONCLUSION

While researchers have been studying various human aspects of software development, the knowledge about programmers' backtracking behavior was very limited. This motivated my empirical studies of backtracking, which are discussed in Chapter 4. Many undo models have been proposed to help users backtrack and facilitate exploration, but none are directly applicable to today's source code editing environments due to their limitations. The existing undo models do not describe how to handle the edit operation conflicts in a code (or text) editing context. Version control systems can help programmers backtrack their code changes, but only if those target changes have already been committed and are well separated from the other irrelevant changes. In Chapters 3 & 5, I explain how AZURITE can make use of the fine-grained code edits to provide and handle edit operation conflicts.

Moreover, there is little evidence that the existing undo models are actually usable and useful for the users. In Chapters 6-8, I describe a set of novel user interfaces designed for selective undo, and then evaluate those designs in Chapter 9.

3.

CAPTURING FINE-GRAINED CODING EVENTS FROM THE CODE EDITOR[15]

Little was known about the backtracking behaviors of programmers when this research started. I first looked for existing research methods or data that I could analyze to gain more insights about backtracking, but none of them were suitable for the purpose of understanding backtracking behaviors. Therefore, I created my own fine-grained coding event data collection tool, which is the main topic of this chapter. First, the existing research methods and data sets are reviewed, and why they were not suitable for the backtracking research is explained.

3.1. RELATED WORK

There are many different sources of programmers' usage data, each with its own strengths and weaknesses. One way is to directly ask the programmers 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 programmers (e.g., the undo command), 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 [Beyer 1997] 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 [Ko 2004][Ko 2005a][Coman 2008][Ko 2003]) 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. For example, many researchers have used this method to gain insights about code clones [Kim 2005a][Aversano 2007][Bettenburg 2009] and how the programmers refactor [Murphy-Hill 2009][Xing 2006][Kim 2011]. 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 programmers might have used to

[15] Portions of this chapter appeared in [Yoon 2011].

change the code from one revision to the next. Also, some of the popular version control systems such as Git provide the ability to edit the existing commit history (e.g., rebase, squash) and thus there is a high chance that the public repository does not show the software evolution history as it actually happened [Bird 2009].

When studying the backtracking behavior of programmers, mining software repositories is inadequate and having access to the low-level code editing and/or tool usage data becomes even more important, because it is likely that much backtracking is done as part of some experimentation *locally*, without being committed to public source code repositories. Although there were several existing methods for gathering tool usage data, there was none that was suitable for analyzing fine-grained code editing history without requiring laborious manual inspection.

3.2. FLUORITE: FINE-GRAINED CODING EVENT LOGGER FOR ECLIPSE

In order to address these limitations, I built a publicly available event logging plug-in for Eclipse called FLUORITE[16] as part of my research. 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 common metadata such as command IDs and the timestamps indicating when the command was executed, but also additional parameters specific to the type of command. For example, a **`Find`** command has additional **`searchText`** and **`replaceText`** parameters. In the case of text editing events, the inserted and/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 programmers' complex code editing strategies which are often composed of *sequences* of commands. For example, it was seen in the collected logs (see Section 4.1) that backspace was 12.41% of all the keystrokes in code editing, and it was 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 change-log histories, or other high-level logging tools.

Although the Eclipse Usage Data Collector (UDC) data provides the detailed timestamp of the editor commands executed, which enables the event sequence analysis, the data does not contain the actual source code or the fine-grained textual changes. In contrast, with FLUORITE logs, 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 what situation a command was executed.

[16] FLUORITE is a mineral, and here it stands for: **F**ull of **L**ow-level **U**ser **O**perations **R**ecorded **I**n **T**he **E**ditor.

Since the FLUORITE log files contain the actual source code in them, the tool should not be used in a situation where the source code is confidential. This is what makes it difficult to collect data from the software industry, where most of the code is proprietary. It may also be inappropriate to share the collected data, and we could not publicize the logs collected during our empirical studies for the same reason (Chapter 4).

To mitigate this problem, FLUORITE does not upload log files automatically, so in a field study, the investigator would have to ask the study participants to send the log files whenever the code being edited is not confidential.

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 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 [Ko 2005a], 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, as described later in Chapters 5 & 6, FLUORITE's logging and analysis can be used in real-time to support novel code editing operations that depend on the history.

3.3. FLUORITE IMPLEMENTATION

FLUORITE is implemented as an Eclipse plug-in because Eclipse is one of the most widely used integrated development environments (IDEs). The FLUORITE code was based off of an open source Eclipse plug-in called Practically Macro,[17] but it was not complete enough because some important commands and parameters were missing (e.g. the **`FileOpen`** command), and it was not stable enough to record long sessions. Therefore, I augmented it to record all the commands and their parameters, increased its stability, and I also 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 3-1.

3.3.1. TYPES OF LOGGED EVENTS

There are three types of events that FLUORITE logs: *commands*, *document changes*, and *annotations*. The full list of different types of events is shown in Table 3-1.

[17] http://sourceforge.net/projects/practicalmacro/

```
<Command __id="2" _type="MoveCaretCommand" caretOffset="142" docOffset="142"
timestamp="3977"/>
<Command __id="3" _type="EclipseCommand" commandID="eventLogger.styledTextCommand.SE-
LECT_LINE_DOWN" timestamp="5598"/>
<DocumentChange __id="4" _type="Delete" docASTNodeCount="22" docActiveCodeLength="125"
docExpressionCount="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 3-1. Example log generated by FLUORITE. The developer (1) moved the cursor by clicking the mouse button, (2) selected one line by **Shift+DownArrow**, (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 the **DocumentChange** event.

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.

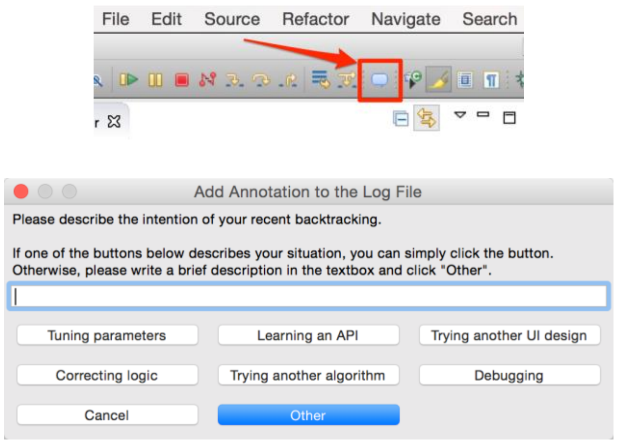
A *document change* event is logged whenever the active file is changed by any executed command. Each document change event contains the actual deleted or inserted text. This is needed because it is not always possible to correctly reproduce the snapshots of the files by capturing only the commands. For example, when the programmer 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 of

Event Type	Detailed Type	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
	Junit	Run/Debug Junit tests
	MouseWheel	Scroll the code editor with the mouse
	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

Table 3-1. List of the different types of events captured by FLUORITE.

getting the actual change results for each command: just reading the preceding[18] 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 programmer 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 3-2, 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.



Please describe the intention of your recent backtracking.

If one of the buttons below describes your situation, you can simply click the button.
Otherwise, please write a brief description in the textbox and click "Other".

Figure 3-2. Annotation toolbar button and its dialog box.

3.3.2. 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 (Table 3-2) 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 (Table 3-3), in order to keep track of the code size changes.

Parameter	Description
id	Unique ID (sequentially incremented)
type	Detailed event type (cf. Table 3-1)
timestamp	Timestamp relative to the session start time
timestamp2	(*optional*) Timestamp of the last merged event
repeat	(*optional*) Number of events merged together

Table 3-2. List of the common parameters.

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

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

Table 3-3. List of the code size metrics logged for the document change events.

3.3.3. Merging Consecutive Events

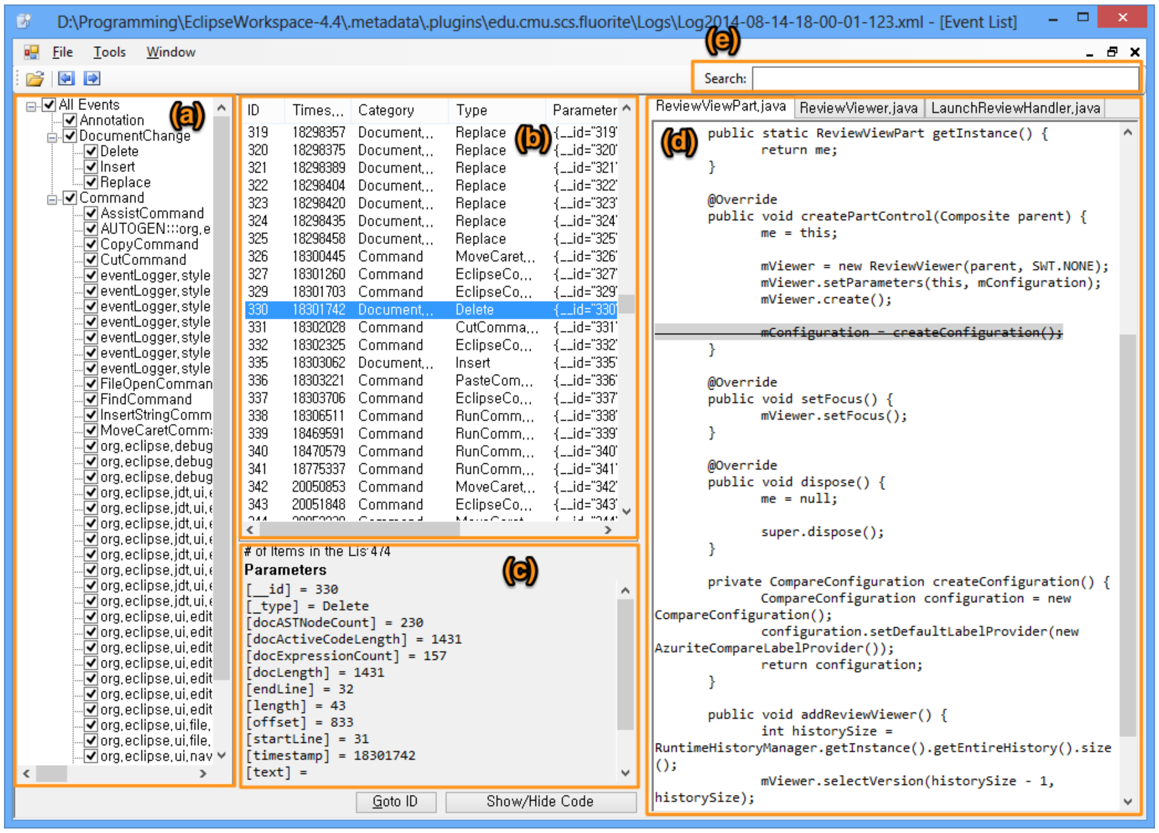
In order to prevent the log files from being unnecessarily large, FLUORITE merges multiple events of the same type in a row whenever possible. For instance, when the programmer moves the cursor to ten lines by holding down the up arrow key, the ten events are merged together as one XML element and its **`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, which is set to 2 seconds by default, and is configurable. This is similar to the way character typing sequences are merged for the undo command in other text and code editors.

3.4. Fluorite Analyzer

Along with the FLUORITE logger for Eclipse, a FLUORITE log analyzer is also provided on our website (Chapter 12), which makes it much easier to manually inspect the logs and produces several types of basic analysis reports and visualizations. In this section, the basic analysis features of FLUORITE analyzer are demonstrated.

3.4.1. Event List

FLUORITE analyzer provides an event list interface, where all the coding events in the log file are displayed (Figure 3-3). The main event list area is in the center of the screen (b). The events are displayed in chronological order in this list. The events can be filtered by their types in the leftmost panel (a). For instance, checking only the document change events and unchecking the rest will make the event list display only the code changes. When an event is selected in the list (the row highlighted in blue), the detailed parameters are displayed in the bottom panel (c), and the right panel shows the source file which was active when the selected event was generated (d). The source code panel also indicates the last code change made in that file. In the example, one line of code was deleted by the selected document change event. The search panel (e) allows searching for events having any parameter values containing the search text.



ID	Times...	Category	Type	Parameter
319	18298357	Document...	Replace	{__id="319"
320	18298375	Document...	Replace	{__id="320"
321	18298389	Document...	Replace	{__id="321"
322	18298404	Document...	Replace	{__id="322"
323	18298420	Document...	Replace	{__id="323"
324	18298435	Document...	Replace	{__id="324"
325	18298458	Document...	Replace	{__id="325"
326	18300445	Command	MoveCaret...	{__id="326"
327	18301260	Command	EclipseCo...	{__id="327"
329	18301703	Command	EclipseCo...	{__id="329"
330	18301742	Document...	Delete	{__id="330"
331	18302028	Command	CutComma...	{__id="331"
332	18302325	Command	EclipseCo...	{__id="332"
335	18303062	Document...	Insert	{__id="335"
336	18303221	Command	PasteCom...	{__id="336"
337	18303706	Command	EclipseCo...	{__id="337"
338	18306511	Command	RunComm...	{__id="338"
339	18469591	Command	RunComm...	{__id="339"
340	18470579	Command	RunComm...	{__id="340"
341	18775337	Command	RunComm...	{__id="341"
342	20050853	Command	MoveCaret...	{__id="342"
343	20051848	Command	EclipseCo...	{__id="343"

# of Items in the Lis 474

Parameters

```
[__id] = 330
[_type] = Delete
[docASTNodeCount] = 230
[docActiveCodeLength] = 1431
[docExpressionCount] = 157
[docLength] = 1431
[endLine] = 32
[length] = 43
[offset] = 833
[startLine] = 31
[timestamp] = 18301742
[text] =
```

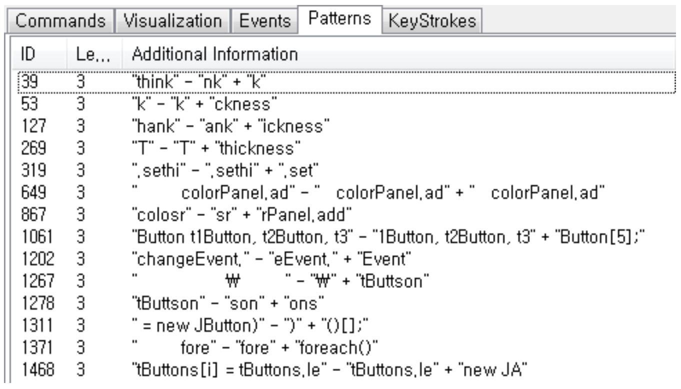
```
	public static ReviewViewPart getInstance() {
		return me;
	}

	@Override
	public void createPartControl(Composite parent) {
		me = this;

		mViewer = new ReviewViewer(parent, SWT.NONE);
		mViewer.setParameters(this, mConfiguration);
		mViewer.create();

		~~mConfiguration = createConfiguration();~~
	}

	@Override
	public void setFocus() {
		mViewer.setFocus();
	}

	@Override
	public void dispose() {
		me = null;

		super.dispose();
	}

	private CompareConfiguration createConfiguration() {
		CompareConfiguration configuration = new
CompareConfiguration();
		configuration.setDefaultLabelProvider(new
AzuriteCompareLabelProvider());
		return configuration;
	}

	public void addReviewViewer() {
		int historySize =
RuntimeHistoryManager.getInstance().getEntireHistory().size
();
		mViewer.selectVersion(historySize - 1,
historySize);
```

Figure 3-3. The event list interface of FLUORITE analyzer.

3.4.2. *Code Editing Pattern Detection*

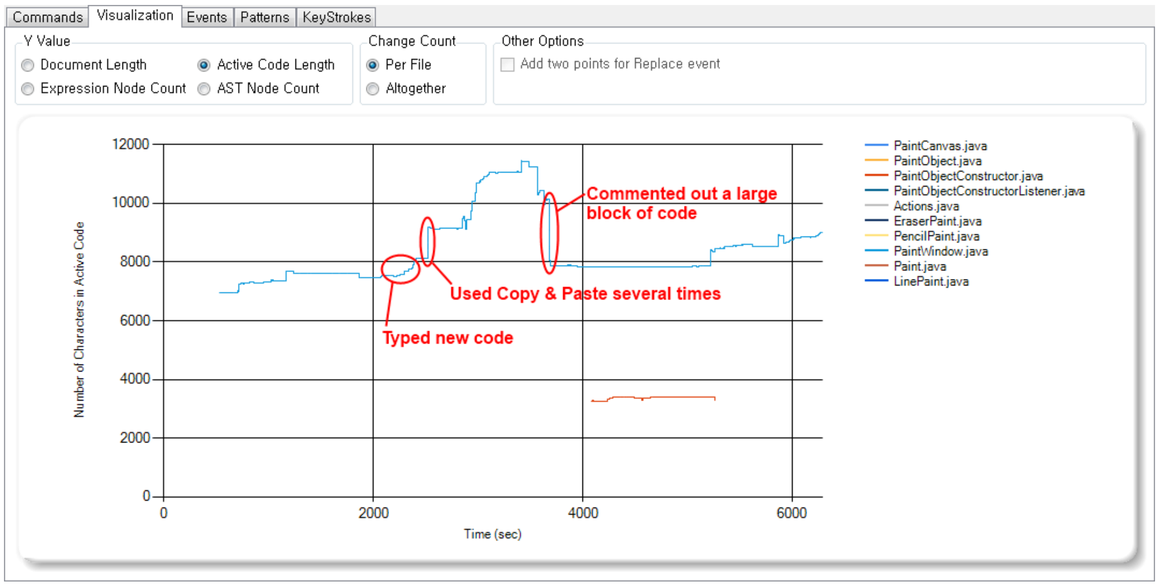
It is possible to detect various code editing patterns which are composed of sequences of commands. As an example, our analyzer can detect *fixing typo patterns* from the logs. Some fixing typo patterns can be detected by looking at three consecutive document change events as follows: 1) Any `Insert` 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 3-4 shows a few sample fixing-typo patterns detected by this algorithm. Some of the detected patterns are not merely typo corrections. For example, for the pattern starting from ID 1061 in Figure 3-4, we can see that the programmer decided to declare an array instead of declaring multiple variables. Double-clicking one of the detected patterns shows the corresponding event in the event list (Section 3.4.1), in order to make it easier to investigate the code editing pattern with the surrounding context.

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. More sophisticated code editing pattern detection could also be implemented. For example, an abstract syntax tree (AST) based automatic pattern analyzer was implemented to detect backtracking instances within collected FLUORITE logs (Section 4.3).


Commands
Visualization
Events
Patterns
KeyStrokes

ID	Le...	Additional Information
39	3	"think" – "nk" + "k"
53	3	"k" – "k" + "ckness"
127	3	"hank" – "ank" + "ickness"
269	3	"T" – "T" + "thickness"
319	3	",sethi" – ",sethi" + ",set"
649	3	" colorPanel,ad" – " colorPanel,ad" + " colorPanel,ad"
867	3	"colosr" – "sr" + "rPanel,add"
1061	3	"Button t1Button, t2Button, t3" – "1Button, t2Button, t3" + "Button[5];"
1202	3	"changeEvent," – "eEvent," + "Event"
1267	3	" ₩ " – "₩" + "tButtson"
1278	3	"tButtson" – "son" + "ons"
1311	3	" = new JButton)" – ")" + "()[];"
1371	3	" fore" – "fore" + "foreach()"
1468	3	"tButtons[i] = tButtons,le" – "tButtons,le" + "new JA"

Figure 3-4. Examples of detected typo fixing 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 patterns starts so the investigator can jump to the events list and see what was happening around that time.



Y Value: Document Length; [✓ Active Code Length]; Expression Node Count; AST Node Count

Change Count: [✓ Per File]; Altogether

Other Options: Add two points for Replace event

Figure 3-5. Example active code length graph drawn from one of the logs by the FLUORITE analyzer. Some interesting points are marked using red circles and the corresponding code editing strategies are described. Y-axis value can be one of the metrics described in **Table 3-3**. Only line graphs of the files that have been changed during the session are drawn. The graph can be zoomed with the mouse wheel, and the user can double click on a point to jump to the events-list view.

3.4.3. 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 Table 3-3. From the collected logs, it was noticeable that the *code length* graph and the *active code length* graphs differ significantly, which indicates that programmers often comment out or uncomment code. The graphs also show some interesting editing trends. In **Error! Reference source not found.**, the steadily increasing part indicates that the programmer was typing new code, small fluctuations mean the programmer 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 more thorough investigation is needed to see what was happening, the point can be double-clicked to jump to the event list. The event whose timestamp is closest to the selected point on the graph is highlighted to facilitate manual investigation.

3.4.4. Keystroke & Command Distribution Report

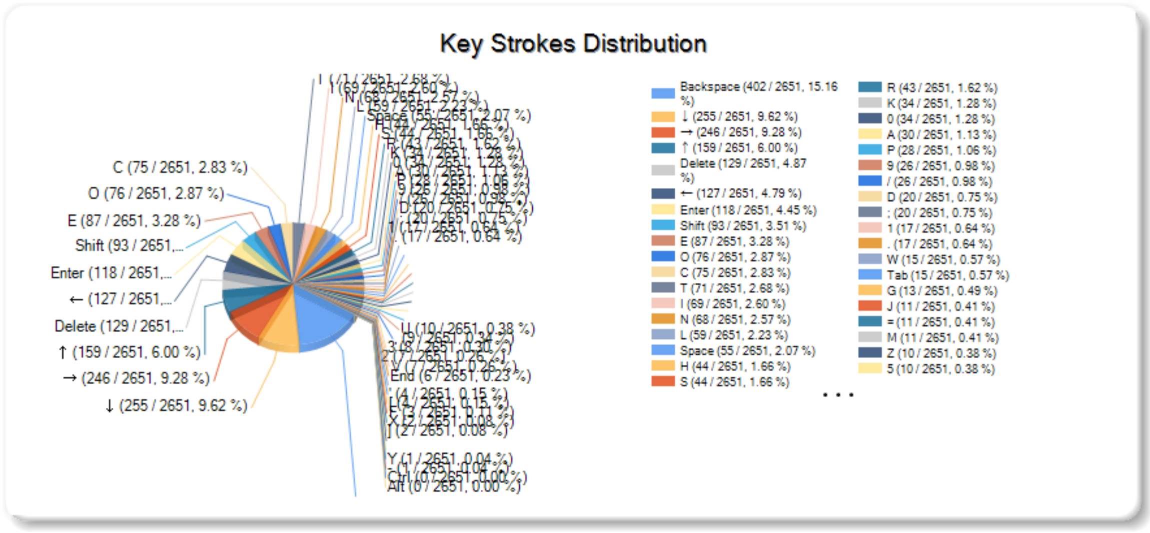
The keystroke distribution report gathers all the keystroke data from the logs and draws a pie chart showing the frequency of various types of keystrokes. The command distribution report is similar, but differs from the keystrokes in that it focuses on Eclipse commands rather than just keystrokes. It is similar to the commands report of the Eclipse Usage Data Collector (UDC), but FLUORITE also includes the commands missing from UDC.

Here, these two features are demonstrated using a particular set of FLUORITE log data collected during an exploratory lab study (see Section 4.1). In the lab study, 12 student participants performed some small editing tasks for about 2 hours each. These tasks used the Paint program from [Ko 2005a][Fogarty 2005], and had users add some new features. Figure 3-6 shows example screenshots of a keystroke distribution and a command distribution report generated from a single participant's log file from the study.

In the collected data from all the 12 participants, 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 programmer holds down a key and it auto-repeats, it is still interesting to see that programmers navigate a lot within a file using arrow keys. This result is consistent with Ko *et al.*'s observation [Ko 2005a] that developers spend about 16% of their time navigating dependencies.

Another interesting observation is that programmers 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, MacKenzie and Soukoreff's report that 7.10% of keystrokes were backspaces [MacKenzie 2002]. This provides further evidence, as mentioned in [Ko 2005b], that editing code is different than editing documents.

Consistent with the keystroke report, the five most frequent commands used by the 12 participants 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. More detailed results and their implications regarding backtracking will be discussed in Section 4.1.4.



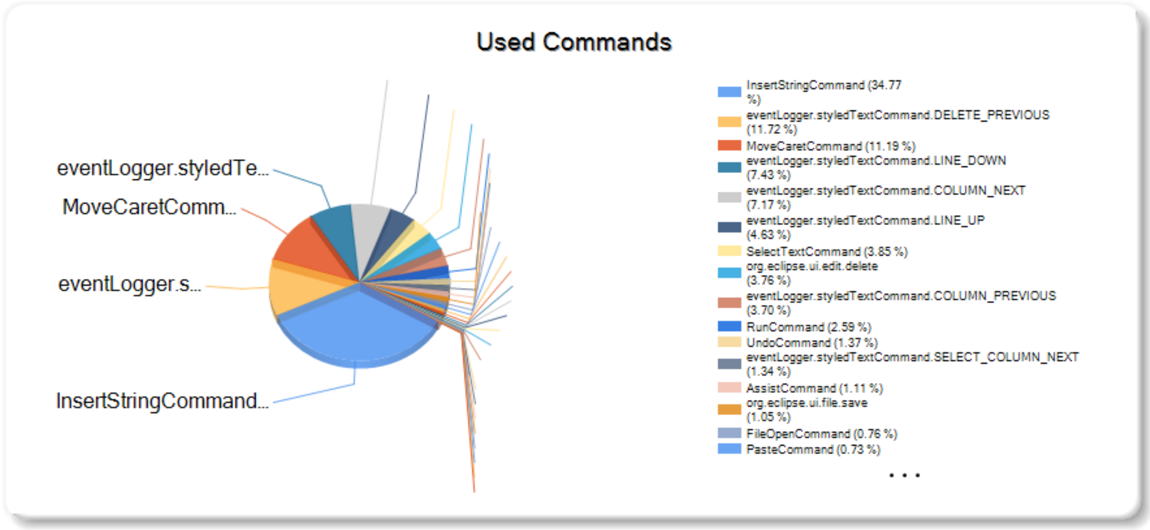


Figure 3-6. Example keystroke / command distribution reports generated by FLUORITE analyzer showing the distributions for one participant. The reports are also provided in comma-separated values (CSV) format, which can easily be imported into spreadsheets for more analyses.

3.5. DISCUSSION

3.5.1. DETECTING CODE CHANGES MADE OUTSIDE OF THE IDE

Since the system is dealing with individual incremental changes instead of full snapshots, a single missing item in the edit history can confuse the entire history. However, source code can be changed even outside of the IDE for many reasons. For example, the code can be modified by the external version control system while the IDE is not running, to revert to an earlier version, or updated to reflect the changes made by another team member. Sometimes, users might edit the code with a plaintext editor instead of using the IDE. In addition, if a file is closed without saving, then the last known snapshot of the file kept in FLUORITE would be out of sync when the file is reopened later.

To avoid this problem, FLUORITE detects such situations by keeping the initial snapshot and the last known snapshot of each file that was open in the current session. When a file is re-opened, FLUORITE compares the new snapshot with the last known snapshot, and, if they are different, extracts diffs between those two snapshots to fill in the missing changes. This process is done using the Google-diff-match-patch open-source library [Fraser 2012][Myers 1986].

3.5.2. CODING EVENTS NOT CAPTURED BY FLUORITE

There are several issues with FLUORITE which were only discovered after conducting several studies with it. FLUORITE only captures the document changes generated from the currently active source file (i.e., the file currently open in the active editor). Whenever a new source file is open, a **`FileOpen`** command is logged with the initial snapshot, and all the following document change events belong to the last open source file. This approach works well for most of the coding situations, but this approach has a few problems. First, when the programmer makes some code changes across multiple files with a single command (e.g., refactoring commands), only the changes made in the active file are logged, missing the changes made in the other files. These missed changes may be captured by FLUORITE later, only when the corresponding source file is open in the editor and becomes the active file, in which case the timestamp of the document change would be incorrectly logged as being when the file was opened. Moreover, when there are files that are changed by some command but never are opened, the document changes in that file would not be captured at all.

Another related limitation is that FLUORITE only captures the files that were open at least once during the editing session. While the rationale behind this decision was to keep the logging tool non-intrusive and the log files as small as possible, this makes it difficult to understand the bigger picture of the entire project when analyzing the log files in isolation. Coding events can also happen outside the IDE, which are not captured by FLUORITE. A common example situation is when programmers are invoking version control system commands through command-line or third-party clients, instead of using the IDE plug-ins.

3.5.3. WRITING LOG ENTRIES AS THE EVENTS ARE CAPTURED

The old versions of FLUORITE used to keep the coding events in memory during the editing session and write the log file when the IDE is being closed. However, this behavior has been changed to writing the log entries as the events are captured for two reasons. First, by flushing the events from the memory to the disk frequently, FLUORITE's memory usage is no longer increased proportionally to the number of events captured so far. Second, even when the IDE crashes in the middle of the editing session, the log file is still kept on the disk without any loss of data. This change did not result in any noticeable extra time or delay in using Eclipse with FLUORITE running.

3.5.4. Evaluation

FLUORITE and the analyzer tool have been publicly released (see Chapter 12 for the URL) since Fall 2011. Since then, FLUORITE has been used by many researchers from CMU and other institutions for different purposes. For example, it has been used for enhancing code search mechanisms [Martie 2013], monitoring and analyzing the students' coding behaviors [Fuchs 2014], and visualizing participants' actions performed during some user studies [Kwan 2013]. Some people used FLUORITE in addition to their primary data collection method during their studies, in order to make sure that they do not miss any important coding events. For example, Dörner et al. used FLUORITE for the evaluation study of Euklas [Dörner 2014], and three other researchers contacted me and told that they were using FLUORITE for their own studies.[19]

Since the FLUORITE log files are written in XML format, other researchers have been able to implement their own analyzers with ease. In addition, a group of researchers at Oregon State University[20] developed a FLUORITE log replayer, which they used for converting FLUORITE logs into another data format that they could analyze more easily, by replaying the FLUORITE logs while the other capturing tool is running.

Personally, I have been running FLUORITE in my Eclipse environment for developing FLUORITE and AZURITE, as a "dogfooding" practice [Harrison 2006] for about 4 years, as of the writing of this dissertation. It has been running without any noticeable problems and has not interfered with my own work. My own FLUORITE log data collected in this way were analyzed together with the logs collected from the study participants in a longitudinal study (Section 4.3). Details about the average size of the FLUORITE logs can be found in Section 9.3.1.

3.6. Conclusion

The FLUORITE logger for Eclipse and the analyzer were developed in the hopes that they will be useful to the community for when detailed analyses of programmers' edits are required. During our empirical studies of backtracking, FLUORITE reduced significant amount of manual analysis, and helped uncover interesting results (Chapter 4). Later, FLUORITE was also used as the input source of our selective undo tool AZURITE, discussed later.

[19] When FLUORITE was used as an extra data collection mechanism, it was not always cited.

[20] Irwin Kwan, David Piorkowski

4.

EMPIRICAL STUDIES OF BACKTRACKING[21]

As a first step towards supporting more robust backtracking in modern IDEs, I wanted to first know more about when and how programmers backtrack when they write source code. However, there has been no thorough study about backtracking in the software development context. This chapter describes three empirical studies of programmers' backtracking conducted in order to understand backtracking better. First, an exploratory lab study was conducted to gather baseline knowledge about backtracking (Section 4.1), and then a follow-up online survey was performed to get a better idea of the backtracking frequency and tactics (Section 4.2). Realizing that these two studies have some limitations, I also conducted an extensive, longitudinal study to complement the previous two studies and see if these backtracking situations arise when programmers are working on their own programming projects (Section 4.3), not just the artificial programming tasks given in a lab. These three studies showed that backtracking happens quite frequently and often there are difficulties when programmers are backtracking, suggesting that programmers would benefit from better backtracking tools.

4.1. PRELIMINARY LAB STUDY OF BACKTRACKING

First, an exploratory lab study was conducted to study when and how programmers backtrack using today's tools, and to identify barriers that they face while backtracking. The focus of this study was to answer the following research questions.

RQ1-1. How do programmers backtrack?
RQ1-2. How do programmers know where to backtrack to?
RQ1-3. What are the barriers to successful backtracking that a new tool might alleviate?

4.1.1. STUDY DESIGN

This study was a 2 hours long exploratory lab study where participants were asked to finish two pairs of feature-adding tasks and think aloud during the study. The editing screens and their voice were recorded for further analyses. In addition, FLUORITE (Chapter 3) was used to capture all the low-level editing events. Participants used the Eclipse IDE version 3.6.2 (He-

[21] Portions of this chapter appeared in [Yoon 2012] and [Yoon 2014]

lios) on a laptop PC 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 Java API Documentation.

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

For this study, 12 graduate students were recruited from the School of Computer Science at Carnegie Mellon University. The participants were required to (1) have professional development experience or at least two internships as a software programmer, 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.

4.1.2. The Paint Program

As the code base of the study, a *Paint* program (Figure 4-1) was used which has been previously used by other researchers [Fogarty 2005][Ko 2005a]. This is a simple Java Swing based painting application composed of 10 Java files and a total of 452 lines of code.

Using the *Paint* program as the code base had several advantages. First, graphical user interface (GUI) development tends to be exploratory (i.e. involves extensive experiments with code), which means that the programmers would often need to backtrack during the 2 hour period. 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.

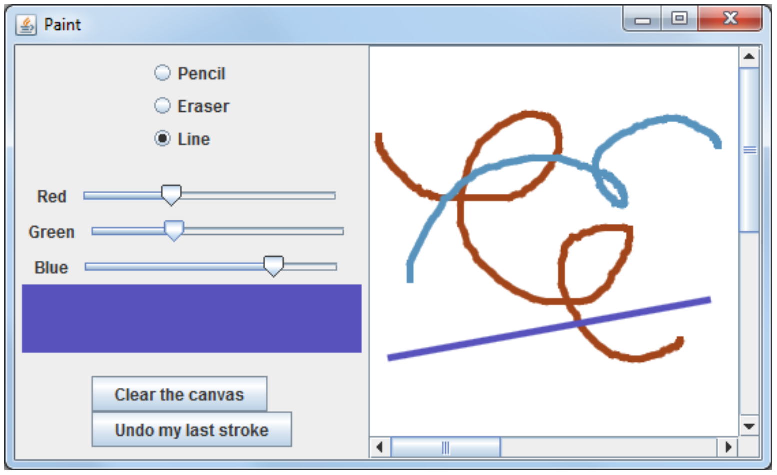


Figure 4-1. A screenshot of the Paint program used during the lab study.

4.1.3. TASKS

The participants were asked to add new features to the *Paint* program. In order to get as much backtracking data as possible in 2-hour lab study, the tasks were designed so that they would lead the programmers to backtrack regardless of any occurrences of their own exploration. To achieve this goal, an imaginary scenario was set up where a whimsical boss first asks the participants to implement a feature, changes her mind after testing the feature and asks them to implement the same functionality using a different user interface element. Because it did 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 from scratch 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 sets of 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).

Another issue investigate was whether the programmers would behave differently if they knew they might need to backtrack later. Therefore, the participants were first asked to implement one of the features F_A-1, without knowing that they might have to backtrack later. Then, they were asked to implement F_A-2 instead. Next, they were asked to go back to F_A-1 implementation, in order to see how they would restore the previous version. Finally, they were given *both* F_B-1 and F_B-2 simultaneously and asked to implement one at a time, using any tactic they wanted, to see if they behave differently when they knew in advance that they would need to backtrack. Whether participants used Feature 1 as F_A and Feature 2 as F_B (Group 1) or vice versa (Group 2) was randomized. The study procedure and group settings are shown in Table 4-1. All the task sheets provided to the participants can be found in Appendix A.

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

Table 4-1. Participant groups and the tasks of the preliminary lab study

As mentioned above, all the code edits performed by the participants were logged using FLUORITE (Chapter 3). Using this data, several code editing patterns composed of sequences of commands could be detected, which are closely related to backtracking. Having this data has many advantages. Not only does it reduce the time to inspect the videotapes significantly [Kim 2004], it also enables various automatic analyses.

4.1.4. RESULTS

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

The 4 different features were meant to 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.[22] The reason for F1-1 being the easiest might be 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, their data were not excluded because the participants still backtracked to some extent while trying to figure out how to get the tasks completed. The following sections summarize the key observations related to each research question.

4.1.5. RQ1-1: HOW DO PROGRAMMERS BACKTRACK?

4.1.5.1. COMMAND STATISTICS & KEYSTROKE DISTRIBUTION

In order to investigate how frequently programmers used backtracking related editor commands, I first analyzed the FLUORITE log data to obtain the frequency of each IDE command execution and each keyboard key press. Table 4-2 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 such as delete and undo, indicated as inverted. 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, it is shown that backtracking related commands are very frequently executed. The command statistics are somewhat different from those observed by Murphy et al. [Murphy 2006] because the two logging tools differ in what types of commands are logged. However, the rank orderings of commands are consistent if only the main editor commands such as Delete, Save, Copy, Paste, and Assist are compared.

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

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

Table 4-2. Commands and keystroke distributions. The top twenty entries are listed for each category. Shaded entries are related to code navigation, and the inverted entries are related to backtracking.

Table 4-2 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.

4.1.5.2. *Deleting vs. Commenting Out*

There were also some interesting backtracking related behaviors observed during the study. 7 of the 12 participants habitually commented out their code rather than deleting it, whether or not they thought the 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`, and the .NET Framework provides the `Conditional` attribute which allows programmers to conditionally activate a certain method according to the current build configuration. Although, Java also supports conditional compilation, participants were not aware of this feature and they could only use conventional comments.

4.1.5.3. 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 participants commented out code because they knew that the code being commented out might be used again. This includes the situation where the code was one of the variations and the programmer wanted to be able to switch to another variation. Also, when the programmer 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 programmer wants to keep the code anyway. This could happen when the programmer 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 programmers 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 would be needed later on.

Finally, programmers 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 later.

4.1.5.4. WHEN THEY KNOW THEY NEED TO BACKTRACK LATER

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 they might be asked to go back to F_A-2 again.

Only 5 out of the 9 participants who started task 4[23] 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 the participants were asked 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,

[23] The remaining three participants gave up before getting to task 4.

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.

4.1.6. RQ1-2: HOW DO PROGRAMMERS KNOW WHERE TO BACKTRACK TO?

The participants often remembered one or more aspects of the deleted code, especially when they wanted to restore a specific code fragment that was recently deleted. What they remembered included the original location from where the code was deleted, how the surrounding code looked, the names of one or more code elements in the deleted code, or what the desired code looked like. This suggests 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.

4.1.7. RQ1-3: WHAT ARE THE BARRIERS TO SUCCESSFUL BACKTRACKING?

The study participants faced various problems when they were trying to backtrack. First, the participants 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 the pasted code. 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 manually labeled the related code fragments using comments. It would be even more difficult for the programmers 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 participants who used console output did not.

Several problems were identified with restoring code during the 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 the 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 said he could not remember what the deleted code looked like, and he failed to restore the code even after he correctly found where it went. He had tried to restore the deleted code[24] 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 commands 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, and that in Eclipse, the copy command does not affect the undo stack, so redo was still available after undoing and performing copy.

In addition, 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.[25] 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.

4.1.8. Limitations

This study, however, had some limitations. In order to maximize the chance of observing programmers' backtracking behavior in a short (2 hours) lab session, the tasks were artificially designed so that they require the participants to backtrack. As a consequence, the study results does not really tell us how frequently those problems would occur in real development situations. In addition, because the code base used for this study was a particular type of program, a GUI application, it is left unknown whether the findings from this study would generalize to other types of programming tasks.

[24] 6 lines of code, excluding the blank lines and the lines only containing "}".

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

4.2. Online Survey

In order to get more feedback from general programmers besides just graduate students at Carnegie Mellon, an online survey was conducted as a follow-up, which took about 15~20 minutes to complete. The key research questions for this online survey were the followings:

RQ2-1. What is the perceived frequency of backtracking?
RQ2-2. What are the common backtracking tactics used by programmers?
RQ2-3. What tools or features could help programmers backtrack better?

The survey was posted on several online programmer forums including reddit.com,[26] dzone.com,[27] and others. A total of 103 programmers 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. The questionnaire used for this study can be found in Appendix B.

4.2.1. Demographics / Traits of Their Work

The survey was composed of three parts. First, the respondents were asked demographic information including their gender, age, and prior experience in software development. The demographics of the respondents are summarized in Table 4-3. 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.

Category	**Value**	**Count**	**Percentage**
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
Total respondents		48	

Table 4-3. Demographics of the online survey respondents.

[26] http://www.reddit.com/r/programming
[27] http://www.dzone.com/

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 programmers' work was for different activities, and the results are summarized in Figure 4-2. 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.

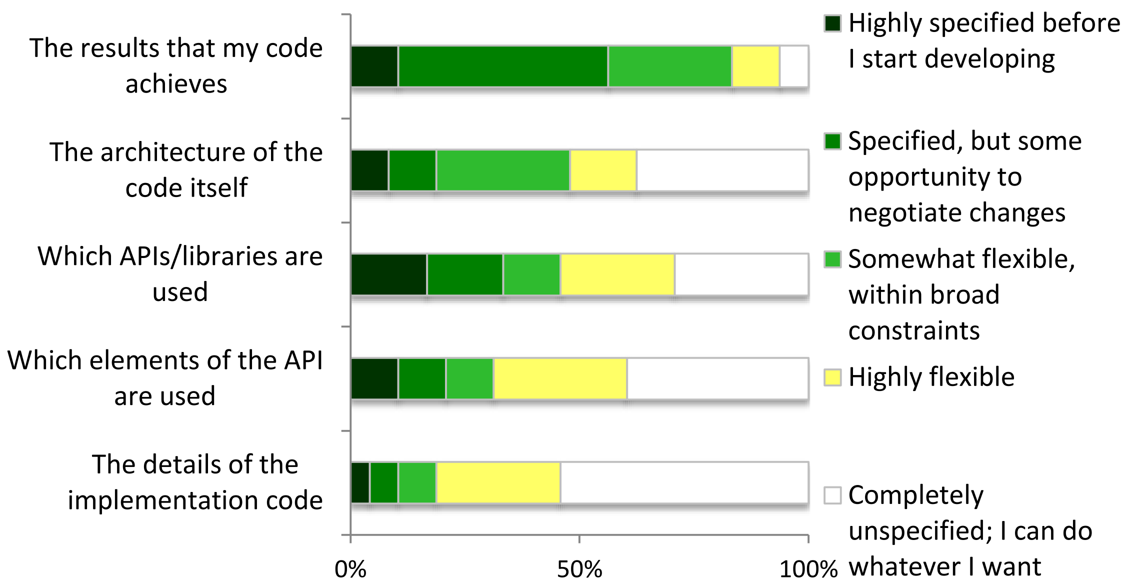


Figure 4-2. The responses for the question "For each of the following, please specify how often you need to experiment, iterate, and/or explore *while you are developing*." The lighter color represents more flexibility.

I speculated that there might be more flexibility if the programmer works alone or as part of relatively small groups. So, I investigated if there is any correlation between the sizes of the groups in which the programmers worked, and the flexibility of their work, but could not find any statistically significant correlation. Even when the programmer worked alone, often the work was assigned by the boss or the customers and we did not find that the programmer had much freedom. Only one of the respondents who always worked alone expressed that everything is completely unspecified and he could do whatever he wants.

4.2.2. RQ2-1: WHAT IS THE PERCEIVED FREQUENCY OF BACKTRACKING?

In the second part of the survey, seven different situations were presented where the programmers might need to backtrack. For each situation, the respondents were first asked how often they faced the given situation. Figure 4-3 shows the responses for these questions. We can see that programmers face these backtracking situations quite often. Roughly $^3/_4$ of the programmers face these situations at least "sometimes."

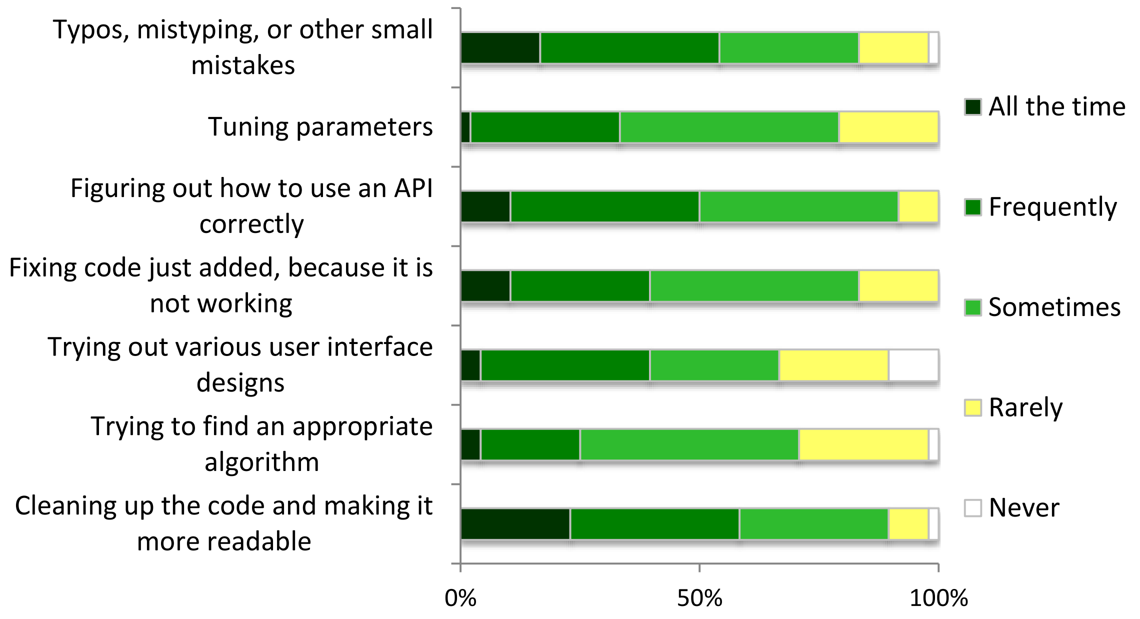


Figure 4-3. The backtracking situations shown to the survey respondents, and their answers.

The respondents were also asked what other backtracking situations they faced. One person mentioned backtracking while writing a new interface file from scratch. Although the exact reason for backtracking was not explicitly mentioned, it may be because it is often not clear what would be the correct interface to be exposed. Two responses were about reorganizing and simplifying code structure. Another two responses said that they mainly backtrack because they find new corner cases or missed input values during testing.

4.2.3. RQ2-2: WHAT ARE THE COMMON BACKTRACKING TACTICS USED BY PROGRAMMERS

Next, they were asked what types of tactics they use to backtrack. Eight different potential tactics related to backtracking were shown, and they were asked how often they think they use each tactic to solve the given situation using a 5-point Likert scale ranging from “Never” to “Pretty much every time”. The result showed that only a few tactics are primarily used for each situation. When fixing typos and small mistakes, the most frequently used tactics 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 said 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.

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

There were two other tactics mentioned by the respondents which were not observed in the lab study. Two 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. Five people mentioned writing unit tests using mock objects to see how the API works. It is possible that our lab study participants did not use these tactics because the code base was fairly small and they had time limitations.

RQ2-3: What Tools or Features Could Help Programmers Backtrack Better?Finally, they were asked to provide ideas on what types of new features or commands for an IDE could help with experimenting and backtracking. Two people wanted a tool where programmers can type in small code snippets and run them, just as they can with scripting languages. Two people wanted an IDE feature that allows programmers to take snapshots across multiple files at any point, and switch among those snapshots, as a light-weight version control system (provided as the tagging feature in AZURITE, Section 6.2). Two other people wanted an even lighter version control system that can keep multiple versions of each method or class and allow users to select one of those versions easily (provided as the code history diff view in AZURITE, Section 8.1). One person wanted an undo tree model instead of the conventional linear undo model.

4.2.4. LIMITATIONS

Although the participants responded how frequently they face various backtracking situations in the online survey, it is possible that the programmers backtrack unconsciously and the survey results may not correctly reflect what actually happens in a real development. In addition, the survey cannot provide details about how and under what circumstances programmers backtrack.

4.3. LONGITUDINAL STUDY OF PROGRAMMERS' BACKTRACKING

This section describes an extensive, longitudinal study conducted to further investigate programmers' backtracking behavior, as a follow-up to the previous two studies. The goals of this new study were twofold. First, this study was aimed at obtaining backtracking statistics in order to quantify the need for backtracking tools. For this study, the focus was on collecting *quantitative* data, as our previous studies were mostly qualitative. The second goal of this study was to identify backtracking situations that are not very well supported by existing programming tools, and to extract useful design implications for developing better backtracking tools that might improve programmer productivity.

The analysis was performed with the following research questions in mind:

RQ3-1. How frequently do programmers backtrack in a real programming environment? (Section 4.3.2.1)
RQ3-2. How large are the backtrackings? (Section 4.3.2.2)
RQ3-3. How exactly do programmers perform backtracking?
For example, do programmers backtracking manually? (Section 4.3.2.3)
RQ3-4. Is there evidence of "exploratory programming"? (Section 4.3.2.4)
RQ3-5. Are there backtrackings happening across multiple editing sessions? (Section 4.3.2.5)
RQ3-6. Are there *selective* backtrackings, which cannot be performed using the undo command? (Section 4.3.2.6)
RQ3-7. Do programmers backtrack to the same code repeatedly? (Section 4.3.2.7)

The key idea of this study was to observe programmers in their normal development environment, rather than in a lab, over a long period of time by using the FLUORITE logger. The following subsection presents the analysis method that was devised to answer these questions (Section 4.3.1). Using the fine-grained code edit logs collected from 21 people, totaling 1,460 total hours of programming time, the analyzer could track the entire histories of all abstract syntax tree (AST) nodes in their source code, and use the per-node history data to detect backtracking instances.

The next subsection presents the backtracking related information found from the analysis, corresponding to the research questions listed above (Section 4.3.2). The results show that programmers were backtracking 10.3 times per hour on average, and the backtracking size varied from a single character to more than a thousand characters, which spans from backtracking out of simple parameter value changes to significant algorithmic changes. Programmers were backtracking manually by deleting or typing code in 34% of all backtracking cases. In 20% of the backtracking cases, programmers first changed some code, ran the application, and then backtracked the changes, suggesting that they were experimenting with their code. About 97% of all backtrackings were done within the same editing session. 9.5% of all backtracking instances were selective, which means that the conventional undo command could not handle them, because they were intermingled with other changes that the programmers did not want to lose. Finally, only 15% of the backtracked nodes were reverted to the same state again later.

4.3.1. ANALYSIS METHOD

4.3.1.1. LOG DATA

The data set we used for this study was collected using FLUORITE (Chapter 3). A total of 21 participants were recruited for the study (Table 4-4). Since the goal was to investigate how programmers backtrack for their own programming projects, the participants were asked to install FLUORITE on their own Eclipse IDE and perform their own programming tasks as usual.

Group	#	Description	Coding Time (hours) (min / avg / max / **total**)[a]
G0	1	Myself	294 / 294 / 294 / **294**
G1	13	Graduate students at CMU	3 / 40 / 216 / **520**
G2	5	Research programmers / systems scientists at CMU	6 / 118 / 446 / **588**
G3	2	Graduate students at the University of Pittsburgh	6 / 29 / 51 / **57**

a. min / avg / max: per-user values
total: the sum of all the participants' coding times in each group

Table 4-4. Participant groups of the longitudinal backtracking study

They were periodically asked to archive and send their logs to us, and were paid up to $50 for their participation. I also collected my own logs, which were analyzed together with the data collected from the participants. The data was collected from April 2012 to January 2014, and contains 1,460 hours of coding activities, excluding all the idle time exceeding 5 minutes. All of the participants were programming in Java using Eclipse (v3.6 or higher) across a variety of Windows and Macintosh machines.

Among the participants, about 8 out of 13 in G1 were Masters students who were working on their Studio projects with real clients, and the participants in G2 are professional programmers whose primary job is programming but in an academic context.

4.3.1.2. AST-NODE BASE CHANGE HISTORY TRACKING

The collected data set contained 1,345,241 coding events in the logs, which made it almost impossible to manually inspect the logs. Therefore, an automated analysis approach was used for this study. By the definition of backtracking (Chapter 1), a trivial backtracking detector would detect any pairs of edit operations <op1, op2> where the later performed operation op2 reverts what op1 did earlier, at least partially. For example, if an earlier edit operation inserts "foo" in the code (op1) and that "foo" is deleted anytime later by another operation (op2), this pair of operations could be considered as a backtracking instance. The initial data analysis attempt followed this approach. However, this naïve approach was unsuccessful due to the following major limitations:

- Too many false positives were found, caused by auto-formatting, organizing import statements, auto-generated comments, etc.
- It could not detect multi-step backtracking, where a code fragment is reverted to an earlier state by multiple edit operations.
- It could not tell syntactically equivalent code fragments very well, as it was purely text-based. For example, if a method call foo(); is deleted and then later put back as foo␣(); with an additional space before the parentheses, it could not know the two statements are actually identical and failed to detect such backtracking.
- It treated comments and source code the same way.

- It was difficult to tell the high-level intent of the detected backtracking automatically, and required substantial manual inspection to get useful information about the detected backtracking instances.

In order to address these issues, a more sophisticated analysis approach was devised which utilized the abstract syntax tree (AST) of the source code. The basic idea of this approach is to keep the evolution history of individual AST nodes of interest throughout the lifetime of the nodes.

The automated analyzer[28] processes the logged edit events sequentially *off-line*, separate from Eclipse. When a new file open event is seen from the log, the analyzer keeps the snapshot of that source file and parses the snapshot (using ASTParser from the Eclipse JDT) to store all the AST nodes of interest, for example all the *statement* nodes in the source file. For each AST node, the analyzer remembers the start position and length of the node within the file, and the initial snapshot of that node. When keeping the snapshot, the analyzer normalizes all the formatting such as whitespaces and indentations.

Whenever an edit operation is seen, the analyzer applies that operation on the last known snapshot of the file in order to get an updated snapshot, and parses the updated version. Then the analyzer determines all the AST node(s) that were affected by this change and updates the start position and length of the node. In addition, the analyzer adds the new version to the evolution history of that specific node, but only when the normalized snapshot differs from the last known snapshot.

Because the log data contains character-level edit operations rather than AST-level differences, a source file can be in an incomplete state (i.e., containing parse errors) after applying an edit operation. In such cases, the analyzer tries to update only the start position and length values of all the known nodes according to the edit offset and length. When the file returns to a parseable state after some subsequent edits, the analyzer again tries to match all the previously known nodes and the current nodes to find all the affected nodes. By doing this, the analyzer can keep track of the per-node evolution history over time.

When an AST node gets removed from the AST tree (i.e., the corresponding code fragment is deleted from the source code), the node can no longer have more evolution history in the future. At that point, the analyzer checks whether the evolution history of that node contains any backtracking instances. Let the evolution history $h \coloneqq v_1 v_2 \ldots v_n$, where v_i is the i-th snapshot of the node. A backtracking instance b is then defined as a sublist $v_f \ldots v_l$ of h where:

$$f + 1 < l \wedge v_f = v_l \wedge \forall i \in (f, l) \cdot v_f \neq v_i$$

In summary, a backtracking instance is a sub-history of a node where the **f**irst and the **l**ast snapshots are the same, and all the **i**ntermediate snapshots are different from the first

[28] The automated backtracking analyzer described in this section is different from the FLUORITE Analyzer tool described in Section 3.4.

snapshot. This indicates that this node digressed from the first version and then backtracked to the first version. Figure 4-4 shows an example node history which contains three backtracking instances. Note that $v_1 \dots v_6$ does not count as backtracking, because there is an intermediate version v_4 which is the same as v_1 and v_6. Throughout the paper, v_f will be referred to as the *first version*, v_l as the *last version*, and any v_i $(f < i < l)$ as an *intermediate version* of a backtracking instance. When there is no need to distinguish the first and last version (because they have the same code by definition), the term *original version* will be used to refer to either version.

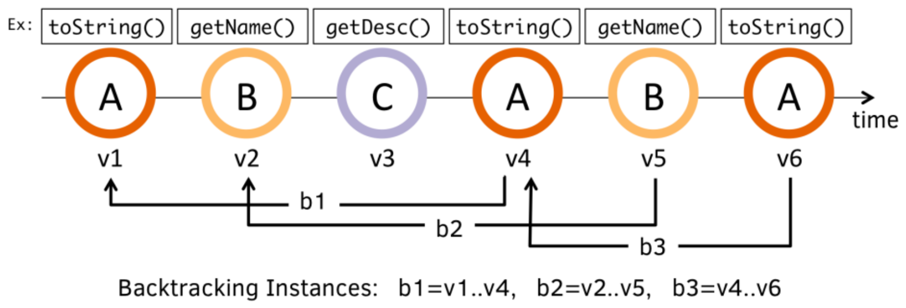


Figure 4-4. An example of a node evolution history, which contains three backtracking instances. The node first appeared in the code as "`toString();`" (v1), changed a few times (v2 through v5), and finally ended up back at the original code (v6). The different contents are symbolized as capital letters A, B, and C. There are three back-tracking instances in this node history, indicated as black backward arrows.

