CODESKIMMER: A NOVEL VISUALIZATION TOOL FOR CAPTURING, REPLAYING, AND UNDERSTANDING FINE-GRAINED CHANGE IN SOFTWARE

BY

CONNOR L SIMMONS

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Adviser:

Professor Brian Bailey
ABSTRACT

Understanding change in software development is a time-intensive yet essential task. Developers must understand the evolution of code in order to properly write, revise, and refactor their own additions. Historically, version control systems were the central method for interpreting how a software project has changed over time. However, these systems are limited in the amount of information they can convey and in the ways they allow users to understand change.

My thesis is a new way of understanding code change. We created CodeSkimmer, a plug-in for Eclipse that can replay every action a developer took while using the editor. The plug-in offers several ways to aid in interpreting the playback. A visualization shows users, graphically, the types of changes that occurred while social, textual, and temporal filters are provided to mark points of interest in the set of recorded operations. These interesting operations are then highlighted through automatic alteration in the playback speed as well as through different visual methods.
Dedicated to my Father -

the most genuine, intelligent, and loving man I've ever known.
ACKNOWLEDGEMENTS

I would like to whole-heartedly thank my adviser, Professor Brian Bailey, for his support and guidance throughout this research process. He constantly shared his strong insight and provided great direction while still giving me the freedom to explore and make key decisions on my own. His passion and pride in his work sparked my interest in this area and drove me forward. I would also like to thank Danny Digg and Stas Negara, whose work I built upon. They provided a really fun project and contributed their own insight and expertise whenever asked. I hope this project helps propel their exciting work forward.

Finally, a great thank you to my friends and family who encouraged me and aided me in my work every step of the way. All your smallest of impacts made the biggest of differences.
# TABLE OF CONTENTS

## CHAPTER 1 INTRODUCTION

### SECTION 3.1: TIMELINE VISUALIZATION

### SECTION 3.2: FILTER AND INFORMATION PANEL

### SECTION 3.3: PLAYBACK

## CHAPTER 2 RELATED WORK

## CHAPTER 3 USER INTERFACE DESIGN

### SECTION 3.1: TIMELINE VISUALIZATION

### SECTION 3.2: FILTER AND INFORMATION PANEL

### SECTION 3.3: PLAYBACK

## CHAPTER 4 USAGE

### SECTION 4.1: LOADING AND UNLOADING OPERATION FILES

### SECTION 4.2: VIEWING INFORMATION FOR A SELECTED OPERATION

### SECTION 4