```
    Backtracking instance: [184263, 184629]
v1  [184263]return new Point(getWidth(),getHeight());
v2  [184555]return new Point(getWidth() - MARKER_SIZE,getHeight());
v3  [184567]return new Point(getWidth() - MARKER_SIZE,getHeight() - MARKER_SIZE);
v4  [184623]return new Point(getWidth()~~ - MARKER_SIZE~~,getHeight() - MARKER_SIZE);
v5  [184629]return new Point(getWidth(),getHeight()~~ - MARKER_SIZE~~);
```

Figure 4-5. An example output of the analyzer, showing the history of a statement node. Each row maps to each version (v1, v2, ..., v5). This node contains a single backtracking instance, which is v1...v5. Note that the version numbers (v1-v5) are not part of the output, and added here for the purpose of explanation.

Figure 4-5 shows an example history of a statement node that contains a backtracking instance. On the left side are the IDs of the edit operations unique within a single programmer's entire log (the numbers in the square brackets). Each line shows the snapshot of the node at a given time, and it shows the evolution history of this specific node from top to bottom. The green shaded code indicates newly inserted code, and the pink shaded strikethrough code indicates deleted code compared to the previous version. In this example, there was one backtracking instance [184263, 184629], which means that the normalized snapshot of this node at ID 184263 – the first version – was identical to the one at 184629 – the last version.

The analysis was performed on three different levels of granularity of AST nodes: *statement* level, *block* level, and *type definition* level. A type definition is any of a whole **`class`**, an

`interface`, or an **`enum`**, which is usually a whole Java file with the exceptions of nested type definitions. The log analysis was performed at these levels for two reasons. First, this was to know the granularity of backtracking instances. Second, this is needed to detect certain backtracking instances. For example, when a statement (s1) is deleted from the code and then the identical statement (s2) is put back at the same position in the future, the statement level analyzer would consider s1 and s2 as separate nodes, and thus not detect this as backtracking. On the other hand, if we run the analysis at the block level as well, both the deletion of s1 and insertion of s2 would be in the evolution history of the surrounding block, thus the block-level detector would successfully detect this as a backtracking instance. In addition, block level analysis can detect the changes spanning across multiple statements. Similarly, when a code block or an entire method is removed and restored later, or when multiple code blocks are changed together and backtracked, the type definition level detector would catch those instances.

Comments in the source code were excluded from the analysis, as we were mainly interested in actual code changes and exploratory programming. However, when the programmer comments out some code and uncomments it (or vice versa), the analyzer still detects this as a backtracking, because it is seen by the analyzer as if the commented out code disappeared and was put back in.

4.3.1.3. Data Preparation & Removing Duplicated Results

The log data needed to be cleaned up before performing the analysis, in order to get more meaningful results. First, all of the minor typo correction instances were removed from the logs using the typo correction detector described in Section 3.4.2. Even though typo corrections are backtracking instances by definition, these are not very interesting in that it is hard to imagine a useful programming tool that helps fixing typos any better than current mechanisms. There were 40,229 code edit events removed as part of typo corrections, out of a total of 343,685 edits (11.7%).

The second clean-up process applied was removing the noise related to the "Rename Refactoring" command of Eclipse. When the rename refactoring is invoked, all the occurrences of the element being renamed (e.g., a variable name) are highlighted directly in the code editor, and they all change together as the user types. When the user confirms the renaming by hitting the enter key, however, Eclipse automatically reverts all the character-level changes made during the renaming process, and turns them into word-level changes so that users can undo the renaming with a single command. Since all the intermediate character-level changes are also logged in the FLUORITE logs, they resulted in many false positives in the analysis results. Thus, all of these instances were cleaned up before the real analysis.

In addition, the analyzer was designed in a way that it takes extra care to remove possible duplicate instances from the analysis results. When running the analysis on different levels of granularities (statement < block < type definition), the same backtracking instance can appear in multiple levels. For instance, when a statement is changed and immediately backtracked, this instance will appear in all three levels. When counting the backtracking instances at a coarse level of granularity, the analyzer excluded all the instances that were already detected at a finer grained level. There can also be duplications within the same level of granularity, because a code block can contain another block, and a type definition can contain a nested type definition inside. In such cases, the analyzer only counted the backtracking instances at the innermost code element.

Finally, because the programming language used was Java, there can also be duplications when a statement contains one or more type definitions or code blocks. In Java, anonymous class instances can be defined and assigned to a variable or passed as a parameter of a method *inline*. The analyzer excluded these types of duplications as well by not counting such statements.

4.3.2. Results

This section presents the analysis results, which are summarized in Table 4-5. The backtracking instances were investigated with several specific questions in mind, each of which is explained in the following subsections.

4.3.2.1. RQ3-1: Frequency of Backtracking

Overall, the analyzer detected a total of 15,095 backtracking instances within the 1,460 hours of coding activities, which gives an average rate of 10.3 instances per hour. That is, programmers were backtracking every six minutes on average, which is quite frequently considering that this number excludes all the trivial typo correction types of backtrackings. The rate varied across participants (min=3.8/h, max=28.4/h), but *all* participants were backtracking frequently.

4.3.2.2. RQ3-2: Size of Backtracking

The size of each backtracking was measured, in terms of how far the intermediate versions digressed from the original version. Minor backtrackings might not need much tool support, but it would probably be very helpful to have better tools for relatively larger backtrackings. To measure this size, the Levenshtein distance [Levenshtein 1966] was used, which is commonly referred to as the *edit distance* between two strings. In a backtracking instance, the size of the instance was obtained by first calculating the edit distances between the original version and each of the intermediate versions, and then taking the maximum value among them as the size of the backtracking. For instance, if an instance had a version history of A→B→C→A, then the size of the instance would be the maximum value of the distance between (A, B) and (A, C). In the earlier example presented in Figure 4-5, the backtracking size is 28, which is the edit distance between v_1 and v_3.

PID	Group	Time (h)	BI	BI/h	CRR	SR
P0	G0	294.1	2278	7.7	37.5%	10.1%
P1	G1	68.1	961	14.1	6.0%	11.8%
P2	G1	216.2	2450	11.3	26.3%	13.7%
P3	G1	2.6	73	28.4	0.0%	6.8%
P4	G1	64.2	1616	25.2	45.4%	8.1%
P5	G1	13.7	110	8.0	29.1%	17.3%
P6	G1	16.4	164	10.0	8.5%	6.1%
P7	G1	25.3	486	19.2	11.7%	8.4%
P8	G1	29.5	296	10.0	45.6%	10.8%
P9	G1	22.5	380	16.9	36.3%	11.6%
P10	G1	19.5	193	9.9	3.1%	14.5%
P11	G1	22.7	87	3.8	13.8%	10.3%
P12	G1	5.5	65	11.8	13.8%	1.5%
P13	G1	14.0	126	9.0	4.0%	1.6%
P14	G2	5.7	47	8.3	42.6%	6.4%
P15	G2	87.3	622	7.1	19.0%	11.4%
P16	G2	446.0	4179	9.4	4.3%	5.8%
P17	G2	28.0	116	4.1	44.0%	8.6%
P18	G2	21.2	186	8.8	4.8%	4.3%
P19	G3	51.2	605	11.8	2.0%	14.9%
P20	G3	6.2	55	8.9	0.0%	7.3%
Total/Mean		1459.9	15095	10.3	20.4%	9.5%

BI: Number of backtracking instances
BI/h: Backtracking instances per hour
CRR: Cross-run backtracking instances rate
SR: Selective backtracking instances rate

Table 4-5. Summary of the analysis results of the longitudinal backtracking study

Figure 4-6 shows the distribution of backtracking sizes. The horizontal axis (which is non-linear) represents the groups of backtracking sizes in ascending order, and the vertical axis shows the total number of backtracking instances in each group. It is shown that 1,304 backtrackings were a single character, which were 8.6% of all backtrackings. The most common backtrackings were between 10 and 49 characters. There were also 220 backtrackings that were larger than or equal to 1,000 characters.

The analysis determined that of all the single character backtrackings, 36% were performed on variable or method names, 26% on number literals, and 13% on string literals. For the larger size categories, 50 instances were randomly sampled from each categories and manually inspected to get a better sense of what kinds of backtrackings are represented at the different sizes. The 2-9 and 10-49 groups seem to be dominated by simple parameter/expression changes as in Figure 4-5, followed by simple name changes on methods or variables. The majority of 50-99 group seem to be single statement changes, and some instances were about surrounding existing code with control blocks (e.g., if, try-catch) and reverting it. The 100-

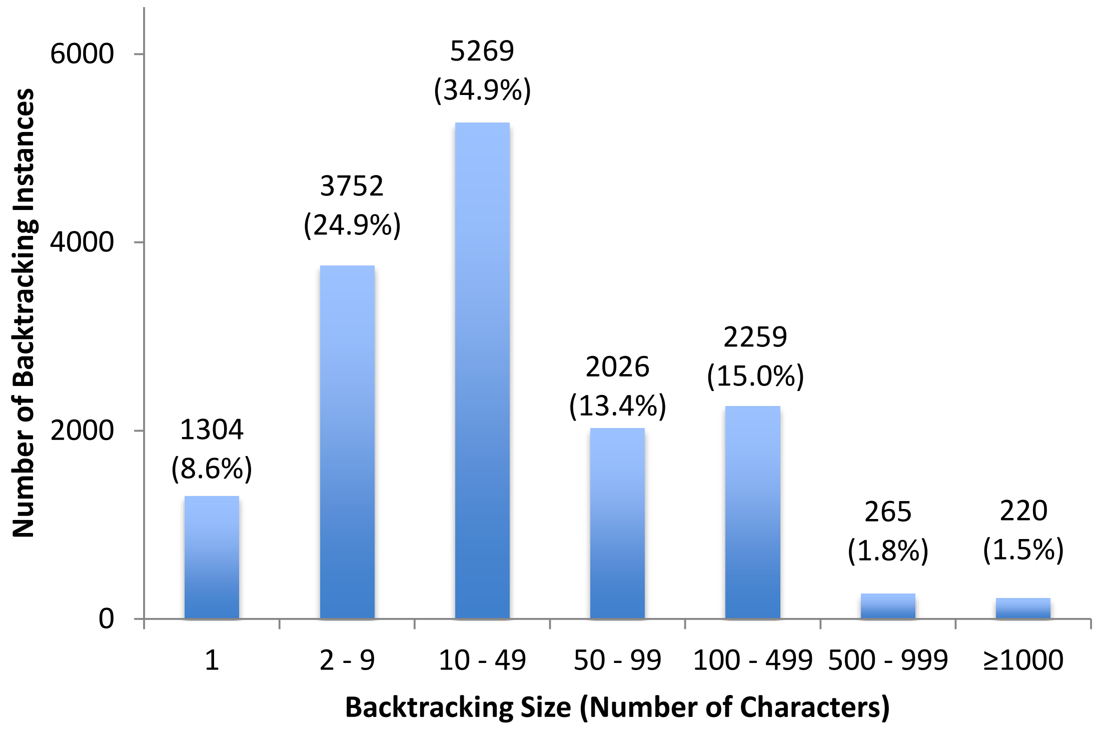


Figure 4-6. Distribution of all the detected backtracking sizes.

499 group instances seem to be mostly adding/removing/modifying multiple statements. The instances with sizes larger than 500 seemed to be significant algorithmic changes, adding or removing multiple methods, and so on.

4.3.2.3. RQ3-3: BACKTRACKING TACTICS

The ultimate goal of this research was to provide useful backtracking tools for programmers. Therefore, it was important to understand how programmers are backtracking now, and to determine whether there is room for improvement. Although this was investigated in the previous studies (Sections 4.1 & 4.2), this time it was possible to identify the editor features used for backtracking with actual data. The FLUORITE logs contained not only the code changes but also the editing commands (e.g., copy, undo, etc.), which allowed detecting what types of commands were used to accomplish the backtracking. The term *tactics* will be used to refer to the low-level editor features used for each backtracking, in accordance with prior research [Bates 1990][Grigoreanu 2010].

The analyzer reported the editing command that caused all the *backward changes* of the instance, which are the changes following the intermediate version with the greatest edit distance with the original version (Figure 4-7). When all the commands were of same type, we determined that command to be the backtracking tactic. Otherwise, we marked the tactic as *multiple*. We automated this process, but the analyzer could not determine the backtracking tactics for 9.43% of the instances. This happened when the logs did not have enough information, for example when the source code changed outside of Eclipse.

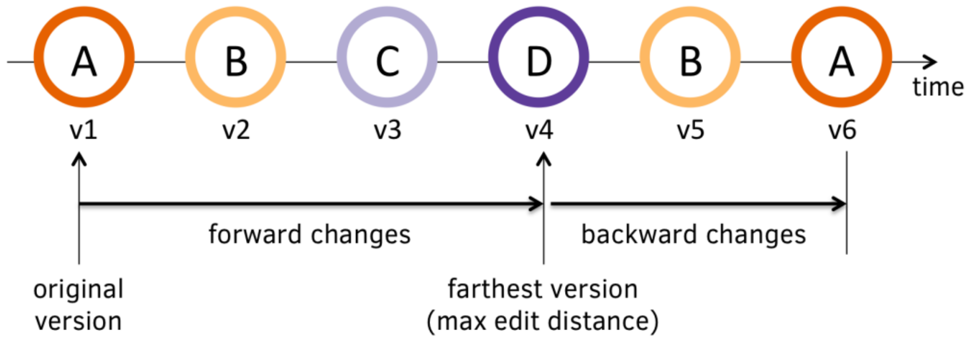


Figure 4-7. A backtracking instance illustrated. The analyzer determines the farthest version within each instance, and considers all the changes following the farthest version as backward changes.

Figure 4-8 shows the identified backtracking tactics. The most frequently used backtracking tactic was using the undo and redo commands, constituting 36.63% and 2.57% of all the backtracking instances, respectively. This implies that these instances were already supported by existing editor features. We also noticed that many of the undo commands were invoked repeatedly in sequence. In other words, the participants often used the undo command multiple times in order to revert the source file to an earlier state.

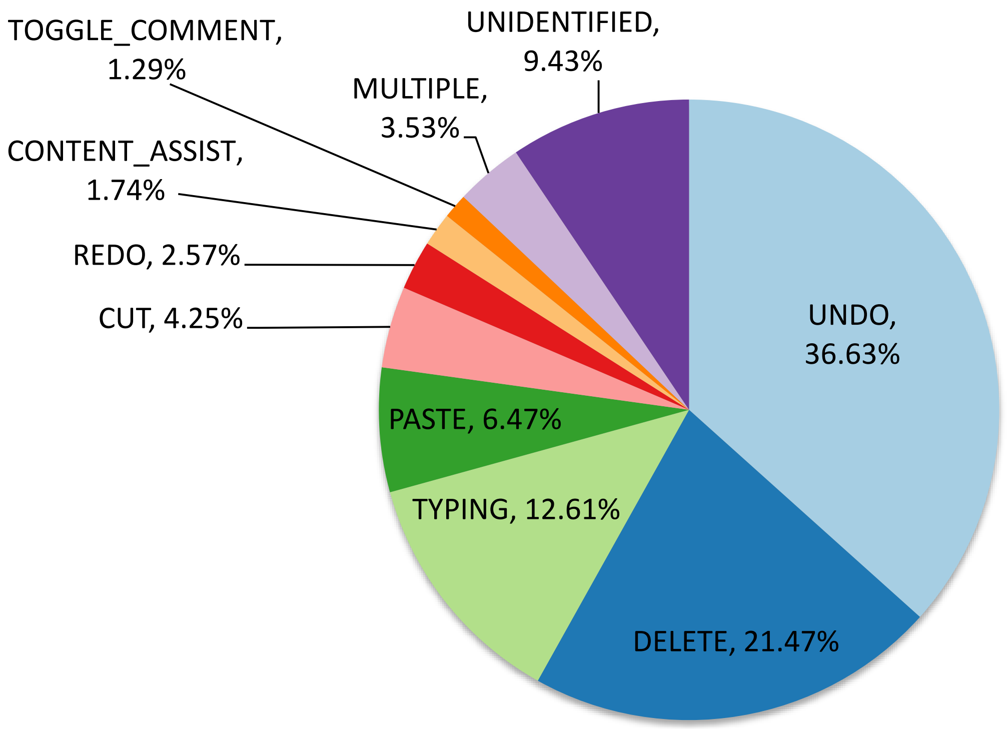


Figure 4-8. The identified backtracking tactics

The next most frequently used tactic was deleting some text from the code, constituting 21.47% of all instances. This includes using backspace key (16.33%), delete key (3.74%), delete line command of Eclipse (0.91%), and some combinations of these (0.49%). For these cases, once the code is located, backtracking itself is trivial; the programmer can easily select the code and delete it. Therefore, backtracking tools should help programmers *locate* the right piece of code to be removed, because that is the biggest challenge in this case.

The third most popular tactic was manually typing the desired code, which was used in 12.61% of all instances. The analyzer reported the cases with typing and deleting intermixed as typing, since these deletions can be seen as minor corrections happened while typing. The finding that manual typing is the third most popular tactic is particularly interesting, because it shows the lack of tool support for restoring deleted code. Manually restoring the deleted code or reverting modified code to the original version relies entirely on the programmers' memory, and thus can be error-prone. Even if the programmer knows exactly what she has to do, manual typing would be less efficient compared to just undoing – even selectively – that piece of code to the desired version.

Cutting (4.25%) and pasting (6.47%) were the next most popular tactics. Of all the backtracking performed by pasting, 41% were just undoing the preceding cut commands, which can be considered as an alternative way of copying the code.

3.53% of the instances were marked as multiple, which means that more than one type of editing commands were used to accomplish the backtracking. The rest of the identified tactics were using content assist features such as code completions and quick fixes (1.74%), and using the toggle comment feature (1.29%).

4.3.2.4. RQ3-4: Cross-Run Backtracking Instances

In situations such as designing a system or learning an unfamiliar API, programmers must explore and try out multiple alternatives, which has been called exploratory programming [Sandberg 1988]. It was interesting to investigate in how many of the detected backtracking instances were performed as part of exploratory programming. One of the ways of experimenting with code is to make some changes, run the application, and revert the code back to the way it was before if the code does not behave as desired. The analyzer checked whether there was an application run command between the first change and the last change of a backtracking instance, which we call *cross-run* instances. The FLUORITE logs contain run commands, such as launching the application under development and running unit tests within Eclipse, which made it possible to count such instances.

The cross-run instance rates are shown in Table 4-5, in the column titled "CRR" (cross-run rate). Overall, 20.4% of all instances were cross-run instances. This rate varied among participants, and two of them (P3 and P20) had no cross-run instances at all. Interestingly, the logs from these two participants were relatively short (2.6h and 6.2h, respectively) compared to the other logs, and did not contain any run commands at all. This could mean that these two people ran the application outside Eclipse for some reason, or did not run the application at all. All the other participants had some number of cross-run instances.

P14 had a 100% cross-run backtracking rate from the statement level analysis. Manual inspection of these six cross-run instances discovered that all of them were parameter tuning instances, for example adjusting some threshold value from 0.3 to 0.4 and then back to 0.3.

4.3.2.5. RQ3-5: Cross-Session Backtracking Instances

In most code editors, the undo feature works only within the same editing session, and users cannot undo the changes made in the past sessions (where a session is from launching the IDE to exiting). To determine whether it would be useful to make backtracking tools work across sessions, I counted the number of cross-session backtracking instances, where the editing sessions of the first change and the last change were not the same.

The analysis showed that 96.7% of backtrackings were done within the same editing session, and only 3.3% of all instances were cross-session backtracking. We also measured how many sessions did each instance go back, which is depicted in Figure 4-9. The graph shows that 99.0% of all backtracking was done within 3 editing sessions. In other words, a backtracking tool would work for the most (97.0%) cases with only the history within the same editing session, and providing the history of the last 3 sessions would cover 99.0% of the cases.

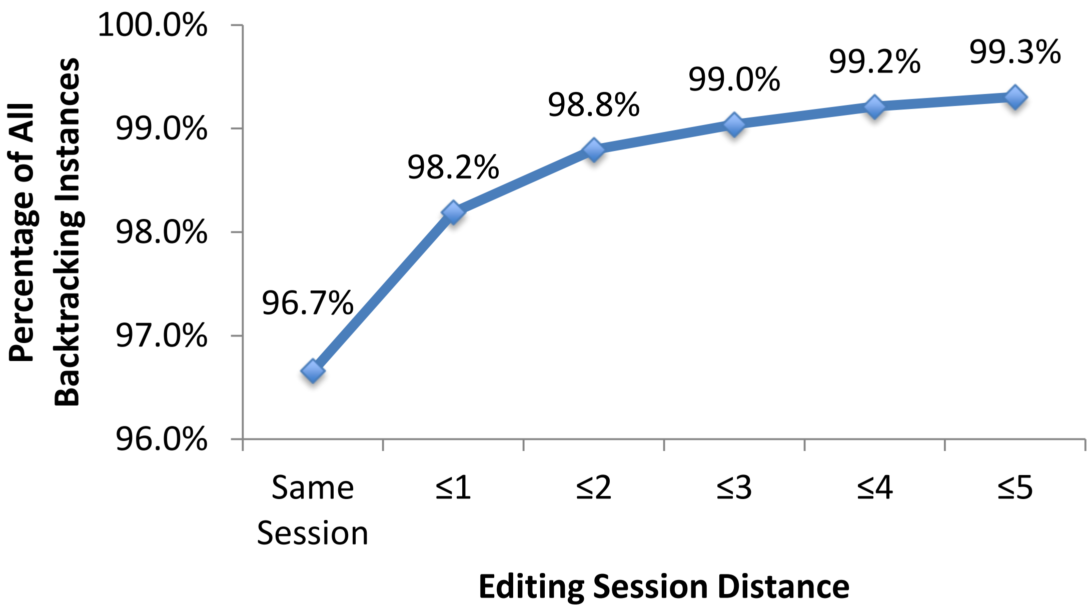


Figure 4-9. Cumulative percentage of all backtracking instances with different editing session distances. 96.7% of all backtrackings were performed within the same editing session. 99.0% of all instances have less than or equal to a 3 session distance.

4.3.2.6. *RQ3-6: Selective Backtracking Instances*

The analysis also investigated whether each backtracking instance was *selective* in nature. A backtracking instance is determined as selective when there are edits in the middle of the backtracking that change *other* parts of the same file, and that are not backtracked together. I was interested in this, because the restricted linear undo command [Berlage 1994], the conventional undo command used in most editors, cannot handle selective backtracking.

When determining selectiveness, the analysis was performed as conservatively as possible, even when there are intermixed changes to other parts of the code. First, all the backtracking done by undo/redo commands were automatically excluded. There were other subtle cases that needed to be excluded. For example, Figure 4-10 shows two possible backtracking scenarios in a source file. The source file has three statement nodes, s1, s2, and s3. In both scenarios, the three nodes are modified in turn and they are reverted back to the original version in various orders. We did not want to mark these instances as selective, because the undo command could be used multiple times to handle these cases. The first scenario shows such a case. However, the second scenario is also possible when the user performs backtracking by hand. To exclude these cases, we also looked ahead and checked the changes immediately following the last change of a backtracking instance, to see if those changes are reverting the intermixed changes to the other parts.

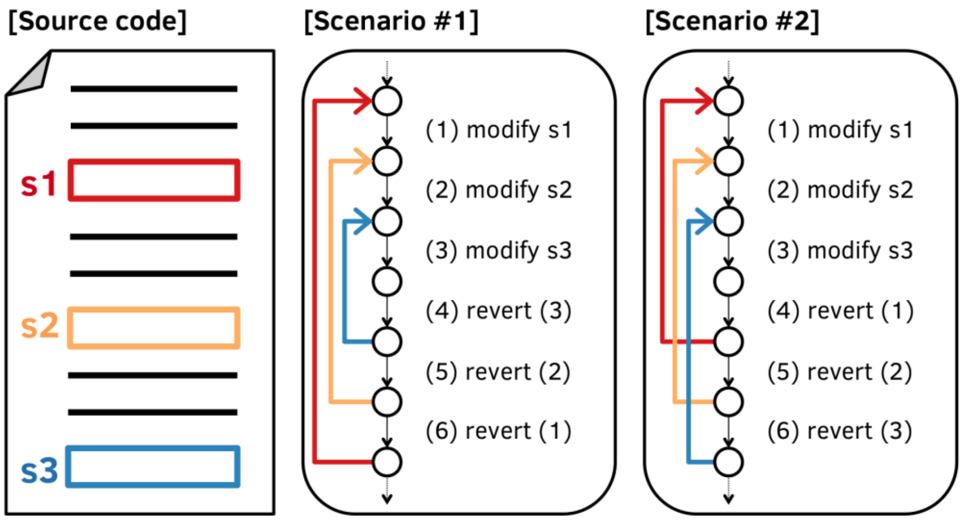


Figure 4-10. Two possible backtracking scenarios, whose backtracking instances are *not* selective. The source file has three different statement nodes being affected (s1-s3). Each backtracking scenario has three backtracking instances in each node. Except for the backtracking instance in s3 in scenario #1, all the backtracking instances have some changes to other parts of the same file within their timespan. Nevertheless, these are not selective because the undo command *can* handle both cases.

The results showed that 9.5% of all instances were selective, as indicated in Table 4-5 in the "SR" (selective rate) column. As explained in Section 4.3.2.3, the analyzer also reported the tactics of all the selective backtrackings. 63% of selective backtrackings were performed by manually deleting or typing the desired code, suggesting the need for backtracking tools. No significant difference was found between selective and non-selective backtrackings in terms of their sizes (that is, the selective backtracking had small and large sizes in a similar proportion as shown in Figure 4-6).

4.3.2.7. RQ3-7: Repeat-Count

When a certain node comes back to the same version more than once, the analyzer kept track of the repeat count of each backtracking instance. For example in Figure 4-4, *b*1 and *b*2 are of repeat count 1, but the repeat count of *b*3 is 2, because the node content backtracked to *A* for the second time.

Figure 4-11 shows the repeat counts of all backtracking instances. The blue bars indicate the number of instances in each repeat count group. When a backtracking happens more than once, the first time is counted in the first bar, the second time is counted in the second bar, etc. In other words, each bar is included in the previous bar. Each point on the red line is the next bar value divided by the current bar value, indicating the percentage of the instances that come back to the same state again. For example, 12,430 instances have the repeat count value of 1, and only 15.0% (1,868 out of 12,430) of them come back to the same state after some exploration in the future. While the number of instances dramatically falls as the repeat count increases, the revisiting fraction goes up. This implies that when a node comes back to the same state a few times, it is likely that the node will digress and return again.

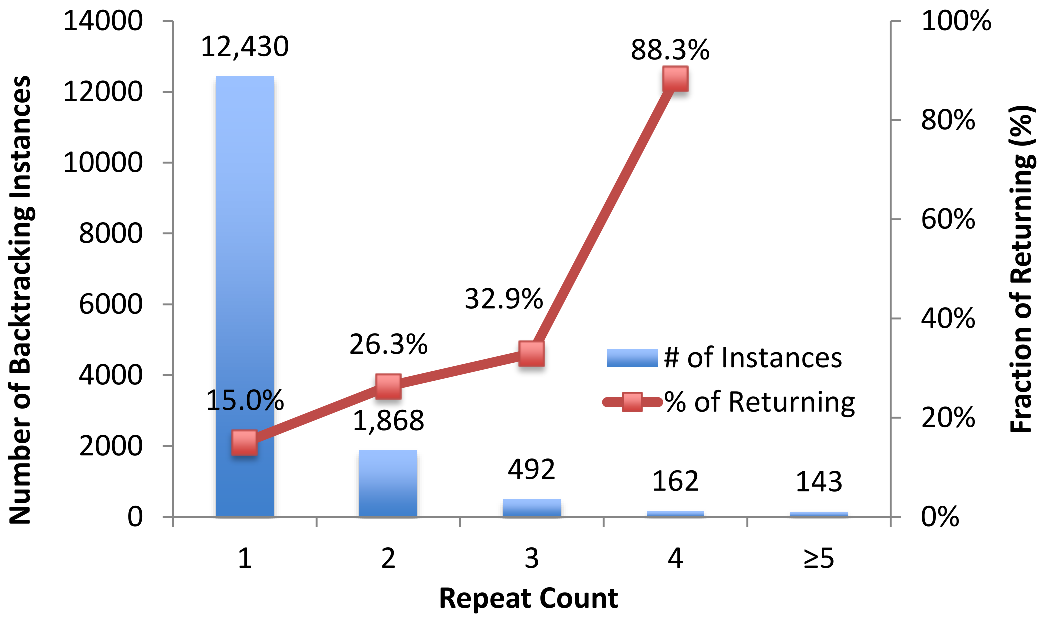


Figure 4-11. Repeat counts of all backtracking instances, along with the percentage fraction of revisiting the same state in the future.

One participant (P2) backtracked a statement to the same state 24 times, which was the maximum value of all repeat counts. This was another parameter tuning instance, where she was experimenting with different width value for a line drawn on a canvas. The next biggest repeat count was 9, where a statement that translated a graphical object on the screen was removed and put back again multiple times.

4.3.3. Limitations

There are a few limitations of the analysis method used in this study. First, the analyzer can only detect *exact* backtracking instances, which has several implications. For instance, when a programmer changes two parts within a statement and reverts only one of them, the analyzer would not be able to catch this backtracking instance because the smallest level of granularity we used was the statement level.

Moreover, all the detected instances are *successful*, and the analyzer cannot detect cases where the programmer intended to backtrack but failed, which occasionally happened in the preliminary lab study (Section 4.1.7). If programmers fail to backtrack correctly, it would imply the need for better backtracking tools even more. The analysis also missed any *near-exact* backtracking instances. For instance, suppose a variable name **`fontSize`** changes to **`rectangleSize`**, then to **`regionArea`**, and finally to **`fontArea`** in that order (changed parts are underlined). Conceptually, this is backtracking because the front part of the variable name has changed from **`font`** and then back. This case is interesting, because even a selective undo tool cannot easily handle this because selectively undoing the first change (**`fontSize`** → **`rectangleSize`**) has what I call a "region conflict" with the following change which deleted a portion of **`rectangle`**. The analysis would not count this case. More detailed discussion of such region conflicts appears in Section 5.1.2.

All the participants were programming in Java. There may be some backtracking patterns that occur more often when coding in Java, and the statistics presented in this analysis may not be generalizable to other programming languages. In addition, if a similar analysis were to be performed for a different programming language, the analyzer might need some language-specific tweaks. For example, we needed to take Java's anonymous class definition into account, in order to filter out duplicated results.

In addition, all the logs were collected within Eclipse, which might have affected the results. The backtracking patterns may vary if programmers are using different code editors or IDEs that provide different code editing features. Nevertheless, considering that most available IDEs provide a similar set of editor commands and all programmers need to do tasks like understanding APIs and determining the right parameters for methods, I believe that the patterns would not be drastically different from the results reported here.

All participants, including the professional programmers, were working in academic settings. It is unknown whether there would be any significant differences in industrial settings. None of the participants were novices learning to program, and studying novice backtracking behaviors would be an interesting area for future work. I included my own logs (P0 in Table 4-5), which might have biased the results. Because my main research focus was on backtracking, I might have performed backtracking more than the average programmers. However, I was able to confirm that excluding my own logs does not change the results much. For example, the backtracking rate becomes 11.0/hour when P0 is excluded, compared to 10.3/hour as we reported in Section 4.3.2.1, and the selective backtracking rate remains the same at about 9%.

There were a few types of information that seemed interesting but could not be obtained. First, because the log data did not record the version control system (VCS) features used by the programmers such as *commit* and *revert*, the analyzer could not determine the fraction of backtracking done by the VCS. If the source code is backtracked using the revert feature of a VCS, the analyzer would still detect the backtracking but mark the tactic as unidentified (see Figure 4-8). If it was known which ones were due to the use of a VCS, the analyzer would have been able to distinguish the types of backtracking which would be better handled by a VCS and those most suited for in-editor features. This would help in designing backtracking tools to work better with existing VCSs.

In addition, I wanted to know whether there were semantic relationships among the backtracking instances. For example, suppose a programmer first renames a method and all the callsites, and then reverts these changes later on. The analyzer would detect the individual backtracking which occurred in the method definition and the callsites separately. It would be clear that these instances are all related to each other by manually inspecting these instances, but the analyzer could not automatically determine such relationships.

I also wanted to learn how programmers navigate to the backtracking location before they actually perform backtracking changes. However, this information could not be determined partly because there was some missing information needed in the logs, and also because it required too much manual investigation. There are diverse ways of navigating in Eclipse, including keyboard keys, mouse clicks and scrolling, and other Eclipse commands such as various searches. The logger did not catch mouse scroll events, and when the users are performing searches, only the initial search command is logged but the following events for clicking on one of the search results to actually jump to the relevant code are not logged, which makes it difficult to analyze the exact navigation command used by the programmers. Improving the analysis method and looking for the missing pieces of information remains as future work.

4.4. Conclusion

It is shown from these studies that backtracking is prevalent, and when and how it happens. Backtracking is inevitable in programming, including when programmers are exploring a design space, experimenting with different options, or just make a mistake. The findings from the preliminary lab study (Section 4.1) provided some insights on how programmers back-track and what problems they have while backtracking. The participants commented out code or used the regular undo command when applicable, but there were many situations where they had to manually perform backtracking. Moreover, programmers had some problems while backtracking, such as failing to locate the right code to be backtracked. Since the lab study did not provide any information on how often programmers face backtracking situations, a follow-up online survey was conducted (Section 4.2). The results show that the programmers' perceived frequency of backtracking is quite high. However, it was difficult to tell whether the perceived frequency matches the reality.

To overcome the limitations of the previous studies and get more accurate backtracking data in real settings, a huge data set of programmers' fine-grained code edit history has been analyzed using the abstract syntax tree node history tracking analysis (Section 4.3). The results confirmed that programmers are in fact backtracking a lot (10.3/hour), and 40% of the detected backtrackings were performed manually, indicating that there are backtracking situations not very well supported by existing programming tools.

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 programmers experiment with code effectively, and improve their productivity.

On top of that, the analysis technique used for the longitudinal study may have applications beyond detecting backtracking, and other researchers could benefit from analyzing fine-grained code change patterns to better understand programmers' coding practices and to provide useful tools for programmers.

5. A Selective Undo Mechanism for Code Editors[29]

The results of the backtracking studies in Chapter 4 shows that the current tool support for backtracking is limited, and programmers are facing challenges while backtracking in their code editor. The key insight of this dissertation is that these problems can be addressed by providing *selective undo* in code editors. Users could select specific edit operations performed in the past, for example, the insertions of the print statements for debugging, and invoke the selective undo command to revert only the code affected by the those operations.

However, providing selective undo in a code editor poses major challenges. First, text and source code does not have the notion of identifiable objects but rather has a stream of characters [Berlage 1994]. Unlike in a drawing application, an edit operation is not performed on a specific object, and only the location of the edit matters. Second, there can be regional conflicts among the edit operations. A *regional conflict* can occur when the region of a later edit overlaps with the region of the earlier edit which the user wants to selectively undo. When there is a regional conflict among edit operations, the result of a selective undo may not be well defined.

This chapter describes the technical details of performing selective undo in code editors, addressing the problems mentioned above. First, it describes the internal data structure maintained to provide the selective undo feature which is capable of handling regional conflicts (Section 5.1). Next, the high-level selective undo algorithm is presented, with some implementation details that will help others to replicate the selective undo algorithm (Section 5.2).

The selective undo algorithm described here uses the *inverse model* (Section 2.1.1), which means that the selective undo effectively puts the inverse of the selected operations at the end of the history, as opposed to the script model which pretends that the undone operation has never happened. There are several reasons I chose to use the inverse model over the script model. In the inverse model, the history is represented more like real life: the past is immutable and new events can only be added to the end of the history. This might help users recall the code edit history better, when reviewing the history or backtracking. The inverse model also makes it a lot easier to undo the selective undo operations, just like any other

[29] Portions of this chapter appeared in [Yoon 2015]

operations. Moreover, in order to detect and handle the regional conflicts correctly, the current locations of the past edit operations should be updated as new edit operations are added to the history, as will be discussed in Section 5.1.3. This dynamically updated location information makes it easier to implement the inverse model selective undo, as explained in Section 5.2. On the other hand, it is not clear how the regional conflicts could be handled using the script model.

The mechanisms described in this chapter are implemented in our prototype tool AZURITE.[30] The selective undo user interfaces provided by AZURITE will be discussed in Chapters 6-8.

5.1. INTERNAL EDIT HISTORY REPRESENTATION FOR SELECTIVE UNDO

5.1.1. DEFINITIONS

In text editing, there exist only a few types of primitive edit operations. Here, the following three operations are considered as primitive: *insert*, *delete*, and *replace*. These operations are considered primitive in that these operations are the basic undoable units in the system. A replace operation is treated as a primitive operation even though it can be decomposed into a delete followed by an insert with the same offset. This is because, conceptually, a replace operation is used for changing some code *A* to some other code *B*, but only undoing either the decomposed delete or the insert portion of a replace operation would result in neither *A* nor *B*. Therefore, a replace operation should be undoable as an atomic unit, consistent with the regular undo. All of the higher-level editing commands that change the text, such as find-and-replace and cut-and-paste, result in one or more of these primitive operations. Since AZURITE uses the inverse selective undo model, the history contains the records of *every* operation that happens, and new operations are always added to the end of the history. In particular, regular undo commands and the selective undo commands are also included in the history using these primitives. For example, the undo of a delete operation is added to the end of the history as an insert operation.

However, it is not enough to simply keep the edit history to provide the selective undo feature. When performing a selective undo on some past edit operations, the system must be able to determine which locations *in the current state* correspond to the locations where the operations were originally performed. This is not trivial because the offsets change whenever some text is inserted or removed above the location in the file of the past edit. This problem has been identified by others (e.g., [Reiss 2008]) and arises from the requirement that code be stored as plaintext without embedded meta-information like bookmarks. Thus, before a selective undo can be performed, locations of previous operations in the file may need to be adjusted dynamically. The adjusted location information of an edit operation along with some additional meta-data required for selective undo will be referred to as a *dynamic segment*.

[30] AZURITE is a blue mineral, and here it stands for: **A**dding **Z**est to **U**ndoing and **R**estoring **I**mproves **T**extual **E**xploration.

More formally, a dynamic segment is defined as a 4-tuple (k, o, τ, r), where the value of k is either `ins` or `del`, o is the dynamic offset (i.e., start position) of this segment, τ is the deleted or inserted text, and r is a relative offset value (can be nil) which will be explained in Section 5.1.4. The notion of dynamic offsets here is similar to the notion introduced by Abowd and Dix in the collaborative editing context (which they call *dynamic pointers*) [Abowd 1992], but the system needs extra information for selective undo. In addition to the dynamic offsets, the selective undo system keeps track of dynamic *segments* instead of pointers. There are two types of dynamic segments: *insert segments* ($k = \text{ins}$) and *delete segments* ($k = \text{del}$) – a replace will use both.

Let Ω be the set of all possible primitive edit operations. A primitive edit operation $e \in \Omega$ can be defined as a 3-tuple (t, d, I), where t is the time the operation was performed, d is the delete segment associated with the edit operation (can be nil), and I is the set of insert segments (can be nil). Each dynamic segment is always associated with one edit operation. The segment information is updated whenever a new operation is added to the history. I is a set rather than a single insert segment, because an insert segment can be split by some later performed operation, as will be explained below.

Finally, the edit history is then defined as a chronological history list of edit operations, $H \coloneqq e_1 e_2 \dots e_N$, where N is the number of all edit operations performed so far, and e_N is the last (newest) edit operation.

Before explaining how the dynamic segment is updated as the edits are made in the code editor, the issue of regional conflicts among the edit operations is discussed first, because updating dynamic segment information is also crucial for detecting regional conflicts.

5.1.2. Regional Conflicts of Edit Operations

A selective undo operation may not be well defined when there are *regional conflicts* among the edit operations. Recall the example from Chapter 1, where a replace operation e_1 changes the code from "`myFontSize = 12;`" to "`myRectangleSize = 12;`" and at some time later, another operation e_2 changes it to "`myRegionArea = 12;`". What should be the result of selectively undoing operation e_1 alone? There are multiple options:

A1. Selectively undo e_1 while leaving in the parts changed by e_2, resulting in "`myFontgionArea = 12;`"
A2. Also undo the conflicting operation e_2 and revert the code to the way it was before operation e_1 was performed, resulting in "`myFontSize = 12;`"
A3. Tell the user that there is a regional conflict and do not allow the selective undo, so the code stays as "`myRegionArea = 12;`"

Since it is not entirely clear what the user wants, it would not be appropriate for the system to choose arbitrarily without user intervention. Throughout the dissertation, the notation $e_i \rightarrow e_j$ will be used to represent a regional conflict where the edit region of $e_j = (t_j, d_j, I_j)$

overlaps with the region of $e_i = (t_i, d_i, I_i)$ at least partially, and $t_i < t_j$. For example, $e_1 \rightarrow e_2$ holds in the example above. Note that these two operations need not necessarily be consecutive in time, and there may be arbitrarily many edit operations in between.

As the arrow implies, a regional conflict is always one-directional: the arrow starts from some earlier performed operation and points to a later performed operation. For convenience, when there is $e_i \rightarrow e_j$ conflict, e_i will be called the *conflictee*, and e_j the *conflictor*. Regional conflict detection happens at the segment level. Since there are only two types of dynamic segments, insert and delete, there are total of four combinations of possible regional conflicts (Figure 5-1a-d).

5.1.2.1. INSERT → INSERT CONFLICT

This occurs when the second insertion is performed somewhere in the middle of an existing insert segment (Figure 5-1a). (If the second insert is at either end of the first insert and not in the middle, that is not considered a conflict.) When the user tries to selectively undo the first insert operation, it is not clear whether the user really wants to remove only the text inserted by the first operation, or to remove them both together.

Figure 5-2 illustrates why it is difficult to automatically determine what the user wants, when the user tries to selectively undo the first Insert operation. In the first example (Figure 5-2a), the user first inserted two lines of code to print out some debug messages when the program is entering and exiting the method (conflictee). Then, the user fills in the actual method body (conflictor). Because the method body is inserted between the two debug messages that were

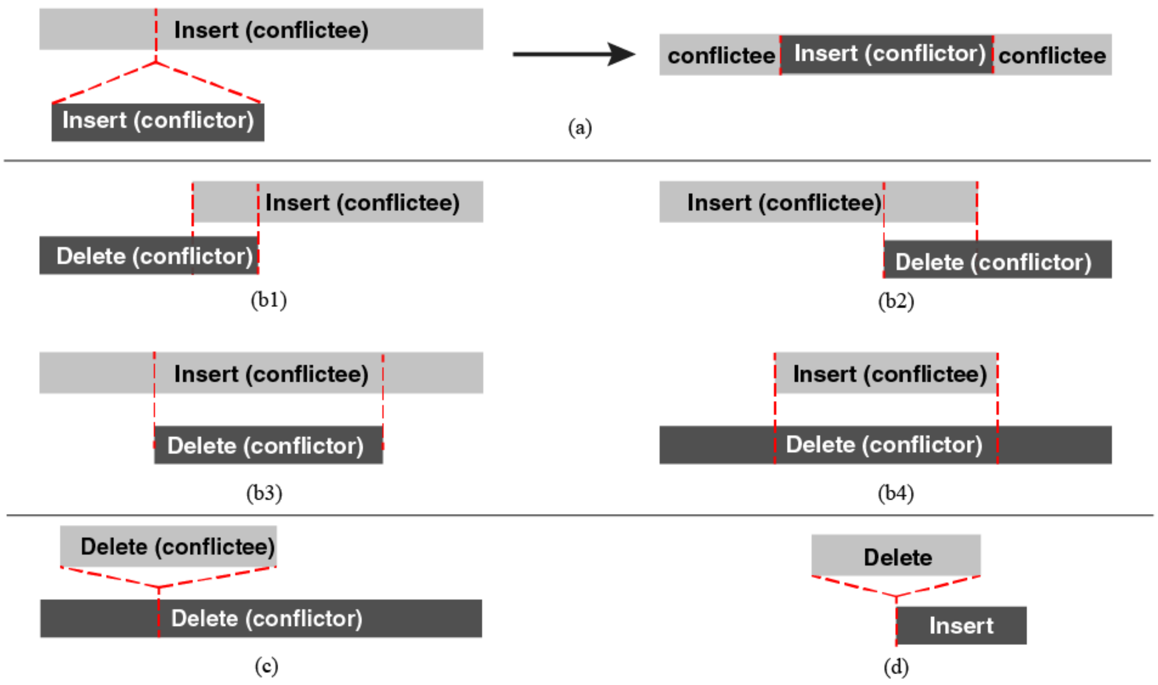
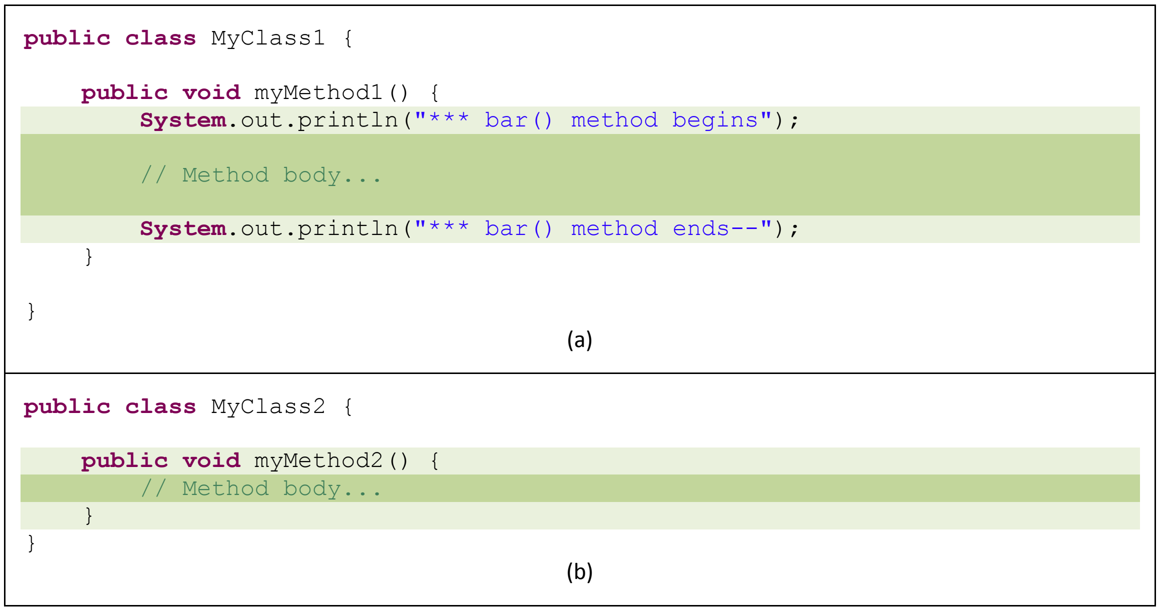


Figure 5-1. Types of regional conflicts illustrated.



```
public class MyClass1 {

    public void myMethod1() {
        System.out.println("*** bar() method begins");

        // Method body...

        System.out.println("*** bar() method ends--");
    }

}
```

(a)

```
public class MyClass2 {

    public void myMethod2() {
        // Method body...
    }
}
```

(b)

Figure 5-2. Ambiguity in the case of Insert → Insert conflicts. In both examples, the lighter shade indicates the code inserted first (conflictee), and the darker shade indicates the code inserted later (conflictor).

initially inserted, it results in an Insert → Insert conflict. Here, if the user later invokes selective undo on the first insert operation which added the two debug messages, it is likely that the user wants to only remove the debug statements while *leaving* the method body code in place.

In the second example (Figure 5-2b), the user first adds a method stub (conflictee), and then writes the method body (conflictor), which again causes an Insert → Insert conflict. However, unlike the previous example, if the user invokes the selective undo on the first operation, it is more likely that she actually wants to remove the *entire* method, because leaving in the method body without the surrounding method signature does not make much sense. This shows why resolving this region conflict must be left to the user (see Section 8.4).

When an Insert → Insert conflict is created (that is, when the user performs the conflictor edit), the insert segment of the conflictee is split into two pieces, as shown in Figure 5-1a.

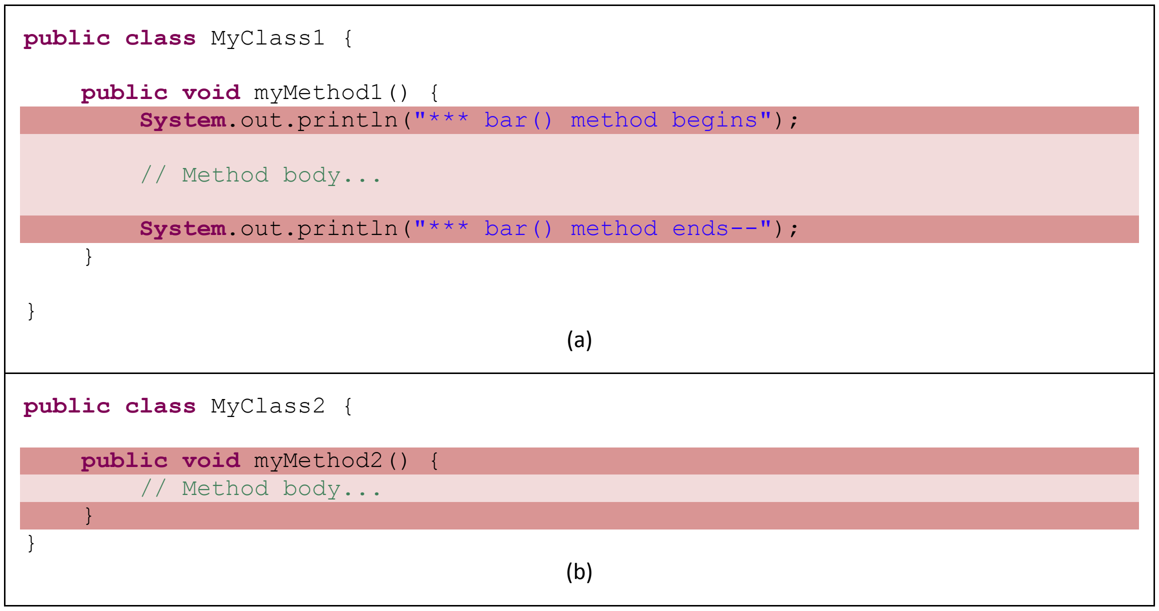
5.1.2.2. Insert → Delete Conflict

An Insert → Delete conflict occurs when the deletion range overlaps at least partially with an existing insert segment, which can happen in four different ways (Figure 5-1b). Fortunately, there is no ambiguity when the user wants to selectively undo the conflictee; all the remaining text that came from that insert operation should be removed.

Similar to the Insert → Insert conflict, an Insert → Delete conflict can also split the first insert segment into multiple (up to three) pieces. In case where the delete operation occurs in the middle of an insert segment (Figure 5-1b3), the existing insert segment is first split into three pieces, and the middle piece becomes an empty segment.

5.1.2.3. Delete → Delete Conflict

A Delete → Delete conflict occurs when the second deletion range encloses the first deletion range (Figure 5-1c). Similar to the Insert → Insert case, when the user tries to selectively undo the first delete operation, it is not clear if the user really wants to restore only the text deleted by the first deletion, or restore the whole text deleted by both operations.



```
public class MyClass1 {

    public void myMethod1() {
        System.out.println("*** bar() method begins");

        // Method body...

        System.out.println("*** bar() method ends--");
    }

}
```

(a)

```
public class MyClass2 {

    public void myMethod2() {
        // Method body...
    }
}
```

(b)

Figure 5-3. Ambiguity in the case of Delete → Delete conflicts. In both examples, the lighter shade indicates the code deleted first (conflictee), and the darker shade indicates the code deleted later (conflictor).

Figure 5-3 illustrates this issue. The code is exactly the same as the one shown in Figure 5-2, but now the code is being deleted rather than inserted, in the opposite order. In the first example (Figure 5-3a), the user first deletes the method body that lies between the two debugging statements (conflictee). Then, the user selects the area in the code where remaining debug statements and then deletes the selected code (conflictor). This results in a Delete → Delete conflict because the second deletion range encloses the location where the first deletion occurred. If the user later invokes selective undo on the first delete operation which deleted the method body, it is likely that the user wants to only restore the logic code while leaving the debug statements deleted.

In the second example (Figure 5-3b), the user first deletes the method body (conflictee), and then deletes the whole method signature (conflictor), which again causes a Delete → Delete conflict. However, when selectively undoing the first delete operation, it is more likely that she actually wants to restore the entire method, because only restoring the method body without restoring the enclosing method signature may not make much sense.

5.1.2.4. Delete → Insert Conflict (Cannot Occur)

As shown in Figure 5-1d, there can never be a Delete → Insert conflict because the range of deletion becomes a single point, and insert operations also only happen at a single point, and two points cannot conflict.

5.1.3. Keeping the Dynamic Segments Up To Date

Each dynamic segment is always associated with an operation, and the segment information is immediately updated whenever a new operation is added to the history. Keeping the dynamic segment information up to date is the essence of the selective undo mechanism. Simply put, when a single segment needs to be selectively undone, the system can easily refer to the current dynamic offset of that segment and apply the inverse operation of that segment.

Updating the dynamic segments is also important for detecting conflicts among the operations. The conflicts can be immediately detected whenever a new edit is made, and these conflict relationships can be stored within the edit operation objects so that the selective undo mechanism can refer to the information when needed. Finally, having these dynamic positions makes it trivial to search for all of the edits performed in a certain area of code.

The dynamic segment management mechanism is illustrated with an example in Figure 5-4. For simplicity, a dynamic segment is denoted as <*offset*, *length*> in this figure. At first, there are no operations in the history. OP1 inserts `println()` to the code. Since there are no existing operations, there is no need to adjust any dynamic segments, and it would simply add a new insert segment <0, 9>. Next, OP2 inserts "`Hello`" within the parentheses, which results in an Insert → Insert conflict with OP1. This essentially splits the existing insert segment into two pieces, pushes the second piece to the right by the length of "`Hello`", and then adds the

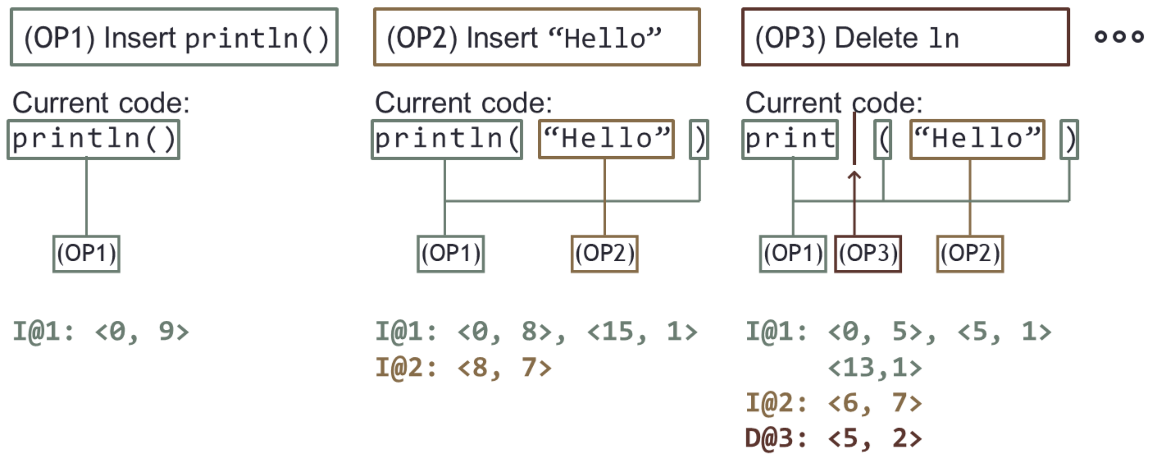


Figure 5-4. Illustration of dynamic segment management. For simplicity, each dynamic segment is denoted as <*offset*, *length*>. OP1 inserts `println()`, OP2 inserts "`Hello`" within the parentheses, and then OP3 deletes `ln` from the method name `println`, in temporal order. Below the code is illustrated how the existing dynamic segments are updated or split as new edit operations are added to the history.

```
def **update**($H$, $newOp = (t, d, I) \in \Omega$):
    forall $e$ in $H$, forall $s$ in $segments(e)$:
        if $d$ is not nil:
            **updateSegment**(s, $d$)
        if $I$ is not nil and $I = \{i\}$:
            **updateSegment**(s, $i$)

    append $newOp$ at the end of $H$

def **updateSegment**($s_1 = (k_1, o_1, \tau_1, r_1)$, $s_2 = (k_2, o_2, \tau_2, r_2)$):
    if $s_2$ precedes $s_1$:
        adjust $o_1$ appropriately
    elif $s_2$ overlaps with $s_1$:
        split $s_1$ if necessary (Figure 5-1)
        close $s_1$ if necessary (Section 5.1.4)
        store conflict info $op(s_1) \rightarrow op(s_2)$
```

Figure 5-5. Pseudo code illustrating the dynamic segment updating algorithm.

new insert segment <8, 7> in the middle. Finally, OP3 deletes **ln** from the method name **println**, which also conflicts with OP1. This first splits the existing insert segment <0, 8> into three pieces, deletes the middle one, and adjusts the third piece by 2. Then all the other dynamic segments on the right side of this delete are adjusted by 2 as well. After all of these adjustments are done, the new delete segment <5, 2> will be added. In case of a delete segment, the length value does not reflect the actual length of the segment in the current context, but it refers to the length of the deleted text. A delete segment is instead represented by a single offset in the current code.

Python-like pseudo code for the edit history management technique is shown in Figure 5-5. **update(**H, $newOp$**)** would be called whenever a new edit operation $newOp$ is performed.

5.1.4. *Segment Closing / Reopening*

When updating the dynamic segment information, a naïve implementation may lose some of the required information for selective undo. Suppose that there was originally "**xyz**" in the code at offset 10, and two delete operations e_a and e_b are performed, whose delete segments are $d_a = (\text{del}, o_a, \tau_a, r_a)$ and $d_b = (\text{del}, o_b, \tau_b, r_b)$, respectively. First, e_a deletes "**y**" in the middle, resulting in $o_a = 11$. Next, e_b deletes the remaining "**xz**", which results in $o_b = 10$ and a Delete → Delete conflict. In addition, o_a is adjusted to 10, so that selectively undoing just e_a would put "**y**" at the correct offset. What would happen if we undo both e_a and e_b at this point?

As will be explained in Section 5.2, the selective undo has to be performed backwards (that is, starting from the latest edit operation among the selected ones) to avoid messing up dynamic segments while performing the selective undo, e_b would be undone first, and "**`xz`**" would be put back at offset 10. Next, e_a would be undone and it would put back "y" at offset 10, which is the current value of o_a. The resulting code would be "**`yxz`**" instead of "**`xyz`**", which is clearly not what the user wanted.

This happens because the dynamic segment for the conflictee (e_a) loses the offset information relative to the conflictor (e_a)'s offset when a Delete → Delete conflict occurs. To solve this problem, the system needs to (1) store this relative offset and (2) restore the offset when the conflictor is undone. These two processes are called *segment closing* and *segment reopening*, which work as follows:

$$\textbf{close:}\quad r_a \coloneqq o_a - o_b \qquad \textbf{reopen:}\quad \begin{aligned} o_a &\coloneqq o_b + r_a \\ r_a &\coloneqq nil \end{aligned}$$

In the previous example, closing d_a will store $o_a - o_b$ = 11 – 10 = 1 into r_a, right before o_a is adjusted to 10. When undoing d_b, d_a will be reopened, which will restore the offset o_a to o_b + r_a = 10 + 1 = 11 and make r_a be nil. Once a segment is closed ($r \neq nil$), then its r value is never updated unless the segment is reopened. The same applies to Insert → Delete conflicts.

5.2. SELECTIVE UNDO ALGORITHM

Provided that the system has all the dynamic segments information for the entire history, it can perform selective undo. Users select one or more edit operations in the past, which are not necessarily consecutive in time and then invoke the selective undo command. The user interface for selecting desired edit operations will be presented later in Chapters 6-8. The selective undo is performed in two phases: *determining code chunks*, and *performing selective undo for each chunk*. For simplicity, this section explains the algorithm with the assumption that all the selected operations are coming from the same source file. When the selected operations come from multiple source files, the same algorithm is applied for each of those files.

5.2.1. PHASE #1: DETERMINING CODE CHUNKS

Unlike the conventional undo, the selective undo mechanism described here allows the user to select *multiple* operations to be undone together. Since there is no guarantee that all the selected operations were performed at the same place, the selective undo mechanism must first find all of the *code chunks* affected by the selected operations. A code chunk consists of one or more operations and their dynamic segments, which constitute a continuous area in code.

Three rules are used to determine the code chunks. First, all the segments associated with a same operation must be added to the same chunk. Second, all the segments located between two segments in a same chunk must also be added to that chunk. Finally, all abutting chunks are merged into a single chunk.

5.2.2. *Phase #2: Performing Selective Undo for Each Chunk*

Once the chunks are determined, selective undo is performed for each chunk independently. For convenience, this is done from the bottommost chunk within the file so as not to affect the dynamic offsets of the segments of the other chunks while performing selective undo. Selective undo is performed in two different ways, depending on the existence of regional conflicts outside of the chunk, which is defined as following:

$$rcExists = \exists e_i, e_j \in \Omega \; such\ that \; e_i \in \Sigma \wedge e_j \notin \Sigma \; \wedge e_i \rightarrow e_j,$$

where Ω is the set of all edit operations, and $\Sigma \subseteq \Omega$ is the set of all the selected operations in the chunk. The point is that regional conflicts can be automatically resolved if the conflictor is also selected ($e_j \in \Sigma$), which would not have been possible if the system only allowed selectively undoing one operation at a time. When there exists a conflictor that is *not* selected together ($e_j \notin \Sigma$), user intervention is required.

5.2.2.1. *If rcExists = false*

In this case, selective undo can always be performed without user intervention by applying the inverse changes of all the operations involved in this chunk, working backwards from the most recent operation to the oldest one. The inverse changes should be applied to each dynamic segment when there are multiple segments associated with an operation. When applying the inverse change of a delete segment, all the segments closed by this segment should be reopened (see Section 5.1.4).

5.2.2.2. *If rcExists = true*

In my selective undo mechanism, the users are provided with the different alternatives of possible resulting code so that they can choose one of them or cancel the selective undo, when there are conflicts (the actual user interface for this will be presented in Section 8.4). Using the font size example from Section 5.1.2, all three alternatives, A1, A2, and A3, should be provided to the user. A3 is simply the same code as it is, so the system only needs to calculate A1 and A2.

A1 is obtained by selectively undoing the conflictees while leaving in the parts changed by the conflictors. It turns out that A1 can be obtained by applying the same algorithm for conflict-free chunks used when **`rcExists=false`** without extra work.

In order to get A2, the chunk needs to be *expanded* to include its conflicting operations. However, there might be other operations conflicting with the ones just included in the chunk. For instance, in our example, there could be another later operation e_3 which changes the code from "**`myRegionArea = 12;`**" to "**`myRegionBreadth = 12;`**", which conflicts with operation e_2 but not with e_1. AZURITE includes all of these transitive conflictors to get the expanded chunk.[31] Once the expanded chunk is obtained, A2 is calculated by applying the selective undo algorithm as for non-conflicting chunks. The selective undo algorithm is illustrated in Figure 5-6.

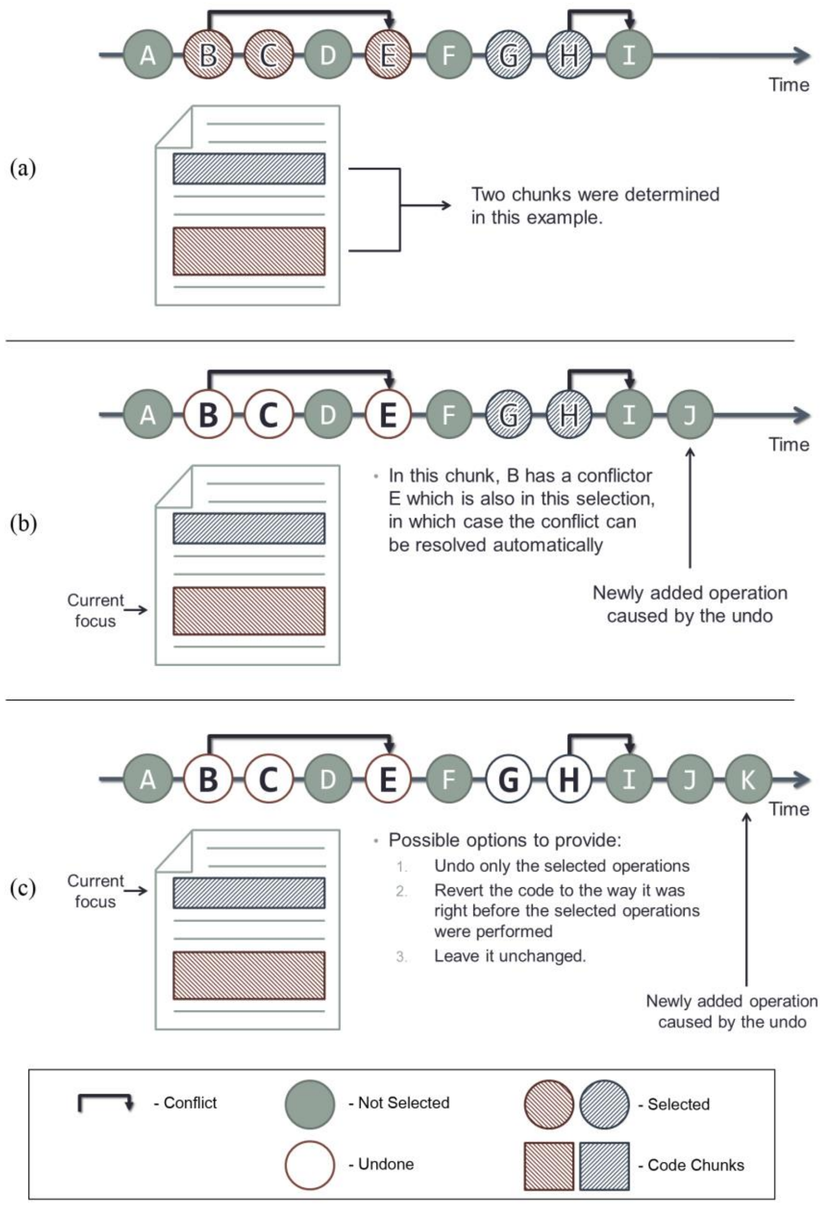

A
B
C
D
E
F
G
H
I
Time
(a)
Two chunks were determined in this example.
A
B
C
D
E
F
G
H
I
J
Time
(b)
In this chunk, B has a conflictor E which is also in this selection, in which case the conflict can be resolved automatically
Current focus
Newly added operation caused by the undo
A
B
C
D
E
F
G
H
I
J
K
Time
(c)
Current focus
Possible options to provide:
1. Undo only the selected operations
2. Revert the code to the way it was right before the selected operations were performed
3. Leave it unchanged.
Newly added operation caused by the undo
- Conflict
- Not Selected
- Selected
- Undone
- Code Chunks

Figure 5-6. Illustration of the selective undo mechanism. First, the algorithm determines the code chunks affected by the selected operations (a), and then performs selective undo on each chunk separately. When there are no conflicts outside of the chunk, selective undo can be performed without user intervention, and the undo operation is added as "j" (b). When there are some irresolvable conflicts, it provides the three alternatives of possible resulting code to the user and if the user selects a change, the operation is added as "k" (c).

[31] In edge cases such as if the user replaces an entire file with an older version, AZURITE extracts the diffs between the two versions and logs the diffs as separate operations to minimize the potential future regional conflicts.

5.3. Discussion

5.3.1. *Ensuring the Correctness of the Dynamic Segments*

In the initial version of the selective undo algorithm built into the AZURITE tool, there were some subtle bugs where the selective undo results were occasionally incorrect, which were very difficult to reproduce. These bugs were identified by randomly generating a large sequence of edit operations and checking if selectively undoing one or more operations from the back of the history correctly returns the document to the version right before those operations were originally applied. This approach identified some non-trivial bugs in the selective undo algorithm, such as the ones explained in Section 5.1.4, where "**`y`**" and then "**`xz`**" were deleted from the original text "**`xyz`**". In the early version of the selective undo algorithm, selectively undoing the two deletion operations resulted in "**`yxz`**" instead of "**`xyz`**". After identifying this problem using the automated random testing approach, I could fix it by adding more metadata (the relative offset) to the dynamic segments to correctly perform selective undo in all situations.

5.3.2. *High-Level Architecture*

All the mechanisms mentioned above do not assume any knowledge about the granularity of the recorded changes. AZURITE uses the fine-grained editing changes as recorded by FLUORITE (Chapter 3), but in theory, the same approach could be used with more coarse-grained changes as well. For instance, the same set of visualizations could be used with the logs provided by version control systems, or the Eclipse local history. The high-level architecture of the entire system is shown in Figure 5-7.

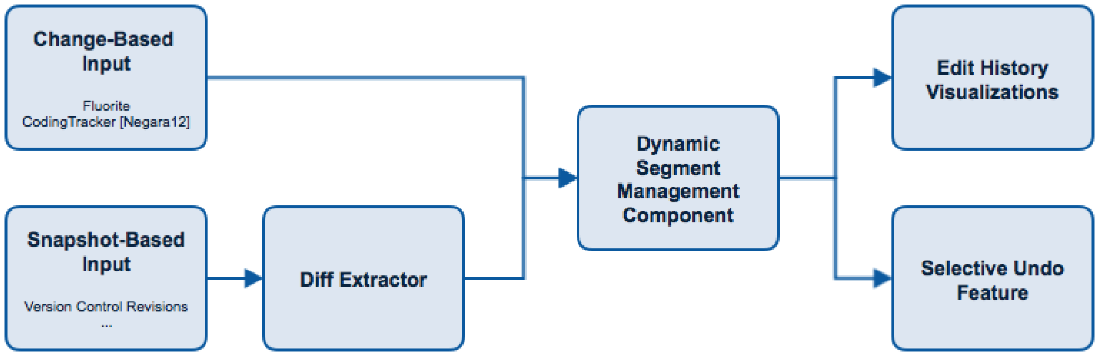


Figure 5-7. The high-level architecture of the selective undo system.

5.3.3. Granularity of Edit Operations

Having different approaches for merging / dividing edit operations might affect the usability of a selective undo tool. On the one hand, if each character-level edit is logged individually, the tool could be less usable because all of the individual operations would have to be tediously identified and selected character by character. AZURITE combines consecutive character typing or deletion operations performed within 2 seconds (configurable), similar to the way typical text/code editors do.[32] On the other hand, if the operations are too coarse-grained, then there would be many more regional conflicts among the operations. This would also give less control to the users, which was a real problem observed during the pilot for the evaluation user study (Section 9.2). Based on our observation that users wanted to control the undo range at least at the line level, we decided to prevent a single operation from spanning across multiple lines. For example, when the user types multiple lines or pastes a large block, AZURITE divides the operation into multiple insert operations, each having one line of code. This approach worked very well during the actual user study (see Section 9.2).

5.4. Conclusion

This chapter explained the selective undo mechanism for code editors and the issue of regional conflicts of edit operations, independent from the user interface aspects. In order to provide selective undo, the system must keep the dynamic segment information up to date and remember the regional conflict relationships among the operations. Then, selective undo can be performed using this information, based on the operations that the user selected.

The following chapters will present the user interfaces specifically designed for selective undo, which are implemented in our prototype tool AZURITE.

[32] To be more precise, FLUORITE does the merging and then forwards the merged operation to AZURITE (Section 3.3.3).

6.

TIMELINE VISUALIZATION OF CODE EDITS[33]

In the previous chapter, the selective undo mechanism was presented, addressing the challenge of possible conflicts among edit operations. Another major challenge of providing selective undo for code editors is that it is difficult to provide intuitive and usable user interfaces for the user to *find* and *select* what to undo. Many existing tools have explored ways to present the edit history and allow users to review or interact with the history and invoke useful commands on the past operations. However, these are inappropriate for representing fine-grained code edits, because of the issues described below.

For example, many existing selective undo user interfaces for graphical applications present a list of edit operations performed in the past along with human-readable descriptions of individual operations [Berlage 1994][Myers 1996][Myers 1998]. Graphical applications can also use a thumbnail to represent a snapshot of the graphics at a certain point of time, which makes it easier to present the edit history to the user. We used this approach in our selective undo prototype in a painting application (Chapter 10), and many existing graphical editors did the same [Kurlander 1988][Klemmer 2002][Terry 2004][Kurlander 1990][Chii 1998].

Another set of tools presenting edit history are version control systems (VCSs). Programmers use VCSs such as Subversion and Git to keep the history of how the source code has evolved over time. Programmers manually commit each changeset consisting of a set of changes along with human-readable comments describing the changes. Having these software evolution histories is useful for many purposes. First, programmers can better understand the source code by looking at the evolution histories (e.g., [Lanza 2001][Hindle 2007]). This can be useful when reviewing code changes or before modifying any existing codebase written by others. Second, programmers can execute many commands on each changeset (or revision) of the software code. For instance, when some recent changes are discovered to be wrong, then the entire project can be easily reverted to one of the previous revisions that was correctly working. Another example operation would be merging a changeset made in one branch into another branch, for example from a programmer experimenting with different implementations or from different programmers working independently. Finally, the histories are not only useful for the programmers, but are also useful for the researchers who are interested in how software has developed over time. Mining software repositories [Kagdi 2007] is known to be an effective research method and there is even a whole conference on this topic.

[33] Portions of this chapter appeared in [Yoon 2013] and [Yoon 2015]

However, these approaches are not suitable for providing selective undo with fine-grained code edit histories. Text editing operations are much more fine-grained than graphical editing, so it is hard for the users to interpret the high level edit intent just by looking at the individual text edits. Also, a thumbnail of a piece of a large text file does not give much information to the users. Finally, there is a potential problem of information overload; developers make a huge number of low-level changes while editing source code. Without proper visualization and filtering mechanisms, it would be difficult for programmers to focus on the information they need. This becomes a basic requirement for visualizations and to enable richer editing commands which would be executed on the past changes, such as various forms of searching, undo, and redo.

In summary, there are two main challenges of providing selective undo UIs in code:

- How can the users see the history and select past operations?
- How can the users effectively find the desired edits to be undone?

This chapter presents a novel *timeline visualization* of the fine-grained code edit history, with the specific focus on making it easier to perform selective undo and addressing the first challenge mentioned above. How the second challenge was addressed will be discussed later in Chapter 8.

The timeline visualization is the most basic user interface where the users can see and interact with the code edit history (Figure 6-1).[34] The figure shows how the timeline visualization has been improved, and the explanations in this chapter refer to the most recent version (Figure 6-1a) unless otherwise specified.

6.1. File Rows and Edit Operation Rectangles

Unlike most other tools supporting selective undo that display the edit history in a linear list [Maruyama 2012][Hattori 2011], here the edit history is displayed in a two-dimensional space, in order to provide more contextual information of the history while still using only limited screen space. The horizontal axis represents time, and the time labels are shown along the x-axis. Each row contains the edit history of one source file. Individual edit operations are represented with rectangles. Each rectangle is color-coded according to the type of edit: inserts are green, deletes are red, and replacements are blue. The timeline displays these three primitive edit types because all editing operations that change the code result in one of these three types, and to minimize the information overload as much as possible.

Whenever the user makes a new edit to a file, a new rectangle immediately appears at the right end of the timeline view representing that edit. The horizontal location (x position) and width of a rectangle represents the time and duration of the edit performed. The vertical location (y position) and height of a rectangle within the row represents the relative location of the edit

[34] The initial version of the timeline (Figure 6-1c) was developed by Sebon Koo.

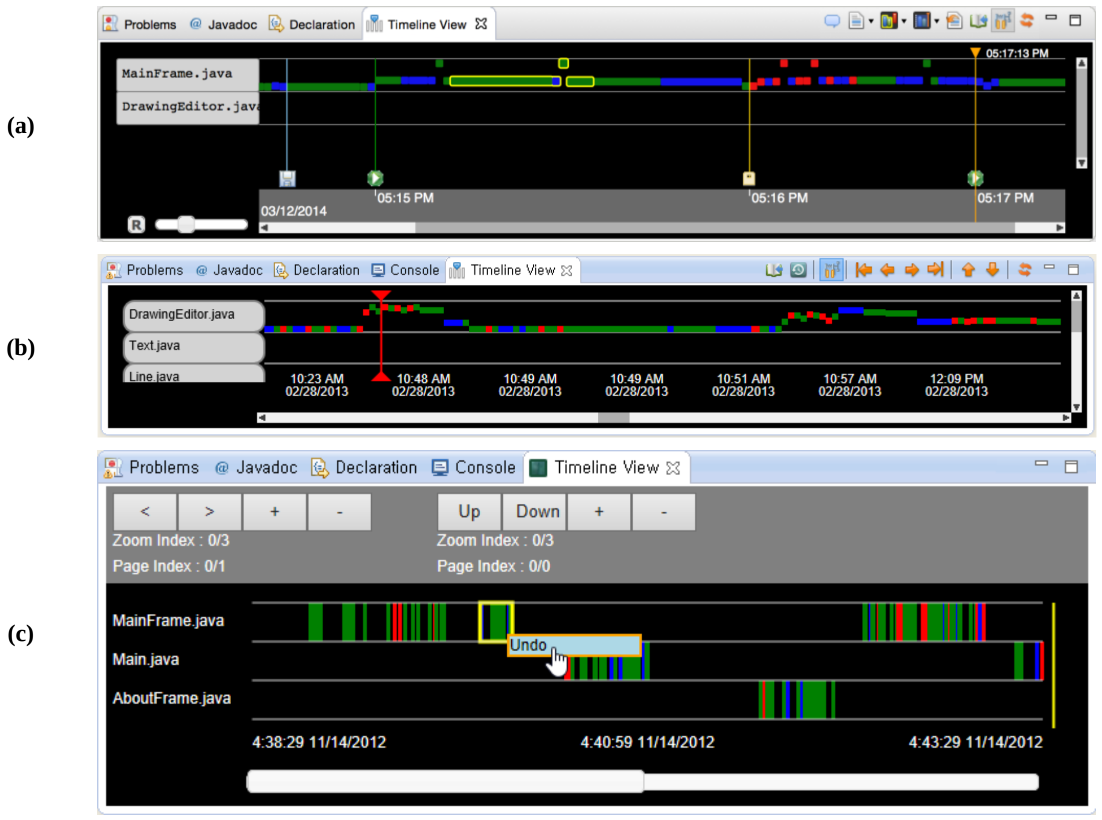


Figure 6-1. Different versions of the timeline visualization shown from the most recent version (a) to the oldest version (c). The design has been improved iteratively based on the user feedback and the changes are discussed in this chapter.

within the file. Originally, the y position and height did not have any meanings in the earlier design (Figure 6-1c), but they were added to help users to identify edits performed in different locations of a single source file.

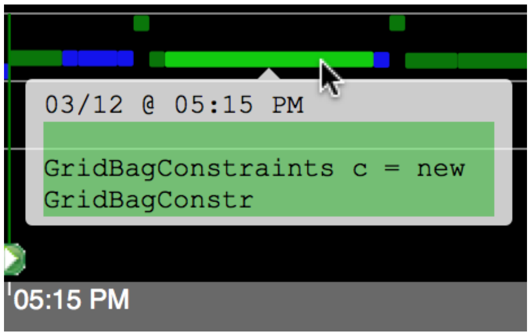
The size of the last added rectangle is updated in real time as the user is typing (or deleting) multiple characters in a row, as they are merged into a single edit event by FLUORITE (Section 3.3.3). However, once the edit operation is *finalized* (that is, when it becomes no longer available to merge with following edits), these dimensions are fixed at that point and no longer will be updated, even when the corresponding dynamic segments are updated later. There were two reasons for this decision. First, dynamically updating all the rectangles in the timeline would slow down the IDE, which is not desirable. Second, it is much more difficult to represent all the dynamic segments because they can be split or even deleted by later edit operations, which is difficult to represent. Therefore, the size of a rectangle is not affected by any later operations, and it always represents the properties it had when the edit was originally performed.

In addition, whenever a new edit is performed in the code editor, the edited file moves to the top row automatically, which enables the user to quickly recognize the most recently edited files by reading the file names from top to bottom. Currently, the rows cannot be reordered manually, but a drag & drop interface could be added.

There is a minimum width and height of a rectangle so that users can easily identify and select even small edits. The timeline is arbitrarily zoomable and scrollable both horizontally and vertically by using shortcuts (Ctrl + Mouse Scroll for horizontal zooming, and Shift+Ctrl + Mouse Scroll for vertical zooming) or menu items, so that the user can see all the files and the entire history of all edits, or the specific details of one editing session. The horizontal zoom scale can also be modified using the zoom slider control (shown at the bottom-left corner of Figure 6-1a, next to the "R" button), in which case the zoom scale can only be set between 0.1x and 3.0x.

Note that unlike the undo stack, the edit history contains *all* the edits that have ever been performed, in chronological order. Any undo operations (including both regular linear undo/redo and selective undo) are added on to the end of the timeline, just like any other operation, and the operation which was undone is still kept in the visualization. This makes it possible to see all previous operations and states of the files.

More detailed information of each edit is available as a tooltip which is shown on mouse hover. The tooltip (Figure 6-2) contains the exact time when the edit was made, and the text that was inserted and/or deleted by that edit.


03/12 @ 05:15 PM

GridBagConstraints c = new GridBagConstr

Figure 6-2. An example tooltip. The timestamp is shown at the top. The inserted code is shown in the light-green box. For a delete operation, the deleted code will appear in a pink box instead. In case of a replacement operation, both boxes appear to indicate the deleted / inserted code.

In the timeline, users can control which files are shown using various filtering options, which can be invoked by right-clicking one of the file labels at the left of the timeline. Currently, AZURITE provides four file filtering options: (1) show only this file, (2) show all files in the same project, (3) show all files edited during a specified time period (see Section 6.5), and (4) unhide all files. The "unhide all files" button is displayed in the timeline only when there are one or more files that are currently hidden.

6.2. CODING EVENTS DISPLAYED ALONG THE TIMELINE

One of the common ways of backtracking is to go back to a certain point in the past when a specific event happened. As Kent Beck says in his book, "it would be great if the programming environment helped me with this, working as a checkpoint for the code every time all of the tests run" [Beck 2002]. To support this, AZURITE detects significant coding events and displays them in the timeline view.[35] Table 6-1 lists the coding events currently displayed in the timeline.

[35] More precisely, FLUORITE captures all the Eclipse commands and forwards those to AZURITE, which specifies which of those commands should be displayed in the timeline.

Icon	Description
✓	Successful JUnit test run
✗	Failed JUnit test run
▶	Application run or debug
💾	File save (disabled by default)
⇄	Version control commands, such as Commit
■	User-defined tag

Table 6-1. List of significant coding events displayed in the timeline view

An event is displayed on the timeline as a vertical line with an icon representing that event at the bottom (Figure 6-1a). Further event types can be trivially added in the future. Users can also “tag” the current or any previous point in time, which was one of the most requested features from the field trial (Section 9.1). The tag also shows an icon (Figure 6-1a at 5:16PM), which can be named (shown on mouse hover), or stay anonymous. Users can left-click on any icon to move the orange time marker to that point to see how the code looked then. Right-clicking any of the icons shows a context menu providing useful commands such as “Undo All Files to This Point,” which can be viewed as a lightweight, automatic versioning feature. The context menu items will be explained in Section 6.5.

Any of these pre-defined set of event icons can be turned on or off in the user preference page. For example, file save icon is turned off by default, because many programmers tend to save source files very frequently, in which case the timeline could become messy with all the file save lines and icons.

6.3. Layout Modes

The timeline visualization supports two layout options: *real-time mode* and *compact mode*. In real-time mode, the rectangles are horizontally located proportionally to the actual time that they were made. This is a trivial option in terms of implementation and was the only option in the early version (see Figure 6-1c), but it turned out there is a significant problem with this approach. There are many gaps between the changes because programmers use only about 20% of their time actually editing code [Ko 2005a], which makes it difficult to navigate through the edit history in the timeline.

To resolve this problem, the timeline visualization provides a compact mode, which is used by default in AZURITE (Figure 6-1a-b). In compact mode, all the horizontal gaps between rectangles are removed so that times when the user is not editing are not displayed, and all the edits are shown contiguously. This mode is better for handling longer histories, since it dramatically reduces the need for horizontal scrolling.

In contrast, real-time mode could be better for short histories because users can better reconstruct their previous working context, for example by seeing the size of the gaps and the grouping of edits temporally. Users can switch between the two modes at any time using a menu command or toolbar icon.

When the compact mode layout was first introduced (Figure 6-1b), some problems were discovered with the time indicators shown below the file rows, which were not very evident in the real-time mode. Since the horizontal axis became non-linear in the compact mode, it was unclear as to which side of the time indicator each label is pointing to. In addition, when scrolling the timeline horizontally, the time indicators stayed in the same locations and only the numbers were being changed, which was also confusing.

Inspired by various timeline user interfaces of video editing software, the time indicators were redesigned to tackle these problems (Figure 6-1a). Each time indicator has a small time tick on the left side, showing where exactly it is pointing to. The time indicators are scrolled together with the timeline, which is a lot less confusing to the users. The indicators are arranged so that there are always about 4 time indicators appearing on the screen. Extra care is taken to prevent two time indicators from overlapping with each other.

6.4. Selecting Rectangles

Users can click on a rectangle to select it, or drag to select multiple rectangles at once. Additional rectangles can be toggled in the selection using the control key. The current selection is highlighted with yellow outlines (Figure 6-1a). Note that, unlike regular text or code editors, disconnected sections of the timeline can be selected. Once some of the operations are selected, the user can invoke a popup context menu by right-clicking.

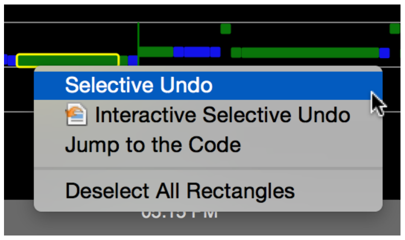


Figure 6-3. Context menu for the selected rectangles. Users can invoke various commands, such as "Selective Undo". The third command, "Jump to the Code" appears only if a single rectangle is selected.

The first command in the menu is "Selective Undo" which was extensively discussed in Chapter 5. This command undoes only the edit operations corresponding to the selected rectangles while keeping the other changes unaffected. Note that AZURITE allows rectangles to be selected across multiple files (i.e., multiple rows in the timeline) and undone together, which is a significant advantage over conventional undo which only works on a per-file basis. Note that all the inverse operations added by the "Selective Undo" command are put into the timeline like any other operations, so users can easily change their mind and undo them.

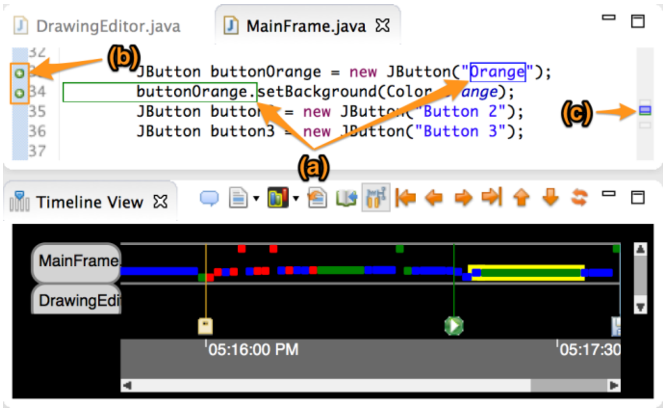
The "Interactive Selective Undo" command allows users to interactively select which operations to be undone while seeing a preview of the selective undo result. This will be explained in more detail in Section 8.4.

The "Jump to the Code" command appears only when there is exactly one selected operation. This command opens the source file corresponding to the selected operation in the code editor and moves the cursor to the location where the operation was performed in the current code. The same command can be invoked by double-clicking a rectangle in the timeline. This is implemented by moving the cursor to the current dynamic offset of the first dynamic segment attached to the selected operation (Section 5.1).

Finally, the "Deselect All Rectangles" command is used for clearing the current rectangle selection. When users want to clear the current selection, this command should be explicitly invoked, since clicking on an empty space in the timeline does not discard the selection, as in some other graphical applications such as Photoshop. This was a deliberate decision to prevent the users from accidentally losing the selection.

There are multiple ways of selecting rectangles in the timeline, other than just manually selecting them using the mouse. For example, users can select some region of code, and then invoke the "Select Corresponding Rectangles" command, which searches through the history for all the edit operations performed on the currently selected region of code. Users can also search for specific code or text in the code through the history, which is called "History Search" (see Section 8.3). The resulting rectangles are selected on the timeline for both of these commands.

Another requested addition is that when one or more rectangles are selected in the timeline, the corresponding areas of code are highlighted in the code editor (Figure 6-4). This mitigates the users' reported problem that it can be difficult to mentally associate the rectangles with the code. The highlights are shown only if the corresponding files are currently open in the editor, and just selecting the rectangles from the timeline does not automatically open the files. When a file with some selected rectangles is later opened, the highlights are immediately shown in the file. Due to the problem of Eclipse where it gets slower when many of the boxes are added or removed in real time, this feature is turned off by default, but can easily be turned on from the preference dialog of AZURITE, whenever the user wants.



```
JButton buttonOrange = new JButton("Orange");
buttonOrange.setBackground(Color.orange);
JButton button2 = new JButton("Button 2");
JButton button3 = new JButton("Button 3");
```

Figure 6-4. The code corresponding to the selected rectangles (with yellow outlines) in the timeline are indicated by (a) the boxes in the code editor, (b) the small icons on the left ruler, and (c) the markers on the scrollbar on the right side. The colors of the boxes match the rectangle colors in the timeline.

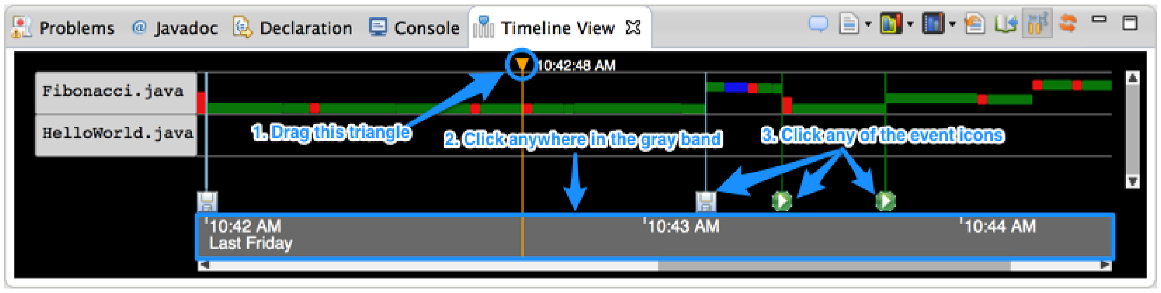
6.5. Selecting Times or Time Ranges

Besides selecting rectangles, users can also select a *point in time* in the history, using the vertical orange marker in the timeline (Figure 6-5). The marker can only be positioned between any two consecutive rectangles, and it can be moved by either (1) dragging the triangle at the top, (2) clicking or dragging within the gray time band area, or (3) clicking any of the event icons. Right-clicking the marker shows a context menu specific to the time marker.

The "Tag This Point" command is used to tag the currently selected time (i.e., where the marker is currently pointing at) for future reference. "Undo All Files to This Point" is a convenient command to revert all the files that has been edited since the selected time, which is essentially a shortcut for selecting all the rectangles after the selected time and invoking selective undo. The next command in the menu launches the interactive selective undo dialog (Section 8.4) rather than performing selective undo directly. The last four commands can be used for selecting / deselecting rectangles on either side of the selected time.

It is also possible to select a *time range* in the timeline by first selecting the start time with the normal time selection method above, and then selecting the end time while holding the Shift key while clicking in the gray time band. An example screenshot is shown in Figure 6-6.

The first four context menu items for a selected time range are used for selecting / deselecting rectangles either inside or outside the selected time range. For example, users can select a range of time, select all rectangles inside the range, and invoke selective undo to revert all the changes made in that time period. Conversely, users can also selectively undo everything except for the edit operations within the time period.



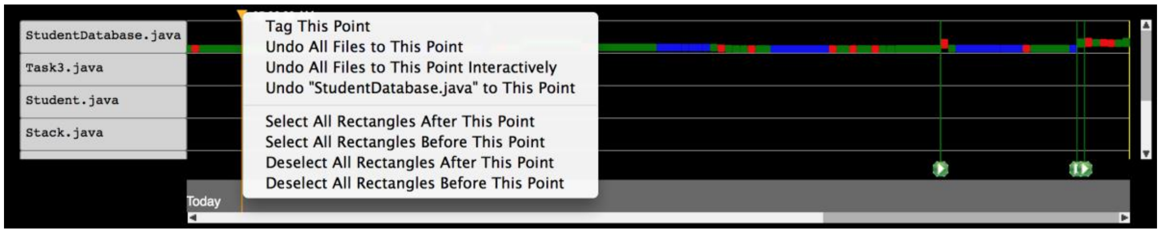

StudentDatabase.java
Task3.java
Student.java
Stack.java
Today
Tag This Point
Undo All Files to This Point
Undo All Files to This Point Interactively
Undo "StudentDatabase.java" to This Point
Select All Rectangles After This Point
Select All Rectangles Before This Point
Deselect All Rectangles After This Point
Deselect All Rectangles Before This Point

Figure 6-5. The time selection marker, which is the orange vertical bar with a triangle-shaped handle attached to the top. Right-clicking the marker brings up a context menu with various commands.

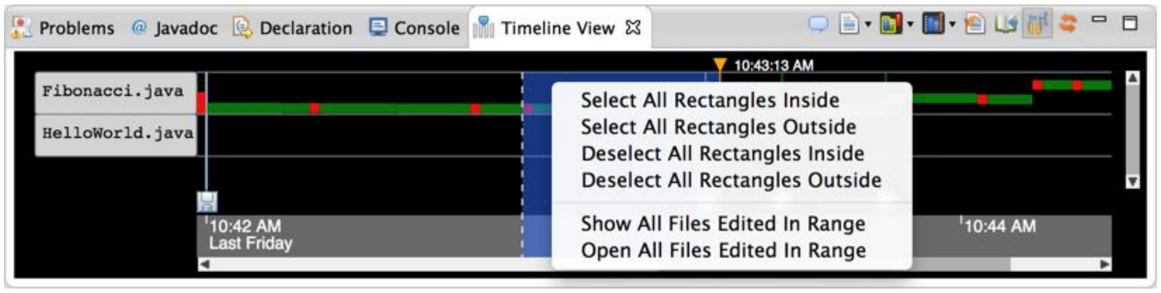

Problems
Javadoc
Declaration
Console
Timeline View
10:43:13 AM
Fibonacci.java
HelloWorld.java
Select All Rectangles Inside
Select All Rectangles Outside
Deselect All Rectangles Inside
Deselect All Rectangles Outside
Show All Files Edited In Range
Open All Files Edited In Range
10:42 AM
Last Friday
10:44 AM

Figure 6-6. An example screenshot of a time range selection. The start time is indicated as white, dotted vertical line, and the end time is indicated with the same time marker used for time selection.

The "Show All Files Edited in Range" command is a file filtering command (Section 6.1), which would only show the source files that were edited in the selected time range while hiding all the other files. Finally, "Open All Files Edited in Range" command opens all the source files in the range, as the name suggests.

6.6. IDE-Independent Implementation of the Timeline

AZURITE'S timeline visualization is written in HTML-5 / CSS / JavaScript, and it communicates with the backend through the embedded browser interface of Eclipse. This approach was previously used for another research prototype tool that I was part of ([Omar 2012], Figure 6-7), and it has some advantages over using IDE-specific APIs such as SWT and JFace.

First, since web development is very popular, the visualization toolkits for the web tend to be more mature than the IDE-specific toolkits. Since the timeline visualization needed to be capable of displaying fairly complex information, going in this direction was a reasonable choice. The drawing part of the timeline is written using the Scalable Vector Graphics (SVG) standard[36] and D3.js,[37] which made it a lot easier to implement the zooming and scrolling features because SVG supports two dimensional affine transformations.

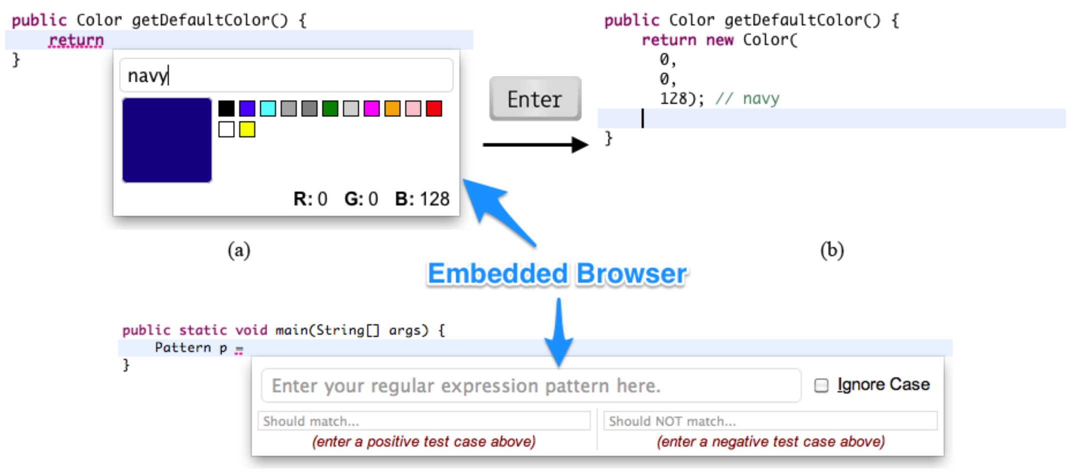
Second, the HTML-based user interface parts can mostly be developed in a stand-alone web browser. Modern web browsers provide a rich set of web development tools, which are very useful for UI development. In addition, the entire plug-in does not need to be compiled or launched when the UI is being updated, which makes the implement-test cycle much shorter.

Finally, using HTML technologies makes the user interface reusable across multiple IDEs, because it does not have any dependencies on the IDE-specific APIs.

The high-level view of the relationships between the IDE, the AZURITE tool, and the timeline visualization is depicted in Figure 6-9. The HTML-based UI part was developed independently from the plug-in API provided by the IDEs. The plug-in side of the tool is the part where the core

[36] http://www.w3.org/TR/2011/REC-SVG11-20110816/

[37] http://d3js.org/



```
public Color getDefaultColor() {
    return
}
```

```
public Color getDefaultColor() {
    return new Color(
        0,
        0,
        128); // navy
}
```

```
public static void main(String[] args) {
    Pattern p =
}
```

Enter your regular expression pattern here.

Should match...
(enter a positive test case above)

Should NOT match...
(enter a negative test case above)

Figure 6-7. Embedded browser control used in the Graphite project [Omar 2012]. The color palette and the regular expression pattern builder were implemented using standard web technologies and then embedded into the Eclipse Code editor using the `Browser` control in SWT.

business logic of the tool and all the IDE-specific UIs reside. Also in the plug-in side is an embedded browser widget (e.g., `Browser` in the SWT library), that works as a mediator between the HTML-based UI and the plug-in.

As shown in Figure 6-9, the plug-in needs to be able to communicate with the HTML-based UI via the embedded browser control in both directions. Therefore, the embedded browser control should support at least the following two features:

- invoking JavaScript functions on the HTML-based UI from the plug-in side
- being notified in the plug-in side code when some event happens from the HTML side

Most existing embedded browser controls available in the APIs provided by the IDEs or the main programming language used by the IDEs meet these two conditions, including the Eclipse, Visual Studio, and NetBeans IDEs. As a proof of concept, the timeline visualization UI has been successfully tested on Visual Studio, independently from the selective undo core logic, and a screenshot is shown in Figure 6-8. All the features implemented within the visualization itself were fully functional, including the zooming, scrolling, and various selections.

This approach has some disadvantages as well. First, some core functionalities might have to be implemented in both sides (plug-in side and the UI side) in two different languages. This problem can be mitigated by moving the functionalities to the UI side as much as possible. Another problem is that the look-and-feel of the user interfaces in the embedded browser are different from the other user interfaces of the enclosing IDE. Moreover, there could be cross-browser compatibility issue as well, because different IDEs use different web rendering engines internally. This problem could be reduced by having a separate cascading style sheets (CSS) for each IDE and operating system combination to mimic the native look-and-feel of the IDE as much as possible.

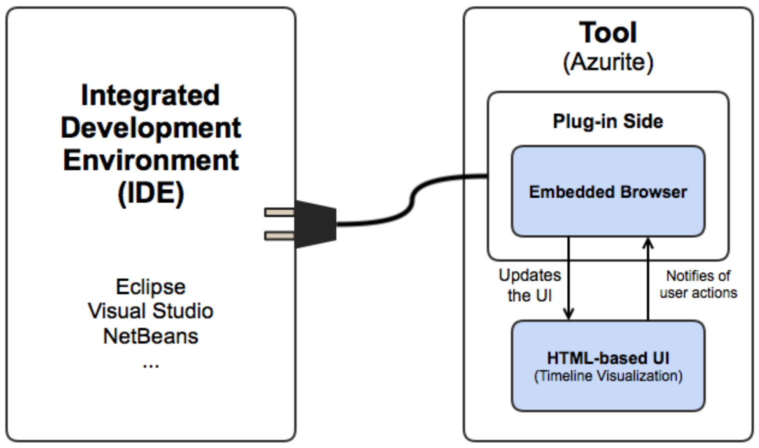


Figure 6-9. High-level architecture of the HTML-based user interface in an IDE plug-in.

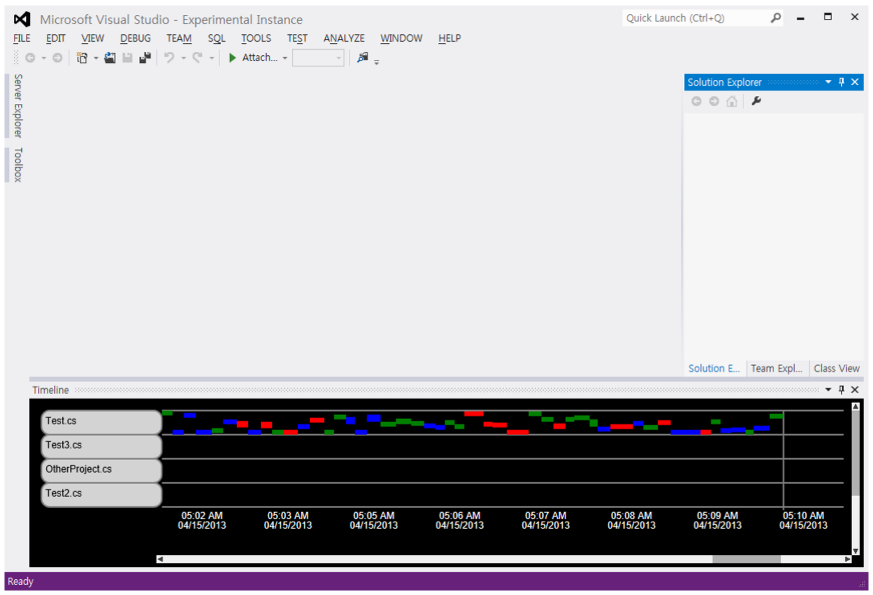


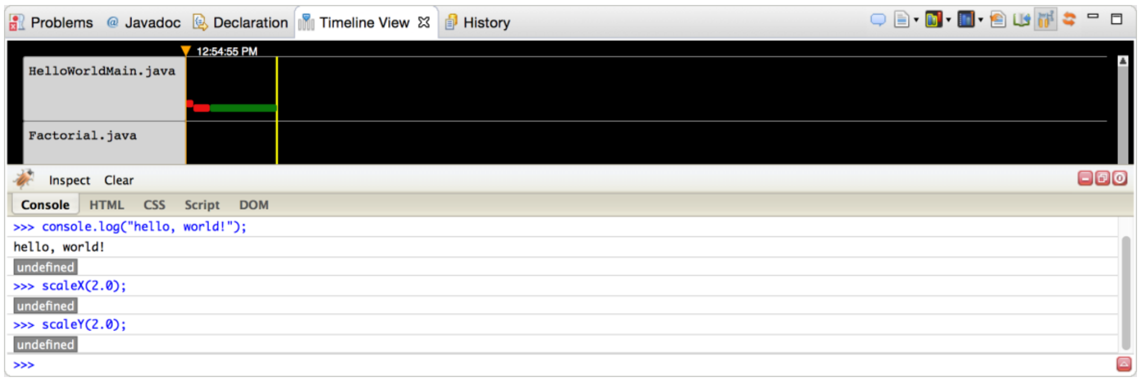
Figure 6-8. Timeline visualization of AZURITE loaded in Microsoft Visual Studio 2012.

Another problem is that the UIs implemented within the embedded browser cannot display any contents outside the boundary of the browser control. For example, this was an important issue when implementing the popup context menus, because the JavaScript-based context menu could be larger than the browser boundaries, which often resulted in cut off context menu items. To address this problem, we needed to re-implement the context menu by having the timeline visualization send a message to the plug-in side indicating that the right mouse button was clicked in the timeline, and then showing the context-menu on the current mouse cursor position using the context menu APIs of SWT.

Debugging the UI while it is running within the plug-in could be a challenge, because the embedded browsers do not support the development tools provided by stand-alone browsers.

Fortunately, there exist portable web development toolkits such as Firebug Lite[38] that can be embedded directly into the UI, as shown in Figure 6-10. Any JavaScript code can be executed in the developer console.

Finally, running heavy computation on the JavaScript side may block the entire UI thread of the IDE until the computation is finished, in case the embedded browser control does not support executing JavaScript code from the plug-in side asynchronously. Therefore, it is desirable to carefully design the JavaScript functions in a way that each function does not take too long and any significant computation work is divided in smaller pieces and a progress bar is shown when needed.



```
>>> console.log("hello, world!");
hello, world!
undefined
>>> scaleX(2.0);
undefined
>>> scaleY(2.0);
undefined
>>>
```

Figure 6-10. Firebug Lite loaded within the Eclipse IDE. The developer console is fully functional, and the DOM elements can be navigated within this UI.

6.7. DISCUSSION

6.7.1. LINEAR VS. TREE/GRAPH BASED HISTORY

One might raise the question why AZURITE does not keep the edit history as a tree or graph instead of as a linear list when the user undoes multiple steps and makes a new edit, as in US&R [Vitter 1984] or Git. The linear history model was chosen intentionally, where all the code changes (including the undo commands themselves) are added to the end of the timeline, for two reasons. First, although several text editors and plug-ins have provided tree-structured visualizations that allow users to move among different nodes [Losh 2012][Cubitt 2010], it is difficult to understand the tree as the history gets bigger. This is because the nodes do not provide useful information for the user to navigate the tree, which is why we believe these have not caught on in popularity. In contrast, the linear history model and timeline visualization of AZURITE would match programmers' episodic memory [Parnin 2012] and have been shown to be understandable in our studies (see Section 9).

[38] https://getfirebug.com/firebuglite

Second, it is not trivial to represent a selective undo operation in a tree-structured history. Unlike the regular undo, selectively undoing some changes does not result in one of the previously visited nodes in the history tree. Rather, it creates a *new* node that has never existed before. Also, a graph-based history as provided in Git would be inappropriate because a selective undo operation is not used for merging.

6.7.2. Limitations

There are some known limitations in the current version of the timeline visualization. When multiple types of coding events happen in sequence without any code changes in between, those event icons are displayed on top of each other in a way that only the most recently executed command may be seen while the other events are hidden behind it. For example, when the source code is first tested by running some JUnit tests and then committed to the version control repository, the JUnit test icon would not be shown to the user because the commit event icon will be on top of it. This could be problematic when the user is trying to backtrack to the last time when all the unit tests passed, for example. Furthermore, since no code edits happen in between, zooming in would not help separate the icons in compact mode.

In compact mode, the horizontal time gaps are completely removed, which means some contextual information is being lost. For example, a user might want to undo every code changes made after the lunch break. While the user could switch the timeline to the real-time mode to find the big gap around the lunch time, it would be more convenient if the size of the gaps were indicated even in the compact mode using techniques such as a “broken” axis or taking logarithm values of the gap sizes to differentiate relatively large gaps from the smaller ones.

6.8. Conclusion

Despite the recent trends to exploit more fine-grained code editing histories, their use in existing tools has mostly been limited to replaying the history, or analyzing the data for research purposes. This chapter demonstrated that these fine-grained histories can also be directly useful for the programmers with proper visualization and editor commands such as selective undo, which are tightly integrated with the history. In the future, this approach could also be applied to regular text editors (see Section 11.4).

Although the timeline visualization was shown to be useful in many situations in several user studies (Sections 9.1 & 9.2), users agreed that the timeline visualization would be more useful if the edits could be somehow collapsed to be shown at a higher-level. In the next chapter, a real-time edit collapsing algorithm will be introduced, and integrated with the timeline visualization to provide a “semantic zooming” technique.

7.

REAL-TIME EDIT COLLAPSING AND SEMANTIC ZOOMING

The timeline visualization can be zoomed in and out arbitrarily in both directions. In the current version of AZURITE, the timeline visualization also supports automatic edit operation collapsing and semantic zooming, which are the techniques to dynamically adjust the level of detail presented in the timeline depending on the current horizontal zoom scale. In earlier versions without semantic zooming, it was difficult to see the bigger picture of the recent code edits by looking at the timeline, even when the timeline visualization was zoomed out significantly, because the individual edits were still too fine-grained. In addition, the AZURITE users from the field trial (Section 9.1) and the evaluation study (Section 9.2) mentioned that they often wanted to see the code changes at a higher-level, such as the level of adding a field, editing an existing method, and so on. Therefore, the real-time edit collapsing and semantic zooming features were added to AZURITE, as described in this chapter.

There are existing tools that summarize the code changes when two different snapshots of code (often from the version control system) are given (e.g., [Ren 2004][Kim 2009]). However, these techniques could not be directly applied to AZURITE for a number of reasons. First, the existing techniques use a top-down approach: they extract conceptual edits from the complete snapshots. However, AZURITE needs to use a bottom-up approach: it needs to collapse and summarize multiple, fine-grained edit operations into a more meaningful, conceptual edit. Another aspect is that the edits should be summarized and displayed in the timeline in *real-time*, as the user actively edits the code. This means that the algorithm should be capable of handling the *stream of edits* in an efficient, incremental manner.[39]

The semantic zooming (see Section 7.3.1) feature was inspired by the Semantic Zoom Multi View feature of SATIN [Hong 2000], where multiple views are defined for different zoom levels, and the appropriate view is automatically chosen based on the current zoom level. To implement similar semantic zooming in the timeline visualization, our edit collapsing mechanism identifies what kinds of code changes the user is making in the code editor, and uses that information to determine whether to collapse multiple edits at a certain collapse level (see Section 7.2.2). The different kinds of code changes presented in Table 7-1 are similar to

[39] Note that the features described in this chapter are only relevant when the input data are fine-grained edit operations made in the code editor (see Figure 5-7).

the categories of atomic code changes that appear in [Ren 2004], but there are several differences. First, in their atomic change model, adding and deleting changes always represents adding or deleting an empty element. For example, suppose that a programmer adds a new method **`foo`** and then fills in the body of the method. In Ren's model, this set of changes would be represented as two consecutive atomic changes: an Add Method change that adds the empty method **`foo`**, and a Change Method change for filling in the body. However, in our model, the same edits can be represented as a single Add Method change that contains the body as well. Another big difference is that our classification can represent special cases such as Non-code changes (NCC) or Unknown (UKN). By representing these additional cases in the history, they can also be selected and undone by the user. For example, a code reformatting edit (NCC) could later be selectively undone.

Others have developed ways to extract high-level code changes from two versions of code, using AST tree differencing [Fluri 2007][Neamtiu 2005]. One of the biggest challenges of AST differencing is the problem of matching code elements between the two versions [Kim 2005b][Kim 2006]. For example, when a method is renamed between the two versions, the change extraction algorithm can only *guess* that there was a renaming edit, using various heuristics. This happens because the change extraction algorithm does not know about the actual edit operations performed in the editor. In contrast, the change extraction algorithm used in AZURITE uses the actual edit operations (Figure 7-5), which makes it possible to always correctly match the code elements, even when there are renaming operations.

Our real-time edit collapsing algorithm is described in the following sections. To better illustrate the algorithm and our solution, a sample programming task and a code editing script for performing the task will be presented first, and then the explanations will refer to the script and show how the example edits are collapsed. Suppose that a programmer wants to write a program for calculating factorial numbers. For example, she might implement the factorial calculator by taking the following steps in order:

(a) Write **`factorial`** method using a **`for`** loop (Figure 7-1a)
(b) Test the function with a constant value (Figure 7-1b)
(c) Change the **`factorial`** method to use recursion instead (Figure 7-1c)
(d) Modify the **`main`** method to get user input from the console (Figure 7-1d)

Figure 7-2 shows the state of the timeline visualization after completing each of these steps. The timeline is partitioned into four sections, each corresponding to a programming step described above. The blue partitions are not actually shown in the timeline but were added on the screenshot for the purpose of explanation.

In this example, because the programmer was alternating between the **`factorial`** method and the **`main`** method, these different sections of programming are somewhat distinguishable by the vertical locations of the consecutive rectangles. Thus, if the programmer wants to selectively undo the changes made in step (c), she could visually distinguish the groups of changes and select all the rectangles in the section (c), and then invoke selective undo.

```java
public class Factorial {
    public static void main(String[] args) {
    }

    public static int factorial(int n) {
        int result = 1;
        for (int i = 2; i <= n; ++i) {
            result *= i;
        }
        return result;
    }
}
```

(a)

```java
public class Factorial {
    public static void main(String[] args) {
        System.out.println( factorial(5) );
    }

    public static int factorial(int n) {
        int result = 1;
        for (int i = 2; i <= n; ++i) {
            result *= i;
        }
        return result;
    }
}
```

(b)

```java
public class Factorial {
    public static void main(String[] args) {
        System.out.println( factorial(5) );
    }

    public static int factorial(int n) {
        if (n <= 1) { return 1; }
        return n * factorial(n - 1);
    }
}
```

(c)

```java
public class Factorial {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        System.out.println( factorial(n) );
        in.close();
    }

    public static int factorial(int n) {
        if (n <= 1) { return 1; }
        return n * factorial(n - 1);
    }
}
```

(d)

Figure 7-1. The code changes for the factorial example.

However, it can also be seen that there are relatively many rectangles shown in the timeline (the numbers in the square bracket in Figure 7-2), even for these seemingly simply programming steps. Part of the reason is that often there exists small noise edits made by small mistakes or typos, which may interfere with understanding the intent of the programmer. For example, the first red rectangle appearing in section (a) of Figure 7-2 is a deletion operation of a letter "**`l`**", which was performed because the method name was accidentally typed as **`factoriall`**, and then the programmer wanted to fix the typo by deleting the duplicated "**`l`**" at the end. Even though this is what actually happened, it would be more meaningful for the programmer to see the change with the surrounding context and know that she added a new method named **`factorial`** at that moment.

With the real-time edit collapsing algorithm, the same code edit history can be displayed at higher levels. Figure 7-3 shows how the example edit script shown above would be displayed at different collapse levels but the same zoom level. AZURITE's timeline supports a total of four

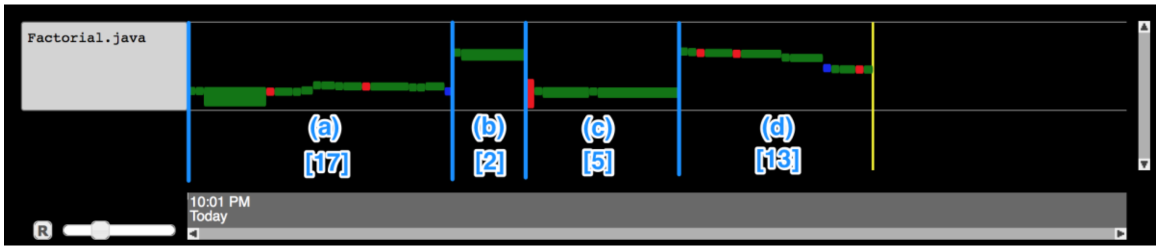


Figure 7-2. The state of the timeline visualization after completing all the four steps in the factorial example, shown at the raw level. The blue vertical separation lines were added on the screenshot for the purpose of the explanation, and are not shown in the actual timeline. The numbers in the square brackets indicate how many rectangles are in each section.

collapse levels: *raw level*, *parse level*, *method level*, and *type level*, listed from the lowest to the highest level. Note that AZURITE automatically chooses which collapse level to use based on the current horizontal zoom scale as a *semantic zooming* feature (see Section 7.3.1), but the zoom level is fixed in Figure 7-3 just for the purpose of illustration.

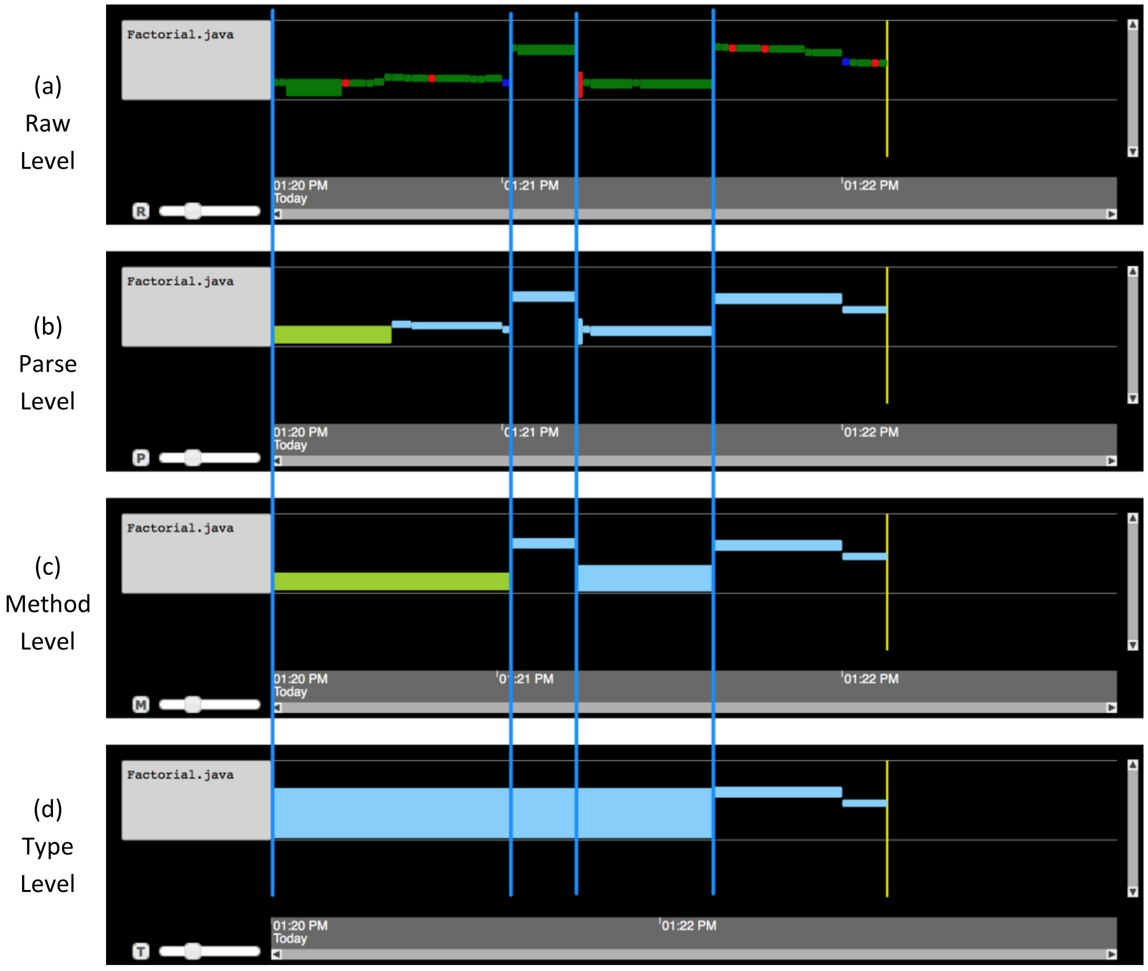


Figure 7-3. The example code edit script for the factorial program shown at different collapse levels but the same zoom level.

There are certain restrictions when collapsing the edit operations, which were established to make the collapsing algorithm compatible with the selective undo algorithm described in Chapter 5. First, collapsing is not allowed to reorder edit operations, so only the edit operations that are consecutive in time can be collapsed together. Second, edit operations that are collapsed at one level cannot be split at a higher level. That is, the collapsed edit operations at one level are considered as the building blocks of the next collapse level. Finally, the edit operations are not collapsed across user-defined tags or coding events that are displayed in the timeline (Section 6.2).

7.1. THE FOUR COLLAPSE LEVELS

In this section, each of the collapse levels – raw level, parse level, method level, and type level – are described in more detail. The edit collapsing rules for each level and the rationale behind providing each level will be explained. Then, the following section will explain the detailed collapsing algorithm.

7.1.1. RAW LEVEL (NO COLLAPSING)

The raw level (Figure 7-3a) displays the fine-grained edits as they arrive, without collapsing any of the edits. This level used to be the only option, and it was used during the evaluation studies of AZURITE (Sections 9.1 & 9.2). The rectangles shown at this level are the basic undoable units of the selective undo mechanism in AZURITE.

Although the raw level has several limitations as described above, it is still available as an option, because it can be useful in certain situations. When some code is edited and then reverted quickly, these edits are going to be collapsed at the parse level and higher and essentially considered as a no-op (i.e., identity operation). If the programmer wants to restore the first edit by selectively undoing the reverting edit, it can only be done at the raw level.

7.1.2. PARSE LEVEL

The parse level (Figure 7-3b) is the new default collapse level used in the timeline visualization. The basic idea of constructing the parse level is to collapse consecutive edit operations so that all the intermediate versions of code (i.e., the versions between any two consecutive collapsed edits) are parseable. In other words, when a new edit operation is added to the history buffer and the resulting code turns out to be unparseable, the system collapses the edit operation with the following edit operations until the source file gets to a parseable state again. One exception to this rule is that the two consecutive (in time) edits are not collapsed when their edit ranges are apart by more than one line of code, because they are most likely to be editing separate statements.

There are significant advantages of using the parse level compared to the raw level, which is the reason for making the parse level be the new default. By using the collapsing rules above, individual collapsed edits are much more comprehensible independently, because they are usually at the level of a single statement change, variable addition, an empty method stub addition, and so on. In addition, the number of edit operations appearing in the timeline is significantly reduced, which reduces the problem of information overload. Besides, most of the noise edits such as typo corrections are naturally collapsed with the surrounding edits.

The parse-level collapsing rule, however, does not work very well for certain situations. For example, when editing an existing string literal value in the code, the code is always in a parseable state, thus making the edit operations not collapsed at all, which is not desired. In order to mitigate this potential problem, the parse level collapsing rule has an exception: even when an incoming edit gets to a parseable state, the edit is collapsed with the following edit if the following edit is contiguous in location and made within a set amount of time (currently 2 seconds, configurable).

7.1.3. METHOD LEVEL

The method level (Figure 7-3c) is a collapse level that is higher than the parse level. The main idea behind the method level is to collapse all the consecutive edits made in the same method (or same class field) into a single edit.

The method level is great for discriminating the conceptual units of code edits, assuming that programmers divide the code logic into relatively small methods. For example, in the method level visualization shown in Figure 7-3c, each step of the example factorial programming matches with a single rectangle in the timeline, because the programmer was alternating between the **`factorial`** method and the **`main`** method after completing each step. At this level, she could backtrack by selecting the code changes that modified the **`factorial`** method to use recursion (the third big rectangle from the left) with a single mouse click, and then invoke the selective undo command to revert the method to its **`for`** loop version.

Note that the code changes in step (d) are still split into two rectangles even though all the changes were made in the same **`main`** method. This happens because the method level collapser needs to wait until the collapsing process is fully completed by the parse level collapser. This issue will be described in more detail in Section 7.2.1.

7.1.4. TYPE LEVEL

The highest collapse level provided in AZURITE is the type level, which is shown in Figure 7-3d. Similar to the method level, the main idea of this level is to collapse all the consecutive edits in the same type (for example, every operation in the same class, interface, or enum) into a single edit.

The rationale behind providing type level is to make it easier to review or interact with the code edit history when the programmer is working with nested types (e.g., inner classes in Java), or when working with multiple types simultaneously. A great example of such situations is when the programmer is using the State or Strategy design patterns [Gamma 1994], which are often implemented as nested classes in Java. However, the type level view may not be very useful when the programmer is working on a single class file for a long time. For example, in Figure 7-3d, the example code changes for the factorial program are all collapsed into a giant big box, which may not be useful for the purpose of selective undo.

The last two small rectangles at the end of the timeline are not yet collapsed into the previous edits for the same reason mentioned in the method level.

7.2. Collapsing Algorithm

This section describes the edit collapsing algorithm used for the different collapse levels introduced in the previous section. The raw level is excluded from the discussion in this section, because it does not require any edit collapsing.

7.2.1. Overall Collapse Mechanism

The collapsing algorithm works in each of the collapse levels separately, and the overall algorithm is the same, but only the logic to determine whether to collapse new incoming edit(s) with the previous edits is different for each level, which will be referred to as the *collapse test*. The key idea of this collapsing algorithm is to keep a list of pending edits (*pending list*, hereafter) for each level. The edit operations in the pending list are already determined to be collapsed together at that collapse level, but still pending in the sense that the following incoming edit(s) can also be collapsed with them.

Once a new edit operation is added to the history buffer, the edit operation is first considered by the parse level collapser. There can be three different outcomes when this happens. (1) If the current pending list is empty, then the incoming edit is simply added to the pending list. Alternatively. when there are some existing pending changes, then the parse level collapser runs the collapse test to determine whether the incoming edit should be collapsed with the currently pending edits or not. For the parse level, the rules described in Section 7.1.2 are used for the collapse test. (2) If the edit should be collapsed, then it is added to the end of the pending list. (3) If the edit should *not* be collapsed, then all the currently pending edits are finally marked as collapsed, the pending list is emptied, and the new incoming edit is added

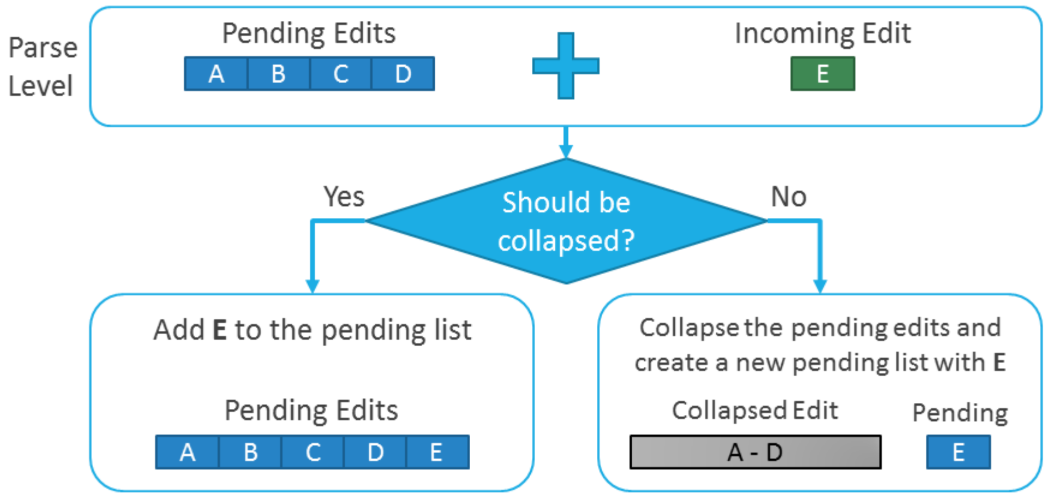


Figure 7-4. Illustration of the overall collapse mechanism for the parse level. When there is an incoming edit operation, the parse level collapser runs the collapse test to see if the new edit should be added to the pending list or if the existing pending edits should finally be marked as collapsed. The newly collapsed edit (A-D) is taken to the next level collapser as the incoming edit, and the same process is followed.

to the pending list. When this happens, the edits that were just collapsed are considered by the next level (the method level, in this case) collapser as the new incoming edits. The same process is followed by the method level collapser, except for the collapse test part, and the edits collapsed at the method level are then considered by the type level. The tests for these levels is discussed in the next section, and the whole process is illustrated in Figure 7-4.

Because the method and type level collapsers receive the incoming edits only when their immediate lower-level collapser finally marks a set of edits to be collapsed, some of the code edits at the end of the history may not appear immediately as collapsed even though they will be collapsed eventually. This problem can be shown in Figure 7-3c and Figure 7-3d. There is one more special rule that applies to the overall process. As briefly mentioned above, when one of the significant coding events such as JUnit test run occurs that would be displayed in the timeline as an event icon (Section 6.2), all the current pending edits at every collapse level are immediately marked as collapsed, and all the pending lists are emptied. In other words, this rule makes sure that the code edits are never collapsed across one of these coding events, with the assumption that the coding events could be considered as implicit checkpoints, and the programmer is likely to make code edits something conceptually different from the previous set of edits after one of these coding events.

7.2.2. Collapse Test For the Method Level and Type Level

At the method (or type) level, the basic collapse rule is to collapse the consecutive edits made in the same method or field (or type). In order to run this collapse test successfully, the collapser should first extract some detailed information about each edit, such as what kind of code change the edit entailed, and whether the edit range is bound to a certain method or a class.

The change detail extraction process is illustrated in Figure 7-5. In order to extract change details of a set of code edits, the system takes the two snapshots, before and after the edits, and the edit operations as input. Because there are two sets of edits in consideration, pending edits and the incoming edit, the collapser needs to analyze three versions of code snapshots before and after each set of edits: *initial snapshot* (before the pending edits), *intermediate snapshot* (after the pending edit but before the incoming edit), and the *final snapshot*. Each of these snapshots are first parsed into an abstract syntax tree (AST) using the **`ASTParser`** class provided in the Java Development Tools (JDT) plug-in in Eclipse. Then, the pending change details are extracted from the AST trees of the initial and intermediate snapshots and the pending edits themselves. Similarly, the incoming change details are extracted from the intermediate and final AST trees, and the incoming edit itself. There are a total of 12 kinds of edits determined by the collapsers, summarized in Table 7-1.

Because the focus was on the correctness of the algorithm for the current research prototype, and not much on the performance, there are many optimizations that could be trivially implemented. For instance, the current algorithm redundantly parses the three different snapshots whenever a new incoming edit is being processed, even though the initial and intermediate ASTs could be cached and reused. Also, the final snapshot could be incrementally parsed from the intermediate AST to enhance performance. The performance analysis of this unoptimized implementation is provided in Section 9.3.3.

Once the change kinds are determined for the pending changes and the incoming change, then the collapser uses the collapse test matrix to see if their change kinds are compatible (Table 7-2 & Table 7-3). A collapse test matrix defines whether the different kinds of changes can be collapsed at the particular collapse level. The content of each cell indicates the resulting change kind after collapsing the pending changes and the incoming change. For example at the method level (Table 7-2), an AM change followed by a CM change on the same method results in a collapsed AM change. When an AM or CM is followed by a DM change on the same method, the collapsed changed becomes a no-op, which is considered as a Non-Code Change. The same logic applies to the fields. At the type level, more kinds of changes can be collapsed than at the method level. For example, an AM change followed by another AM change can be collapsed at the type level, provided that they were added to the same enclosing type. The gray cells indicate they are not collapsible.

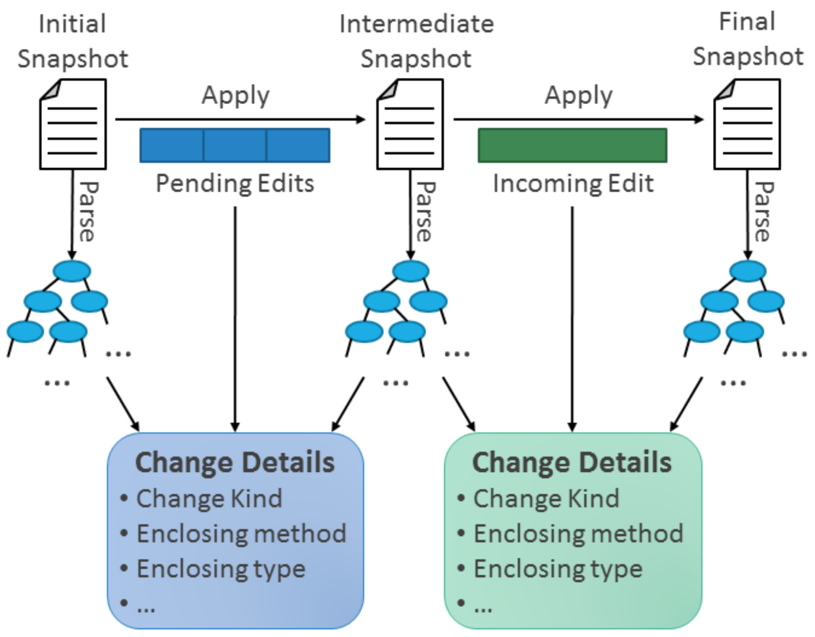


Figure 7-5. Illustration of the change detail extraction process.

Kind of Edit	Abbrev.	Description
Add Field	AF	Adding a new field to a class (possibly with an initializer)
Change Field	CF	Changing an existing field by altering its type, name, or value
Delete Field	DF	Deleting an existing field
Add Method	AM	Adding a new method (possibly with the method body)
Change Method	CM	Changing an existing method by altering its signature or body
Delete Method	DM	Deleting an existing method
Add Type	AT	Adding a new type declaration (class, interface, or enum)
Change Type	CT	Changing an existing type declaration
Delete Type	DT	Deleting an existing type declaration
Change Import Statement	CIS	Adding, changing, or deleting one or more import statements
Non-code Change	NCC	Editing code without altering the abstract syntax tree structure (e.g., editing a comment section, reformatting the code)
Unknown	UNK	All the other changes (e.g., when the edit range spans across multiple code elements)

Table 7-1. Different kinds of code edits determined by the collapsing algorithm.

		Incoming Change											
		AF	CF	DF	AM	CM	DM	AT	CT	DT	CIS	NCC	UNK
Pending Changes	AF		AF	NCC									
	CF		CF	NCC									
	DF												
	AM					AM	NCC						
	CM					CM	NCC						
	DM												
	AT												
	CT												
	DT												
	CIS										CIS		
	NCC											NCC	
	UNK												

Table 7-2. Collapse test matrix used for the *method level* collapse test.

		Incoming Change											
		AF	CF	DF	AM	CM	DM	AT	CT	DT	CIS	NCC	UNK
Pending Changes	AF	CT	CT*	CT*	CT	CT	CT		CT	DT			
	CF	CT	CT*	CT*	CT	CT	CT		CT	DT			
	DF	CT	CT	CT	CT	CT	CT		CT	DT			
	AM	CT	CT	CT	CT	CT*	CT*		CT	DT			
	CM	CT	CT	CT	CT	CT*	CT*		CT	DT			
	DM	CT	CT	CT	CT	CT	CT		CT	DT			
	AT	AT	AT	AT	AT	AT	AT		AT	NCC			
	CT	CT	CT	CT	CT	CT	CT		CT	DT			
	DT												
	CIS										CIS		
	NCC											NCC	
	UNK												

* or the value specified in the method level matrix, if the changes are made on the same code element

Table 7-3. Collapse test matrix used for the *type level* collapse test.

7.3. Integration with the Timeline Visualization

As briefly shown in Figure 7-3, the collapsed edits can be displayed in the timeline visualization. The collapsing algorithm is tightly integrated with the timeline visualization and the selective undo mechanism of AZURITE in many ways described in this section. For clarity, a rectangle representing a set of collapsed edits will be called a *group rectangle*, and the raw-level edits constituting the group rectangle as *member edits*. The group rectangles are color-coded according to their change kind. All the Adds (AF, AM, AC) are colored as yellow-green, all the Changes (CF, CM, CC, CIS) are sky-blue, and all the Deletes (DF, DM, DC) are pink. The other kinds of changes (NCC, UNK) are shown as grey.

7.3.1. Semantic Zooming

Even before the collapsing mechanism was implemented, the timeline visualization supported arbitrary zooming in both directions. In the current version of AZURITE, the timeline visualization supports semantic zooming using the edit collapsing mechanism when the user changes the horizontal zoom level, for example by using the horizontal zoom slider (see Figure 7-6). By default, the horizontal zoom scale of the timeline is set at 1.0x, and the parse level is used for displaying the edit history. When the timeline is zoomed in to more than 1.5x scale, the collapse level is automatically switched to the raw level, so that the user can see the finest-grained changes from the timeline. The method level is used when the zoom scale becomes less than 0.7x, and the type level when

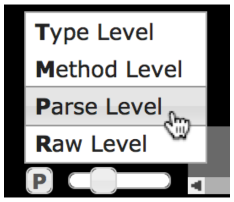


Figure 7-6. The horizontal zoom slider and the collapse level controller (the letter “P” and the popup menu above it), located at the bottom-left of the timeline.

it becomes less than 0.4x. These threshold zoom scale values are chosen empirically as a proof of concept. In the future, different threshold values would need to be tested with users to determine the most appropriate values.

Users might want to examine the history at a different collapse level without necessarily changing the zoom scale. In this case, users can manually switch to another collapse level by clicking the collapse level controller button (the "P" in Figure 7-6) and then selecting the desired collapse level (Figure 7-6). An abbreviation of the current value of the current collapse level (e.g., "P" for the parse level) is always displayed as the button label.

7.3.2. Width of a Group Rectangle in the Compact Layout Mode

When implementing the semantic zooming feature, extra care was taken to make sure that a group rectangle is reasonably sized. Recall that the width of a rectangle in the timeline at the raw level indicates the time it took to perform that edit, which is calculated by subtracting the start timestamp from the end timestamp of the edit operation (Section 6.1). However, it could be problematic if we simply did the same for calculating the width of a group rectangle by subtracting the start timestamp of its first member edit from the end timestamp of its last member edit, especially in the compact layout mode (Section 6.3). This is because there can be arbitrary large gaps between the member edits, the resulting group rectangle could become excessively wide.

In order to address this problem, the width of a group rectangle is calculated by taking the sum of all the width values of the individual member edits that they would have at the raw level. Therefore, the meaning of the width of a rectangle at the higher collapse levels is different from that at the raw level: the width indicates the actual time the programmer spent to make the group edit, excluding all the idle times in-between.

Because of the way width values are calculated, the timeline visualization does not abruptly jump to a different location even when the user is manually switching the collapse levels. For example, the screenshots in Figure 7-3 are taken by keeping the horizontal zoom scale at 1.0x and manually switching the collapse levels, and it is clearly shown that the horizontal locations of the rectangles are well aligned across the different collapse levels.

7.3.3. Selection of the Group Rectangles

One of the most important aspects of the timeline visualization is that users can select some of the past edit operations in the timeline and invoke useful commands that operate on those selected edits such as selective undo. However, it is unclear how the rectangle selection feature would work in the presence of the various collapse levels. For example, suppose that the timeline is currently at the raw level and showing four operations in the history: A through D. The user selects three rectangles, A, B, and C, and then switches the collapse level to the parse level, which might show two group rectangles: [A-B] and [C-D]. What should happen to the rectangle selection in this case? Especially, how do we handle the second group rectangle [C-D], where only some of its members were originally selected?

One way is to only allow selecting a group rectangle as a whole, which could be achieved in multiple ways. The system could discard the rectangle selection before performing the collapse level change. This is the easiest option in terms of implementation, but it is not desirable because the user could lose all the selections just by changing the horizontal zoom scale. Alternatively, the system could choose to keep some of the group rectangles as selected, whose member edits were all originally selected. In this case, the first group rectangle [A-B] will be marked as selected, because all of its members, A and B, were selected in the previous collapse level. For the second group rectangle [C-D], where only some of its members were selected, the system could either choose to discard C from the selection, or else add D to the selection and mark the group rectangle [C-D] as selected in the new level. However, these two options are not very desirable either, because the selection may be altered against the user's wishes.

Alternatively, the system could retain the original selection as is and indicate the group rectangles with only some selected members as *partially selected*, which is what AZURITE does in the current version. A partially selected rectangle is indicated with a dotted yellow outline in the timeline (Figure 7-7). Bringing up a context menu and performing an operation, such as selective undo, will still operate on the originally selected rectangles. However, if a partially selected rectangle is clicked with the left button, then it becomes fully selected. Clicking a fully selected rectangle discards the rectangle from the selection, as it would normally do at the raw level.

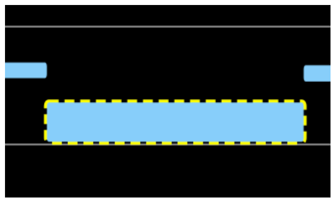
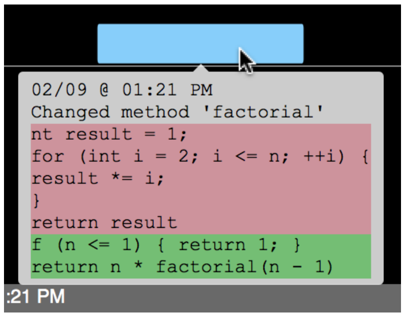


Figure 7-7. An example composite rectangle which is partially selected. If the user clicks on this rectangle, it becomes fully selected.

When a group rectangle is fully selected, the context menu items work exactly the same way as they would when all the member rectangles are selected at the raw level. As an exception, invoking the "Jump to the Code" command on a group rectangle, either by the context menu or by double-clicking the group rectangle, moves the edit cursor to the location of the first member edit.

7.3.4. SUMMARIZING THE COLLAPSED EDITS IN THE TOOLTIPS

As discussed in Section 7.2.2, the higher-level collapsers need to determine the kind of change made by the edit operations, such as Add Method (Table 7-1). This information about the kind of change that was extracted from the edits can be useful not only for successfully performing the collapse test, but also for showing the summary of edits to the users. In AZURITE, when the user hovers the mouse over a group rectangle, the one-line summary of the edit is displayed at the top, followed by the actual code changes. In Figure 7-8, the summary is shown as "Changed method 'factorial'". As in this screenshot, the name of the relevant code element (field, method, or class) is also displayed. If the code changes are too big, the tooltip box may be expanded beyond the boundary of the timeline visualization, which is a known issue with the current implementation. In the future, this problem could be resolved by adding scrollbars directly within the tooltip area, or by allowing the tooltip to be launched as a flowing window.



```
02/09 @ 01:21 PM
Changed method 'factorial'
nt result = 1;
for (int i = 2; i <= n; ++i) {
result *= i;
}
return result
f (n <= 1) { return 1; }
return n * factorial(n - 1)
```

Figure 7-8. An example tooltip shown for a composite rectangle in the timeline. The one line summary also shows the method name `factorial` in which the edits were performed.

7.4. Log Analysis

The real-time collapsing mechanism is fully presented in the previous sections, but there are many interesting questions unanswered. How well does the collapsing mechanism work for real code edit histories? How many rectangles would be collapsed at each level, and what are the reduction rates? What are the relative occurrences of the different kinds of code changes? What do these numbers imply?

To explore the answers to these questions, the collapsing mechanism was tested with the entire code editing transcripts obtained from the longitudinal study (Section 4.3). The edit collapsing component of AZURITE processed the edit operations in the transcripts as if they were made in the code editor. All the coding events other than the code edits are also fed into the collapsing component to make sure that the edits are not collapsed across significant coding events, as explained above in Section 7.2.1.

Table 7-4 summarizes the number of edit operations resulted in each collapse level for all the participants. There were a total of 282K edit operations at the raw level. The number of edit operations is considerably reduced when going up one level. On average, the parse level collapser reduces the number of edit operations from the raw level by 55%, resulting in 128K operations. In turn, the number is reduced by 50% to 64K operations at the method level, and reduced by 26% more to 47K operations at the type level, which is the highest collapse level. This implies that the collapsing mechanism would be useful in minimizing the potential information overload by reducing the number of rectangles displayed in the timeline.

The distribution of different kinds of code changes was also investigated, and the results are shown in Table 7-5. Not surprisingly, the vast majority of the code edits are Change Method (CM) edits, which would mostly be about the actual program logic. There are about 98K CM edits at the parse level, which is reduced to 39K at the method level, accounting for most of the edit operation reduction at the method level. This implies that programmers work on the same method for a while before moving to another part of the code, considering that some coding events such as application runs split the preceding and following set of code edits.

PID	Raw Level	Parse Level	Method Level	Type Level
p00	66,400	29,280	19,902	17,312
p01	1,657	784	384	218
p03	8,041	2,811	1,162	796
p04	21,421	11,069	5,316	3,482
p05	70,550	30,338	9,979	5,577
p06	917	314	207	164
p08	18,302	7,666	4,852	3,669
p107	27,196	14,319	5,283	3,134
p110	740	427	127	78
p112	12,516	6,321	2,795	2,111
p201	2,939	1,374	600	460
p202	4,344	2,198	1,844	1,710
p204	9,639	3,708	2,154	1,742
p205	6,705	3,199	2,214	1,920
p206	4,032	1,526	918	799
p207	3,756	1,482	688	533
p209	4,174	1,690	955	788
p210	862	415	346	335
p301	3,045	1,799	649	457
p303	7,856	4,223	1,906	842
p304	7,103	2,740	1,703	1,257
Total	282,195	127,683	63,984	47,384
Avg. per hour	193/hr	88/hr	44/hr	32/hr
% Reduction from the Previous Level		**55%**	**50%**	**26%**
% Reduction from the Raw Level		**55%**	**77%**	**83%**

Table 7-4. Number of edit operations at each collapse level, obtained from the log data set used in the longitudinal study of backtracking. The number of edit operations is significantly reduced at each collapse level.

At the type level, all of the field level changes (AF, CF, DF) and method level changes (AM, CM, DM) are significantly reduced, because they would be collapsed with other changes within the same type to constitute a Change Type (CT) change. Still, the number of CT changes is less than the previous level, indicating that many of the method level changes were collapsed into a single CT change.

About 6% of the changes are non-code changes (NCC) which do not change the AST structure of the code, and less than 2% of the changes are classified as Unknown (UNK), meaning that the change detail extractor failed to identify what kinds of changes they were, most often because there are different kinds of changes mixed in a single change.

Change Kind	Parse Level	Method Level	Type Level
AF	1,598 (1.3%)	1,571 (2.5%)	820 (1.7%)
CF	3,654 (2.9%)	2,345 (3.7%)	713 (1.5%)
DF	573 (0.4%)	549 (0.9%)	190 (0.4%)
AM	2,753 (2.2%)	2,698 (4.2%)	1,713 (3.6%)
CM	97,941 (76.7%)	**39,356 (61.5%)**	28,758 (60.7%)
DM	799 (0.6%)	742 (1.2%)	361 (0.8%)
AT	105 (0.1%)	107 (0.2%)	107 (0.2%)
CT	7,714 (6.0%)	7,735 (12.1%)	**5,867 (12.4%)**
DT	44 (0.0%)	44 (0.1%)	44 (0.1%)
CIS	5,275 (4.1%)	4,439 (6.9%)	4,420 (9.3%)
NCC	6,287 (4.9%)	3,458 (5.4%)	3,451 (7.3%)
UNK	940 (0.7%)	940 (1.5%)	940 (2.0%)
Total	127,683	63,984	47,384

Table 7-5. Distribution of the different kinds of code changes at each collapse level.

7.5. LIMITATIONS AND FUTURE WORK

The edit collapsing mechanism described in this chapter has a number of limitations. First, the mechanism never reorders the edits, so the collapsing mechanism may not work well when the programmer is jumping around multiple locations in code. For example, a programmer might add or delete import statements (CIS in Table 7-1) using the Organize Imports feature of Eclipse while writing a method body (CM in Table 7-1). According to the collapse test matrices (Table 7-2 & Table 7-3), these changes would not be merged and the CIS change will appear in the middle of two CM changes, which may not be what the user wants. The main reason for not reordering the edits in AZURITE is because reordering edits in the history may break the correctness of the dynamic segment information. In the future, a history refactoring mechanism as in Historef [Hayashi 2012][Hayashi 2015] could be implemented and extended to be compatible with the dynamic segment management and the selective undo mechanism presented in Chapter 5 to support edit collapsing in a smarter way.

While the general idea behind the collapsing mechanism is programming language independent, the actual mechanism described in this chapter is tied to Java. Nevertheless, the change kind table (Table 7-1) and the collapse test matrices (Table 7-2 & Table 7-3) could be adjusted for other object-oriented programming languages with minimal effort. Another limitation is that the collapsing mechanism only detects a few kinds of changes. In the future, more specific kinds of changes could be detected as in [Fluri 2006]. In addition, the visualization of the collapsed edits might be improved by coding more information about the changes with colors.

The change detail extraction process could also employ other techniques to improve performance or extract additional useful information. Currently, the entire source files are parsed to determine whether the files are currently parseable and to extract the change details from two versions of code (Figure 7-5). The performance could be improved by optimizing the parser to only process the changed area of code instead of the entire file, for example by using island grammars [Moonen 2001]. Alternatively, the AST parser could be configured to resolve the binding information to determine the connection between different parts of code, which could be useful for edit collapsing. For example, if the collapser could determine that two consecutive code edits are semantically related (e.g., renaming a method and its call sites), those edits could be collapsed as a *semantic unit* of code changes. However, resolving binding information incurs considerable cost, and it may not be feasible to be used for a real-time collapsing algorithm. Moreover, when the code is actively being edited, there is no guarantee that the source code being parsed is in a stable state, which would make it very difficult to resolve the binding information correctly.

There could be other types of collapse levels which are orthogonal to the four collapse levels provided. For instance, when the programmer is using a task tracking system such as Mylyn [Kersten 2006], all the code changes made for the same task could be collapsed in the history.

7.6. CONCLUSION

This chapter described the real-time edit collapsing algorithm used in the AZURITE timeline, which provides four different collapse levels with different granularities. By using the collapsing mechanism, users can quickly review or interact with the recent code change history at a higher level. Although the collapsing algorithm was designed with the goal of making it easier to understand and navigate the history and/or to perform selective undo, the resulting collapsed edits could be used by other programming tools as well. For instance, Section 2.2.4 described the choice edit model, where each code edit operation is translated into a choice in a variational document. One of the problems with this model is that translating each of the fine-grained edits might make the underlying variational document unnecessarily complicated and messy. By using the edit-collapsing mechanism, the choice edit model could take coarser-grained edits and create a more practically useful variational document on the fly.

8.

USER INTERFACES FOR SELECTIVE UNDO[40]

In Chapter 6, two key challenges of providing selective undo UI in code editors were presented:

- How can the users see the history and select past operations?
- How can the users effectively find the desired edits to be undone?

The timeline visualization of the code edit history is mainly focused on addressing the first challenge. It enables users to review the code changes and selectively undo some of the past edit operations by selecting them in the timeline and invoking the selective undo command on the selection. However, there is a still big question remaining unanswered: how can the users select the desired edits to be undone effectively and accurately? If the users need to spend a long time on the timeline figuring out which of the past edit operations should be selected in order to achieve the desired selective undo result, using selective undo for backtracking may not be an effective solution.

The observations from our initial lab study showed that programmers remember certain aspects about the code edits that they want to revert (Section 4.1.6). For instance, they remembered when they made the code changes, the location of those changes, what the surrounding code looked like, and the names of some of the code elements (e.g., variable name, method name) that they were editing. Although the timeline visualization can help in finding the desired edits for some cases (e.g., finding the changes made after the last unit test execution), there are many cases where the programmers cannot directly find the information from the timeline itself.

Our goal is to enable users to express what they remember about the code changes in a more direct way. For example, when the user remembers where the code edits were made, then the user should be able to navigate to that code using the familiar code editor, and then select the area of code and ask the tool to find the code changes made in the selected code. This chapter introduces a number of user interfaces designed with the specific focus to make it easier to find the code changes in the history and to perform selective undo more effectively. AZURITE includes four user interfaces which are described in this chapter: the *code history diff view*, the *regional undo shortcut*, the *history search*, and the *interactive selective undo dialog*. These user interfaces closely interact with the timeline visualization of code edits and the internal selective undo mechanism to provide a better user experience for selective undo. All of these features work with the edit history of the current editing session, but the history from the past sessions

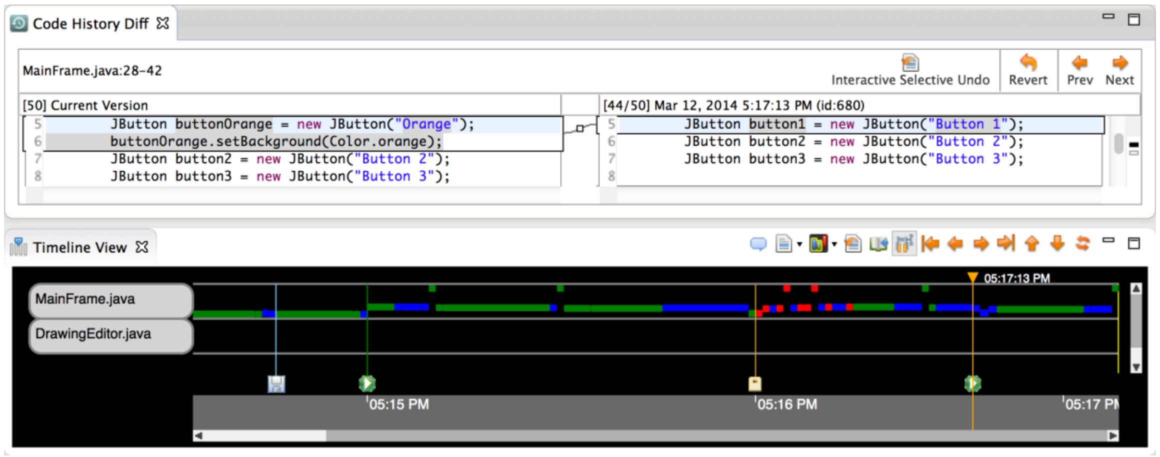
[40] Portions of this chapter appeared in [Yoon 2013] and [Yoon 2015]

can be loaded, if needed. In order to show the feasibility of these user interfaces, they are all implemented in the AZURITE prototype tool. These user interfaces are shown to be usable in our user studies (Chapter 9).

8.1. CODE HISTORY DIFF VIEW

Suppose that a programmer faces a backtracking situation, but only remembers the area of code where the changes were made which she wants to undo. It may be very difficult for the user to find out what rectangles in the timeline were made from the region of code that she wants to undo.

The code history diff view, which is shown in Figure 8-1, is designed to solve this problem. Users can select an arbitrary code snippet from a regular Eclipse code editor window and launch this view, which is a code-compare view with two juxtaposed panels. The left-hand panel always shows the current version[41] (i.e., the most recent version) of the code snippet. The right-hand panel shows some version of the code in the past along with the version number, and the exact time when the change was made.[42] In the figure, there were 50 versions of that specific region, and the user is currently seeing version 44, which was made at 5:17:13pm. The diffs between the two code snippets are marked. The right side code can go all the way back to when the code did not even exist, assuming that this is in the history.



MainFrame.java:28-42

[50] Current Version

```
        JButton buttonOrange = new JButton("Orange");
        buttonOrange.setBackground(Color.orange);
        JButton button2 = new JButton("Button 2");
        JButton button3 = new JButton("Button 3");
```

[44/50] Mar 12, 2014 5:17:13 PM (id:680)

```
        JButton button1 = new JButton("Button 1");
        JButton button2 = new JButton("Button 2");
        JButton button3 = new JButton("Button 3");
```

Figure 8-1. The code history diff view of AZURITE. The most recent version of the selected region of code is always shown in the left panel, and the version of the code from the selected time is shown in the right panel. The currently selected time is indicated by the orange time marker in the timeline at 05:17:13pm.

Users can move back and forth through the history in two ways. First, they can drag the orange time marker (see Section 6.5) within the timeline with mouse to see how the code looked at the time at which the marker is pointing (see bottom of Figure 8-1). This design was inspired by

[41] A new version is added for each fine-grained code edit performed in the selected region of code in the past. Here, the meaning of the term *version* is different from that of the version control systems such as Git.

[42] The left and right panels were switched after it was first presented in [Yoon 2013], to make it consistent with the Interactive Selective Undo dialog which shows the selective undo preview in the right panel.

the time marker in video editors. The code snippet shown on the right panel changes instantaneously as the marker is moved, and diffs are recalculated as well. Alternatively, users can use the navigation buttons above the code (Prev / Next) to move back and forth through different versions incrementally. This is useful when there are many rectangles in the timeline between versions, which are irrelevant to the code snippet that the user is interested in. Whenever the version changes using the navigation buttons, the marker position in the timeline is also updated correspondingly.

The code history diff view can be used for several different purposes. First, it can be used to simply understand how the code has evolved. The results from the experiment conducted by Hattori et al. shows that people can understand the code base faster with a fine-grained replay tool [Hattori 2011]. The code history diff view can be considered as a replay tool for a specific area of code showing the evolution history of that region. Second, this view can be used to look for some deleted code in the history, copy the desired code from the preview panel on the right to the clipboard, and then reuse it in the current code by pasting it. This is useful when the code should be restored in a different place from where it was originally deleted (so selective undo would not put it in the right place). Finally, users can revert the code snippet to one of its previous versions simply by moving to the desired version and clicking the "Revert" button. Instead of reverting the entire code snippet to the version currently shown in the right side, users can also interactively select what to undo and what not to, using the "Interactive Selective Undo" button (described below in Section 8.4).

When the user navigates between different versions of code in the code history diff view, the view respects the current collapse level used by the timeline visualization. For example, when the Prev and Next buttons are used, the version jumps between the group rectangles shown at the current collapse level. This makes it possible to review the evolution history of a specific region of code at a user-selectable coarser or finer granularity.

To implement this view, AZURITE first searches for all edits performed on the selected code, just like when the "Select Corresponding Rectangles" command (Section 6.4) is invoked. Then, AZURITE reconstructs all the intermediate snapshots by incrementally applying selective undo with the searched edits. This process is fast, and the code history diff view is launched without any noticeable delay for any practical size code history diff view.

8.1.1. SCOPE OF CODE SNIPPETS

While most other tools such as local history features of IDEs or version control systems only provide file-based history or method-based history at best, the code history diff view of AZURITE can show the history of an arbitrary code snippet of any size. For example, users might investigate how an **`if`** block was originally written, how the parameters to a function call have changed, or even how a mathematical expression within a single line has evolved over time. Conversely, the history of an entire source file can also be investigated using the code history diff view.

One limitation of our code history diff view is that it can only handle a single contiguous block of code. The reason is because Eclipse does not allow non-contiguous blocks to be selected at the same time. To partially overcome this problem, AZURITE allows multiple code history diff views to be launched as separate tabs which can be dragged to be side-by-side or even in a window outside of Eclipse. This is useful when there are multiple code snippets coupled together around a certain feature, which has been referred to as a working set [Ko 2005a]. In this case, all open code history diff views share the same time marker in the timeline, and users can intuitively see how those code snippets as a whole have evolved over time. Users can also revert all the code snippets in the code history diff views to the versions shown in their preview panels in the right side with a single command.

8.2. REGIONAL UNDO SHORTCUT

During the initial field trial of AZURITE (Section 9.1), I found that the most popular form of selective undo is reverting a specific region of code to an old version, which has been referred to as "regional undo" [Li 2003]. In the current version of AZURITE, users can select some region in the regular code editor and use a keyboard shortcut (Ctrl+R by default, ⌘R on a Mac) one or more times to perform selective undo on that region directly within the code editor. This can be a faster way of performing regional undo, compared to using the code history diff view described above. When this regional undo command is invoked multiple times on the same region, the repeated commands are automatically collapsed into a single command, so that the resulting undo performed by multiple invocations of the regional undo command can also be undone with a single command, if desired.

8.3. HISTORY SEARCH

Recall the motivating example presented in Section 1.2, where the programmer wanted to restore the temporary code she wrote about **`GridBagLayout`**. How could she effectively find the deleted code from the history? She could potentially investigate the timeline and see what rectangle contains the desired code, but it could be very tedious especially when the history size is large.

To address this problem, AZURITE provides a *history search* feature, where users can search the edit history to find the information they need. The history search dialog (Figure 8-2) is invoked from the code editor menus, and the search results are shown in the timeline as selected edit operations. AZURITE provides three history search options. First, users can search for all edits performed on a selected area of code, which we found to be the most desired operation during the field trials (Section 9.1). The scope of this search is not limited to structural code elements such as a class or a method; the search can be performed on an arbitrary region of code that the user selects. Internally, this is the same search mechanism also used by the code history diff view (Section 8.1).

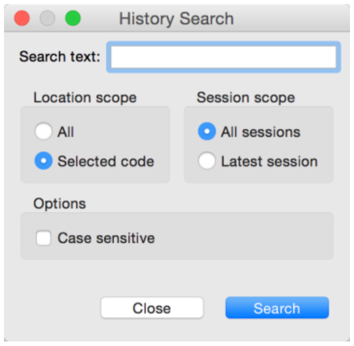


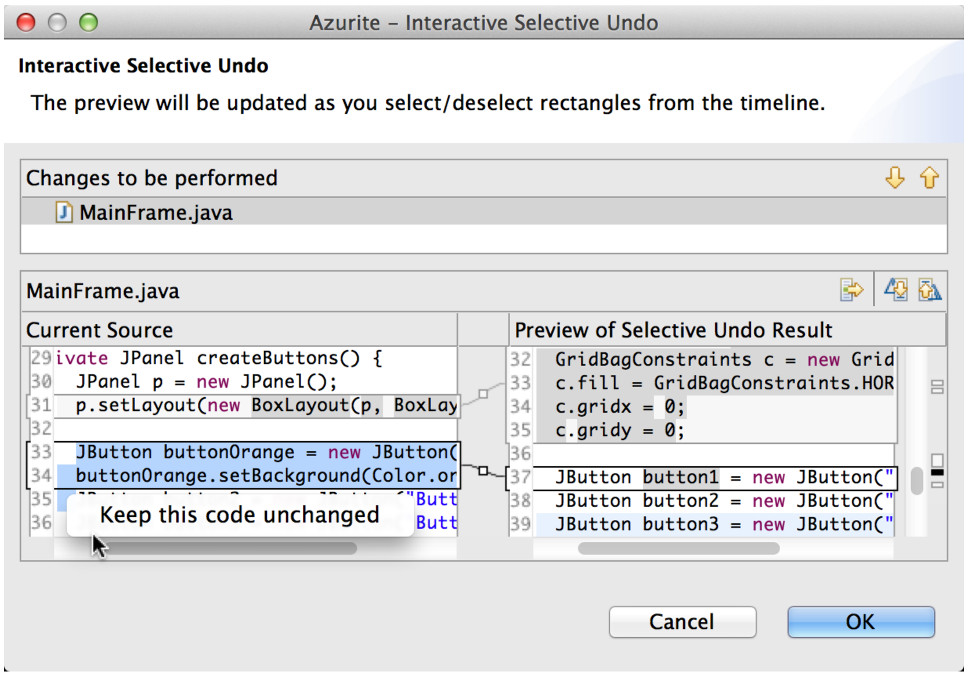
Figure 8-2. The history search dialog of AZURITE. Users can search through the history to find out the time range in which a certain text existed in the code.

Second, users can search for all edits that happened during a time interval where a certain code (or text) existed. Note that, in this case, the searched-for text does not necessarily have to exist now in the code, so this is not the same as searching the current code base for the text. It is also not sufficient to search for the text within the stored deleted / inserted text for each operation, because the text being searched for may be partially in the edit and partially in the code (for example, searching for **`DrawRectangle`** when the code now says **`PaintRectangle`** and an operation is "replace **`Draw`** with **`Paint`**"). To make this search possible, the history search tool internally uses the selective undo feature to calculate the snapshot of the code at each point in time, just like when launching the code history diff view (Section 8.1), and checks if the snapshot contains the desired code or not. Although the history search results were instantly shown for normal usage, the history search could be slowed down when the size of the source file is large and there are a large number of operations in the current history. In the future, a more optimized way of performing history search could be implemented by searching only in the surrounding region of the changed area of each operation locally, instead of searching the entire source file all the time.

Finally, users can limit the search scope to the current session or include the past sessions. In the latter case, only the history of past sessions that are already loaded (see Section 8.5) are considered.

8.4. INTERACTIVE SELECTIVE UNDO

In response to the feedback from the field trial (Section 9.1), a new user interface was created which is called the *interactive selective undo* dialog (Figure 8-3). The design was inspired by Eclipse's refactoring wizard, which shows a preview of all changes to be made before actually changing the code. Similar to a typical refactoring wizard, our interactive selective undo dialog shows a side-by-side "diff" view where the left panel shows the current code and the right panel shows the preview of the selective undo based on the currently selected rectangles in the timeline. On the top panel is the list of all the affected files. When a file is selected on the top panel, the source panels below update their contents to show the selected code and the preview for the selected file.



Azurite – Interactive Selective Undo

Interactive Selective Undo

The preview will be updated as you select/deselect rectangles from the timeline.

Changes to be performed

MainFrame.java

MainFrame.java

Current Source

```
ivate JPanel createButtons() {
  JPanel p = new JPanel();
  p.setLayout(new BoxLayout(p, BoxLay

  JButton buttonOrange = new JButton(
  buttonOrange.setBackground(Color.or
                                     "Butt
                                      Butt
```

Keep this code unchanged

Preview of Selective Undo Result

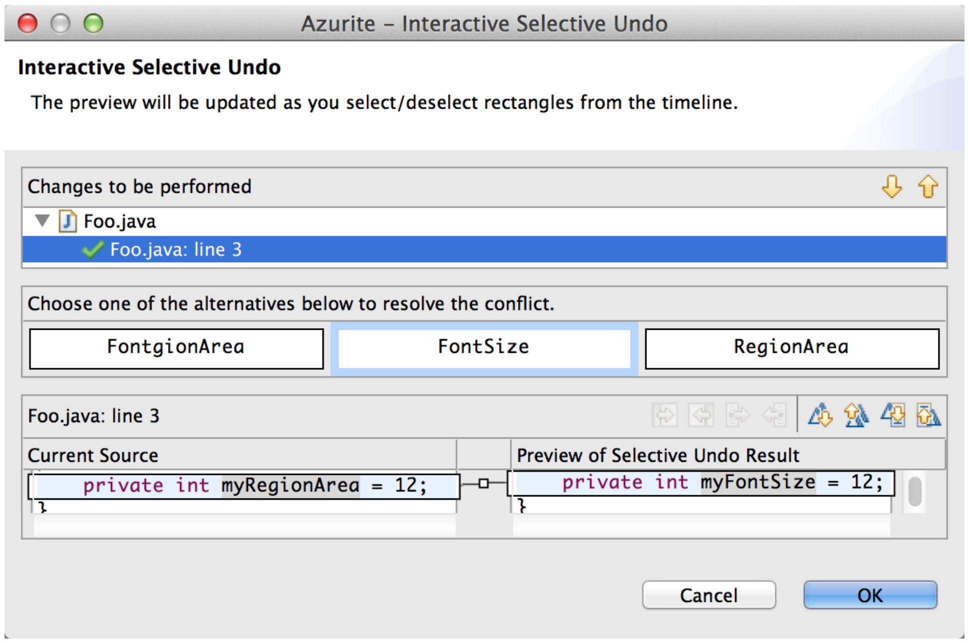
```
  GridBagConstraints c = new Grid
  c.fill = GridBagConstraints.HOR
  c.gridx = 0;
  c.gridy = 0;

  JButton button1 = new JButton("
  JButton button2 = new JButton("
  JButton button3 = new JButton("
```

Figure 8-3. The interactive selective undo dialog of AZURITE. Users can mark some code in the left panel, and ask to "Keep this code unchanged", which can be repeated until the preview in the right matches what is desired.

The interactive selective undo dialog is modeless, and the rectangles can be added to and/or removed from the selection while the dialog is open, which immediately updates the preview results shown in the dialog. By allowing this, users need not worry about selecting the exact set of rectangles on their first attempt. Users can manipulate the selection until the preview shows the desired result.

Additionally, users can select an arbitrary region of the code in the left panel of the interactive selective undo dialog, right-click to bring up the context menu, and select "Keep this code unchanged" (Figure 8-3). Doing this searches for all the edit operations affecting that selected region of code and excludes those operations from what will be undone. This feature provides a significant usability improvement compared to requiring users to select the exact set of operations, because users can easily get the desired results by roughly over-specifying the selected operations, and then marking all the code fragments desired to be in the resulting code.

This dialog is also capable of dealing with regional conflicts. When there is a chunk which contains regional conflicts, the dialog shows a red X icon (⊗) beside the chunk, and the OK button is disabled temporarily. Once the user selects the chunk from the top panel, the dialog shows the three alternative options described in Section 5.2.2.2 so that users can choose which one they want. Once an option is chosen, the chunk is marked as resolved with a green check icon (✓, Figure 8-4), and the OK button becomes enabled when the user resolves all existing regional conflicts. Note that in most use cases of AZURITE, regional conflicts will not come up since rectangles will be selected with the aid of AZURITE features such as the history search, the code history diff view, and the "keep this code unchanged" feature explained


Azurite – Interactive Selective Undo
Interactive Selective Undo
The preview will be updated as you select/deselect rectangles from the timeline.
Changes to be performed
Foo.java
Foo.java: line 3
Choose one of the alternatives below to resolve the conflict.
FontgionArea
FontSize
RegionArea
Foo.java: line 3
Current Source
Preview of Selective Undo Result
Cancel
OK

```
private int myRegionArea = 12;
}
```

```
private int myFontSize = 12;
}
```

Figure 8-4. The interactive selective undo dialog when there is a chunk with regional conflicts. The user can choose one of the provided options to resolve the conflicts. Here, the second option (`FontSize`) is chosen by the user, which is indicated by the blue outline.

above. The conflict-resolution interface is provided for the sake of soundness, when the user manually selects or deselects some of the past operations from the timeline.

There are multiple ways of launching the interactive selective undo dialog. Essentially, for all the situations where the selective undo command can be invoked, the interactive selective undo dialog can be launched instead to review the selective undo results and fine-tune the edits to be selective undone if necessary.

8.5. Reading the History of Past Sessions

AZURITE keeps the history separately for each session, where a session starts with the IDE being opened and ends when the IDE is exited. By default, the timeline displays the history of the current session only. Users can manually invoke the "Read Previous History" command to load the code change history of previous sessions when needed. When this command is invoked, AZURITE looks for the recent FLUORITE logs in the current workspace and reads the edit history into the timeline visualization. Between each two adjacent sessions, a gray vertical line is shown to indicate the boundary between those two sessions. A vertical line at the right edge of the current session indicates "now", which is drawn in yellow to be distinguishable from other sessions. The edit operations loaded from the past sessions can be selected and undone, just like the normal edit operations of the current editing session.

Similar to when some source files are changed outside Eclipse (Section 3.5.1), AZURITE compares the final snapshot of each file in the past sessions and the known initial snapshot of that file in the next session when reading the history of past sessions in order to check the validity of the history. If they do not match, then AZURITE extracts the diffs between the two version using the Google-diff-match-patch library [Fraser 2012], and adds the diffs to the timeline as if they were normal edit operations.

8.6. LIMITATIONS AND FUTURE WORK

There are many different user interface features and editor commands provided by AZURITE. Although these features provide a great level of flexibility in performing various backtracking tasks, having too many features might make it difficult to learn when and how to use these features correctly. On top of that, the user interfaces introduced in this chapter, except for the regional undo, present some additional views or dialogs with which the user needs to interact. Previous research suggests that programmers are less likely to use development tools with complex user interfaces [Vakilian 2012][Lee 2013]. Simplifying the user interfaces and making them work directly within the code editor would improve the usability of those interfaces and minimize the learning overhead (which is why the regional undo shortcut was added (Section 8.2).

Neither the code history diff view nor the interactive selective undo allows editing the selective undo result. Especially in case of the interactive selective undo dialog, half of the evaluation study participants tried to manually edit the code within the dialog, which caused them to take even longer to complete the provided backtracking task (see Section 9.2.6). Allowing users to edit the code within the dialog or making it work within the editor itself would mitigate this problem.

8.7. CONCLUSION

This chapter demonstrated various user interfaces and editor commands that are tightly integrated with the fine-grained code editing histories and are specifically designed for selective undo. I believe that providing more refined editor commands with more history search and filtering options (see Section 11.2) would make programmers more comfortable in code editing, fostering more exploration and more reliable backtracking.

9. EVALUATION OF AZURITE

In order to gain insight on how to improve the user interfaces and to evaluate the usability and usefulness of AZURITE, I conducted two user studies: field trial with the initial user interfaces and a formal comparative evaluation study of a later version of the user interfaces. In addition, the performance of AZURITE was measured in various aspects in order to demonstrate that it does not notably slow down the IDE.

9.1. FIELD TRIAL WITH THE INITIAL USER INTERFACE DESIGN

After implementing the initial version of the timeline visualization (Figure 6-1b) and code history diff view (Figure 8-1) features in AZURITE, I conducted a field trial of AZURITE in order to get feedback on the usability and usefulness of AZURITE. 8 participants[43] were recruited from the Masters of Software Engineering program at Carnegie Mellon and asked to try using AZURITE while working on their regular development tasks over the course of the summer. The entire user interactions related to AZURITE were recorded using FLUORITE. Unfortunately, because the participants were not enforced to use AZURITE features, four of the participants never used any of the features during the study. Two other participants only tried each AZURITE feature once when they first installed AZURITE. Only the remaining two participants actively used AZURITE features including selective undo, so these two people were interviewed individually (I1 and I2), in order to get more detailed feedback.

The two participants provided both positive feedback and suggestions for improvement. I1 expressed favorable opinions about the timeline visualization while I2 did not like it as much. This was because I1 used the timeline visualization mainly for quickly checking the list of the most recently edited files (as shown in the leftmost column of filenames), whereas I2 wanted to use it for selective undo. They both described that they tried to use selective undo multiple times by manually selecting rectangles from the timeline, but failed to get the desired result and had to undo the selective undo operation. The reasons for this was that (1) it was difficult to determine what the resulting code would look like, just by looking at the selected rectangles, and (2) sometimes the edits represented by the rectangles were too small.

For these reasons, I2 used the code history diff view rather than directly selectively undoing from the timeline, because the code history diff view worked as a "preview" of the selective

[43] Originally, 10 participants were recruited, but 2 of them did not submit their log files and quit the study. The log data from the remaining 8 participants were also included in the data set analyzed for the longitudinal backtracking study (Section 4.3, P10-P17 of Table 4-5).

undo result for him. He mentioned that he liked to use the code history diff view even when the regular undo command could be used, because it was convenient to see a preview before undoing.

I2 also mentioned that he often wanted to keep the comments and selectively undo only the code because for him the comments are usually the high-level description of a certain algorithm and only the code may be wrong. They both wanted a feature to "tag" the current point in time in the timeline.

These concerns were addressed in the newer version of AZURITE, by adding the following features: the interactive selective undo (Section 8.4), the tagging feature in the timeline (Section 6.2), highlighting the corresponding areas of code when there are selected rectangles in the timeline (Section 6.4), and the real-time edit collapsing features (Chapter 7). The two interviewees were asked to try out the interactive selective undo and the tagging feature of AZURITE, and I2 appreciated the new features:

> *I would say [the interactive selective undo] is a killer. Combined with the [tagging feature], I have total control over what I want in the code and what I want out.*

9.2. EVALUATION STUDY

After implementing all the new user interfaces presented in Chapter 6 (but not the edit operation collapsing feature of Chapter 7), I conducted a controlled lab study to answer the following two research questions:

RQ1. Is AZURITE *usable*?
RQ2. Is AZURITE *useful*?

The second research question asking the usefulness of AZURITE was decomposed into the following three aspects:

RQ2-1. Can AZURITE users perform backtracking tasks *more accurately*?
RQ2-2. Can AZURITE users perform backtracking tasks *more quickly*?
RQ2-3. Is AZURITE *perceived as useful*?

A between-subjects design was used for this evaluation study with two groups: AZURITE and Eclipse only (as the control). All the tasks were performed using the Eclipse 4.4 IDE on a 15" Macbook Pro machine running OS X 10.9. The study took about 1.5 to 2 hours for each participant.

9.2.1. PARTICIPANTS

A total of 12 programmers were recruited at Carnegie Mellon, 8 males and 4 females (median age=27), who were randomly assigned to either group. All participants reported that they had at least 3 years of programming experience in general (median = 5 yrs), and at least 2 years of Java experience (median = 4 yrs). No one had previously seen or used AZURITE before the study.

9.2.2. TASKS

There were a total of 6 Java programming tasks which were derived from the empirical studies of backtracking (Section 4). The tasks were performed in the same order by all participants, which are summarized in Table 9-1. A separate code base was provided for each task. For tasks 1 through 5, each task was composed of a series of steps: a number of normal (non-backtracking) programming steps described in Table 9-1, followed by a backtracking step to revert the changes made in the *underlined step* while keeping the changes from the other steps. This was because we wanted to have the participants create the edit history themselves, and wanted to ensure there were backtracking situations that would not be trivial to perform using regular undo. For example, for T1, participants were asked to revert the factorial method back to the for loop version after finishing step (3). For task 6, the participants were provided with an existing code edit history and then asked to perform backtracking from there. The full task materials provided to the study participants can be found in Appendix C.

Because it would be impossible to gather backtracking timing data if the participant becomes stuck in one of the non-backtracking steps, we set up a 4-minute limit for each step, and the experimenter helped the participant after the time limit. No help was provided for the backtracking steps from which the reported results are measured.

Task	Individual Steps	SEL	MF
T1	(1) Implement factorial method with a loop *(2) Modify factorial method to use recursion* (3) Make a few more independent changes	√	
T2	*(1) Delete some existing sorting code* (2) Make a few more independent changes	√	
T3	(1) Implement a simple Number class (unit tests given) *(2) Modify Number to be an immutable class*		√
T4	(1) Implement a Stack with inheritance (unit tests given) *(2) Modify Stack to use composition instead*		
T5	(1) Layout GUI controls using GridLayout *(2) Change the layout manager to GridBagLayout* (3) Add another GUI control	√	
T6	Remove all the debugging specific code (e.g., println in multiple places) while keeping the actual bug fix code. An edit history is provided to begin with.	√	√

SEL: selective backtracking (cannot be done with regular undo)

MF: multiple files are involved

Task T5 is similar to the motivating example of Section 1.2.

Table 9-1. Summary of the evaluation study tasks.

9.2.3. *Study Procedure*

After obtaining the informed consent and demographic information, for both groups we alternated between one tutorial session and two programming tasks, resulting in a total of 3 tutorial sessions and 6 tasks.

In the tutorial sessions, the AZURITE group learned how to use the regional undo shortcut (Section 8.2), the "Undo All Files to This Point" feature from the timeline view (Section 6.5), and the interactive selective undo dialog (Section 8.4). The tasks were designed in a way that they can be effectively performed by using the feature they learned from the previous tutorial session, but the participants were told that they were free to use any tactics. In other words, Tasks 1-2 were easier to complete with the regional undo shortcut, tasks 3-4 with the "Undo All Files to This Point" command, and tasks 5-6 with the interactive selective undo dialog.

The control group learned about the local history feature of Eclipse,[44] which is a built-in feature that keeps a per-file local history of every saved version of the source code, and is the closest existing feature that can help with the tasks. In the three tutorial sessions, this group learned how to manually perform selective undo using the local history (by copying the desired code from the compare view), how to replace the entire source code with one of the saved revisions, and how to perform selective undo in multiple files (by performing selective undo in one file at a time).

It seemed possible that the participants might have wanted to please the investigator and behave in favor of the AZURITE tool if they knew if it was developed by the investigator. To mitigate this problem as much as possible, the participants were never told that AZURITE was developed by the investigator. Interestingly, no one in the control group was previously aware of the Eclipse local history feature, and two of them mistakenly thought that the feature was developed by the investigator, which might have also mitigated the bias.

All participants in both groups were familiar with and were able to use the regular linear undo as well. After each tutorial, the participants were given a written document explaining the feature they learned during the tutorial with screenshots, so that they could refer to it later.

9.2.4. *RQ1: Is Azurite usable?*

In general, the participants in the AZURITE group could learn the AZURITE features and use them correctly as intended, after they finished the tutorial sessions. For the regional undo shortcut, all six participants understood the feature quickly, and used the feature for completing the backtracking tasks. There was one participant who mistyped the shortcut key twice during T1, but the same participant had no problems while completing T2 using the

[44] http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.platform.doc.user%2FgettingStarted%2Fqs-55.htm

regional undo shortcut. All six participants could learn and use the "Undo All Files to This Point" feature without any noticeable problems.

However, three out of six participants faced some usability problems while completing T5. This task was meant to be completed using the interactive selective undo dialog, but the participants had to also manually edit the resulting code after using the dialog, due to the subtle difference between the two layout managers they were using. Two of the participants quickly realized this and roughly performed selective undo with the dialog and then manually edited the resulting code, which resulted in somewhat faster completion time. One participant used the regional undo command of AZURITE instead, which also worked. The other three participants in the AZURITE group tried to complete the backtracking using the interactive selective undo dialog exclusively, which resulted in much longer time because the current interactive selective undo dialog does not support manual editing within the dialog itself. On the other hand, the "compare view" of the Eclipse local history does support manual editing, and the resulting code can be manually edited within the compare view. So this is a feature that should be added to AZURITE in the future.

9.2.5. RQ2-1: CAN AZURITE USERS PERFORM BACKTRACKING TASKS MORE ACCURATELY?

All the participants in both groups successfully completed all the provided backtracking steps, thus no meaningful comparisons could be made between the two groups regarding the accuracy of backtracking.

9.2.6. RQ2-2: CAN AZURITE USERS PERFORM BACKTRACKING TASKS MORE QUICKLY?

To see if the AZURITE users can perform backtracking more quickly, I compared the completion time of the backtracking step of each task. An independent-samples t-test was conducted to compare the backtracking completion time between the two groups. Over all tasks, the AZURITE group took significantly less time (mean=386.3 seconds) to perform all the backtracking steps compared to Eclipse only group (mean=768.8s, $p < 0.01$), which is roughly twice as fast. Figure 9-1 shows the average backtracking completion time for the individual tasks for the 6 participants in each group. For tasks 2, 4, and 6, the AZURITE group was significantly faster compared to the control group ($p < 0.05$). For T1, the AZURITE group was generally faster but it was not statistically significant ($p=0.25$), mainly because one participant in the AZURITE group mistyped the shortcut key twice, so he started over and took much longer (173s).

T3 & T4 were non-selective backtracking tasks, meaning that they could have been performed by using the regular undo command multiple times. One participant in the control group in fact used the regular undo command instead of using the local history feature. Still, the AZURITE group was generally faster because they could quickly skim through the timeline to find the last successful unit test run and then undo all files to that point with a single command. There was one participant in the AZURITE group who did not use AZURITE for T3 and

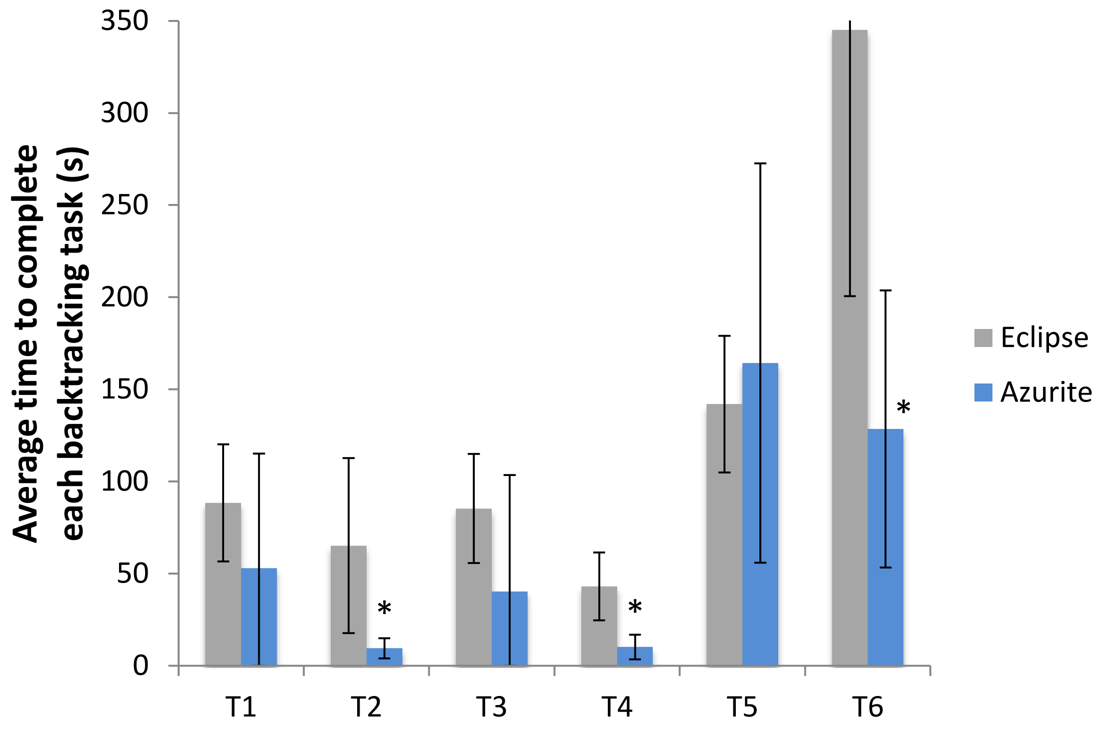


Figure 9-1. The average backtracking completion time for each task. The error bars indicate the standard deviations. *differences are statistically significant (p < 0.05).

took 167s reproducing the code manually, which heavily affected the mean value and made the average time difference statistically non-significant (p=0.16) for T3.

The AZURITE group did not perform better for T5 (p=0.65), and the completion time varied much more (sd=108.4) than the control group (sd=37.11). T6 was also meant to be completed with the interactive selective undo dialog, but in this case everything could be done solely with the dialog, and thus the AZURITE group dramatically outperformed the control group (mean=128.5s v. 345.0s, p < 0.02).

Another interesting point shown in Figure 9-1 is that for each pair of tasks that were testing the same feature (T1&T2 pair, T3&T4 pair, and T5&T6 pair), the second task was always completed significantly faster by the AZURITE group compared to the control group (p < 0.05). This may indicate that the AZURITE users became more familiar with the feature after completing the first task and were able to use that knowledge to complete the second task faster.

9.2.7. RQ2-3: IS AZURITE PERCEIVED AS USEFUL?

After finishing all the tasks, the participants were asked whether the tool they used was useful for them with a 5-point Likert scale. A Wilcoxon rank sum test showed that AZURITE was more useful (median=5) than the Eclipse local history (median=3.5, p<0.05).

9.2.8. Summary of the Evaluation Study

In this formal evaluation study, the usability and usefulness of AZURITE was tested. The participants in the AZURITE group could quickly learn and use the features during the study, but there was one usability problem with the interactive selective undo dialog discovered during the study. The usefulness of AZURITE was evaluated in three aspects. In terms of accuracy of backtracking, there was no meaningful difference between the two groups. The overall backtracking task completion time of the AZURITE group was about half of that of the control group, which was statistically significant. Finally, users perceived AZURITE to be more useful compared to the Eclipse local history feature.

9.3. Performance Feasibility

One of the concerns of using IDE plug-in tools in general is that the tools may slow down the IDE to the extent that the user becomes less productive. It could be problematic if AZURITE introduces noticeable performance degradation of the hosting IDE. Therefore, I sought answers to the following research questions regarding the performance:

RQ3. How much disk space do FLUORITE log files take?
RQ4. Does AZURITE notably slow down the IDE?

For RQ4, I measured specifically how much time it takes to perform all the calculations needed for providing selective undo, whenever a new edit operation is added to the history. There are three different logics that run when a new edit operation is made. First, the dynamic segments of all the past operations are updated (Section 5.1.3), the real-time edit collapsing logic runs to determine whether the new edit should be collapsed with the previous ones (Section 7.2), and a new rectangle representing the new edit is added to the timeline (Section 6.1). The calculation time for these three logics were measured separately, which will be discussed in more detail below.

9.3.1. RQ3: Disk Space Used by Fluorite Logs

AZURITE uses the fine-grained code change history generated by FLUORITE (Chapter 3). During our studies (Chapter 4), there was no noticeable 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 total size of FLUORITE log data from 21 programmers, containing 1,460 hours of active coding activities, collected for the longitudinal backtracking study (Section 4.3) was 377MB, which gives a log size growth rate of about 264.5KB/hour during active editing. Given the spacious hard drives used nowadays, this will not be a critical issue for most users. Moreover, the same study discovered that 99% of the backtracking instances are performed within 3 editing sessions, implying that purging old editing histories would be safe enough in most cases if the main use for the logs was for backtracking (as opposed to using them for understanding the changes in the code).

9.3.2. RQ4-1: Performance of the Edit History Management Algorithm

The edit history management algorithm is described in Section 5.1.3 and Figure 5-5. The **`update(`**H**`,`** $newOp$**`)`** function is called whenever a new edit operation $newOp$ is performed. Each new operation can add at most 4 dynamic segments: 1 delete segment, 1 insert segment, and up to 2 more segments (Figure 5-1b3) if it happens to split an existing insert segment. Since the update function iterates through all the segments in the history and performs constant time update work for each segment, the worst-case time complexity of update function is $O(N)$, where N is the total number of edit operations in the current history H.

This algorithm gets slower as the history gets bigger because of this. The actual time it takes to add a new operation to the history was measured[45] under varying sizes of N, and it took 3.93ms when N = 10,000, and 39.58ms when N = 100,000, which confirms the linear time complexity. According to the collected log data, 10,000 operations approximate one week of coding work, assuming 40 hours of work a week (avg. # of edits: 193/hour, from Table 7-4).

This edit history management is needed for providing selective undo. Considering that 97% of the backtrackings are performed within the same editing session (Section 4.3.2.5), the edit history management algorithm, even unoptimized, will work in practice without causing significant delay.

9.3.3. RQ4-2: Performance of the Real-Time Edit Collapsing Algorithm

When the real-time edit collapsing algorithm (Chapter 7) is running, the collapse logic runs whenever a new edit operations is performed in the code editor. Note that an edit operation usually contains one or more tokens, not just a single character, because FLUORITE merges consecutive character typing operations into a single edit (Section 3.3.3). The entire collapsing logic is illustrated in Figure 7-4, and the running time is dominated by the collapse test logic, which requires parsing the current source file (Section 7.2.2). Unlike the edit history management algorithm above (Section 9.3.2), the running time of the collapsing algorithm is not affected by the current history size. Rather, the size of the current source file matters, because parsing the code is the key element of the collapse test algorithm.

In order to measure the performance of the collapse logic in average use, I measured the time it takes to run the edit collapsing logic at each level, using the data set from the longitudinal study (Section 4.3) on the same machine as above.

Table 9-2 summarizes the mean time it takes to run the collapse logic at each collapse level. Here, I took the trimmed mean values by trimming 10% of the data from each end in order to remove some extreme outliers from the data.[46] The parse level logic, which mainly tests if the current code is parseable, takes about 7ms on average. The method and type level logic, which

[45] Measured on a PC running Windows 8 with a 2.60GHz CPU.

[46] This performance analysis was done offline by feeding the log data to the edit collapsing logic. The analyzer running on the JVM was using a lot of memory to process all the huge log files, and occasionally froze for a long time for garbage collection, which resulted in some extreme outliers.

requires more sophisticated change detail extraction process as in Figure 7-5, takes about 11~12ms.

Collapse Level	Running Time
Parse Level	7.08 ms
Method Level	11.29 ms
Type Level	11.80 ms

Table 9-2. Running time of the collapse logic at each collapse level (in milliseconds).

From the data presented earlier in Table 7-4, we can obtain the invocation rates of the collapse logic at each level. The parse level collapse logic is called every time when a new edit is made. The method level logic is called 0.45 times/operation, and the type level logic is called 0.23 times/operation on average.[47] Combining invocation rate values with the measured time, the time it takes to run the collapse level per edit operation can be calculated by taking a weighted average as follows:

$$7.08\text{ms} \times 1/\text{op} + 11.29\text{ms} \times 0.45/\text{op} + 11.80\text{ms} \times 0.23/\text{op} = 14.87\text{ms}/\text{op}$$

This means that whenever a new edit is made, the collapsing logic runs for about 15ms on average, which is acceptable. The current implementation of the collapsing algorithm is not very well optimized, and the running time would be considerably reduced when we implement some simple performance improvements such as AST caching (see Section 7.2.2).

9.3.4. RQ4-3: Performance of the Timeline Visualization

The timeline should not significantly affect the response time of the code editor either. The response time was measured for several important operations of our timeline with 700 and 3,500 rectangles, which approximates one day and one week of work *at the parse level*, respectively (88 rects/hr from Table 7-4). The time was measured on the same machine as above running Internet Explorer 10.[48] The results are summarized in Table 9-3.

Note that the measured time in this section does *not* include the time it takes to compute which edits should be collapsed, which was discussed above in Section 9.3.3. The collapse results are forwarded from the edit collapsing component to the timeline, and the timeline is merely responsible for correctly displaying the rectangles. In other words, the performance of the timeline operations listed below is only dependent on the number of rectangles in the timeline, independent from the current collapse level.

[47] Calculated by dividing the total number of operations at the previous level, by the total number at the raw level. See Table 7-4 for the detailed numbers. (Method Level: 127,683 / 282,195 = 0.45, Type Level: 63,984 / 282,195 = 0.23).

[48] The browser information here is relevant because the timeline runs in an embedded browser.

Overall, the compact mode is much slower than real-time mode because of the calculations required to remove the gaps. The only operation that is automatically performed while the programmer is editing code is Add Rectangle, which only takes a negligible time (6ms) even in compact mode with 3,500 rectangles, which means that the timeline is non-intrusive in average use cases. The other three operations in Table 9-3 are called only when the user interacts with the visualization. A Layout routine recalculates the positions of all rectangles, and it is called when the file filtering option is changed, when the layout mode is toggled, or when the collapse level is being changed. Although Layout takes more than 2 seconds for 3,500 rectangles in compact mode, this is not likely to be an issue in practice because the number of rectangles was a huge overestimation, and the Layout operation is not needed for most use cases of the tool.

	Compact mode		Real-time mode	
	# of rectangles		# of rectangles	
Operations	700	3,500	700	3,500
Add Rectangle	3	6	3	6
H-Scroll	3	9	10	31
V-Scroll	2	6	6	24
Layout	174	2,852	112	593

Table 9-3. Summary of the measured response time (in milliseconds).

9.3.5. RQ4: Putting It All Together

Assuming that a user keeps her Eclipse environment open for a week and writes code for about 40 hours (which is an overestimation), a new edit operation in the code editor can cause all the dynamic segments to be updated (4ms), the edit collapsing logic to run (15ms), and a corresponding rectangle to be added to the timeline view (6ms). Adding these three values, the average response time of AZURITE for adding a new edit operation is 25ms. This means that AZURITE would not notably slow down the Eclipse IDE.

9.4. Example Use Cases

This section lists some example use cases where the timeline visualization in conjunction with the various selective undo user interfaces of AZURITE can be used to solve real-world problems that previous research shows that software programmers face.

9.4.1. Answering History-Related Questions Programmers Ask

Prior research has identified many hard-to-answer questions programmers ask as part of their development activities [Fritz 2010][Ko 2007][LaToza 2010]. These include the following history-related questions (quoted directly from [LaToza 2010]), which can be easily answered using AZURITE's visualizations. Note that answering these history questions can also be an effective strategy for answering higher-level rationale questions, as pointed out in [LaToza 2010].

Q1. When, how was this code changed or inserted?

Q2. How has it changed over time?
Q3. Has this code always been this way?

These three questions can be easily answered by selecting the code and launching the code history diff view, which would show how the code has changed over time with the exact timestamp of each change.

Q4. What recent changes have been made?

This question can be answered using the timeline visualization. Programmers can load the history of the past editing session to see what recent changes have been made to the project (see Section **Error! Reference source not found.**). Since the most recently changed file is always shown at the top of the timeline, the recently changed files can be easily identified by reading the file names from the top. If more detailed information is needed, the user can jump to the specific code by double-clicking the last rectangle of each file and the code history diff can be viewed to see the actual code edits.

Q5. What else changed when this code was changed or inserted?

First, the code history diff view can be used to determine when this code was changed or inserted. Next, from the timeline, the user can select the surrounding time range of interest and invoke "show all files edited in range" command (Section 6.5) to determine all the other files that were also edited within that timeframe.

Q6. How did this ever work?

Although it would not directly answer this question, AZURITE can help provide some clues by allowing programmers to quickly identify *when* the code was introduced, and go back to see the way the whole project was at that point in time, so that they can test the program under the configuration where the code was introduced. If the code still does not work in that situation, then it is likely that the code has never actually worked correctly.

9.4.2. SELECTIVE UNDO SCENARIOS

In our empirical studies, we identified several problems programmers face while backtracking (Chapter 4). Based on those observations, this section lists some scenarios where the visualizations and selective undo features can help with backtracking tasks we observed. When these scenarios occur in between version control commits, the version control systems would not help in these situations.

9.4.2.1. REVERTING TO A PREVIOUSLY USED LAYOUTMANAGER

When programming GUIs, one often ends up having to experiment with different layout managers to get the UI to look as desired. Recall the example scenario described in Section 1.2. A programmer is implementing a GUI dialog in Swing. She first writes the code with

`GridBagLayout`, and then changes to a simpler **`BoxLayout`** manager. Then she realizes **`BoxLayout`** does not look right, and wants to revert back to **`GridBadLayout`**. Assuming the target operations will be close together either in location or time, this backtracking task can be completed using AZURITE. History search will find the point in time where **`GridBagLayout`** existed in the code snippet, and the user can use the timeline and/or code history diff view to find the exact point to backtrack to, and then revert the code with the "Undo All Files to This Point Interactively" command. Note that this works even if other, unrelated changes are made after or interspersed with the edits to the layout manager, which would make using conventional undo or a version control system inappropriate.

This could be achieved without AZURITE if the programmer commented out the **`GridBagLayout`** code instead of deleting it. However, it was observed from the preliminary lab study that even the programmers who explicitly said they regularly commented out code, occasionally deleted code which turned out to be needed later (Section 4.1) and programmers occasionally had trouble uncommenting the code correctly.

9.4.2.2. *RESTORING DELETED CODE IN GENERAL*

Searching for the code like **`GridBagLayout`** is not the only way to restore the deleted code. I also noticed that programmers often remember where the code was deleted, or what the surrounding code looked like, in which case the code history diff view or the regional undo shortcut can be used to find and then restore the desired code.

9.4.2.3. *REMOVING TEMPORARY DEBUGGING CODE*

One popular debugging strategy is to add print or logging statements in various locations to see how the values change as the program executes. In many cases, these statements are temporary and should not be committed to the main repository. Removing all the recently added **`println`** statements, however, can be a tedious task if they are spread across multiple locations. If the **`println`** statements were consecutive in the history (i.e., they were inserted at the same time), this can be done quite easily with AZURITE. First, the user can locate one of the **`println`** statements from the code editor with regular search, for example. Then, the user can identify the point in time when the statement was inserted with history search or code history diff view. Since all the **`println`** insertions would appear near to each other in the timeline, they can be easily selected together and undone at once, even if they are across multiple files. This also works if there are other **`println`** statements mixed in the code that were *not* part of this debugging and thus should be kept unchanged, in which case using regular text search would be less useful.

9.4.2.4. *ABORTING OR UNDOING A MANUAL REFACTORING*

Researchers have discovered that programmers perform refactoring manually a majority of the time, even when there are automatic refactoring tools available [Murphy-Hill 2009][Vakilian 2012]. Aborting a manual refactoring in the middle, or undoing it sometime later could be tedious, because there can be many steps involved often across multiple files to achieve the refactoring. Using AZURITE, a manual refactoring can be reverted in various ways. Users can use the code history diff view to navigate to the desired version of the code and revert the snippet to that version. If there are multiple files involved, users can find the other files that were edited together using AZURITE's filtering feature and selectively undo them together. Depending on the type of refactoring, other types of history search can help. For instance, if the refactoring was Extract Method, one could easily find the point in time right before the refactoring was performed by searching for the time when the extracted method name first existed.

9.4.3. Other Benefits

Programmers often need to remember the previously attended locations in code when resuming from an interruption or when switching between tasks [Parnin 2012]. The timeline can be used as automatic bookmarks of recently edited locations in such a situation. Using the timeline, programmers can quickly see which files were the most recently edited in a project or in a workspace as a whole. For each file, the last edited location can be easily visited by double-clicking on the last rectangle that appears in the timeline. Unlike manual bookmarks, this works well even when there are a great many files in the programmer's workspace.

9.5. My Own Experience of Using Azurite

This section describes my own experience using AZURITE as a user. I have been using AZURITE for my own development of FLUORITE and AZURITE since the field trial version. First of all, there was no measurable performance loss while using the Eclipse IDE with AZURITE installed. In addition, I could analyze my FLUORITE logs to see how frequently I used each AZURITE feature and reflect on how I used the features. Table 9-4 summarizes the execution count for each AZURITE command in 2014. Note that all the log files created while testing or debugging are excluded from the analysis, because they do not reflect my normal use of AZURITE as a user. Excluding all the testing and debugging logs, my logs contained 146 hours of coding activities.

AZURITE Feature	**Count**
Timeline interactions	36
Selective undo (from the timeline)	9
Regional undo	21
Code history diff view	17
Interactive selective undo	7
History search	5
Undo all files to this point	3

Table 9-4. Frequency of all the AZURITE commands that I used during 2014.

The timeline interactions (36 times) were used for understanding the recent code edit histories, or jumping to the code location associated with some rectangle in the timeline, or selecting rectangles to be undone. The most basic form of selective undo – selecting rectangles from the timeline and invoking the selective undo command – was only used 9 times. The regional undo (21 times) and the code history diff view (17 times) were the two most frequently used forms of selective undo. The regional undo shortcut was the most convenient when I knew the exact location of code that I wanted to undo. I used the code history diff view instead in several situations: (1) when I wanted to see the preview of the undo result, (2) when I only wanted to check how the code looked in the past, and (3) when I wanted to restore some deleted code, but wanted it to be restored in a *different location*. In the last situation, I first launched the code history diff view, navigated backwards through the history to find the code that I wanted, copied the desired code to the clipboard, and then pasted the code to the desired location. I used the interactive selective undo (7 times) and the history search (5 times) less frequently, compared to the regional undo features. This was because most of the backtrackings that I performed were simple enough to be completed with the regional undo. Finally, I occasionally used the "Undo All Files to This Point" command from the timeline (3 times) to revert multiple source files to a certain point in the past.

Excluding the timeline interactions and the history search invocations which were used only to select what to undo, I performed selective undo a total of 57 times over the course of 146 hours of coding. In other words, my average backtracking rate using AZURITE was 0.40/hour. Although this rate is smaller than the selective backtracking rate of 0.98/hour that I obtained from the longitudinal study, I personally found AZURITE to be useful for my own development overall.

9.6. CONCLUSION

The selective undo mechanism for code editors described in Chapter 5 was implemented in our prototype tool AZURITE, with all the user interfaces introduced in Chapters 6, 7, and 8. In this chapter, I evaluated its usability, usefulness, and performance feasibility. The field trial and the interviews showed that users liked the features in AZURITE, and also provided valuable insights on what features should be improved and how. The formal user study suggests that this selective undo tool for code editors can make programmers more effective and efficient when they are performing backtracking tasks. The performance was measured to demonstrate the feasibility in various aspects: edit history collection, maintaining the dynamic segment information to be used for selective undo, and the timeline visualization. In addition, I showed various situations where selective undo mechanism could be useful while coding, and also shared my own positive experience of using AZURITE for my development.

These results show that the selective undo mechanism helps programmers when they face some backtracking tasks. Combining these results with the observations from the empirical studies of backtracking which showed that programmers backtrack a lot (Chapter 4), it can be concluded that the selective undo mechanism would improve the general productivity of programmers.

The idea of selective undo could be applied to many fields other than programming. As an example, the next chapter will discuss our investigation of design issues around applying selective undo in painting applications.

10. SELECTIVE UNDO SUPPORT FOR PAINTING APPLICATIONS[49]

Although the previous chapters discussed our selective undo mechanism in the context of code editing, there are many other types of applications where selective undo could be useful. In order to see if the selective undo idea can be applied to other domains and if the knowledge about selective undo issues can be transferred between different domains, we investigated selective undo in the painting domain.

10.1. MOTIVATION

The undo operation has long been understood to be a required command in applications, especially to support *creative exploration*. Studies have shown that when users have the ability to undo, they are more comfortable exploring and trying new commands [Kuttal 2011].

Ideally, any previous operation should be able to be undone. However, as discussed earlier, most applications use the same restricted linear undo model [Berlage 1994] which has limitations. A key reason more elaborate models have not become more popular is primarily due to the user interface challenges of presenting a more powerful undo model in a way that it is easy for the user to understand.

The limitations of linear undo have motivated research on *selective undo*, where the user can specifically select which of the previous operations to undo and which to keep (Section 2.1.1). However, this prior work has primarily focused on object-oriented drawing programs, like PowerPoint or Adobe Illustrator [Berlage 1994][Myers 1998], where selective undo is more easily achieved by selecting objects and restoring their properties. New work has started to address selective undo in text as in the previous chapters and in [Li 2003], but there is little work on selective undo in *painting* applications like Photoshop which lack the structures used by object editing programs, and where operations tend to affect overlapping spans of pixels. This makes it much more difficult for users to identify which operations to undo (since there may be many small edits) and to predict the outcome of selectively undoing something in the

[49] Significant portions of this chapter appeared in [Myers 2015]. Ashley Lai and Tam Minh Le conducted the initial semi-structured interviews, and mainly implemented the Aquamarine prototype. Throughout this chapter, I intentionally use the pronoun “we” instead of “I” to indicate the work was highly collaborative.

past. Furthermore, it is challenging to provide understandable user interfaces for any selective undo model. Note that throughout this chapter, we are deliberately distinguishing the term *drawing* from *painting*. Drawing programs have identifiable objects in the picture that can be selected and manipulated later, whereas painting programs convert objects into pixels immediately, and subsequent selections and operations are at the pixel level. Many painting operations cannot be naturally converted into objects with a distinct shape. For example, the paint bucket tool and filters do not create or even necessarily modify objects; instead they manipulate pixels. Note that modern image manipulation programs like Photoshop actually have a mixture of drawing and painting features. The prototype described in this chapter focuses on painting only.

Another interesting issue is the distinction between the *inverse* and *script* models of selective undo (Section 2.1.1). In the *inverse model*, supported by most previous selective undo systems [Berlage 1994][Myers 1998] and AZURITE, a selective undo causes the inverse of the selected operations to be added to the end of the history. For example when the user selectively undoes a delete, a create operation is performed and added to the end of the history. In contrast, the *script model* removes the selected operation from the history and performs all subsequent operations as if that operation had never been performed [Kurlander 1988]. This can cause semantic problems in drawing programs (for example, what is the meaning of a change-size operation applied to an object if the previous creation of that object is selectively undone, and thereby removed from the history?), but may be more appropriate when painting (since there are no objects to which operations are applied). We are not aware of any previous research on supporting selective undo in a paint program using the script model.

We created *Aquamarine*[50] (Figure 10-1) to study these issues. Aquamarine is a new prototype painting program that supports selective undo using the scripting model. We first performed semi-structured interviews with student and professional graphic artists to see what they would want and expect in such a tool. We then implemented the Aquamarine prototype based on those findings, and ran usability evaluations to assess its features and design decisions.

[50] Aquamarine is a gemstone, and here it stands for: **A**llowing **Q**uick **U**ndoing of **A**ny **M**arks **A**nd **R**epairs to **I**mprove **N**ovel **E**diting

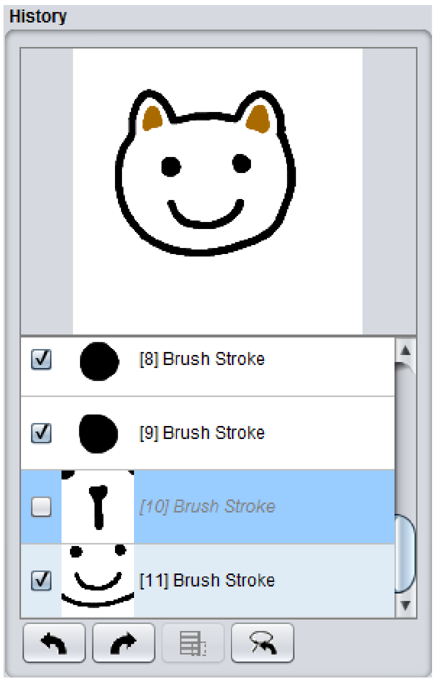


Figure 10-1. Aquamarine's history panel with operation #10 (brush stroke for the nose) selectively undone.

The contributions of this chapter include:

- Results of semi-structured interviews, which show that users have a significant need for selective undo capabilities in painting programs, and compensate with workarounds such as using many layers, saving the whole file to disk, or just starting over.
- A discussion of the design space for selective undo and tradeoffs among the design decisions.
- A design and implementation for a selective undo mechanism using the script model in a prototype paint program called Aquamarine, which explores the design space for how the script model should operate.
- A usability evaluation that demonstrates the aspects of the selective undo mechanism which are most usable and useful, and areas where more research is required. All users expressed a desire to have a form of selective undo in their everyday programs.

10.2. Initial Semi-Structured Interviews

In order to get a better idea about how people might use a selective undo mechanism in painting applications, we first performed semi-structured interviews of nine people. Our collaborator from Adobe (Joel Brandt) reports that there are three primary uses of Photoshop: (1) to design and prototype user interfaces, (2) as a painting tool to create art, and (3) to edit and retouch photographs. We were careful to recruit participants from all three groups (see Table 10-1).

ID	Experience	Student or Prof.	Des?	Art?	Photo?	Ctrl-Z	Step-Back	Hist. Panel	Hist. Brush
1	Intermediate	student	Yes	Yes			Yes	Yes	Yes
2	Intermediate	student	Yes	Yes		Yes			
3	Intermediate	student			Yes			Yes	
4	Expert	student	Yes			Yes	Yes		
5	Expert	prof.		Yes		Yes	Yes	Yes	
6	Novice	student		Yes			Yes		
7	Expert	prof.	Yes				Yes		
8	Expert	prof.	Yes				Yes		
9	Expert	prof.	Yes				Yes		

Table 10-1. Participants in our semi-structured interviews.

All of our participants were frequent users of Photoshop, with four self-rating as expert users, four as intermediate and one as novice. Four were professionals and five were students. Photoshop provides four undo mechanisms: an old-fashioned toggle undo (Ctrl-Z), which undoes and then redoes the last operation, a normal linear undo, which it calls "Step Backwards" and "Step Forwards" (Shift/Alt-Ctrl-Z), a history panel which enables undoing back multiple steps (with a fixed maximum limit that defaults to 20), and a "history brush" where users select a point in the history panel and then paints, which restores the pixels back to that point in time, as a form of regional undo. As previously mentioned in Section 2.1.5, Photoshop provides an optional non-linear history mode. No participant (in this or the final study of Aquamarine) had ever used or had even heard of this feature, which is turned off by default. Three of the participants reported using the toggle undo, five reported using step-backwards and step-forwards, three used the history panel, and one reported using the history brush.

In the first part of our interviews, we asked participants to work on one of their own tasks using Photoshop, as a form of contextual inquiry [Beyer 1997], and they were encouraged to think aloud. We told them we were particularly interested in situations where they would "explore different ideas and test out different alternatives."

The more experienced users exclusively used the shortcut keystrokes to invoke commands. Only two participants had the history panel displayed, but both of them used it to undo multiple operations. We did not see any use of the history brush. Some participants erased areas rather than using undo, whereas others made significant use of the step-backwards commands. They recognized the limitations:

I don't want to use the undo button if there's a part that I drew after that I like.

Another said:

It's more of a short-term memory [issue] for me. I don't usually undo more than five steps back.

Participants used the layer mechanism to group items together when they anticipated a need to selectively change an area or a shape, and experienced users grouped and named layers to keep the large number of layers organized. When beginning a creative exploration, participants often duplicated layers or saved a version of the whole picture, to facilitate backtracking. Users often hid layers instead of deleting them;[51] one said:

I don't like deleting things; it feels too permanent—even if it was a mistake.

In the second part of our interviews, we gave the participants a series of tasks designed to elicit selective backtracking behaviors and strategies. We told them to imagine they were creating images for a company that is run by a very indecisive boss. We would repeatedly instruct them to create something, then change it, then do other things, and finally to change it back. This cannot be supported with any of Photoshop's existing undo mechanisms. As in the first part, when participants anticipated that something would need to be undone, they put it on a separate layer. If we asked them to undo something that was not on a separate layer, they expressed regret that it was not separated and usually started over from scratch. Users considered using advanced editing tools such as Photoshop's Color Replacement Tool, but often decided it would not work sufficiently well, and just painted over items, or deleted and redrew them from scratch. These results reveal the reliance on layers to simulate selective undo, and the need to start over and lose desired work when undesired operations precede the desired ones.

In the third part of our interviews, we explained the concept of selective undo and asked if they thought it would be useful in their own work. Six participants said it would be useful frequently or almost every day, and the other three said maybe not since they achieved the same functionality with layers or by frequently saving their work to disk.

In the fourth part of our interviews, we asked a series of questions about what they thought would happen as a result of performing a selective undo. In particular, we were exploring whether they expected the *script model* or the *inverse model* of behavior. For example in Figure 10-2, if the stars were created by duplicating the original star, what should happen if the creation of the original star was selectively undone? All participants thought the other stars should *not* disappear, so they were endorsing the *inverse* model in this case, since under the script model, there would be nothing to duplicate. On the other hand, if we added a recolor step for the star between steps 1 and 2, and then selectively undid the recoloring, all but one of the participants expected the stars in step 2 to also change color, requiring the script model to hold. When this inconsistency was pointed out, most participants agreed that their preference would change based on the situation.

[51] This is consistent with the observations from our lab study of programmers' backtracking (Section 4.1.5.2). Hiding layers is comparable to commenting out existing code.

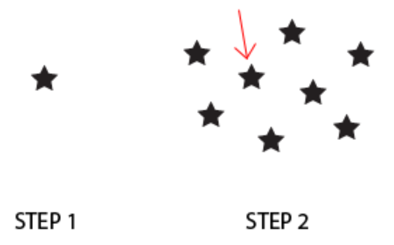


Figure 10-2. Multiple steps to create a drawing.

Finally, we explored a number of features that might be used in a selective undo mechanism. Most participants agreed that the ability to identify which operations had contributed to a selected region of the picture would be useful, as a way to find which operations to selectively undo. About half thought the reverse operation was needed – to select an operation in the history panel and highlight what area of the picture it affected. Few participants saw any need to search for commands in the history by name (to enable searches like "find the last time I used the brush tool"). Also deemed unnecessary would be a need to view different versions of a picture side-by-side, or automatic version control tools, as are commonly used for source code and which are provided for painting applications by third party tools like LayerVault[52] and Pixelapse[53].

10.2.1. Discussion

In an object-oriented drawing program, users can generally get the effect of undoing operations by manually performing the opposite operation. Thus, the "undo" of resizing can be achieved by just resizing it back, which can generally be performed at any time. Photoshop and other painting programs try to provide this capability for as many operations as possible by having many operations act like they are object-oriented instead of pixel based (that is, like a drawing program instead of a painting program). For example, text in Photoshop is kept as objects on its own layer until the user explicitly flattens it into a bitmap. However, this means that many bitmap operations, like blurring or erasing, would not be available until the image is flattened, and changing the text would not be possible after blurring.

Similarly, in our study, we saw participants using layers to try to make painting operations have the ability to be selectively edited. However, layers do have significant limitations. We saw frequent errors where participants would accidentally draw into the wrong layer, merge layers and regret it later, or realize too late that a new layer should have been created. Also, the large number of layers became difficult to manage.

[52] https://layervault.com/

[53] https://www.pixelapse.com/

In addition, users were wary of the Photoshop history tool, since it only keeps a limited number of steps backwards, the history is not preserved when a document is closed, and it can be difficult to identify which step to go back to. One participant noted:

If you rely on history, you're asking to get burned by it.

Therefore, there is an opportunity to provide a new way for users to backtrack in painting programs.

Participants agreed that there would be significant value in having a selective undo tool that would provide the ability to change what was done in the past, without the requirements of pre-planning to use layers and without giving up the advantages of bitmap editing.

10.3. Design Tradeoffs

In designing and implementing the selective undo feature, we had to make a large number of design decisions, across a number of different dimensions. In many cases, we needed to decide what a selective undo for painting applications should mean since this has never previously been explored. This section presents the design dimensions, the various options and tradeoffs, and justifications for decisions implemented in Aquamarine.

10.3.1. Which Operations to Explore?

We classified all Photoshop operations into eight categories with respect to their interaction with Undo (as has also been explored previously by others [Chen 2011]). The categories are:

1. Creation/painting, which cause pixels to be drawn
2. Local-adjustment tools, which change existing pixels
3. Global commands, which affect all pixels in the image
4. View control commands, which do not change pixels
5. Selection tools, which control subsequent operations
6. Mode changes, like picking a color or changing the current layer
7. Conversion tools, like Flatten and Convert to Smart Object
8. Miscellaneous, like 3D tools

The first three are the most relevant to selective undo, so we implemented at least two operations from each of these categories. We expect that operations in Photoshop or other full-featured paint programs would have identical issues with respect to selective undo, so we believe the HCI issues we have identified will generalize to all other commands.

10.3.2. Script Model versus Inverse Model?

The most important decision was which kind of selective undo model to support. As discussed above, the primary choices are to use some form of script model or some form of inverse model.

The inverse model seemed inappropriate for a painting program, since so many operations cannot be undone by adding a new command to the end of the history. For example, take color change. In a painting program, a color change using the Paint Bucket Tool (more formally called flood fill), changes the color of the area defined by the contiguous pixels that match, within a tolerance, the color of the clicked pixel. This operation often *cannot* be undone with a new flood-fill at the end of the history (that is, "now"). For example, in Figure 10-3, the flood fill of step 2 cannot be undone with another flood fill after all of the actions in step 3, because flood-filling now will only change parts of the former blue t-shirt, and the long sleeves added to the shirt might also be flood-filled. Similarly, other bitmap editing operations, like blurring the image, cropping, etc. may be impossible to reverse in the current state. Other operations, even creating new shapes, cannot necessarily be undone in the current state, because of other paintings drawn on top of them may compute the new pixels using the existing pixels.

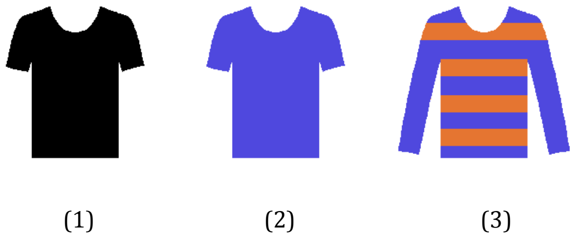


Figure 10-3. In a painting program, (1) paint a shirt, (2) flood fill it with a new color, (3) then do a variety of other actions.

However, a script model *can* be used to selectively undo the flood fill operation—we can undo all the operations back to before step 2 was done (so the image looks like step 1), skip doing step 2, and then do the remainder of the operations.

Note that using layers to separate operations as in Photoshop and other programs, so the user can simulate selective undo by turning off or deleting layers, will not help with selective undoing of painting-based operations like flood fill or blur, since these operations must work on the same layer as the pixels to be modified.

Since Aquamarine specifically focuses on these painting operations, we decided to use the script model. Another motivation is that the script model has rarely been investigated in previous research or commercial systems, and it brings up many more interesting and challenging design decisions, which are discussed next.

Note that providing selective undo with either model can work with the regular linear undo command. Thus, the selective undo/redo commands can be ignored entirely if the user does not use them, and the normal way of using undo/redo can be utilized.

Another issue is that some commands have side effects *external* to the editor. For example, File Save should not be re-executed each time the system reruns the script to perform a selective undo. Fortunately, it turns out that these operations are actually not put into the undo stack anyway (see also the discussion of copy and paste below). For a few *internal* operations with side effects, like Create Layer, we must disable all the other operations on that layer if the create is selectively undone.

10.3.3. Handling Region Conflicts among Operations

As discussed in Section 2.1.1, an important concern in object-oriented editing is dealing with dependencies and conflicts [Berlage 1994][Myers 1998][Cass 2005]. Bitmap operations do not have this kind of conflict, since they work on pixels, which will still be there. However, it is possible to have *region conflicts* (Section 5.1.2), which are where a later operation's scope overlaps the selected operation in such a way that it is not clear what the selective undo should do. For example, in Figure 10-3, if the user tried to selectively undo the painting operation (step 1), what should the flood fill (step 2) do, since the t-shirt is no longer on the screen?

There are a variety of possible outcomes for step 2 in a script model when step 1 is removed:

1. Cancel the selective undo due to the conflict. That is, do not perform the selective undo at all.
2. Perform the selective undo, and also undo the conflicting operations, which includes at least step 2.
3. Perform the conflicting operations as dictated by the script model with the selected operation removed. In this case, it would probably flood-fill the whole canvas since there would be nothing (only background) at the pixel where the flood fill operation happened.

There is actually a 4th possibility, which we discuss in Future Work: allow the user to *modify* the conflicting operation or *add* entirely new operations into the history. For example in Figure 10-3, the user might select a new pixel at which to apply the flood fill, or paint a brand new shape where the shirt used to be as a replacement step 1.

We investigated providing a popup dialog to help the user resolve these conflicts and pick one of the three options, as in AZURITE (Figure 8-4), but it proved quite difficult to identify exactly which operations should be marked as conflicting with the undone operation, resulting in very expensive pixel operations and too many false positives. Therefore, we decided to use a much simpler approach of just comparing bounding boxes of all subsequent operations, and highlighting the operations in the history which *might* conflict, to help the user find them if they want option 2 (see Figure 10-4).

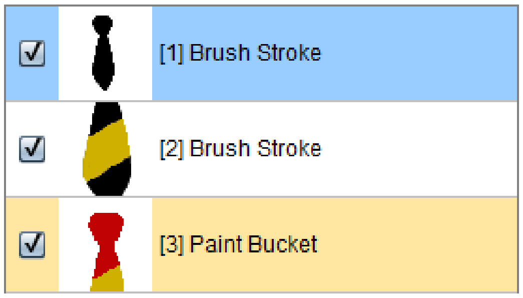


Figure 10-4. Highlighting operation 3 in orange since it conflicts with the selected operation 1 (shown in blue).

To reduce the number of operations that are highlighted, we developed a set of heuristics for the kinds of region conflicts that should be brought to the user's attention based on the operation categories discussed above in Section 10.3.1. The heuristics are as follows: first, we do not highlight creation/painting operations. For example, if a new shape is painted on top of an old shape, and the painting of the old shape is selectively undone, even though overlapping pixels are affected, we do not alert the user, and simply remove the old shape and repaint the new shape. Similarly, global operations affect all pixels, but we do not alert users to these region conflicts either. The main issue is when the later operation is a *local-adjustment tool*, which modifies only some of the pixels of the picture. Flood-fill and smudge are examples of this, and there are many others. When such operations affect the same area that the selective undo will affect, we alert the user of all conflicting operations using orange highlights.

As future work, we plan to support allowing the user to select *multiple* operations to be selectively undone together as in AZURITE. For example, the user might select both Steps 1 and 2 together in Figure 10-3. When all the region conflicts are internal to the selected operations, there is no ambiguity of what to do, so the user does not have to be alerted. In our usability evaluation, this was a much-requested feature.

10.3.4. Copy and Paste

Another interesting design issue revolves around copy and paste. In most existing applications, the copy operation is *not* put on the undo stack, and whatever is copied is retained independent of what the user subsequently undoes. For example, in Microsoft Word, a user can type some text and copy it, and then invoke undo one or more times so that the text is all removed. However, the clipboard will still retain the text, which can be pasted later. Of course a cut operation goes on the undo stack, but only the deleting part of the cut is undone—again the clipboard is not affected by undoing the cut operation. Users have developed strategies for clever ways to use this feature, so we decided to retain it in Aquamarine.

Therefore, Aquamarine makes what we call a *deep copy* of whatever is selected when the user performs a copy operation, and the copy operation is not put into the undo stack. That means that the script model will not re-execute the copy operation, so the clipboard will continue to have a copy of the pixels as they were originally copied, no matter what happens subsequently to that part of the image. Similarly, the paste operation retains a deep copy of what was in the clipboard when the operation is first invoked, so it continues to paste the *same* picture if it is reinvoked later due to the script model.

Consider this sequence of operations:

1. Paint a blue star
2. Recolor the star to be red with the paint brush
3. Copy the red star
4. Paste the clipboard into a new part of the screen
5. Selectively undo step 2

Given our model, the same picture will be pasted by step 4 even after the user in step 5 subsequently selectively undoes step 2, so the original star is blue, but the new pasted star stays red.

Another reason we felt that we needed to adopt this model for copy is that the clipboard is globally shared by all programs, and the user might change the clipboard by doing a copy in another program, and in that case, we would not want to change the clipboard as a result of a selective undo.

In the above example, if the user copied a picture of a circle in PowerPoint between steps 4 and 5 above, when the system reruns the operations as part of the script model, we do not want the clipboard to be changed, and we want the paste in step 4 to continue to paste a star and not a circle.

10.3.5. SELECTIVE UNDO/REDO OPERATIONS IN THE HISTORY PANEL

In Topaz [Myers 1998] and AZURITE and many other systems that implement the inverse model, the selective undo and redo operations themselves are added to the end of the undo history, and shown in the history panel (see Figure 10-5). This seems to make sense in those programs since the inverse operation is actually performed at that point in the history, and so a representation like Figure 10-5 should reinforce this mental model for users. Another advantage of this approach is that the selective undo operation itself can be selected and undone, which would then add another selective undo to the end of the history, restoring the effect of the original operation. Alternatively, the user could select the original operation (#5 in Figure 10-5) and selectively *redo* that operation, which has the same effect as selectively undoing the undo. A final advantage of this design is that it is clearer what a regular linear undo command will do – it works up from the bottom, undoing each of the operations in turn (selective and regular operations alike).

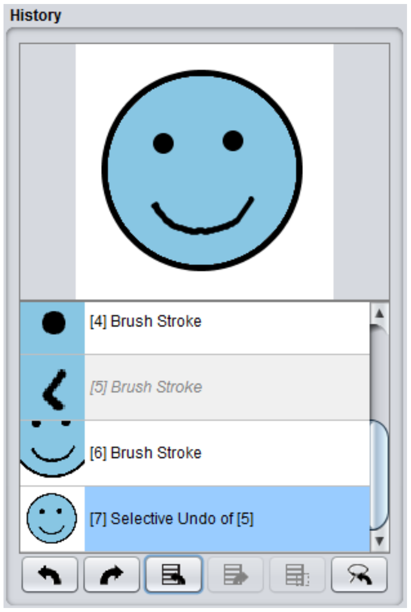


Figure 10-5. An alternative form of history panel where selective undo/redo operations are included in the history panel.

Given these advantages, we first tried adapting this style of history panel for the script model of Aquamarine, as shown in Figure 10-5. In this design, the user can select an operation (e.g., #5), and invoke selective undo on it, which adds the selective undo operation to the end of the history (shown as operation #7). Unlike in the inverse model, the selective undo is not really on the history, but instead the referenced operation is skipped (in Figure 10-5, only operations 1 through 4 and 6 are executed). We try to visualize this by graying out the referenced operation. If the user wants to reverse this operation, the selective undo (#7) can be selectively undone or the original operation (#5) can be selectively redone, and these operations would be also added to the history. However, we felt it was getting complicated to understand what was done and not done, and we also felt that this design might give the wrong mental model to users, since the selective operations are not really in the history.

Therefore, we designed a different history panel, shown in Figure 10-1, where the original operations can simply be unchecked to selectively undo them, or rechecked to be selectively redone. This matches the design for the layers panel in Photoshop, where layers can be made visible or not with a toggle button next to each layer's name. This should do a better job of matching the appropriate mental model for users: that the current picture results from the operations being executed from the top down, skipping operations that are not checked. Another advantage is that we do not need explicit selective undo or selective redo commands, but instead the user just toggles the appropriate checkbox.

It is less clear what the regular linear undo should do in this model. We decided linear undo should undo the most recent operation shown in the history panel that is still in effect. Therefore, the selective undo is not directly undoable by Ctrl-Z. This again mimics the default behavior of turning on and off layers, which are not put on the history stack by default (although

Photoshop has a setting to enable this). Similarly, the regular linear *redo* moves down the history stack re-enabling operations, no matter how they were undone. Thus, it redoes operations that were disabled either by selective undo or regular undo. Of course, the user can always go to any individual operation and toggle its checkbox, no matter how it was disabled.

Since the "right" design for this feature is not clear, we decided to include questions about this in our usability evaluation, discussed below, which came out strongly in favor of the checkbox version shown in Figure 10-1.

10.3.6. Thumbnail Images

One small design issue is what to display for each operation in the history panel. A common complaint about Photoshop's history panel from our initial semi-structured interviews was that it only shows an icon for the tool used, so it is impossible to tell which operation is for which part of the drawing, for example when there are many brush strokes. Therefore, we wanted to show a thumbnail representation of what the operation did, along with some context (i.e., surrounding pixels), as has been done in earlier systems [Kurlander 1988].

A new complication for Aquamarine that has not been previously reported arises from the scripting model of selective undo—what should be shown in the thumbnail when a selective undo causes a previous operation to be turned off? Should the thumbnails of the subsequent operations be changed? Updating all the thumbnails to make them correctly represent what the operation does in the current state should make it clearer the effect of undoing it. However, updating all the thumbnails might unduly slow down selective undo, especially when an early operation is selected in a long history, and keeping the thumbnails constant might make them more recognizable for users if they go back to find a remembered operations. We decided to update all the thumbnails, since that is the more correct (and interesting) design. Ironically, in the usability evaluation, users almost universally said they did *not* want *any* context to be shown in the thumbnails, completely eliminating this problem. Instead, they preferred seeing only the specific output of the current operation. The preview window (the upper panel of Figure 10-5), however, should continue to show what the complete picture would look like up to the selected operation.

10.3.7. Identifying Desired Operations

AZURITE provides elaborate ways to *search* for the operations that the user might want to undo (Chapter 8). However, in our initial semi-structured interviews, the key way that users wanted to search for operations in a painting program was by selecting an affected region on the screen. Therefore, we provide commands in Aquamarine for identifying the set of operations that affect the selected region on the screen. We also provide the reverse operation which shows the region on the screen that any operation affects. Note that both of these are based on the specific pixels on the screen, which may no longer hold that operation's result. For example, if the user paints a shape, but later selects and moves it, the region for the original painting operation will remain where the shape started. In a painting program, it seems impossible to identify where the pixels that result from an arbitrary operation might have gone, and it seems more useful to help users find where things used to be, in case they want to get back to that state. Note that if the user selects the region on the painting where the shape was originally painted, both the painting and the move operations will be identified, so users can selectively undo/redo any operations that are desired.

10.4. Implemented System

Our original hope was to implement Aquamarine as a plugin to Photoshop since it supplies a variety of APIs. Unfortunately, none of Photoshop's APIs provided sufficient access to the undo stack and the result of each operation to support the desired features. Therefore, we needed to find an open-source paint program we could modify. We required sufficient functionality so that the design issues discussed above would emerge (which eliminated simple painting programs like the Java Paint program used in Section 4.1.2), and we needed access to the full source to be able to implement the selective undo.

We selected Pixelitor 2.0.0[54] in Java by László Balázs-Csíki (see Figure 10-6). The original version of Pixelitor used the built-in Java Swing undo model, which uses command objects [Myers 1996][Gamma 1994]. We were able to replace this with our own undo implementation which supports our novel selective undo mechanism.

One complication is that Pixelitor, like other painting programs including Photoshop, implements undo by simply having most operations save a bitmap of the picture before the operation is executed. This is sufficient for the regular linear undo model, since it can always be sure that when it places the bitmap into the correct position, that the context will be correct and the full image will be restored. However, this is insufficient for any selective undo model, especially for the script model, since we need to be able to re-execute all commands later. Therefore, we had to modify each command to remember all of its parameters so it can be re-executed when needed. For example, the Shape and Brush stroke tools must save the path and all the properties of the pen (color, transparency, etc.). Similarly, the Paint Bucket tool needs to remember the color, start pixel, etc. Further, we had to refactor the application so all operations could get their parameters from the global widgets (like the current color) or from the saved command objects. We still save the bitmaps so they can be used for regular undo.

[54] http://sourceforge.net/projects/pixelitor/

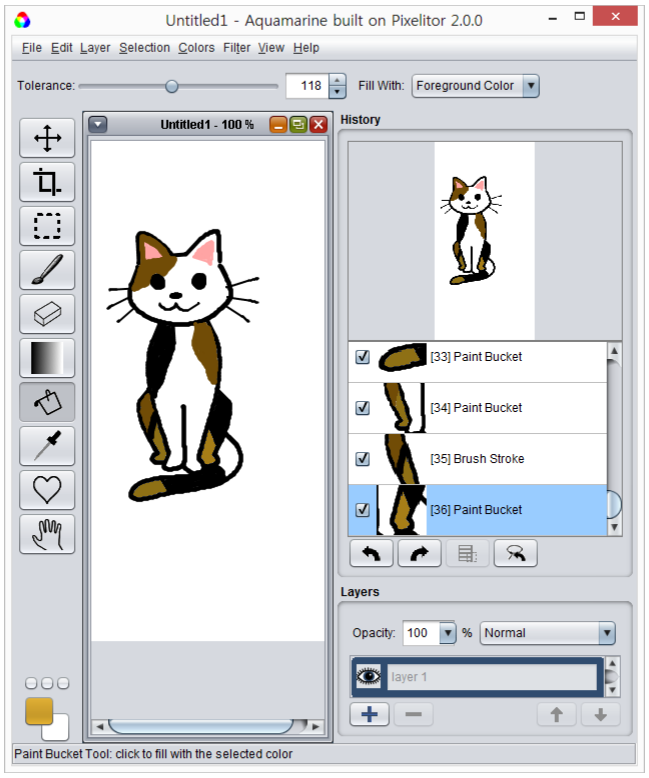

Untitled1 - Aquamarine built on Pixelitor 2.0.0
File Edit Layer Selection Colors Filter View Help
Tolerance: 118 Fill With: Foreground Color
Untitled1 - 100 %
History
[33] Paint Bucket
[34] Paint Bucket
[35] Brush Stroke
[36] Paint Bucket
Layers
Opacity: 100 % Normal
layer 1
Paint Bucket Tool: click to fill with the selected color

Figure 10-6. Pixelitor modified with our history panel.

Another implementation tradeoff was whether each operation should save the entire bitmap of the whole picture, or only the portion affected by this operation. If each operation saved the entire picture, then our script model could implement the selective undo of an operation by simply reinstating the full picture before that operation, skipping the operation, and then re-performing all enabled subsequent operations. However, saving the entire bitmap wastes a lot of memory, since most operations only affect a tiny part of the picture. Pixelitor only stores bitmaps for the regions affected by each operation (which it uses for undo), which we retain. Therefore, implementing the script model selective undo does not require any significant amount of extra space. The tradeoff is that for selective undo, we must internally roll back to the selected operation, by doing a linear undo all the way back, in order to restore the picture to the original state. This is a classic space-time tradeoff, and in practice, we found it to be sufficiently fast. If it turns out not to be fast enough in the future, there are many obvious optimizations that could be implemented.

Some Photoshop operations can be quite slow and expensive, so there is a question about the efficiency of re-applying all the operations each time a selective undo is invoked, which is required for the script model. In our unoptimized prototype implementation, selective undo slows down noticeably on a big image if there are over 100 operations. Although we focus on the user interface aspects here, we have considered some performance optimizations that could be applied in a real implementation. Obviously, the length of the history can be limited, as is already the case in Photoshop, but that is not desirable from a UI standpoint. Instead, slow operations could be approximated on down-sized images, to show quick previews which estimate what the results will be, with the real result calculated in the background once the user stops undoing and redoing.

10.5. Usability Evaluation

Since our initial semi-structured interviews and other prior work had clearly identified problems that our mechanism addresses, we felt the key issue to evaluate was whether users could understand and successfully apply our new script-model selective undo, and whether users can think of strategies that would make effective use of it. Therefore, we felt that the appropriate evaluation would be to do think-aloud usability evaluations of various versions of the features which embody the design decisions discussed above.

We recruited eight participants (see Table 2), one of whom was also part of our initial semi-structured interviews (participant 5 in Table 1 is participant 2 in Table 2). We specifically picked users across a wide range of experience. None of the new participants were students. Four were professional graphic artists and the other four were professionals who only used Photoshop or equivalent occasionally. All participants were familiar with conventional undo models, as supported by Photoshop and most applications, but three participants were not familiar with Photoshop's special undo features.

We asked the participants to "try drawing a few things" with Aquamarine, and then undo operations using a variety of commands. Most participants just drew random lines and shapes, but p5 drew animals like Figures 10-1 and 10-6. We then asked them what they would expect to happen with various undo commands (regular Ctrl-Z and our new selective undo) and then to try them out to see what happens. We had about half of the participants use the checkbox version of the history first (Figure 10-1), and the other half tried the in-history version first (Figure 10-5). Then they tried the other version. The order had no effect, and *all* 8 participants strongly preferred the checkbox version. Everyone understood the operation of the checkboxes and how they would affect the picture, although some expressed a preference to using an "eyeball" icon as used to toggle layer visibility (see Figure 10-6). In the in-history version, they found the presence of the "selective undo" and "selective redo" operations in the history to be confusing, and also felt that this would clutter up the history without being useful.[Hong 2000]

ID	Experience	Prof.	Des?	Art?	Photo?	Ctrl-Z	Step-Back	Hist. Panel	Hist. Brush
1	novice	other			Yes	Yes	Yes		
2	expert	prof.		Yes		Yes	Yes	Yes	
3	expert	prof.	Yes			Yes	Yes	Yes	
4	expert	prof.	Yes	Yes		Yes	Yes	Yes	
5	intermediate	prof.	Yes			Yes	Yes	Yes	
6	novice	other	Yes			Yes	Yes		
7	intermediate	other	Yes	Yes		Yes	Yes		
8	expert	other			Yes	Yes	Yes	Yes	

Table 10-2. Participants in our usability evaluation.

All users found the script model to be understandable and preferable in general, but all users were surprised by the operation of the paint bucket, when the area underneath where it was applied was selectively undone (a situation in which there is a region conflict). In this case, the paint would typically fill the whole background, which was not something anyone wanted. Some participants wanted the paint bucket to just remember the shape it had filled and reuse that, even if the area was no longer there, but most people wanted the system to be smarter about undoing the paint bucket operation with the previous operation. In discussions, however, they agreed this would be tricky in practice, and that the orange highlighting of the conflicting operations (Figure 10-4) would be a reasonable way for users to manually find and fix conflicts.

Interestingly, all but one participant was surprised by the operation of copy-and-paste, even though everyone was familiar with the way copy-and-paste works in other programs (where the copy ignores undo). However, participants agreed that our design was consistent and would sometimes be useful.

The main requested feature was a way to *group* actions. For example, selecting multiple operations together and undoing them all at once, or collecting operations into named groups, like layers can be in Photoshop.

In summary, the evaluation showed that with the check-box version, the script-model selective undo was understandable and usable, and that people understood how the issues from the semi-structured interviews would be addressed by our tool. All participants expressed a strong desire to have this kind of selective undo in Photoshop, and even in their other editing programs. They said that selective undo could substitute for some of the ways they now use layers.

10.6. Conclusion

Providing selective undo in a painting program brings up a surprising number of design challenges. Aquamarine shows that UI research can address those challenges and usable interfaces can be provided that can have benefits for users. We hope this research will inspire others to investigate providing selective undo in a variety of domains.

11. LIMITATIONS AND FUTURE WORK

This chapter investigates possible future work directions. The research ideas include extensions to the methodologies and tools introduced throughout the dissertation, and applying the selective undo idea to other programming tools and other domains (beyond painting described in Chapter 10).

11.1. EXTENSIONS TO FLUORITE

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 a researcher is interested, but it would be even better if there was a way of detecting such patterns without writing code. For instance, researchers could express the coding patterns that they want to detect in a state machine or a regular expression.

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). I speculate that FLUORITE might be useful for analyzing usability problems if it was possible to distinguish the commands executed by keyboard shortcut from the ones executed by clicking menu items for example.

Providing FLUORITE for other popular IDEs is a possible direction of future work. This would enable the collection of usage data from a larger programmer pool and looking at issues that cross IDEs and languages. To this end, FLUORITE is being ported to a cloud-based JavaScript IDE called Cloud9[55] by Andrew Faulring at CMU. The ported logging tool is called CRYOLITE[56], and is currently used by the researchers of the Exploratory Programming Group[57] for their empirical studies of end-user programmers.

As mentioned in Section 2.3.1, there are growing number of tools like FLUORITE that capture fine-grained coding behaviors in IDEs. Unfortunately, most of the tools were developed independently, and the data formats are not compatible among the tools. In the future, creating a public data corpus similar to EUSES spreadsheet corpus [Fisher 2005] would be a great resource to the software engineering research community. Moreover, having a standard data

[55] https://c9.io/

[56] Cryolite stands for: **C**loud9 **R**ecorder of **Y**our **O**perations by **L**istening to **I**nteractions in **T**he **E**ditor.

[57] http://www.exploratoryprogramming.org/

format for these kinds of loggers would make it possible to reuse certain log analyses across multiple data collection tools.

11.2. EXTENSIONS TO AZURITE

11.2.1. *USING STRUCTURAL CHANGES AS INPUT*

While there are development tools which use the abstract syntax tree (AST) level changes of the code as input [Robbes 2007][Omori 2008][Negara 2014], AZURITE uses the textual code changes instead. There are trade-offs between these two choices. On the one hand, by using textual changes as input, the mechanism becomes language agnostic, as is the conventional undo command and most commercially available merge tools [Mens 2002]. Besides, there are certain types of edits that cannot be captured at the AST-level, such as reformatting code or changing a comment section. By using the textual input, our system can capture and undo these types of changes as well.

On the other hand, by using AST-level changes, the selective undo mechanism could use the additional information and better handle *semantic conflicts*. A semantic conflict can occur when a set of edit operations are semantically related to each other in the code. For example, when a method is renamed, its definition and all the call-sites must change together. Selectively undoing only one of these rename operations would not cause any regional conflicts because the changes were all made in totally different locations. However, this would result in inconsistent code containing compile errors and thus could be considered to result in semantic conflicts.

Since these edits are likely to have been performed close together in time, users may be able to easily select them together in the timeline. However, if not, I believe the compile errors would identify these kinds of problems and users would be able to perform the remaining backtracking steps to complete what they wanted to achieve. This situation was observed during the formal user study (Section 9.2), especially in the backtracking step of T5. In order to keep the added GUI control correctly, the users had to keep both the member field declaration and the code for creating the object and adding it to the panel. Some participants only kept the creation code and forgot to keep the declaration, resulting in a compilation error. All of them immediately realized their mistake by reading the error message and went to the field declaration location and performed selective undo there to restore the code.

Even though it was found that these kinds of conflicts are easy to detect and fix, adding features to AZURITE to handle them directly by augmenting the structural information of the changes would be beneficial. In fact, as shown in the longitudinal study of backtracking (Section 4.3) and the edit operation collapsing work (Chapter 7), the textual changes can easily be transformed into AST-level changes with a parser, which means that the tool could use both the textual and the structural changes as input.

11.2.2. Dealing with Moved or Copied Code

The current selective undo mechanism does not treat "move" as a primitive operation; it simply treats the move as separate delete and insert operations when a block of code is moved from one place to another. The moved code then loses the fine-grained history in the new location, which may not be desirable. To address this problem, the selective undo tool could maintain the history of the moved code in the new place, and treat the move operation as an undoable primitive operation. This would require the internal dynamic segment information to be carefully updated.

Copying some block of code and pasting it in a new location is not treated in a special way, either. For the copied code, it is not obvious whether the edit history of that region should also be copied. This issue could be investigated with actual users as we did in the Aquamarine project (Section 10.3.4).

11.2.3. Combining Code Edit History with Regular Code Searches

Modern IDEs provide a rich set of code search features such as "find references" to help programmers navigate the code base. Leveraging the code edit history information kept in AZURITE into the code search mechanisms could be useful for the programmers. For instance, programmers could search for all of the **`println`** statements and sort them by their last edited time to find recently added debugging statements.

In addition, I believe many of the semantic conflicts can also be detected with this approach. For example, suppose that a variable name changes from **`foo`** to **`bar`** in multiple places. When the user selectively undoes the variable renaming in only one place, there will be no regional conflicts but the code will result in a state with compile errors. In order to mitigate this problem, the tool could 1) search for all occurrences of the **`bar`** variable using "find references" feature of Eclipse, 2) search for all recent edit operations in all regions where **`bar`** appear either by sorting them by edit time or using history search. This could detect the variable renaming regardless of how it was done; whether it was renamed with the refactoring tool or manually.

11.2.4. Code Recycle Bin

One of the use cases of selective undo is restoring some deleted code that later turned out to be needed, as observed in the preliminary lab study (Section 4.1) and from my own use case (Section 9.5). In AZURITE, deleted code can be restored in various ways, depending on what the programmer remembers about the deleted code. For example, the desired code could be found using the timeline, history search, or code history view. However, if the programmer wants to quickly review the recently deleted code blocks, a different user interface might work better: a code recycle bin. Just like the recycle bins of the recently deleted files used in most operating systems, the code recycle bin would collect all the deleted code blocks larger than a minimum threshold size, and display them sorted by time or position in the file. Users could review the deleted blocks of code with various filtering mechanisms, select one of the deleted code block and ask to restore that code to its original location or to copy the code to the clipboard.

11.2.5. IMPROVING HISTORY SEARCH

Our current timeline visualization displays an arbitrarily long edit history, and it can be difficult for the users to pick the right set of operations manually. Although the most common kinds of searches are supported through the history already (Section 8.3), it would be even more helpful if more kinds of search options were provided. The goal is to enable users to express whatever they remember about the previous edits or situations that they want to select in the history. I tried to gather possible search options from the actual users, but this attempt was not very successful. When people were asked what they were looking for when backtracking, they could not answer very well because they had forgotten the exact backtracking situation by the time they were being asked. In the future, the firehouse research method [Rogers 2010] could be used to interview the people immediately after they backtrack their code, as used in a study of software bug fixes [Murphy-Hill 2013]. Moreover, a diary study of backtracking could be conducted in order to capture the moments where programmers want to restore some code but do not actually perform backtracking, because they think it might be too difficult to do.

Although there is currently no solid evidence, I believe that it would be useful to allow users to search for points in the history when:

- the application was run or debugged,
- a specific unit test, or all tests, passed or failed,
- version control commands were executed,
- a particular point in real time ("last Thursday"), or a range of time ("last week"),
- a specific or a sequence of edit operations happened (e.g., copy-and-paste from the web, an Extract Method refactoring),
- a particular task was being completed (e.g., as tracked by task management systems such as Mylyn [Kersten 2006]), or
- any combination of these.

Note that the information for the first four bullet items are already available in the timeline, but not integrated as history search options. Using these options, one could search for "all edits since the last commit related to println statements," for example.

11.2.6. SUPPORTING A TEAM DEVELOPMENT ENVIRONMENT

One limitation of AZURITE is that it only handles a single programmer's edits. By storing who made each change, and sharing the edit history among the team members, it could help with answering more history related questions such as "who modified this piece of code most recently?" [Fritz 2010]. In order to achieve this, it would be best if the edit history was kept in the version control system along with the code, and AZURITE would need to be able to deal with the code merging situations.

11.2.7. SUPPORTING VARIATION MANAGEMENT

There are tools for managing multiple variations such as Juxtapose [Hartmann 2008] or Parallel Pies [Terry 2004], as discussed in Section 2.2.3. One of the limitations of these tools is that users must know in advance when they want to add variants. AZURITE has potential to be integrated with these variation management systems and overcome this problem. When a user selectively undoes some edit operations, the tool could ask the user whether she wants to create a new variant with the undo operation, in which case the version before and after performing the undo operation would be considered as two variants. Such variation information could be encoded using Choice Calculus [Erwig 2011] as the underlying model. The tool could also support editing multiple variants at once while editing the code using the techniques such as linked editing [Toomim 2004] or projectional editing [Walkingshaw 2014].

11.2.8. INTEGRATION WITH DYNAMIC EXECUTION INFORMATION

The timeline visualization of AZURITE displays an icon whenever the application under development is run by the programmer. This feature could be improved by augmenting the run events with the dynamic execution trace, for example, as recorded by Timelapse [Burg 2013], to enable users to review different runs, replay the past runs, or to find out the output produced during those run events. Furthermore, the history search could also provide search options to enable users to search for the code when the execution behaved in certain ways or produced certain outputs, for example.

11.2.9. DEPLOYING AZURITE FOR GENERAL USE

Although I conducted a small field trial of AZURITE (Section 9.1) and made AZURITE publicly available (see Section 12 for the URL), it has not been widely deployed for general use. Making the tool more robust and having a long term field study to gather feedback from more users would provide us better insight on how to improve the user interfaces for selective undo and what other features could be implemented on top of the core selective undo mechanism of AZURITE.

11.3. EXTENSIONS TO AQUAMARINE

Currently, Aquamarine is an early prototype, sufficient to explore the design issues discussed in Chapter 10, but not yet ready for deployment. The current work includes making the rest of the features of Pixelitor work with selective undo, and then releasing Aquamarine as open source for general use and a full field test.

Photoshop has an option to add the operations that enable and disable layers onto the undo history, and Topaz [Myers 1998] even could put changes of selections and the “find” operation into the undo history. Some of our participants expressed the desire to have state changes that do not affect the current painting, such as changing the current color or the radius of the brush, included into the history so they could be undone. We propose to explore these in the future.

We also want to investigate how to allow operations in the past to be *modified*. For example, some participants in both studies expressed a desire to be able to change the color of an operation done in the past, and see that propagated through the rest of the edits. The next step is to enable *new* operations to be inserted into the past, or existing operations to be reordered, for example to put something behind another painting in the stacking order. Although other research systems have tried some of these [Kurlander 1988], a key usability challenge remains of how the user would be able to understand, undo and selectively undo changes to the history.

Another requested feature from our participants was some way to collapse long painting histories. For example, text editors coalesce multiple keystrokes into a single undoable action, and we would like to explore the ability to collapse and expand multiple small brush strokes, similar to the real-time edit collapsing mechanism used in AZURITE (Chapter 7). This might be supported both automatically and manually, so the user could achieve the desired level of granularity when navigating.

Finally, we would like to explore saving and restoring histories. The history might be stored in the image document, to enable cross session undoing. This could also enable someone later to explore how an effect was achieved. Similarly, a saved script could be converted into a tutorial [Chi 2012]. Sections of a history could also be selected and converted into parameterizable reusable macros, as in Topaz [Myers 1998].

11.4. APPLYING SELECTIVE UNDO TO OTHER TOOLS AND DOMAINS

The AZURITE tool is specifically implemented as an Eclipse plug-in. The selective undo mechanism presented in Chapter 5 is language agnostic and the selective undo features in Chapters 6 and 8 work well regardless of the programming language used in the source file. However, the real-time edit collapsing mechanism presented in Chapter 7 only works for Java programming language, because the collapse test logic involves parsing the source files and identifying different parts of the code. Implementing AZURITE for other IDEs and languages would benefit a greater pool of users. In addition, the selective undo mechanism and interfaces have a great potential to be well integrated with other programming tools. For instance, it could be integrated with a task management system like Mylyn [Kersten 2006], and then it could retroactively map recent edits to a certain task, in case the user forgot to switch to another task context. Another example would be integrating the idea with CodeBubbles [Bragdon 2010b][Bragdon 2010a] to show only the edits belonging to a certain bubble or allow selectively undoing operations within a bubble. In fact, the main author of CodeBubbles implemented the regional undo feature (Section 8.2) in CodeBubbles after seeing the AZURITE system.[58]

Throughout the dissertation, the idea of selective undo was investigated first in the context of source code editing, and then in painting applications. Selective undo could also be applied to many other types of editors, such as 3D graphic editors, word processors, web page editors, audio editors, and so on. It is very likely that a different set of design issues would have to be considered for each of these domains, and careful designs and user studies would be needed in order to successfully build selective undo systems for these domains.

[58] Steven Reiss, personal communication, March 17, 2015.

12. CONCLUSION

Identifying inefficiencies and pain points within the workflow of programmers and solving those problems is important for improving programmer productivity. One way of doing it is to treat the programmers as the *users* of programming tools, and apply the well-established human computer interaction (HCI) methodologies to identify problems, develop solutions, iteratively improve the solutions, and evaluate the results through user studies. This research approach has been successfully used by our research group, and produced multiple PhD dissertations (e.g., [Ko 2008a][Stylos 2009][LaToza 2012]).

Throughout this dissertation, I investigated the issue of programmers' backtracking using the research approach described above. There existed some evidence that programmers backtrack their code, but little was known about this backtracking behavior. I have conducted a series of user studies to understand this behavior better. In the course of conducting these studies, I have also developed a tool for recording all the coding activities of the participants. The studies revealed that programmers face backtracking situations frequently, but the existing tools are providing only limited support for backtracking. The contributions from these studies are summarized in the following list:

- FLUORITE: Fine-Grained Coding Event Logger for Eclipse (Chapter 3)
  - Created FLUORITE as a logging tool that captures all the low-level coding activities from Eclipse, including the invoked commands and the fine-grained code changes.
  - Created a log analyzer for FLUORITE, which helps understanding the generated log files.
  - Demonstrated its usefulness as a convenient data collection tool for all the user studies presented in this dissertation (except for the online survey presented in Section 4.2).
  - Publicly released FLUORITE so that other researchers may use it for their own studies.
  - Used it as the input source of our selective undo tool AZURITE.
- Findings from the Preliminary Lab Study of Backtracking (Section 4.1)
  - The most frequently used commands are typing and code navigation commands (e.g., arrow keys), followed by the backtracking commands such as undo and delete.
  - Programmers habitually comment out code instead of deleting, for the following three purposes: 1) to re-enable the code later, 2) to keep the code as a template, 3) to keep the code as a bad example.
  - Even the programmers who said they often comment out code accidentally deleted some code that turned out to be needed later.

  - Programmers have difficulties in finding the right code to be backtracked, and in restoring some deleted code.
- Findings from the Online Survey of Backtracking (Section 4.2)
  - Programmers report they face various backtracking situations at least sometimes.
  - Different backtracking strategies are used for different backtracking situations.
- Findings from the Longitudinal Study of Backtracking (Section 4.3)
  - Devised an analysis method which keeps the evolution history of individual abstract syntax tree nodes separately, which was used to analyze 1,460 hours of coding logs.
  - Programmers backtrack 10.3 times per hour on average.
  - Size of a backtracking varies a lot: from 1 character to 1000+ characters, from simple parameter changes to significant algorithmic changes.
  - 34% of the backtrackings are performed *manually* with no tool support.
  - 97% of the backtrackings are performed within the same editing session.
  - 9.5% of the backtrackings are selective: linear undo could not handle them.

As a solution to the problem of lacking tool support for backtracking, I devised a selective undo mechanism for code editors, which was challenging in many ways. The semantics of selective undo was defined even in the presence of edit operation conflicts. In order to discover how to make the selective undo tool usable, I developed a plugin for Eclipse, and iteratively improved the user interface design through user feedback. The effectiveness of the selective undo tool was evaluated through a formal comparative user study. The contributions related to this selective undo tool are as follows:

- Selective Undo Mechanism for Code Editors (Chapter 5)
  - Devised a novel selective undo mechanism for code editors, which involves correctly maintaining the dynamic segments of the past edit operations and applying inverse operations to the target operations when the selective undo is invoked.
  - Defined the regional conflicts of text edit operations, and classified them into four categories to correctly handle each case when the selective undo is invoked.
  - Provided a user interface that supports undoing multiple operations at once, which is not only convenient, but also greatly reduces the need for manual conflict resolution.
- Timeline Visualization of Fine-Grained Code Edit History (Chapter 6)
  - Designed the timeline visualization which displays the fine-grained code edit history.
  - Users can use the timeline to review the code changes, and to select some of the past operations and invoke editor commands such as selective undo.
  - The timeline has an IDE-independent implementation using the standard web development technologies and an embedded browser widget.
- Real-Time Edit Operation Collapsing Mechanism (Chapter 7)
  - Designed a real-time collapsing algorithm that takes a stream of code edits and smartly collapses the fine-grained edits to abstract them to more conceptual edits.
  - Designed a code change distillation algorithm that can always match the code elements between two versions correctly even in the presence of renaming, by taking the actual edit operations into account, in addition to the two snapshots of code.

    - Supports a total of four collapse levels, which is tightly integrated with the timeline using a semantic zooming feature.
- User Interfaces for Selective Undo (Chapter 8)
    - Designed and implemented a set of user interfaces for selective undo, including the code history diff view, the regional undo shortcut, the history search, and the interactive selective undo dialog. These UIs were improved based on user feedback.
- Evaluation of AZURITE (Chapter 9)
    - Conducted a field trial of our prototype selective undo tool AZURITE, which inspired a number of new features and improvements to the existing user interfaces.
    - Conducted a formal A vs. B evaluation study with 12 users, which showed that AZURITE users can perform certain backtracking tasks twice as fast compared to just using conventional Eclipse features.
    - Analyzed the performance of different components of the tool which demonstrated its feasibility for real, long-term use: the log file size, edit history management, edit collapsing, and the various operations in the timeline visualization.

We took it one step further and applied the idea of selective undo in the context of graphical painting applications, which gave us a new set of challenges and design issues. Again, we first conducted formative semi-structured interviews to understand the problem better, developed the prototype tool implementing selective undo in a painting application, and then evaluated how usable the tool is for different groups of users. This shows that ideas from programming tool research can even be extended to other domains (as was also done with Crystal [Myers 2006] from the Whyline [Ko 2004][Ko 2008b]). The contributions of this work are summarized below:

- Selective Undo for Painting Applications (Chapter 10)
    - The results from our initial contextual inquiries, showing the need for better undo support in painting applications such as Adobe Photoshop.
    - Investigated the script-based selective undo model for painting applications, and built the Aquamarine prototype tool.
    - Identified various design issues from the user studies and during our implementation of the Aquamarine prototype.
    - Conducted a usability study with the prototype showing that users can understand and use the selective undo feature.

My hope is that these selective undo tools, combined with other existing programming tools, will help programmers achieve their daily programming tasks more effectively. By having the selective undo tools, programmers would be able to explore different code designs with confidence, since they know they can always revert those changes later whenever they want. In turn, I would expect this would make the programmers more creative and productive, and facilitate *exploratory programming*, which is believed to be a good way of learning or designing APIs or software systems [Fritzson 1986][Gundel 2005].

I also hope that the techniques and methods from research on the human aspects of programming are applied to more and more software engineering problems, so that programmers may be more productive with improved programming tools that are actually useful for them.

Two of the prototype tools introduced in this dissertation, FLUORITE and AZURITE, are open-source and publicly available at the following locations. The hope is that these tools will be useful for the community.

FLUORITE: Eclipse logging plug-in for capturing code edits and IDE interactions

• Project Page:	http://www.cs.cmu.edu/~fluorite/
• Source Code:	https://github.com/yyoon/fluorite-eclipse/
• Analyzer:	https://github.com/yyoon/fluorite-analyzer/

AZURITE: Selective undo tool for Eclipse code editors

• Project Page:	http://www.cs.cmu.edu/~azurite/
• Source Code:	https://github.com/yyoon/azurite-eclipse/

APPENDIX A: MATERIALS FROM THE PRELIMINARY LAB STUDY

A.1. TASK INSTRUCTIONS FOR GROUP 1

Introduction

You are going to add new features on a simple *'Paint'* program, which is written in the Java language and using the Swing toolkit. This program is composed of 10 Java files, and the project is already open in the Eclipse IDE on the screen.

You can run the program by clicking the Run button (▶), or by pressing Ctrl + F11 key. Try running the program and see how the program works.

Now, you will be given several tasks. While you are performing these tasks, you should THINK ALOUD. This means that you should tell me about whatever you are looking at, thinking, doing, and the reasons why you are doing that.

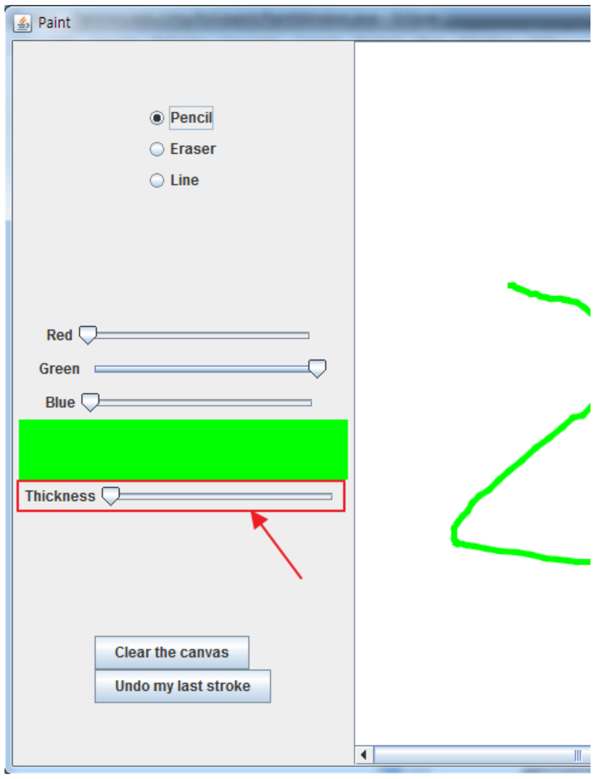
If you do not think aloud for a while, I will remind you to do so.

Task 1.

Suppose that your boss asked you to add a thickness control to the *Paint* program, using a slider widget. The user should be able to set the thickness value from 1 to 9, inclusive.

It should work with both the Pencil tool and the Line tool.

The following is an example screenshot of what it should look like when you are done.

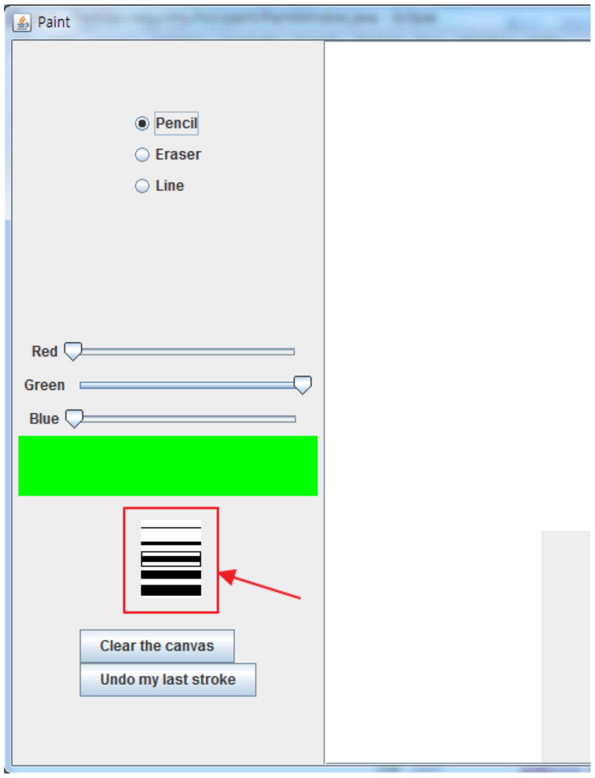


Please implement this feature, test it enough to make sure that it works correctly, and tell me when you think it is done.

Task 2.

After seeing your implementation, your boss changes her mind and asks you to implement the thickness control in an alternative way. This time, implement the thickness control by providing thickness buttons, each previews the corresponding thickness. There should be 5 buttons, of which representing the thicknesses of 1, 3, 5, 7, 9.

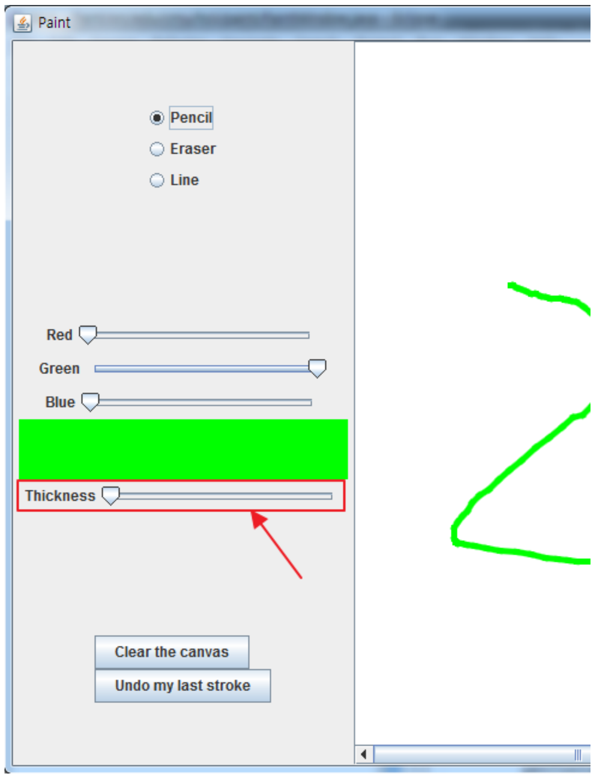
The following is an example screenshot of what it should look like when you are done.



Please implement this new design instead, test it enough to make sure that it works correctly, and tell me when you think it is done.

Task 3.

Your boss decides to adopt the first version, which was implemented using the slider control.



Please make it look like this again, test it enough to make sure that it works correctly, and tell me when you think it is done.

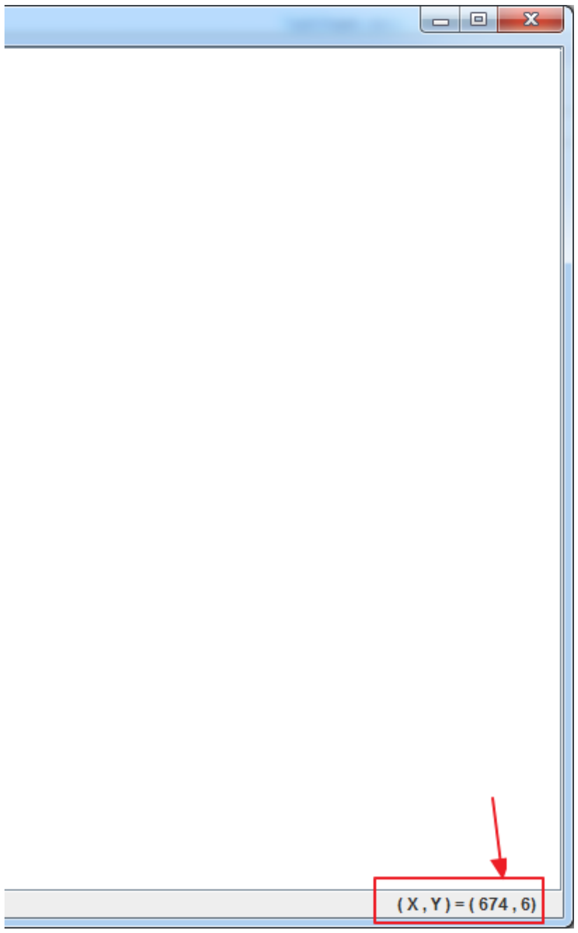
Task 4.

This time, you are asked to implement a totally new feature:
an (x, y) coordinates indicator.

The (x, y) coordinate values should be updated whenever the mouse cursor is moved over the canvas, and they should be measured relative to the upper-left corner of the canvas area.

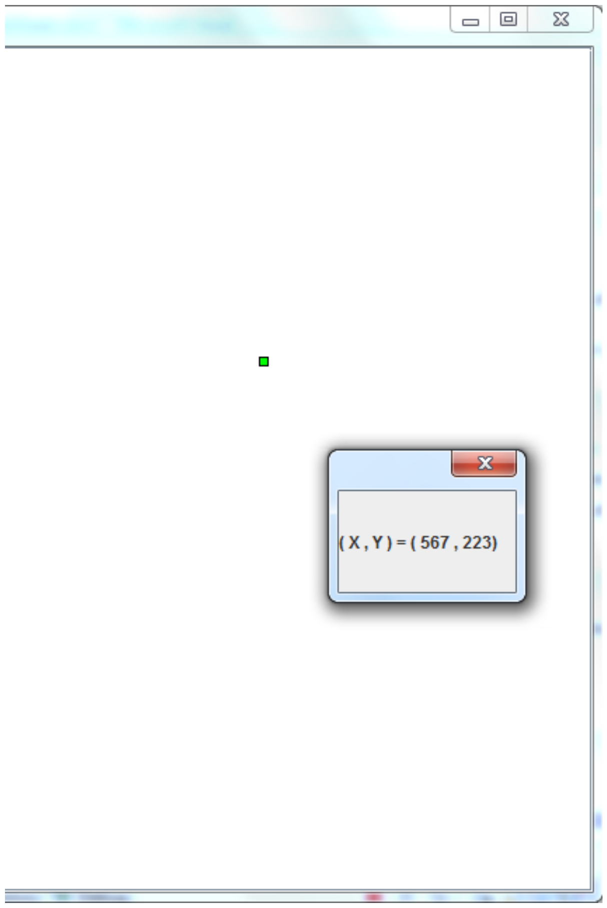
As you did in Tasks 1 & 2, your boss wants you to implement this feature in two different ways. Since only one of them will be adopted, you are required to produce two different versions of the code, each of which has only one of the implementations.

The first way of implementing the (x, y) coordinates indicator is placing a status bar at the bottom of the window as following:


(X , Y) = (674 , 6)

Task 5.

The second way of implementing the (x, y) coordinates indicator is using a *modeless tool dialog* which can be moved by the user at any time.



Please implement these 2 versions, test them enough to make sure that they work correctly, and tell me when you think they are done.

A.2. Task instructions for Group 2

Introduction

You are going to add new features on a simple *'Paint'* program, which is written in the Java language and using the Swing toolkit. This program is composed of 10 Java files, and the project is already open in the Eclipse IDE on the screen.

You can run the program by clicking the Run button (▶), or by pressing Ctrl + F11 key. Try running the program and see how the program works.

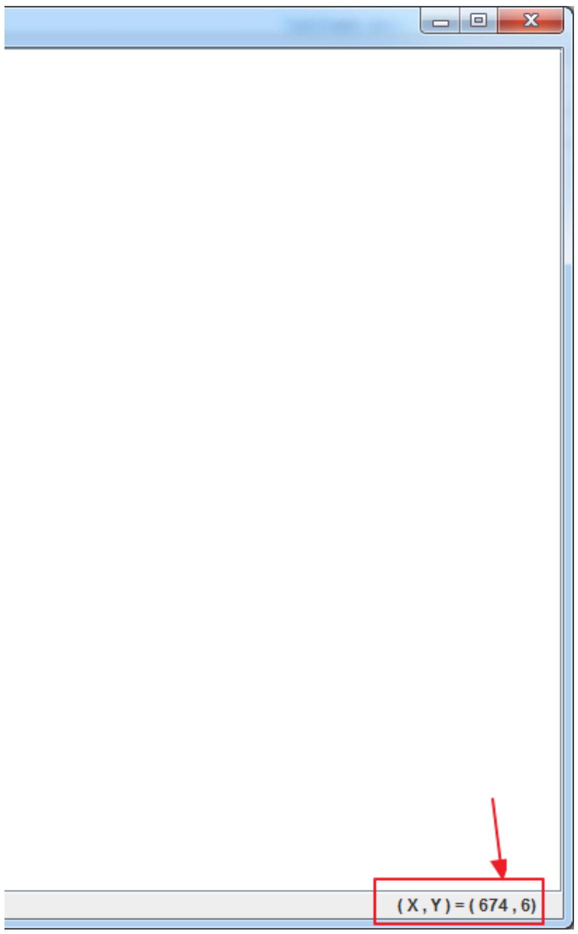
Now, you will be given several tasks. While you are performing these tasks, you should THINK ALOUD. This means that you should tell me about whatever you are looking at, thinking, doing, and the reasons why you are doing that.

If you do not think aloud for a while, I will remind you to do so.

Task 1.

Suppose that your boss asked you to add an (x, y) coordinates indicator to the *Paint* program, by placing a status bar at the bottom. The (x, y) coordinate values should be updated whenever the mouse cursor is moved over the canvas, and they should be measured relative to the upper-left corner of the canvas area.

The following is an example screenshot of what it should look like when you are done.

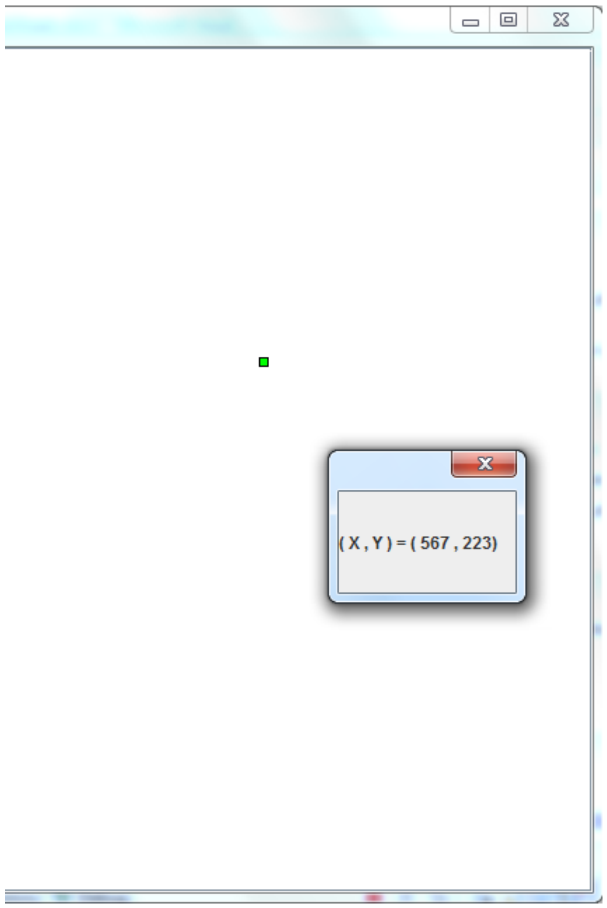


Please implement this feature, test it enough to make sure that it works correctly, and tell me when you think it is done.

Task 2.

After seeing your implementation, your boss changes her mind and asks you to implement the (x, y) coordinates indicator in an alternative way. This time, implement the (x, y) coordinates indicator using a *modeless dialog* which can be moved by the user at any time.

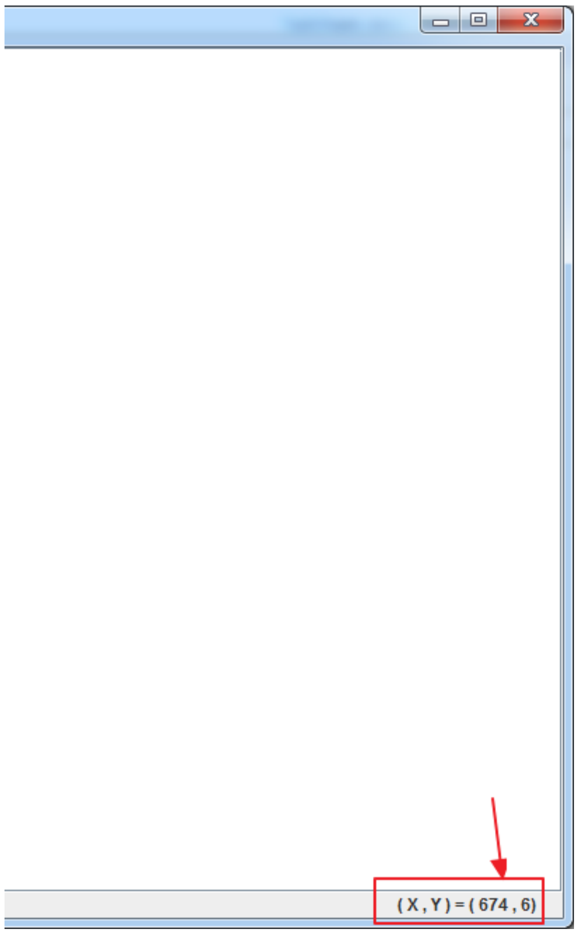
The following is an example screenshot of what it should look like when you are done.



Please implement this new design instead, test it enough to make sure that it works correctly, and tell me when you think it is done.

Task 3.

Your boss decides to adopt the first version, which was implemented using the status bar.



Please make it look like this again, test it enough to make sure that it works correctly, and tell me when you think it is done.

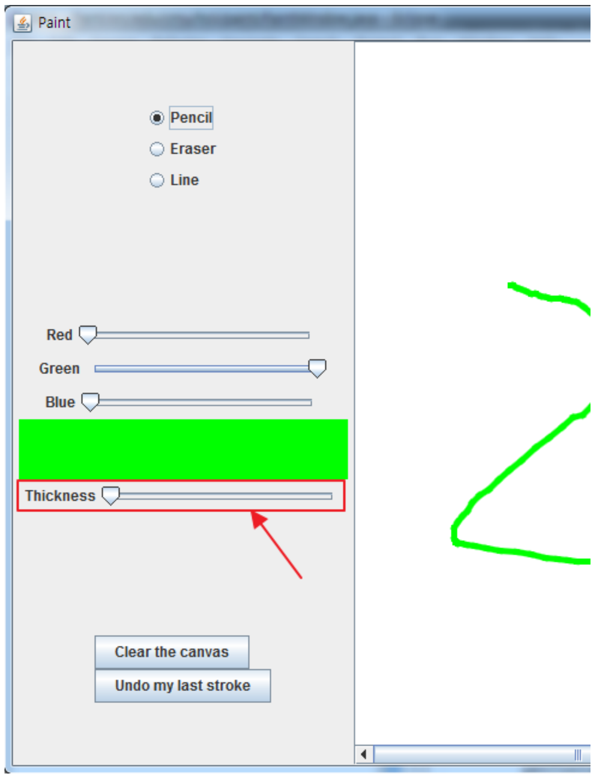
Task 4.

This time, you are asked to implement a totally new feature:
a thickness control.

The user should be able to set the thickness value from 1 to 9, inclusive. It should work with both the Pencil tool and the Line tool.

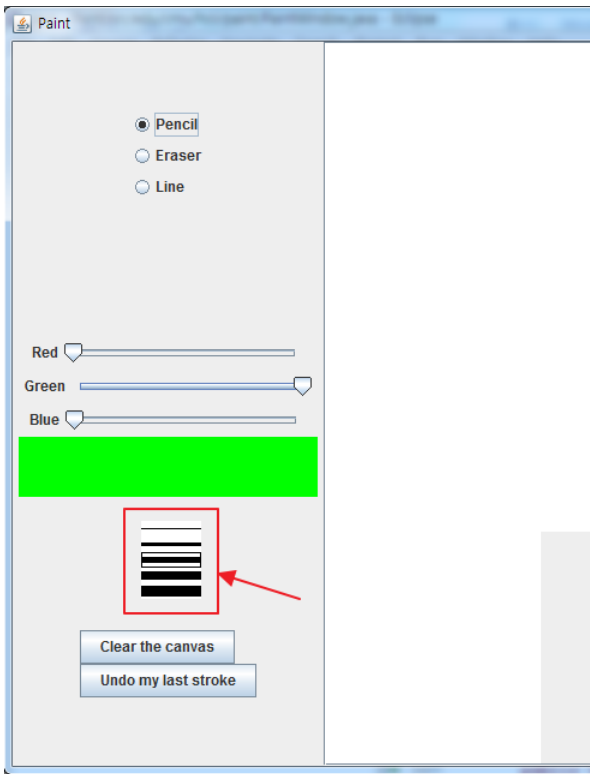
As you did in Tasks 1 & 2, your boss wants you to implement this feature in two different ways. Since only one of them will be adopted, you are required to produce two different versions of the code, each of which has only one of the implementations.

The first way of implementing the thickness control is using a slider widget as following:



Task 5.

The second way of implementing the thickness control is providing thickness buttons, each previews the corresponding thickness. There should be 5 buttons, of which representing the thicknesses of 1, 3, 5, 7, 9.



Please implement these 2 versions, test them enough to make sure that they work correctly, and tell me when you think they are done.

A.3. Questionnaire

Gender / Age

□ Male □ Female | Age:__________

How long have you been programming in general?

□ None □ < 1 year □ 1-3 years □ 3-5 years □ 5-7 years □ 7-9 years □ >= 9 years

Are you majoring (or did you) in Computer Science or an equivalent discipline?

□ Yes □ No

Have you ever worked for a company as a professional programmer?
If so, please specify how long you have worked.

□ Yes: □ No

How much experience do you have with the Java programming language?

□ None □ < 6 mon. □ 6-12 mon. □ 12-18 mon. □ 18-24 mon. □ >= 2 years

How long have you used the Eclipse IDE?

□ None □ < 6 mon. □ 6-12 mon. □ 12-18 mon. □ 18-24 mon. □ >= 2 years

If you normally use other IDEs or text editors for code, please specify:

How much experience do you have with Java Swing toolkit?

□ None □ < 6 mon. □ 6-12 mon. □ 12-18 mon. □ 18-24 mon. □ >= 2 years

If you have experience with other GUI toolkits, please specify:

Have you ever used the `JSlider` class before?

□ Yes □ No

Have you ever used the `JButton` class before?

□ Yes □ No

Have you ever tried subclassing a GUI widget before?

□ Yes □ No

We are studying the issue of ***backtracking*** in general (removing typing that was recently put in, because you decided it was not appropriate for some reason).

How often do you think you do backtracking when you are doing the following coding activities? What techniques do you think you use for each kind of backtracking?

When typing in code, backtracking due to **typos, mistyping, or other small mistakes**:

This happens to me:					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never

When this happens, I use the following strategy to fix it: (for each strategy, check how often you think you use it. Feel free to check as strategies as apply):

Undo (^Z):					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never
Backspace and/or delete keyboard keys:					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never
Selecting with the mouse, and deleting or overtyping:					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never
Commenting out the code:					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never
Using file save before the change, and then loading in the old version of the file:					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never
Using a version control system like CVS to revert to an older version of the file:					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never
What other strategies do you use to fix **typos**:					

Backtracking due to **trying to figure out how to use API calls** (trying different API calls or different parameters to API calls, to learn how they work):

This happens to me:					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never

When this happens, I use the following strategy to fix it: (for each strategy, check how often you think you use it. Feel free to check as strategies as apply):

Undo (^Z):					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never
Backspace and/or delete keyboard keys:					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never
Selecting with the mouse, and deleting or overtyping:					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never
Commenting out the code:					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never
Using file save before the change, and then loading in the old version of the file:					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never
Using a version control system like CVS to revert to an older version of the file:					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never
What other strategies do you use to **figure out APIs**:					

Backtracking due to **trying to figure out how which algorithm to use** (trying different algorithms):

This happens to me:					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never

When this happens, I use the following strategy to fix it: (for each strategy, check how often you think you use it. Feel free to check as strategies as apply):

Undo (^Z):					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never
Backspace and/or delete keyboard keys:					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never
Selecting with the mouse, and deleting or overtyping:					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never
Commenting out the code:					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never
Using file save before the change, and then loading in the old version of the file:					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never
Using a version control system like CVS to revert to an older version of the file:					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never
What other strategies do you use to **figure out which algorithm to use**:					

Backtracking due to **<u>design flaws in the code base</u>** (a new feature cannot be easily added in a modularized way):

This happens to me:					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never

When this happens, I use the following strategy to fix it: (for each strategy, check how often you think you use it. Feel free to check as strategies as apply):

Undo (^Z):					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never
Backspace and/or delete keyboard keys:					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never
Selecting with the mouse, and deleting or overtyping:					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never
Commenting out the code:					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never
Using file save before the change, and then loading in the old version of the file:					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never
Using a version control system like CVS to revert to an older version of the file:					
	□ Pretty much every time	□ Frequently	□ Sometimes	□ Rarely	□ Never
What other strategies do you use to **<u>fix the design flaws in the code base</u>**:					

What are some other reasons you end up backtracking when you are coding?

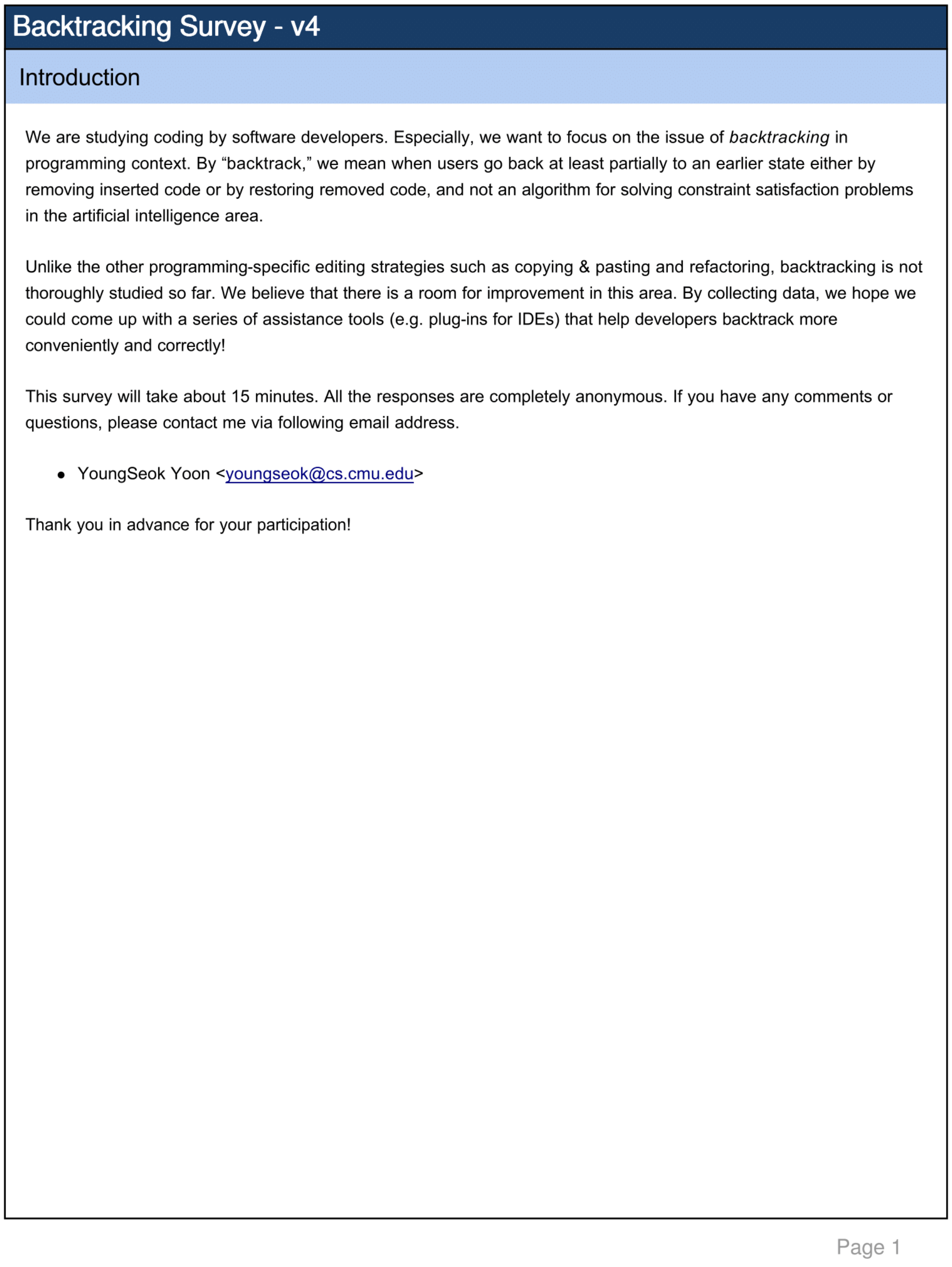
____

____

____

____

APPENDIX B: QUESTIONNAIRE USED FOR THE ONLINE SURVEY



Backtracking Survey - v4

Introduction

We are studying coding by software developers. Especially, we want to focus on the issue of *backtracking* in programming context. 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.

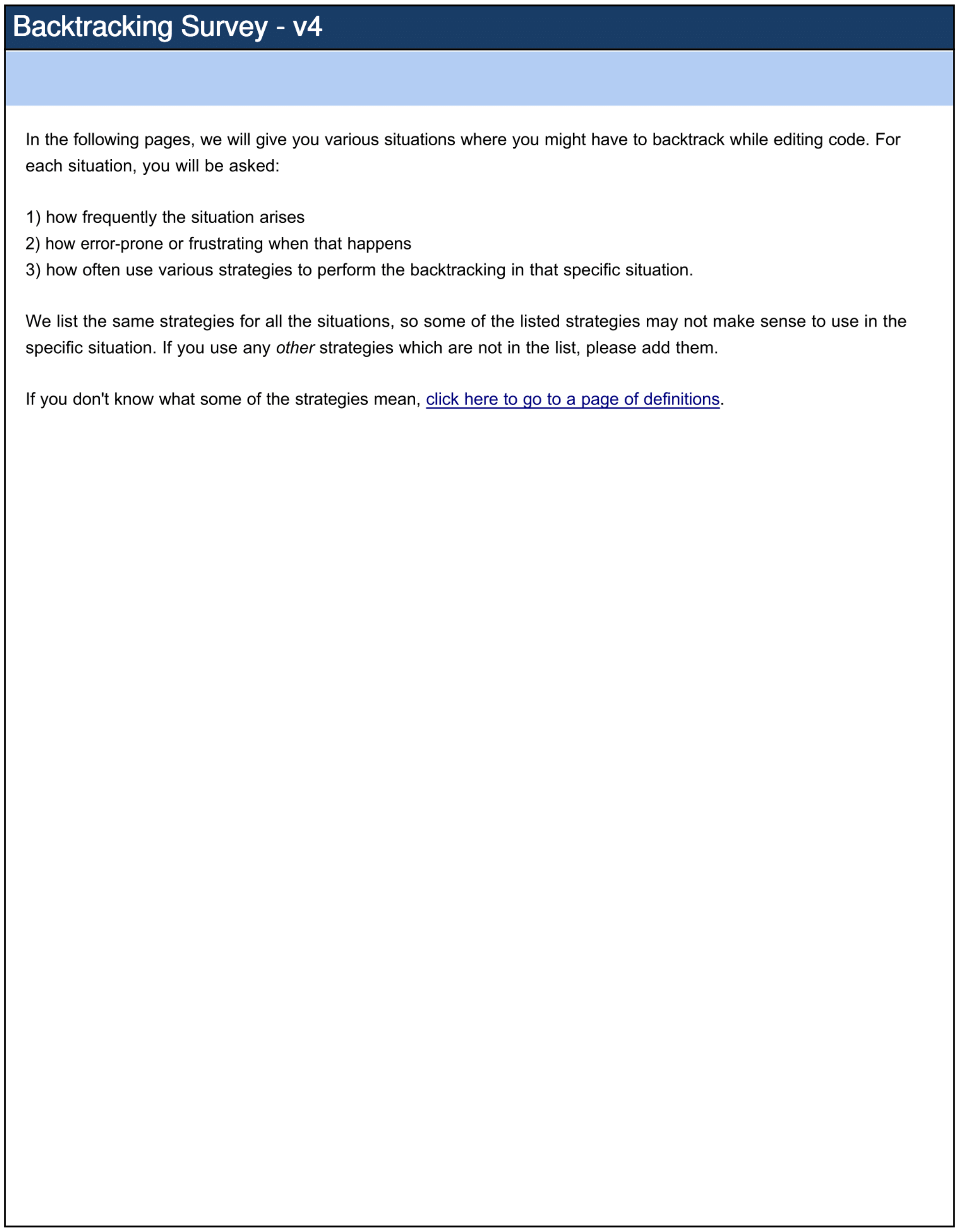
Unlike the other programming-specific editing strategies such as copying & pasting and refactoring, backtracking is not thoroughly studied so far. We believe that there is a room for improvement in this area. By collecting data, we hope we could come up with a series of assistance tools (e.g. plug-ins for IDEs) that help developers backtrack more conveniently and correctly!

This survey will take about 15 minutes. All the responses are completely anonymous. If you have any comments or questions, please contact me via following email address.

- YoungSeok Yoon <youngseok@cs.cmu.edu>

Thank you in advance for your participation!

Page 1



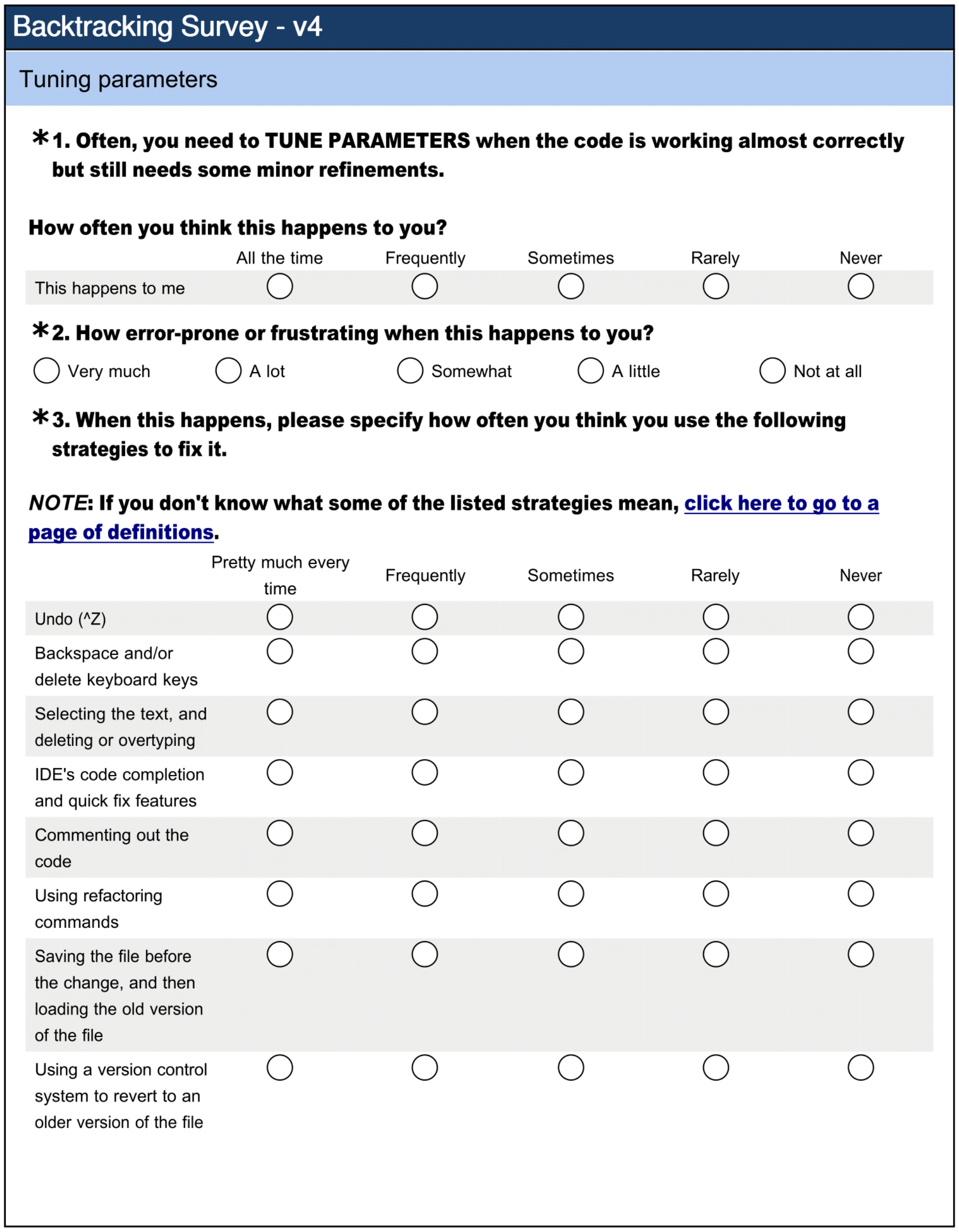
Backtracking Survey - v4

In the following pages, we will give you various situations where you might have to backtrack while editing code. For each situation, you will be asked:

1) how frequently the situation arises
2) how error-prone or frustrating when that happens
3) how often use various strategies to perform the backtracking in that specific situation.

We list the same strategies for all the situations, so some of the listed strategies may not make sense to use in the specific situation. If you use any *other* strategies which are not in the list, please add them.

If you don't know what some of the strategies mean, click here to go to a page of definitions.



Backtracking Survey - v4

Tuning parameters

***1. Often, you need to TUNE PARAMETERS when the code is working almost correctly but still needs some minor refinements.**

How often you think this happens to you?

	All the time	Frequently	Sometimes	Rarely	Never
This happens to me	○	○	○	○	○

***2. How error-prone or frustrating when this happens to you?**

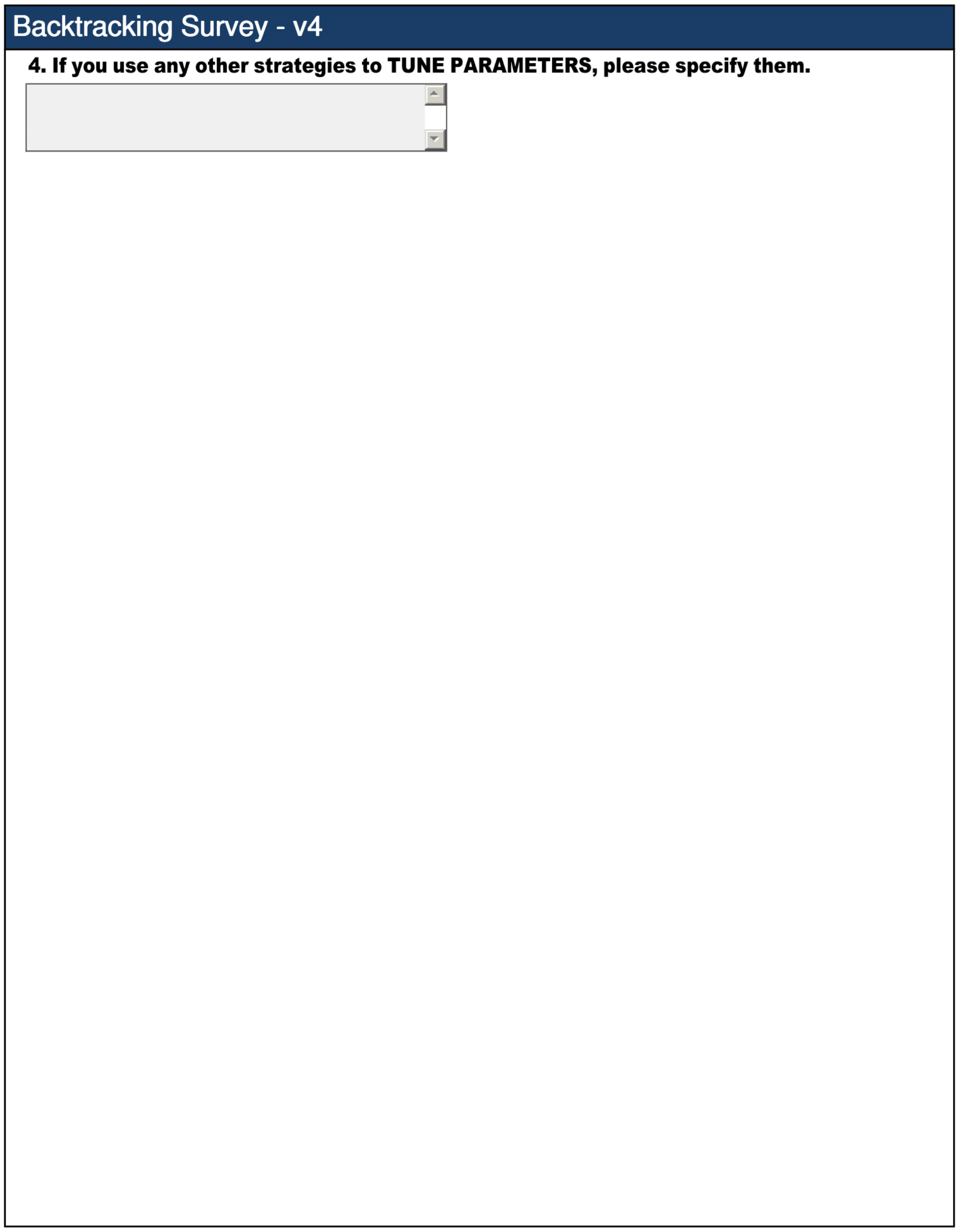
○ Very much ○ A lot ○ Somewhat ○ A little ○ Not at all

***3. When this happens, please specify how often you think you use the following strategies to fix it.**

***NOTE*: If you don't know what some of the listed strategies mean, click here to go to a page of definitions.**

	Pretty much every time	Frequently	Sometimes	Rarely	Never
Undo (^Z)	○	○	○	○	○
Backspace and/or delete keyboard keys	○	○	○	○	○
Selecting the text, and deleting or overtyping	○	○	○	○	○
IDE's code completion and quick fix features	○	○	○	○	○
Commenting out the code	○	○	○	○	○
Using refactoring commands	○	○	○	○	○
Saving the file before the change, and then loading the old version of the file	○	○	○	○	○
Using a version control system to revert to an older version of the file	○	○	○	○	○

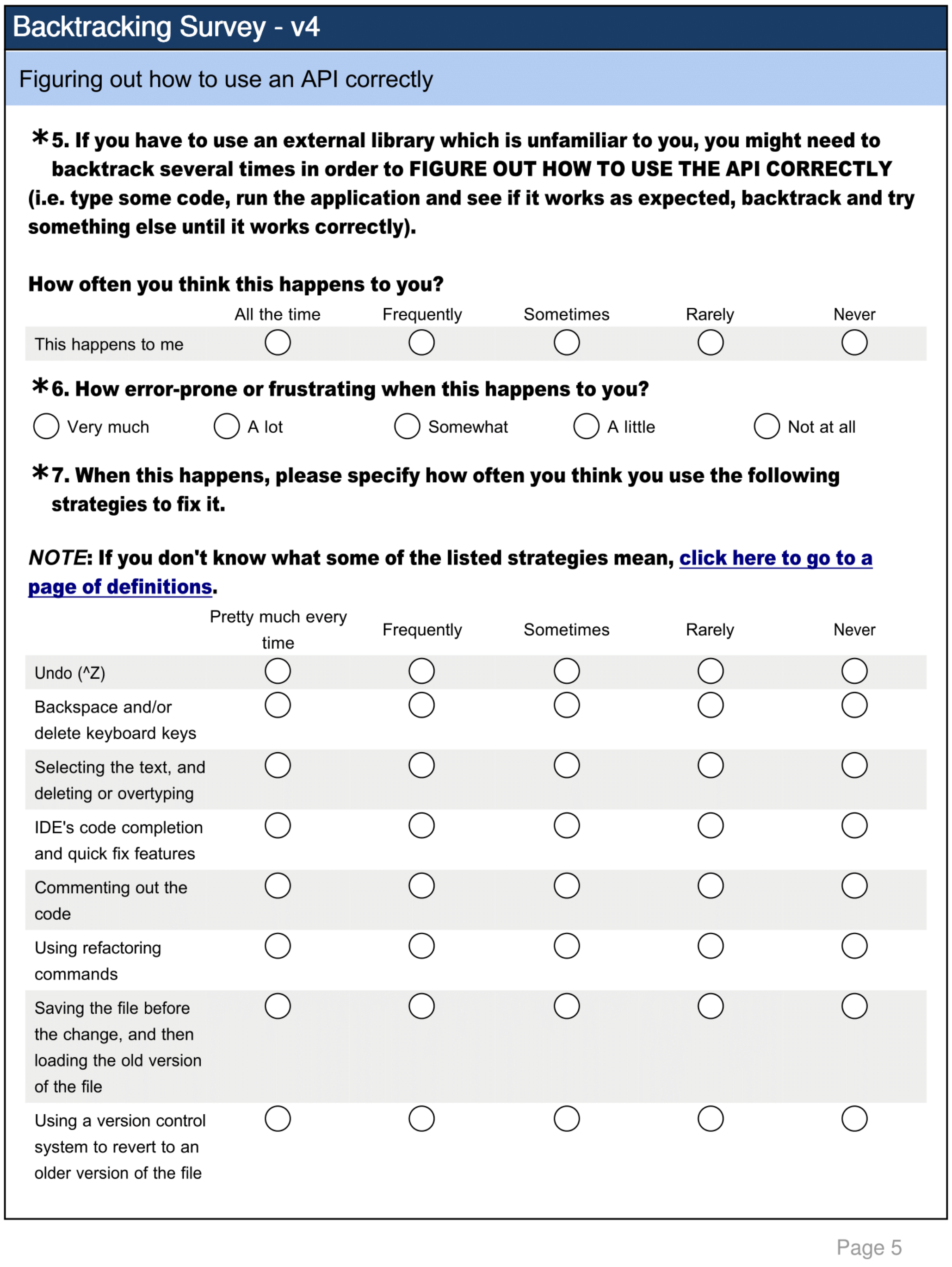
Page 3



Backtracking Survey - v4

4. If you use any other strategies to TUNE PARAMETERS, please specify them.

Page 4



Backtracking Survey - v4

Figuring out how to use an API correctly

***5. If you have to use an external library which is unfamiliar to you, you might need to backtrack several times in order to FIGURE OUT HOW TO USE THE API CORRECTLY (i.e. type some code, run the application and see if it works as expected, backtrack and try something else until it works correctly).**

How often you think this happens to you?

	All the time	Frequently	Sometimes	Rarely	Never
This happens to me	○	○	○	○	○

***6. How error-prone or frustrating when this happens to you?**

○ Very much ○ A lot ○ Somewhat ○ A little ○ Not at all

***7. When this happens, please specify how often you think you use the following strategies to fix it.**

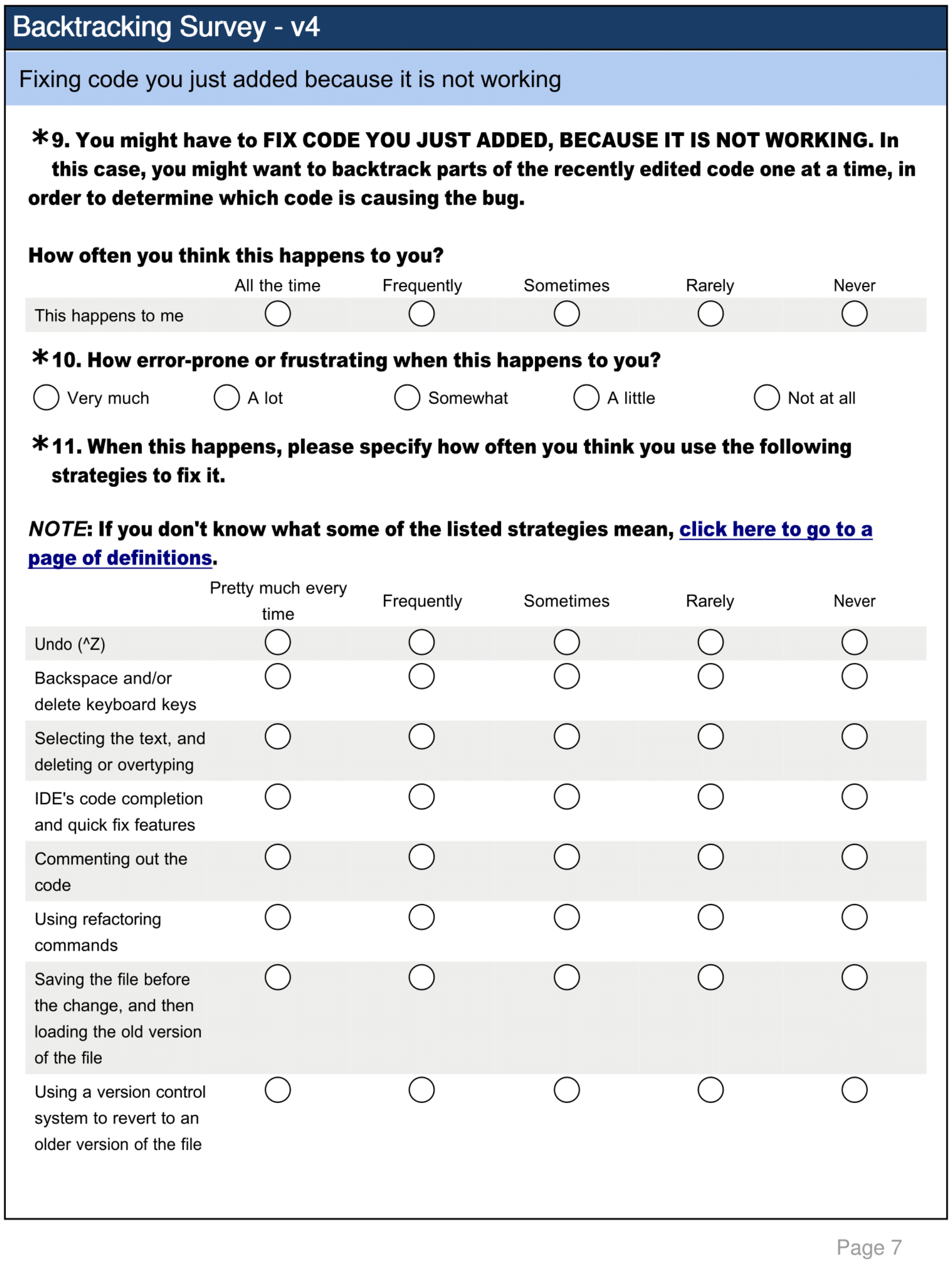
***NOTE*: If you don't know what some of the listed strategies mean, click here to go to a page of definitions.**

	Pretty much every time	Frequently	Sometimes	Rarely	Never
Undo (^Z)	○	○	○	○	○
Backspace and/or delete keyboard keys	○	○	○	○	○
Selecting the text, and deleting or overtyping	○	○	○	○	○
IDE's code completion and quick fix features	○	○	○	○	○
Commenting out the code	○	○	○	○	○
Using refactoring commands	○	○	○	○	○
Saving the file before the change, and then loading the old version of the file	○	○	○	○	○
Using a version control system to revert to an older version of the file	○	○	○	○	○

Page 5

Backtracking Survey - v4

8. If you use any other strategies to FIGURE OUT HOW TO USE AN API CORRECTLY, please specify them.



Backtracking Survey - v4

Fixing code you just added because it is not working

***9. You might have to FIX CODE YOU JUST ADDED, BECAUSE IT IS NOT WORKING. In this case, you might want to backtrack parts of the recently edited code one at a time, in order to determine which code is causing the bug.**

How often you think this happens to you?

	All the time	Frequently	Sometimes	Rarely	Never
This happens to me	○	○	○	○	○

***10. How error-prone or frustrating when this happens to you?**

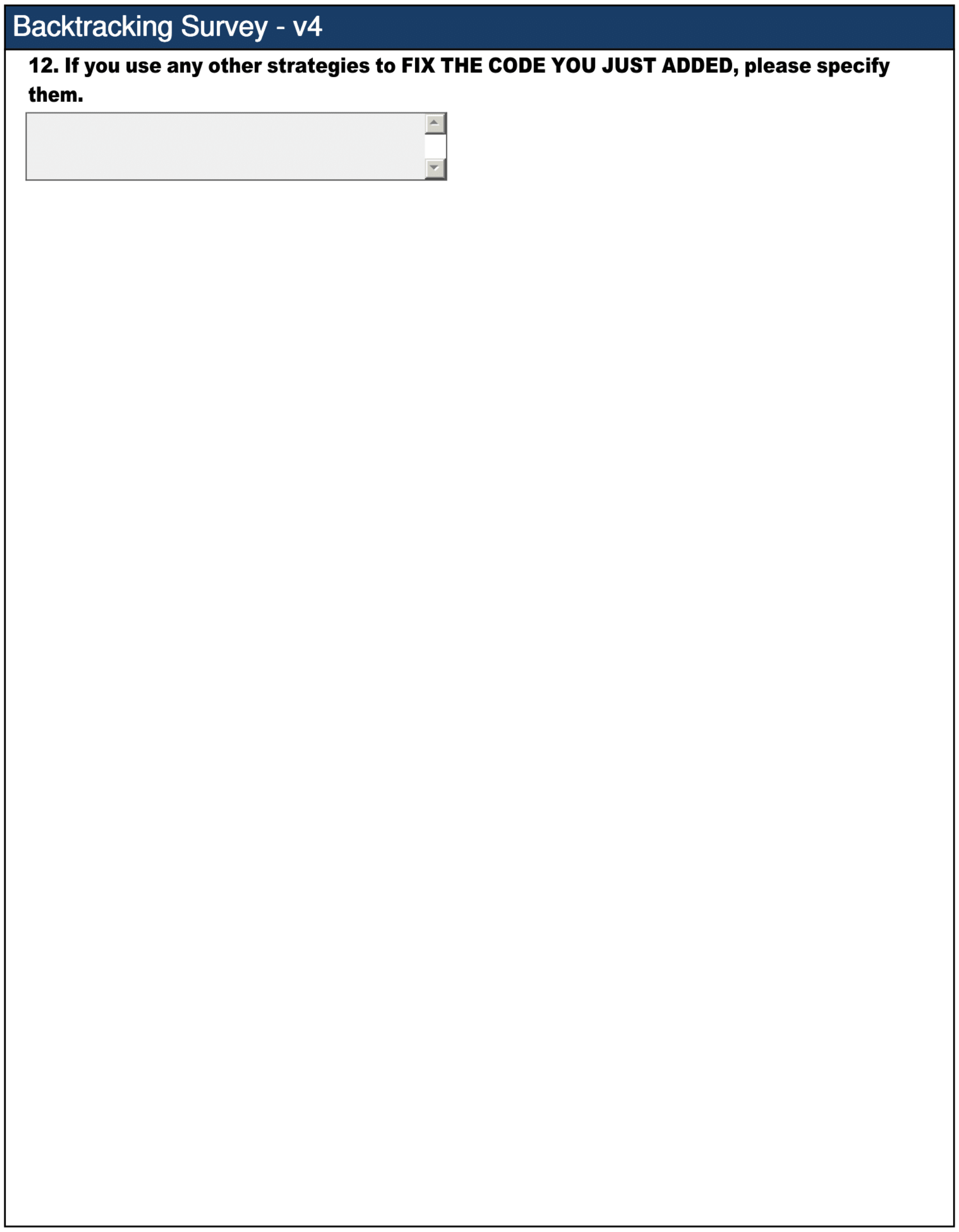
○ Very much ○ A lot ○ Somewhat ○ A little ○ Not at all

***11. When this happens, please specify how often you think you use the following strategies to fix it.**

***NOTE*: If you don't know what some of the listed strategies mean, click here to go to a page of definitions.**

	Pretty much every time	Frequently	Sometimes	Rarely	Never
Undo (^Z)	○	○	○	○	○
Backspace and/or delete keyboard keys	○	○	○	○	○
Selecting the text, and deleting or overtyping	○	○	○	○	○
IDE's code completion and quick fix features	○	○	○	○	○
Commenting out the code	○	○	○	○	○
Using refactoring commands	○	○	○	○	○
Saving the file before the change, and then loading the old version of the file	○	○	○	○	○
Using a version control system to revert to an older version of the file	○	○	○	○	○

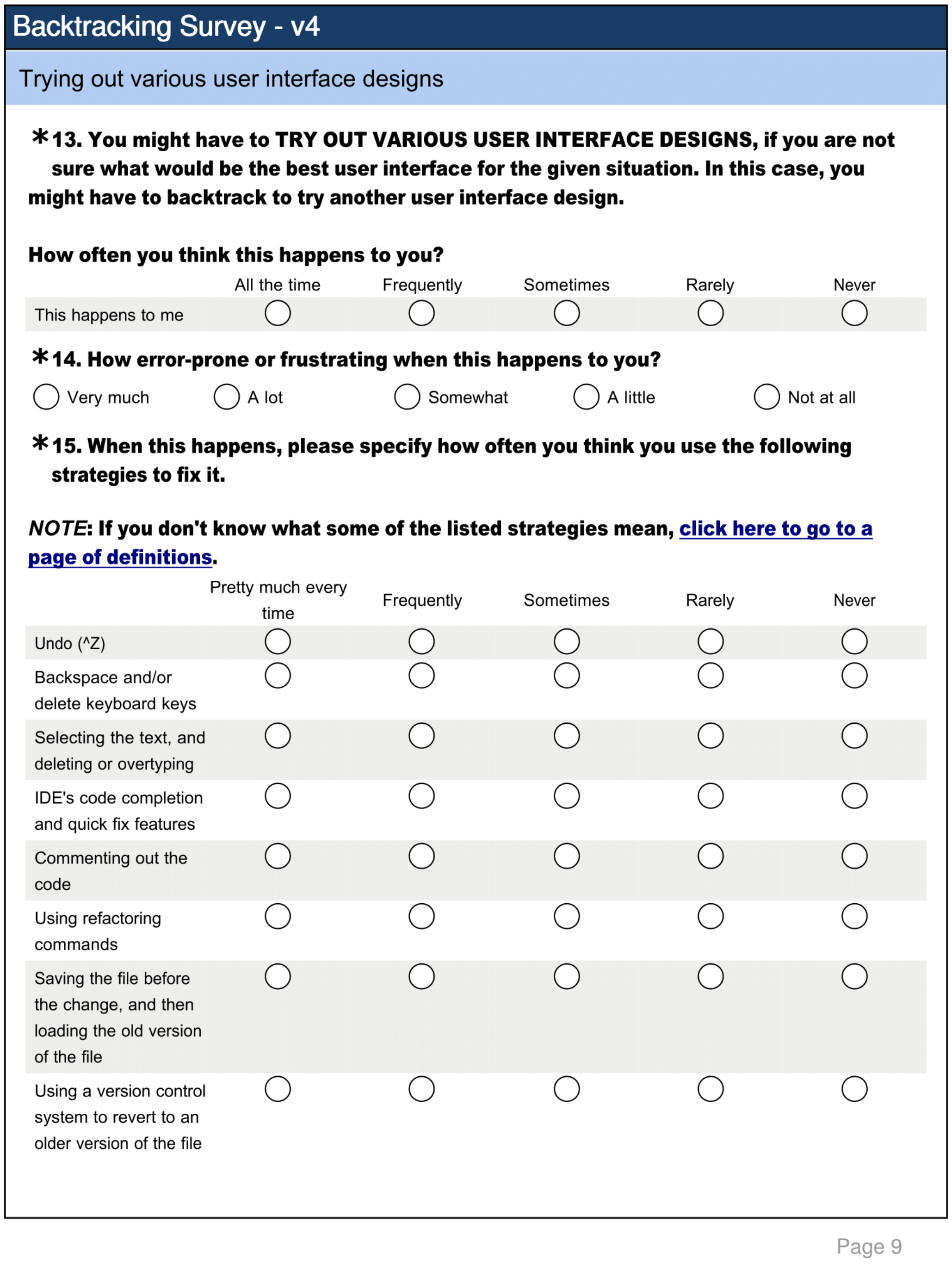
Page 7



Backtracking Survey - v4

12. If you use any other strategies to FIX THE CODE YOU JUST ADDED, please specify them.

Page 8



Backtracking Survey - v4

Trying out various user interface designs

***13. You might have to TRY OUT VARIOUS USER INTERFACE DESIGNS, if you are not sure what would be the best user interface for the given situation. In this case, you might have to backtrack to try another user interface design.**

How often you think this happens to you?

	All the time	Frequently	Sometimes	Rarely	Never
This happens to me	○	○	○	○	○

***14. How error-prone or frustrating when this happens to you?**

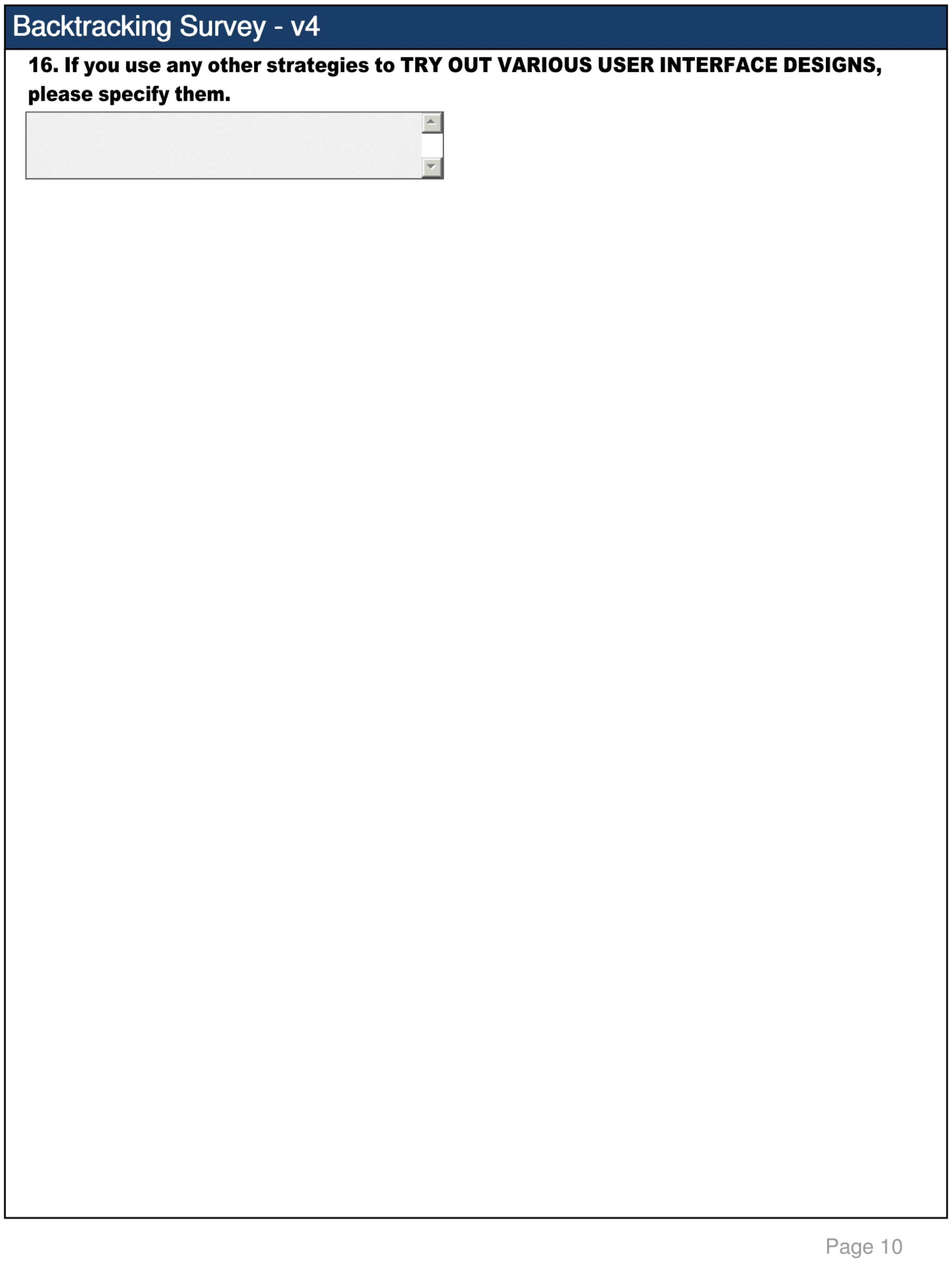
○ Very much ○ A lot ○ Somewhat ○ A little ○ Not at all

***15. When this happens, please specify how often you think you use the following strategies to fix it.**

***NOTE*: If you don't know what some of the listed strategies mean, click here to go to a page of definitions.**

	Pretty much every time	Frequently	Sometimes	Rarely	Never
Undo (^Z)	○	○	○	○	○
Backspace and/or delete keyboard keys	○	○	○	○	○
Selecting the text, and deleting or overtyping	○	○	○	○	○
IDE's code completion and quick fix features	○	○	○	○	○
Commenting out the code	○	○	○	○	○
Using refactoring commands	○	○	○	○	○
Saving the file before the change, and then loading the old version of the file	○	○	○	○	○
Using a version control system to revert to an older version of the file	○	○	○	○	○

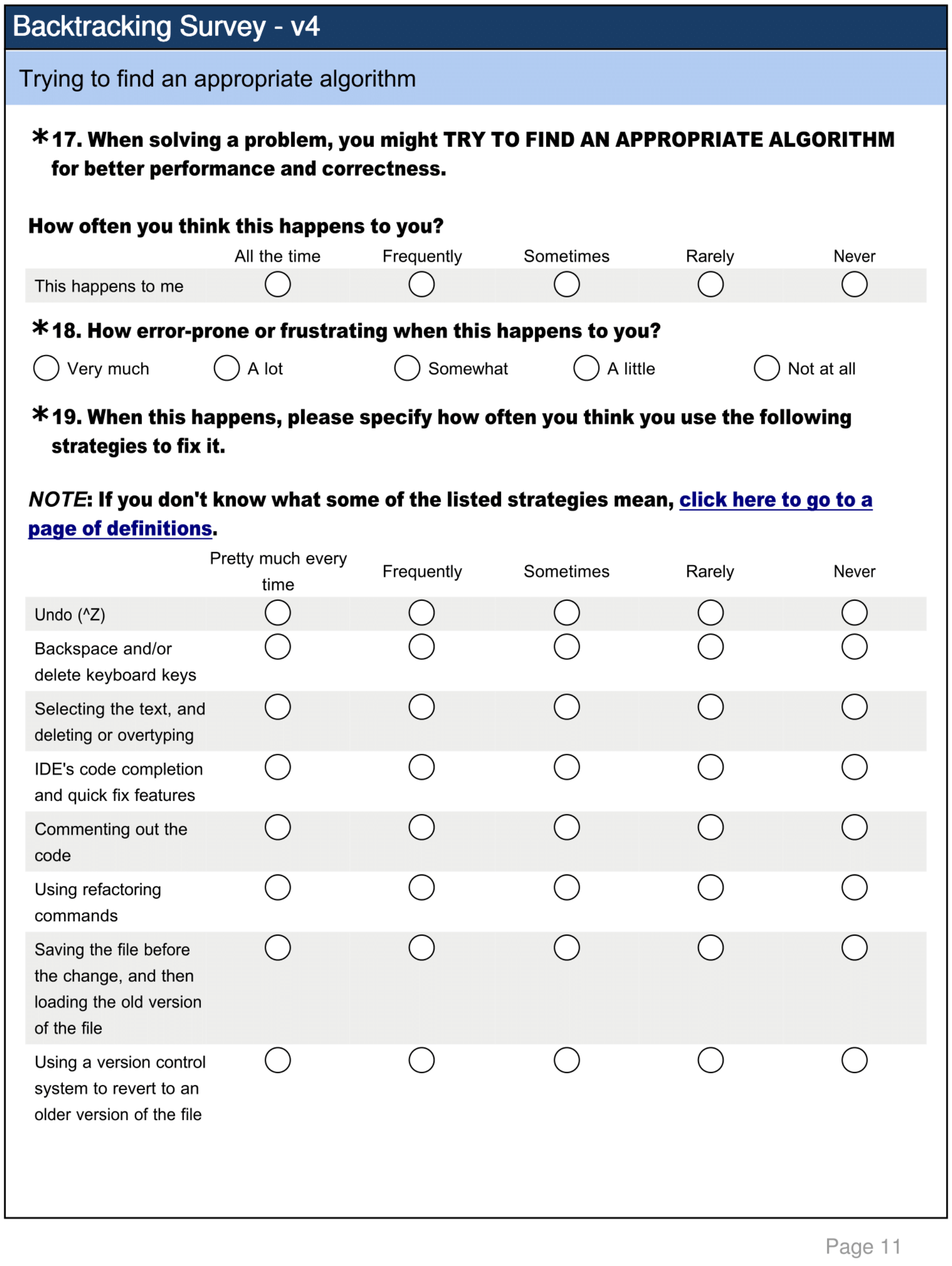
Page 9



Backtracking Survey - v4

16. If you use any other strategies to TRY OUT VARIOUS USER INTERFACE DESIGNS, please specify them.

Page 10



Backtracking Survey - v4

Trying to find an appropriate algorithm

***17. When solving a problem, you might TRY TO FIND AN APPROPRIATE ALGORITHM for better performance and correctness.**

How often you think this happens to you?

	All the time	Frequently	Sometimes	Rarely	Never
This happens to me	○	○	○	○	○

***18. How error-prone or frustrating when this happens to you?**

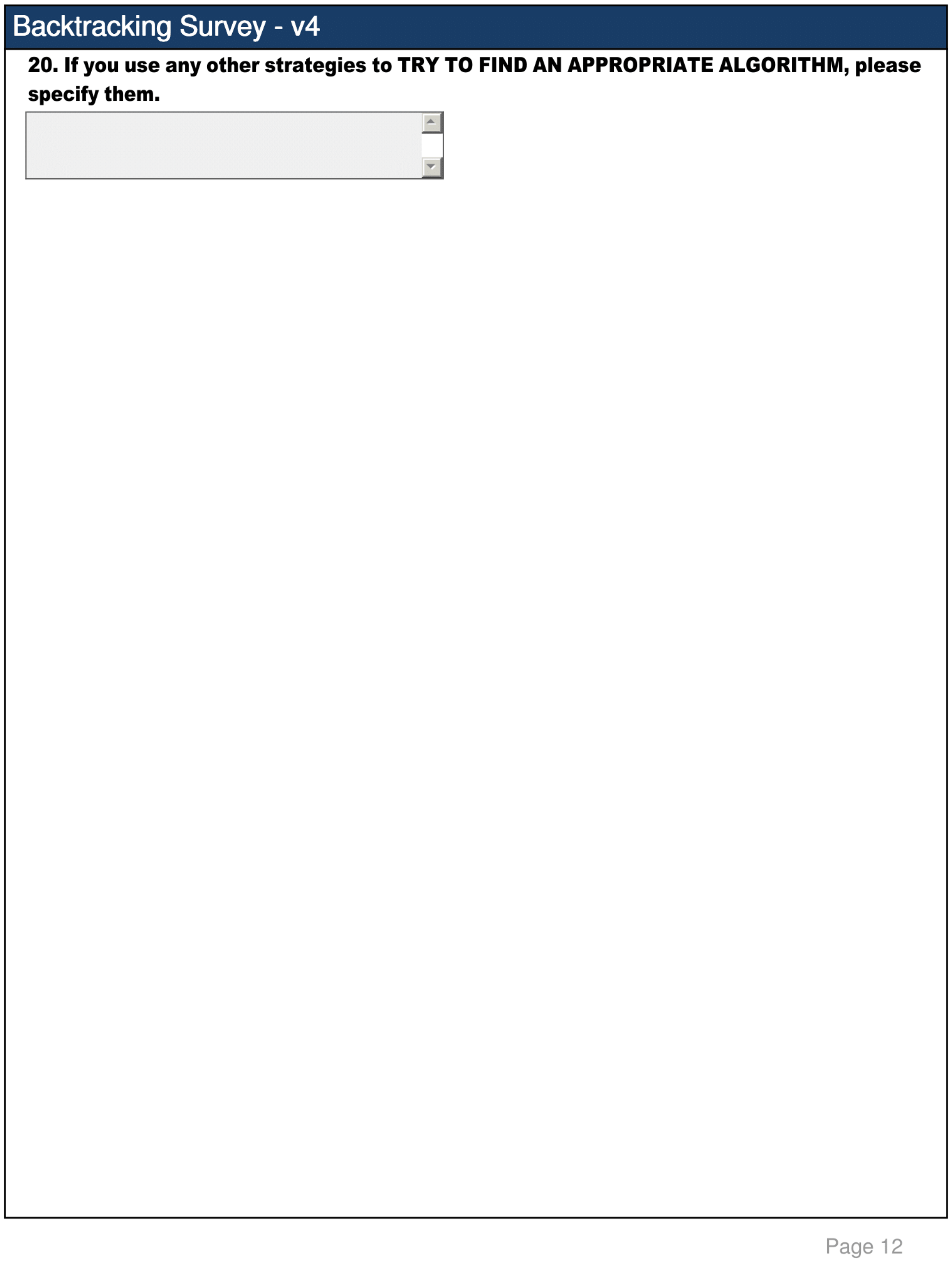
○ Very much ○ A lot ○ Somewhat ○ A little ○ Not at all

***19. When this happens, please specify how often you think you use the following strategies to fix it.**

***NOTE*: If you don't know what some of the listed strategies mean, click here to go to a page of definitions.**

	Pretty much every time	Frequently	Sometimes	Rarely	Never
Undo (^Z)	○	○	○	○	○
Backspace and/or delete keyboard keys	○	○	○	○	○
Selecting the text, and deleting or overtyping	○	○	○	○	○
IDE's code completion and quick fix features	○	○	○	○	○
Commenting out the code	○	○	○	○	○
Using refactoring commands	○	○	○	○	○
Saving the file before the change, and then loading the old version of the file	○	○	○	○	○
Using a version control system to revert to an older version of the file	○	○	○	○	○

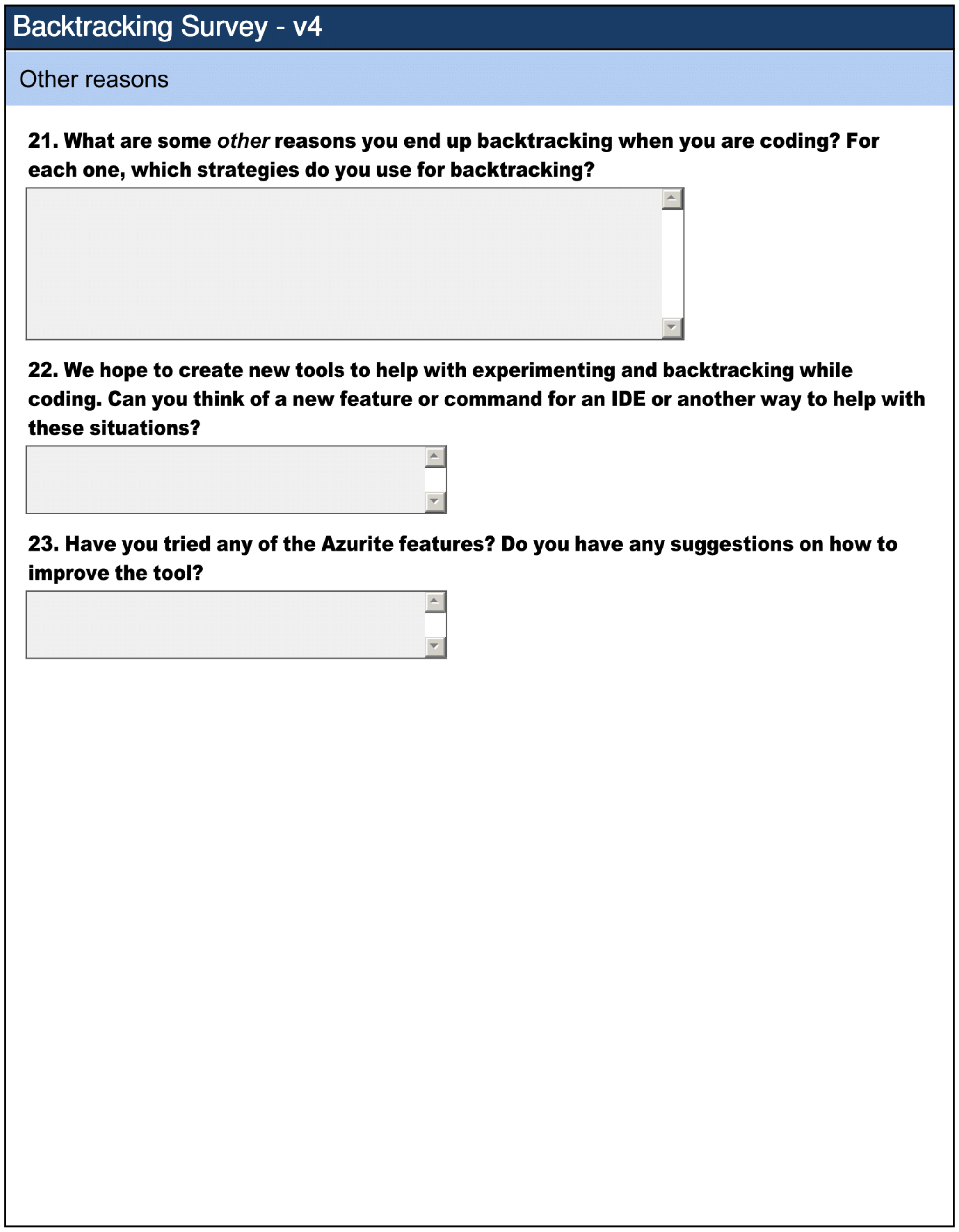
Page 11



Backtracking Survey - v4

20. If you use any other strategies to TRY TO FIND AN APPROPRIATE ALGORITHM, please specify them.

Page 12



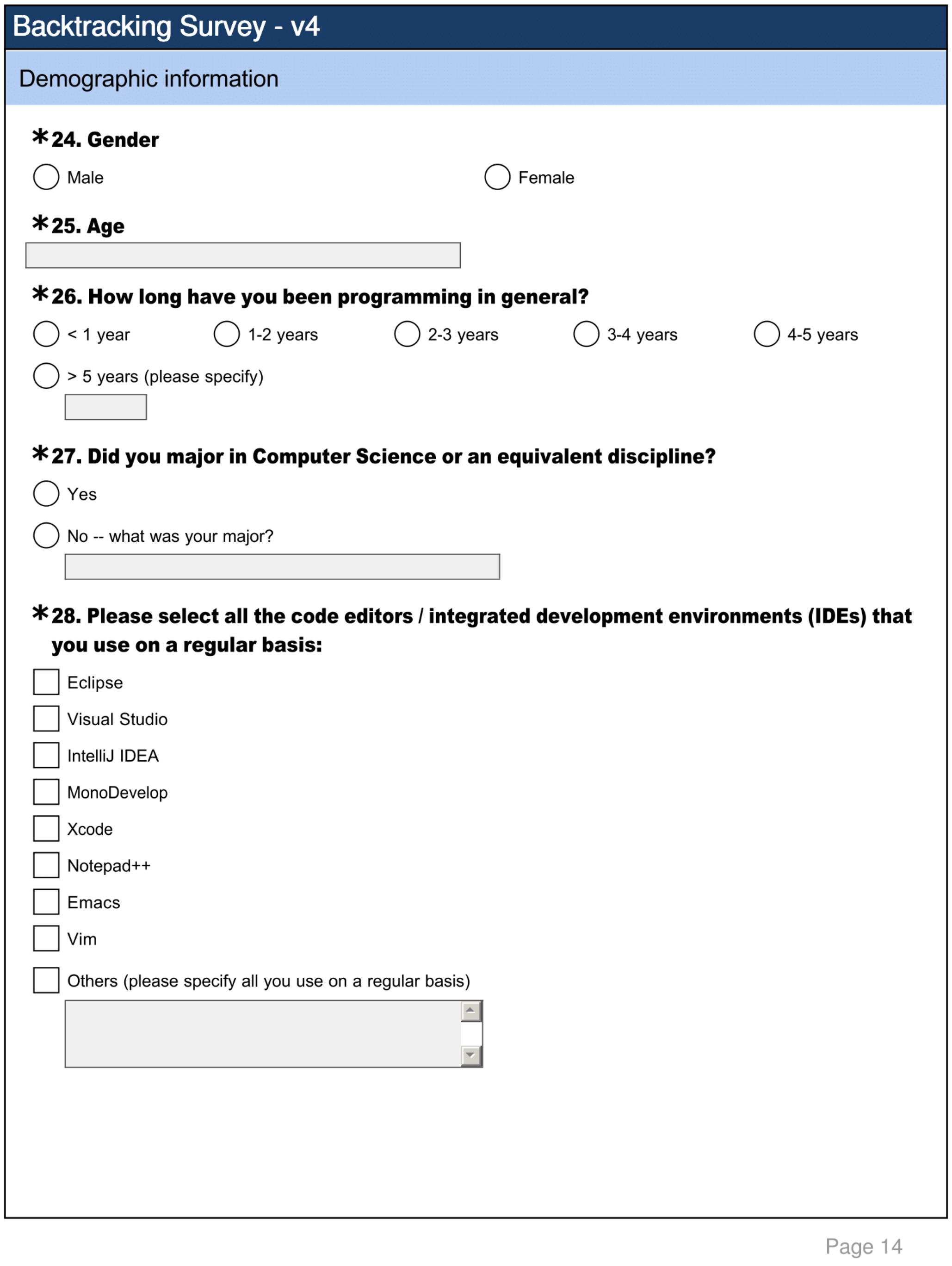
Backtracking Survey - v4

Other reasons

21. What are some *other* reasons you end up backtracking when you are coding? For each one, which strategies do you use for backtracking?

22. We hope to create new tools to help with experimenting and backtracking while coding. Can you think of a new feature or command for an IDE or another way to help with these situations?

23. Have you tried any of the Azurite features? Do you have any suggestions on how to improve the tool?



Backtracking Survey - v4

Demographic information

***24. Gender**

○ Male ○ Female

***25. Age**

***26. How long have you been programming in general?**

○ < 1 year ○ 1-2 years ○ 2-3 years ○ 3-4 years ○ 4-5 years

○ > 5 years (please specify)

***27. Did you major in Computer Science or an equivalent discipline?**

○ Yes

○ No -- what was your major?

***28. Please select all the code editors / integrated development environments (IDEs) that you use on a regular basis:**

☐ Eclipse

☐ Visual Studio

☐ IntelliJ IDEA

☐ MonoDevelop

☐ Xcode

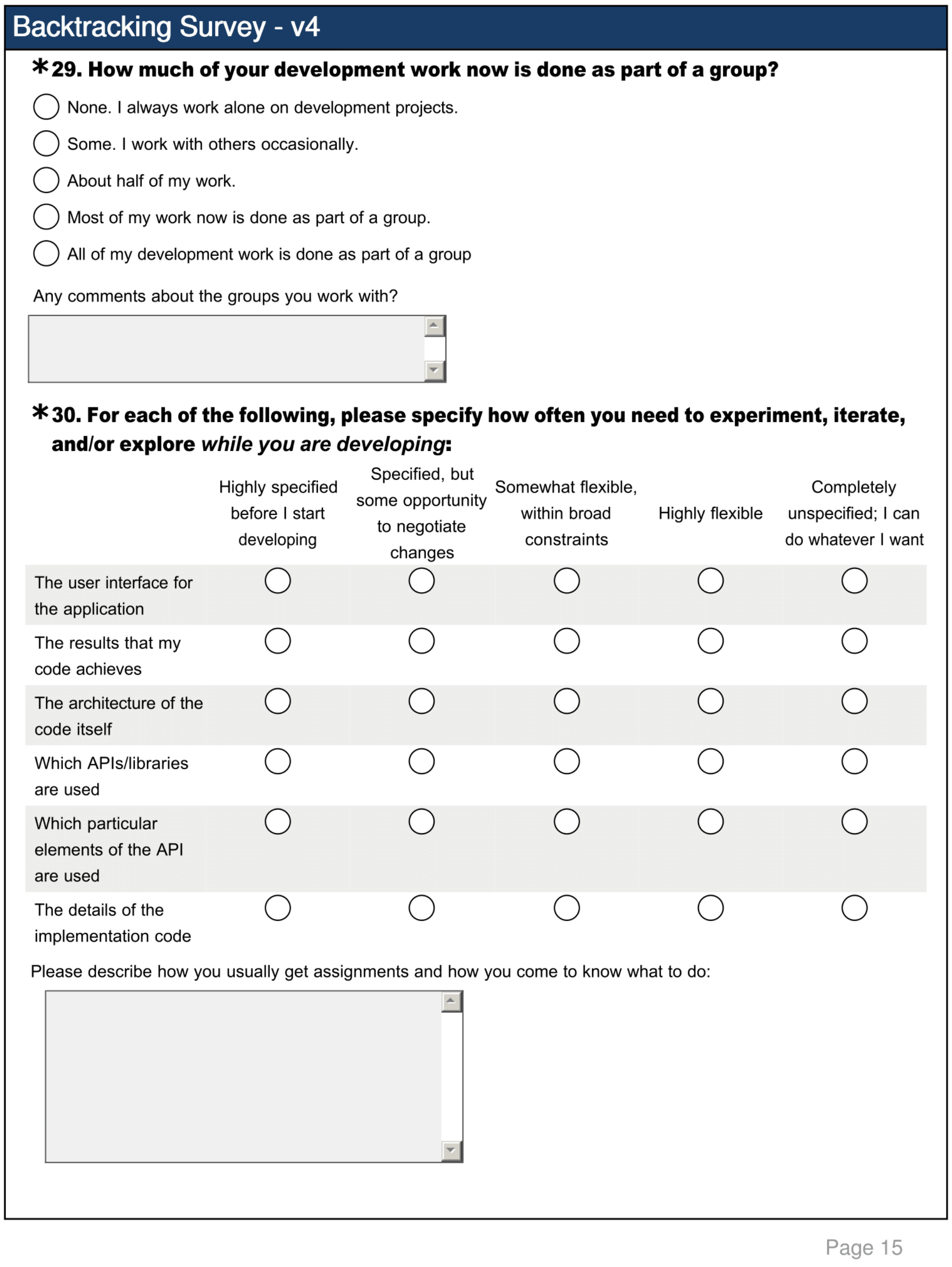
☐ Notepad++

☐ Emacs

☐ Vim

☐ Others (please specify all you use on a regular basis)

Page 14



Backtracking Survey - v4

***29. How much of your development work now is done as part of a group?**

○ None. I always work alone on development projects.

○ Some. I work with others occasionally.

○ About half of my work.

○ Most of my work now is done as part of a group.

○ All of my development work is done as part of a group

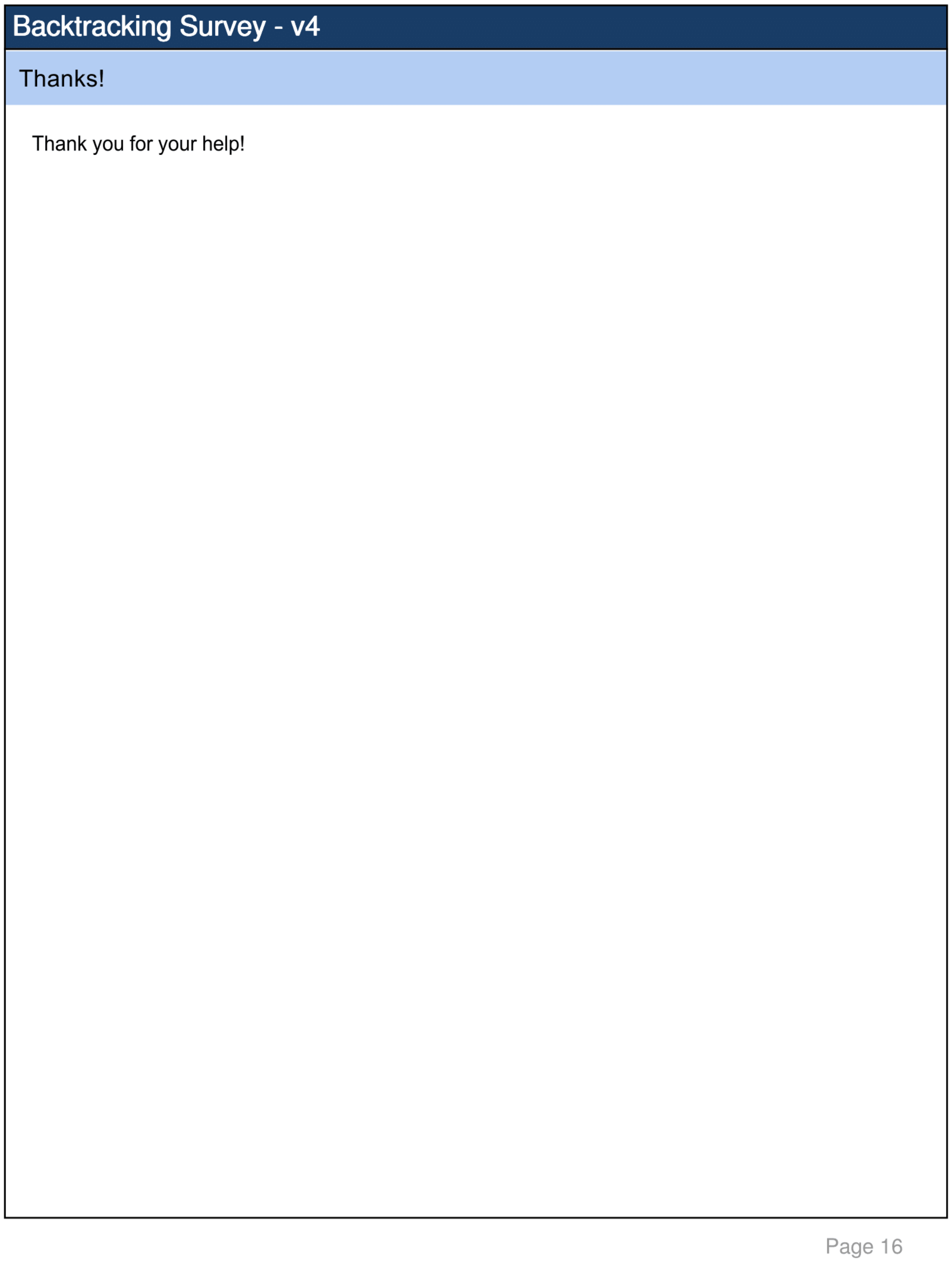
Any comments about the groups you work with?

***30. For each of the following, please specify how often you need to experiment, iterate, and/or explore *while you are developing*:**

	Highly specified before I start developing	Specified, but some opportunity to negotiate changes	Somewhat flexible, within broad constraints	Highly flexible	Completely unspecified; I can do whatever I want
The user interface for the application	○	○	○	○	○
The results that my code achieves	○	○	○	○	○
The architecture of the code itself	○	○	○	○	○
Which APIs/libraries are used	○	○	○	○	○
Which particular elements of the API are used	○	○	○	○	○
The details of the implementation code	○	○	○	○	○

Please describe how you usually get assignments and how you come to know what to do:

Page 15



Backtracking Survey - v4

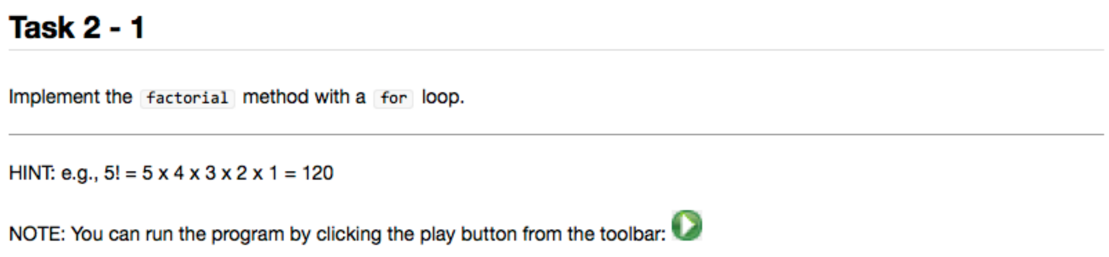
Thanks!

Thank you for your help!

Page 16

APPENDIX C: MATERIALS FROM THE AZURITE EVALUATION LAB STUDY

C.1. TASK SHEETS GIVEN TO THE PARTICIPANTS[59]

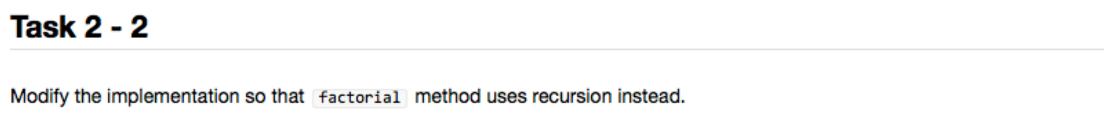


Task 2 - 1

Implement the `factorial` method with a `for` loop.

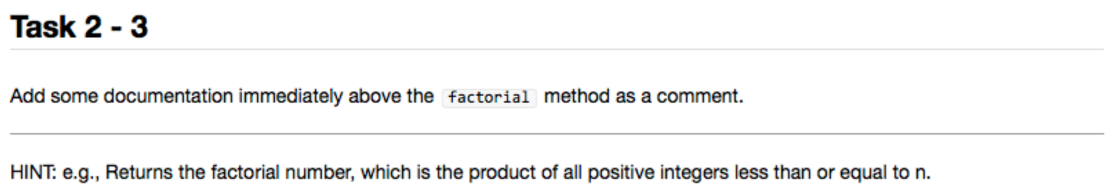
HINT: e.g., 5! = 5 x 4 x 3 x 2 x 1 = 120

NOTE: You can run the program by clicking the play button from the toolbar:



Task 2 - 2

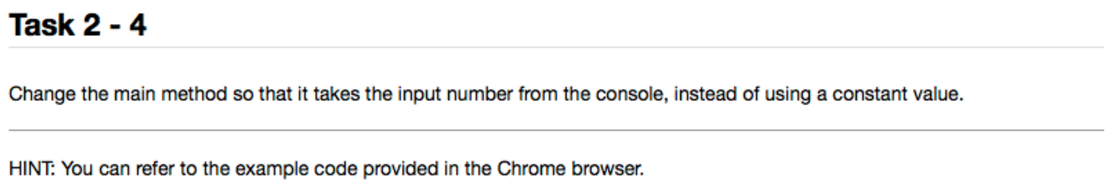
Modify the implementation so that `factorial` method uses recursion instead.



Task 2 - 3

Add some documentation immediately above the `factorial` method as a comment.

HINT: e.g., Returns the factorial number, which is the product of all positive integers less than or equal to n.



Task 2 - 4

Change the main method so that it takes the input number from the console, instead of using a constant value.

HINT: You can refer to the example code provided in the Chrome browser.

[59] Tasks 1, 4, 7 are missing because they were interactive training tasks lead by the experimenter.

Task 2 - 5

Revert the `factorial` method implementation to the for loop version,
while keeping the comment, and the input getting code in place.

Task 3 - 1

Find and delete the code that **sorts** the student database before printing.

Task 3 - 2

Change the `Student` class in a way that `StudentDatabase` prints out each student's First Name, Last Name, and their Email address

Task 3 - 3

Add `getStudentCount()` `clearDatabase()` methods in the `StudentDatabase` class.

- **`getStudentCount()`** : returns the number of students in the database.
- **`clearDatabase()`** : removes all the students from the database.

HINT: You may refer to the `ArrayList` API documentation in the Chrome browser.

Task 3 - 4

Restore the code for sorting the students by their last names before printing, while keeping all the other changes in place.

Task 5 - 1

Finish implementing `Number.java` so that all the existing unit tests pass.

NOTE: You should not be editing the tests themselves.

Task 5 - 2

Change the implementation in a way that `Number` class works as an immutable class.

- Remove `setValue` method
- Change the `add` and `multiply` methods in a way that they return a new object rather than changing the internal value.

HINT: You would have to modify both `INumber` interface and `Number` class.

Task 5 - 3

Revert the code to the point in time in the past,
where mutable version where the `add` and `multiply` methods changed the internal value and passed all the provided tests.

Task 6 - 1

Finish implementing `Stack.java` by inheriting from `ArrayList`, so that all the provided tests pass.

HINT: You can refer to the `ArrayList` API documentation in the Chrome browser.

NOTE: For the purpose of this task, it is not very important to handle exceptional cases properly, such as when `pop()` is called when the stack is empty.

Task 6 - 2

Change the implementation to use composition instead.

In other words, instead of inheriting from `ArrayList`, make the `Stack` have an `ArrayList` object as a member field and use it for the implementation.

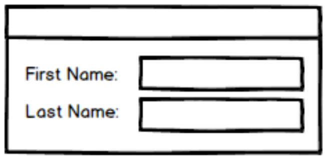
All the provided tests should still be succeeding after finishing this task.

Task 6 - 3

Revert the `Stack` implementation to the version where it inherited from `ArrayList` and passed all the tests.

Task 8 - 1

Lay out the controls using `GridLayout` class, so that the resulting dialog looks as following:



HINT1: You can find the `GridLayout` API documentation page in the Chrome browser.

HINT2: You should set the layout manager on the `mainPanel` object.

Task 8 - 2

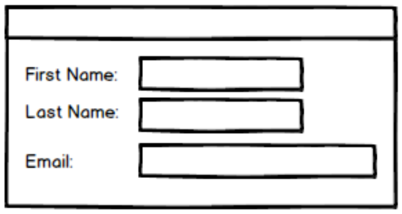
Replace the layout manager to `GridBagLayout`.

HINT: You can refer to the `GridBagLayout` tutorial and the API documentation in the Chrome browser.

HINT2: When setting the `GridBagConstraints` parameters, you may safely assume that you only need to set the `gridx` and `gridy` parameters.

Task 8 - 3

Add an Email field below the Last Name field, as in the following mockup.



NOTE: You do not need to make the text box sizes different.

Task 8 - 4

Revert the layout manager back to `GridLayout` while keeping the Email field.

Task 9

Scenario:

Imagine that you are working for a company as part of a software development team.

One of your colleagues had to leave for a family emergency while fixing a known bug, and you were asked to commit the bug fixing code that she was working on.
The bug fix is complete, but the problem is that she did not have enough time to remove all the code fragments added just for debugging purposes, such as `println` statements and comments.
They are all mixed in the code base.

The local edit history of the bug fix is provided, and this is where she left off.

All you are told is that:

- All the edits made on last Thursday (08/21) are the things you should not be touching,
- All the edits made on last Friday (08/22) are the debugging activities, such as adding the `println` statements and the actual bug fixing.
- The bug fix code is marked as a comment saying `// BUGFIX!`, and this code should be retained.

Task:

Remove all the debugging specific code while keeping the bug fix code.
Tell me when you think you are done.

BIBLIOGRAPHY

[Abowd 1992] G. D. Abowd and A. J. Dix, "Giving Undo Attention," *Interacting with Computers,* vol. 4, 1992, pp. 317-342.

[Akers 2012] D. Akers, R. Jeffries, M. Simpson, and T. Winograd, "Backtracking Events as Indicators of Usability Problems in Creation-Oriented Applications," *ACM Transactions on Computer-Human Interaction* (TOCHI), vol. 19, ACM, 2012, pp. 1-40.

[Appert 2012] C. Appert, O. Chapuis, and E. Pietriga, "Dwell-and-Spring: Undo for Direct Manipulation," In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI'12), ACM, Austin, Texas, USA, 2012, pp. 1957-1966.

[Archer 1984] J. E. Archer, Jr., R. Conway, and F. B. Schneider, "User Recovery and Reversal in Interactive Systems," *ACM Transactions on Programming Languages and Systems* (TOPLAS), vol. 6, ACM, 1984, pp. 1-19.

[Aversano 2007] L. Aversano, L. Cerulo, and M. Di Penta, "How Clones are Maintained: An Empirical Study," In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering* (CSMR'07), IEEE, Amsterdam, Netherlands, 2007, pp. 81-90.

[Bates 1990] M. J. Bates, "Where Should the Person Stop and the Information Search Interface Start?," *Information Processing & Management,* vol. 26, Elsevier Ltd., 1990, pp. 575-591.

[Beck 2002] K. Beck, "Test-Driven Development: By Example," Addison-Wesley Professional, 2002.

[Berlage 1993] T. Berlage and A. Genau, "A Framework for Shared Applications with a Replicated Architecture," In *Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology* (UIST'93), ACM, Atlanta, Georgia, USA, 1993, pp. 249-257.

[Berlage 1994] T. Berlage, "A Selective Undo Mechanism for Graphical User Interfaces Based on Command Objects," *ACM Transactions on Computer-Human Interaction* (TOCHI), vol. 1, ACM, 1994, pp. 269-294.

[Bettenburg 2009] N. Bettenburg, S. Weyi, W. Ibrahim, B. Adams, Z. Ying, and A. E. Hassan, "An Empirical Study on Inconsistent Changes to Code Clones at Release Level," In *Proceedings of the 16th Working Conference on Reverse Engineering* (WCRE'09), IEEE, Lille, France, 2009, pp. 85-94.

[Beyer 1997] H. Beyer and K. Holtzblatt, "Contextual Design: Defining Customer-Centered Systems," 1st ed., Morgan Kaufmann, 1997.

[Bird 2009] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The Promises and Perils of Mining Git," In *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*, IEEE, Vancouver, BC, Canada, 2009, pp. 1-10.

[Bragdon 2010a] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and Joseph J. LaViola, Jr., "Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments," In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (ICSE'10), ACM, Cape Town, South Africa, 2010, pp. 455-464.

[Bragdon 2010b] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and Joseph J. LaViola, Jr., "Code Bubbles: A Working Set-Based Interface for Code Understanding and Maintenance," In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI'10), ACM, Atlanta, Georgia, USA, 2010, pp. 2503-2512.

[Burg 2013] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst, "Interactive Record/Replay for Web Application Debugging," In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology* (UIST'13), ACM, St. Andrews, Scotland, United Kingdom, 2013, pp. 473-484.

[Card 1980a] 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, Elsevier Inc., 1980a, pp. 32-74.

[Card 1980b] S. K. Card, T. P. Moran, and A. Newell, "The Keystroke-Level Model for User Performance Time with Interactive Systems," *Communications of the ACM* (CACM), vol. 23, ACM, 1980b, pp. 396-410.

[Cass 2007] A. Cass and C. Fernandes, "Using Task Models for Cascading Selective Undo," In *Proceedings of the 5th International Conference on Task Models and Diagrams for Users Interface Design* (TAMODIA'07), Springer Berlin / Heidelberg, 2007, pp. 186-201.

[Cass 2005] A. G. Cass and C. S. T. Fern, "Modeling Dependencies for Cascading Selective Undo," In *Proceedings of the IFIP INTERACT 2005 Workshop on Integrating Software Engineering and Usability Engineering*, 2005.

[Cass 2006] A. G. Cass, C. S. T. Fernandes, and A. Polidore, "An Empirical Evaluation of Undo Mechanisms," In *Proceedings of the 4th Nordic Conference on Human-Computer Interaction: Changing Roles* (NordiCHI'06), ACM, Oslo, Norway, 2006, pp. 19-27.

[Chen 2011] H.-T. Chen, L.-Y. Wei, and C.-F. Chang, "Nonlinear Revision Control for Images," In *Proceedings of the ACM SIGGRAPH 2011 papers* (SIGGRAPH'11), ACM, Vancouver, BC, Canada, 2011, pp. 1-10.

[Chi 2012] P.-Y. Chi, S. Ahn, A. Ren, M. Dontcheva, W. Li, and B. Hartmann, "MixT: Automatic Generation of Step-by-Step Mixed Media Tutorials," In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology* (UIST'12), ACM, Cambridge, Massachusetts, USA, 2012, pp. 93-102.

[Chii 1998] M. Chii, M. Yasue, A. Imamiya, and M. Xiaoyang, "Visualizing Histories for Selective Undo and Redo," In *Proceedings of the 3rd Asia Pacific Computer Human Interaction*, IEEE, Shonan Village Center, 1998, pp. 459-464.

[Choudhary 1995] R. Choudhary and P. Dewan, "A General Multi-User Undo/Redo Model," In *Proceedings of the 4th European Conference on Computer-Supported Cooperative Work* (ECSCW'95), Springer Netherlands, 1995, pp. 231-246.

[Coman 2008] I. D. Coman and A. Sillitti, "Automated Identification of Tasks in Development Sessions," In *Proceedings of the 16th IEEE International Conference on Program Comprehension* (ICPC'08), IEEE, Amsterdam, Netherlands, 2008, pp. 212-217.

[Conradi 1998] R. Conradi and B. Westfechtel, "Version Models for Software Configuration Management," *ACM Computing Surveys* (CSUR), vol. 30, ACM, 1998, pp. 232-282.

[Cormen 2009] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms," 3rd ed., 2009.

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

[Dörner 2014] C. Dörner, A. R. Faulring, and B. A. Myers, "EUKLAS: Supporting Copy-and-Paste Strategies for Integrating Example Code," In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools* (PLATEAU'14), ACM, Portland, Oregon, USA, 2014, pp. 13-20.

[Ellis 1989] C. A. Ellis and S. J. Gibbs, "Concurrency Control in Groupware Systems," *SIGMOD Record,* vol. 18, ACM, 1989, pp. 399-407.

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

[Fisher 2005] M. Fisher and G. Rothermel, "The EUSES Spreadsheet Corpus: A Shared Resource for Supporting Experimentation with Spreadsheet Dependability Mechanisms," In *Proceedings of the 1st Workshop on End-User Software Engineering*, ACM, St. Louis, Missouri, 2005, pp. 1-5.

[Fluri 2006] B. Fluri and H. C. Gall, "Classifying Change Types for Qualifying Change Couplings," In *Proceedings of the 14th IEEE International Conference on Program Comprehension* (ICPC'06), Athens, Greece, 2006, pp. 35-45.

[Fluri 2007] B. Fluri, M. Wursch, M. Pinzger, and H. C. Gall, "Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction," *IEEE Transactions on Software Engineering* (TSE), vol. 33, IEEE, 2007, pp. 725-743.

[Fogarty 2005] 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," In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI'05), ACM, Portland, Oregon, USA, 2005, pp. 331-340.

[Fraser 2012] N. Fraser, "google-diff-match-patch - Diff, Match and Patch libraries for Plain Text," 2012; http://code.google.com/p/google-diff-match-patch/.

[Fritz 2010] T. Fritz and G. C. Murphy, "Using Information Fragments to Answer the Questions Developers Ask," In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* (ICSE'10), 2010, pp. 175-184.

[Fritzson 1986] P. Fritzson, "Systems and Tools for Exploratory Programming: Overview and Examples," D. o. C. a. I. Science, Linköping University TR LiTH-IDA-R-86-36, 1986.

[Fuchs 2014] M. Fuchs, M. Heckner, F. Raab, and C. Wolff, "Monitoring Students' Mobile App Coding Behavior Data Analysis Based on IDE and Browser Interaction Logs," In *Proceedings of the IEEE Global Engineering Education Conference* (EDUCON'14), IEEE, Istanbul, Turkey, 2014, pp. 892-899.

[Gamma 1994] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," Pearson Education, 1994.

[Ge 2012] X. Ge, Q. L. DuBose, and E. Murphy-Hill, "Reconciling Manual and Automatic Refactoring," In *Proceedings of the 34th International Conference on Software Engineering* (ICSE'12), IEEE, Zurich, Switzerland, 2012, pp. 211-221.

[Ginosar 2013] S. Ginosar, L. F. D. Pombo, M. Agrawala, and B. Hartmann, "Authoring Multi-Stage Code Examples with Editable Code Histories," In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology* (UIST'13), ACM, St Andrews, United Kingdom, 2013.

[Grigoreanu 2010] V. I. Grigoreanu, M. M. Burnett, and G. G. Robertson, "A Strategy-Centric Approach to the Design of End-User Debugging Tools," In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI'10), ACM, Atlanta, Georgia, USA, 2010, pp. 713-722.

[Grossman 2010] T. Grossman, J. Matejka, and G. Fitzmaurice, "Chronicle: Capture, Exploration, and Playback of Document Workflow Histories," In *Proceedings of the 23nd Annual ACM Symposium on User Interface Software and Technology* (UIST'10), ACM, New York, New York, USA, 2010, pp. 143-152.

[Gundel 2005] C. Gundel, "Exploratory Programming and BASIC," Blog Article, 2005; http://basicprogramming.blogspot.com/2005/01/exploratory-programming-and-basic.html.

[Harrison 2006] W. Harrison, "Eating Your Own Dog Food," *IEEE Software,* vol. 23, IEEE, 2006, pp. 5-7.

[Hartmann 2008] B. Hartmann, L. Yu, A. Allison, Y. Yang, and S. R. Klemmer, "Design as Exploration: Creating Interface Alternatives through Parallel Authoring and Runtime Tuning," In *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology* (UIST'08), ACM, Monterey, CA, USA, 2008, pp. 91-100.

[Hattori 2010a] L. Hattori, "Enhancing Collaboration of Multi-Developer Projects with Synchronous Changes," In *Proceedings of the 32nd International Conference on Software Engineering - Volume 2* (ICSE'10), ACM, Cape Town, South Africa, 2010, pp. 377-380.

[Hattori 2010b] L. Hattori and M. Lanza, "Syde: A Tool for Collaborative Software Development," In *Proceedings of the 32nd International Conference on Software Engineering - Volume 2* (ICSE'10), ACM, 2010, pp. 235-238.

[Hattori 2011] L. Hattori, M. D'Ambros, M. Lanza, and M. Lungu, "Software Evolution Comprehension: Replay to the Rescue," In *Proceedings of the 19th IEEE International Conference on Program Comprehension* (ICPC'11), IEEE, Kingston, Ontario, Canada, 2011, pp. 161-170.

[Hayashi 2012] S. Hayashi, T. Omori, T. Zenmyo, K. Maruyama, and M. Saeki, "Refactoring Edit History of Source Code," In *Proceedings of the 28th IEEE International Conference on Software Maintenance* (ICSM'12), IEEE, Trento, Italy, 2012, pp. 617-620.

[Hayashi 2015] S. Hayashi, D. Hoshino, J. Matsuda, M. Saeki, T. Omori, and K. Maruyama, "Historef: A Tool for Edit History Refactoring," In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering* (SANER'15), Tool Tracks, IEEE, Montréal, Québec, Canada, 2015.

[Hindle 2007] A. Hindle, J. Zhen Ming, W. Koleilat, M. W. Godfrey, and R. C. Holt, "YARN: Animating Software Evolution," In *Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis* (VISSOFT'07), IEEE, Banff, Alberta, Canada, 2007, pp. 129-136.

[Hong 2000] J. I. Hong and J. A. Landay, "SATIN: A Toolkit for Informal Ink-Based Applications," In *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology* (UIST'00), ACM, San Diego, California, USA, 2000, pp. 63-72.

[Kagdi 2007] H. Kagdi, M. L. Collard, and J. I. Maletic, "A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution," *Journal of Software Maintenance and Evolution: Research and Practice,* vol. 19, John Wiley & Sons, Ltd., 2007, pp. 77-131.

[Kawasaki 2004] Y. Kawasaki and T. Igarashi, "Regional Undo for Spreadsheets," In *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology* (UIST'04), ACM, Santa Fe, New Mexico, 2004.

[Kersten 2006] M. Kersten and G. C. Murphy, "Using Task Context to Improve Programmer Productivity," In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (FSE'06), ACM, Portland, Oregon, USA, 2006, pp. 1-11.

[Kim 2004] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An Ethnographic Study of Copy and Paste Programming Practices in OOPL," In *Proceedings of the International Symposium on Empirical Software Engineering* (ISESE'04), IEEE, Redondo Beach, CA, USA, 2004, pp. 83-92.

[Kim 2005a] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An Empirical Study of Code Clone Genealogies," In *Proceedings of the 10th European Software Engineering Conference held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (ESEC/FSE'05), ACM, Lisbon, Portugal, 2005, pp. 187-196.

[Kim 2006] M. Kim and D. Notkin, "Program Element Matching for Multi-Version Program Analyses," In *Proceedings of the 2006 International Workshop on Mining Software Repositories* (MSR'06), ACM, Shanghai, China, 2006, pp. 58-64.

[Kim 2009] M. Kim and D. Notkin, "Discovering and Representing Systematic Code Changes," In *Proceedings of the 31st International Conference on Software Engineering* (ICSE'09), IEEE, Vancouver, British Columbia, Canada, 2009, pp. 309-319.

[Kim 2011] M. Kim, D. Cai, and S. Kim, "An Empirical Investigation into the Role of API-Level Refactorings During Software Evolution," In *Proceedings of the 33rd International Conference on Software Engineering* (ICSE'11), ACM, Honolulu, HI, USA, 2011, pp. 151-160.

[Kim 2005b] S. Kim, P. Kai, and E. J. Whitehead, "When Functions Change Their Names: Automatic Detection of Origin Relationships," In *Proceedings of the 12th Working Conference on Reverse Engineering* (WCRE'05), IEEE, Pittsburgh, PA, USA, 2005, p. 10 pp.

[Klemmer 2002] S. R. Klemmer, M. Thomsen, E. Phelps-Goodman, R. Lee, and J. A. Landay, "Where Do Web Sites Come From?: Capturing and Interacting with Design History," In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI'02), ACM, Minneapolis, Minnesota, USA, 2002, pp. 1-8.

[Ko 2003] A. J. Ko and B. A. Myers, "Development and Evaluation of a Model of Programming Errors," In *Proceedings of the IEEE Symposium on Human Centric Computing Languages and Environments* (HCC'03), IEEE, Auckland, New Zealand, 2003, pp. 7-14.

[Ko 2004] A. J. Ko and B. A. Myers, "A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems," *Journal of Visual Languages & Computing,* vol. 16, Elsevier Ltd., 2004, pp. 41-84.

[Ko 2005a] A. J. Ko, H. Aung, and B. A. Myers, "Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks," In *Proceedings of the 27th International Conference on Software Engineering* (ICSE'05), ACM, St. Louis, MO, USA, 2005, pp. 126-135.

[Ko 2005b] A. J. Ko, H. H. Aung, and B. A. Myers, "Design Requirements for More Flexible Structured Editors from a Study of Programmers' Text Editing," In *Proceedings of the Extended Abstracts on Human Factors in Computing Systems* (CHI'05 EA), ACM, Portland, OR, USA, 2005, pp. 1557-1560.

[Ko 2006] A. J. Ko and B. A. Myers, "Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors," In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI'06), ACM, Montréal, Québec, Canada, 2006, pp. 387-396.

[Ko 2007] A. J. Ko, R. DeLine, and G. Venolia, "Information Needs in Collocated Software Development Teams," In *Proceedings of the 29th International Conference on Software Engineering* (ICSE'07), IEEE, Minneapolis, MN, 2007, pp. 344-353.

[Ko 2008a] A. J. Ko, "Asking and Answering Questions about the Causes of Software Behavior," PhD Dissertation, Human-Computer Interaction Institute, School of Computer Science, Carnegie Mellon University, 2008.

[Ko 2008b] A. J. Ko and B. A. Myers, "Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior," In *Proceedings of the 30th International Conference on Software Engineering* (ICSE'08), ACM, Leipzig, Germany, 2008, pp. 301-310.

[Kojouharov 2004] C. Kojouharov, A. Solodovnik, and G. Naumovich, "JTutor: An Eclipse Plug-in Suite for Creation and Replay of Code-Based Tutorials," In *Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology eXchange*, ACM, Vancouver, BC, Canada, 2004, pp. 27-31.

[Kuhn 2012] A. Kuhn and M. Stocker, "CodeTimeline: Storytelling with Versioning Data," In *Proceedings of the 2012 34th International Conference on Software Engineering* (ICSE'12), IEEE, Zurich, Switzerland, 2012, pp. 1333-1336.

[Kurlander 1988] D. Kurlander and S. Feiner, "Editable Graphical Histories," In *Proceedings of the 1988 IEEE Workshop on Visual Languages*, 1988, pp. 127-134.

[Kurlander 1990] D. Kurlander and S. Feiner, "A Visual Language for Browsing, Undoing, and Redoing Graphical Interface Commands," *Visual languages and visual programming,* Plenum Press, 1990, pp. 257-275.

[Kuttal 2011] S. K. Kuttal, A. Sarma, and G. Rothermel, "History Repeats Itself More Easily When You Log It: Versioning for Mashups," In *Proceedings of the 2011 IEEE Symposium on Visual Languages and Human-Centric Computing* (VL/HCC'11), IEEE, Pittsburgh, PA, USA, 2011, pp. 69-72.

[Kwan 2013] I. Kwan, "Timeline_InformationForagingForks," 2013; https://github.com/IrwinKwan/Timeline_InformationForagingForks.

[Lanza 2001] M. Lanza, "The Evolution Matrix: Recovering Software Evolution Using Software Visualization Techniques," In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, ACM, Vienna, Austria, 2001, pp. 37-42.

[LaToza 2010] T. D. LaToza and B. A. Myers, "Hard-to-Answer Questions about Code," In *Proceedings of the Workshop on Evaluation and Usability of Programming Languages and Tools* (PLATEAU'10), ACM, Reno, Nevada, 2010, pp. 1-6.

[LaToza 2012] T. D. LaToza, "Answering Reachability Questions," PhD Dissertation, Institute for Software Research, School of Computer Science, Carnegie Mellon University, 2012.

[Lee 2013] Y. Y. Lee, N. Chen, and R. E. Johnson, "Drag-and-Drop Refactoring: Intuitive and Efficient Program Transformation," In *Proceedings of the 2013 International Conference on Software Engineering* (ICSE'13), IEEE, San Francisco, CA, USA, 2013, pp. 23-32.

[Levenshtein 1966] V. I. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics Doklady,* vol. 10, 1966, p. 707.

[Li 2003] R. Li and D. Li, "A Regional Undo Mechanism for Text Editing," In *Proceedings of the International Workshop on Collaborative Editing Systems* (IWCES'03), 2003.

[Lieberman 1992] H. Lieberman, "Dominoes and Storyboards Beyond `Icons on Strings'," In *Proceedings of the IEEE Workshop on Visual Languages*, IEEE, Seattle, WA, 1992, pp. 65-71.

[Losh 2012] S. Losh, "Gundo - Visualize your Vim Undo Tree," 2012; http://sjl.bitbucket.org/gundo.vim/.

[MacKenzie 2002] I. S. MacKenzie and R. W. Soukoreff, "Text Entry for Mobile Computing: Models and Methods, Theory and Practice," *Human-computer interaction,* vol. 17, Taylor & Francis, 2002, pp. 147-198.

[Martie 2013] L. Martie and A. Van der Hoek, "Toward Social-Technical Code Search," In *Proceedings of the 6th International Workshop on Cooperative and Human Aspects of Software Engineering* (CHASE'13), IEEE, San Francisco, CA, 2013, pp. 101-104.

[Maruyama 2012] K. Maruyama, E. Kitsu, T. Omori, and S. Hayashi, "Slicing and Replaying Code Change History," In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (ASE'12), ACM, Essen, Germany, 2012, pp. 246-249.

[Mens 2002] T. Mens, "A State-of-the-Art Survey on Software Merging," *IEEE Transactions on Software Engineering* (TSE), vol. 28, IEEE, 2002, pp. 449-462.

[Moonen 2001] L. Moonen, "Generating Robust Parsers Using Island Grammars," In *Proceedings of the 8th Working Conference on Reverse Engineering* (WCRE'01), IEEE, Stuttgart, Germany, 2001, pp. 13-22.

[Murphy-Hill 2009] E. Murphy-Hill, C. Parnin, and A. P. Black, "How We Refactor, and How We Know It," In *Proceedings of the 31st International Conference on Software Engineering* (ICSE'09), IEEE, Vancouver, British Columbia, Canada, 2009, pp. 287-297.

[Murphy-Hill 2013] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan, "The Design of Bug Fixes," In *Proceedings of the 35th International Conference on Software Engineering* (ICSE'13), IEEE, San Francisco, CA, 2013, pp. 332-341.

[Murphy 2006] G. C. Murphy, M. Kersten, and L. Findlater, "How Are Java Software Developers Using the Elipse IDE?," *IEEE Software,* vol. 23, IEEE, 2006, pp. 76-83.

[Myers 1996] B. A. Myers and D. S. Kosbie, "Reusable Hierarchical Command Objects," In *Proceedings of the CHI'96 SIGCHI Conference on Human Factors in Computing Systems*, ACM, Vancouver, BC, Canada, 1996, pp. 260-267.

[Myers 1998] B. A. Myers, "Scripting Graphical Applications by Demonstration," In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI'98), ACM, Los Angeles, California, United States, 1998, pp. 534-541.

[Myers 2006] B. A. Myers, D. A. Weitzman, A. J. Ko, and D. H. Chau, "Answering Why and Why Not Questions in User Interfaces," In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI'06), ACM, Montréal, Québec, Canada, 2006, pp. 397-406.

[Myers 2015] B. A. Myers, A. Lai, T. M. Le, Y. Yoon, A. Faulring, and J. Brandt, "Selective Undo Support for Painting Applications," In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI'15), ACM, Seoul, Korea, 2015.

[Myers 1986] E. W. Myers, "An O (ND) Difference Algorithm and Its Variations," *Algorithmica,* vol. 1, 1986, pp. 251-266.

[Neamtiu 2005] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding Source Code Evolution Using Abstract Syntax Tree Matching," In *Proceedings of the 2005 International Workshop on Mining Software Repositories* (MSR'05), ACM, St. Louis, Missouri, 2005, pp. 1-5.

[Negara 2012] S. Negara, M. Vakilian, N. Chen, R. Johnson, and D. Dig, "Is It Dangerous to Use Version Control Histories to Study Source Code Evolution?," In *Proceedings of the 26th European Conference on Object-Oriented Programming* (ECOOP'12), Springer Berlin Heidelberg, Beijing, China, 2012, pp. 79-103.

[Negara 2014] S. Negara, M. Codoban, D. Dig, and R. E. Johnson, "Mining Fine-Grained Code Changes to Detect Unknown Change Patterns," In *Proceedings of the 36th International Conference on Software Engineering* (ICSE'14), ACM, Hyderabad, India, 2014, pp. 803-813.

[Ogawa 2010] M. Ogawa and K.-L. Ma, "Software Evolution Storylines," In *Proceedings of the 5th International Symposium on Software Visualization* (SOFTVIS'10), ACM, Salt Lake City, Utah, USA, 2010, pp. 35-42.

[Omar 2012] C. Omar, Y. S. Yoon, T. D. LaToza, and B. A. Myers, "Active Code Completion," In *Proceedings of the 2012 34th International Conference on Software Engineering* (ICSE'12), IEEE, Zurich, Switzerland, 2012, pp. 859-869.

[Omori 2008] T. Omori and K. Maruyama, "A Change-Aware Development Environment by Recording Editing Operations of Source Code," In *Proceedings of the 2008 International Working Conference on Mining Software Repositories* (MSR'08), ACM, Leipzig, Germany, 2008, pp. 31-34.

[Parnin 2009] C. Parnin and S. Rugaber, "Resumption Strategies for Interrupted Programming Tasks," In *Proceedings of the 17th IEEE International Conference on Program Comprehension* (ICPC'09), IEEE, Vancouver, British Columbia, Canada, 2009, pp. 80-89.

[Parnin 2012] C. Parnin and S. Rugaber, "Programmer Information Needs after Memory Failure," In *Proceedings of the 20th IEEE International Conference on Program Comprehension* (ICPC'12), IEEE, Passau, Germany, 2012, pp. 123-132.

[Prakash 1994] A. Prakash and M. J. Knister, "A Framework for Undoing Actions in Collaborative Systems," *ACM Transactions on Computer-Human Interaction* (TOCHI), vol. 1, ACM, 1994, pp. 295-330.

[Reiss 2008] S. P. Reiss, "Tracking Source Locations," In *Proceedings of the 30th International Conference on Software Engineering* (ICSE'08), ACM, Leipzig, Germany, 2008, pp. 11-20.

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

[Ren 2004] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: A Tool for Change Impact Analysis of Java Programs," In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA'04), ACM, Vancouver, BC, Canada, 2004, pp. 432-448.

[Robbes 2007] R. Robbes and M. Lanza, "A Change-Based Approach to Software Evolution," *Electronic Notes in Theoretical Computer Science* (ENTCS), vol. 166, 2007, pp. 93-109.

[Rogers 2010] E. M. Rogers, "Diffusion of innovations," Simon and Schuster, 2010.

[Sametinger 1992] J. Sametinger and A. Stritzinger, "Exploratory Software Development with Class Libraries," In *Proceedings of the 7th Joint Conference of the Austrian Computer Society*, Springer, 1992.

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

[Servant 2012] F. Servant and J. A. Jones, "History Slicing: Assisting Code-Evolution Tasks," In *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering* (FSE'12), ACM, Cary, North Carolina, 2012, pp. 1-11.

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

[Stylos 2009] J. Stylos, "Making APIs More Usable with Improved API Designs," PhD Dissertation, Computer Science Department, School of Computer Science, Carnegie Mellon University, 2009.

[Sun 1998] C. Sun and C. Ellis, "Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements," In *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work* (CSCW'98), ACM, Seattle, Washington, USA, 1998, pp. 59-68.

[Sun 2002] C. Sun, "Undo as Concurrent Inverse in Group Editors," *ACM Transactions on Computer-Human Interaction* (TOCHI), vol. 9, ACM, 2002, pp. 309-361.

[Terry 2004] M. Terry, E. D. Mynatt, K. Nakakoji, and Y. Yamamoto, "Variation in Element and Action: Supporting Simultaneous Development of Alternative Solutions," In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI'04), ACM, Vienna, Austria, 2004, pp. 711-718.

[Toomim 2004] M. Toomim, A. Begel, and S. L. Graham, "Managing Duplicated Code with Linked Editing," In *Proceedings of the IEEE Symposium on Visual Languages and Human Centric Computing* (VL/HCC'04), IEEE, Rome, Italy, 2004, pp. 173-180.

[Vakilian 2012] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson, "Use, Disuse, and Misuse of Automated Refactorings," In *Proceedings of the 34th International Conference on Software Engineering* (ICSE'12), IEEE, Zurich, Switzerland, 2012, pp. 233-243.

[Vakilian 2014] M. Vakilian and R. E. Johnson, "Alternate Refactoring Paths Reveal Usability Problems," In *Proceedings of the International Conference on Software Engineering* (ICSE'14), ACM, Hyderabad, India, 2014.

[Vitter 1984] J. S. Vitter, "US&R: A New Framework for Redoing (Extended Abstract)," *SIGSOFT Software Engineering Notes,* vol. 9, ACM, 1984, pp. 168-176.

[Walkingshaw 2013] E. Walkingshaw, "The Choice Calculus: A Formal Language of Variation," PhD Dissertation, Oregon State University, 2013.

[Walkingshaw 2014] E. Walkingshaw and K. Ostermann, "Projectional Editing of Variational Software," In *Proceedings of the 13th International Conference on Generative Programming: Concepts and Experiences* (GPCE'14), ACM, Västerås, Sweden, 2014.

[Xing 2006] Z. Xing and E. Stroulia, "Refactoring Practice: How it is and How it Should be Supported - An Eclipse Case Study," In *Proceedings of the 22nd IEEE International Conference on Software Maintenance* (ICSM'06), IEEE, Philadelphia, PA, 2006, pp. 458-468.

[Yang 1988] Y. Yang, "Undo Support Models," *International Journal of Man-Machine Studies,* vol. 28, 1988, pp. 457-481.

[Yoon 2011] Y. Yoon and B. A. Myers, "Capturing and Analyzing Low-Level Events from the Code Editor," In *Proceedings of the 3rd Workshop on Evaluation and Usability of Programming Languages and Tools* (PLATEAU'11), ACM, Portland, Oregon, USA, 2011, pp. 25-30.

[Yoon 2012] Y. Yoon and B. A. Myers, "An Exploratory Study of Backtracking Strategies Used by Developers," In *Proceedings of the 2012 5th International Workshop on Cooperative and Human Aspects of Software Engineering* (CHASE'12), IEEE, Zurich, Switzerland, 2012, pp. 138-144.

[Yoon 2013] Y. Yoon, B. A. Myers, and S. Koo, "Visualization of Fine-Grained Code Change History," In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing* (VL/HCC'13), IEEE, San Jose, California, USA, 2013, pp. 119-126.

[Yoon 2014] Y. Yoon and B. A. Myers, "A Longitudinal Study of Programmers' Backtracking," In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing* (VL/HCC'14), IEEE, Melbourne, Australia, 2014, pp. 101-108.

[Yoon 2015] Y. Yoon and B. A. Myers, "Supporting Selective Undo in a Code Editor," In *Proceedings of the International Conference on Software Engineering* (ICSE'15), Florence, Italy, 2015.

[Zhang 2009] Y. Zhang, G. Huang, N. Zhang, and H. Mei, "SmartTutor: Creating IDE-Based Interactive Tutorials via Editable Replay," In *Proceedings of the 31st International Conference on Software Engineering* (ICSE'09), IEEE, Vancouver, British Columbia, Canada, 2009, pp. 559-562.

[Zhongxian 2012] G. Zhongxian, "Capturing and Exploiting Fine-Grained IDE Interactions," In *Proceedings of the 34th International Conference on Software Engineering* (ICSE'12), IEEE, Zurich, Switzerland, 2012, pp. 1630-1631.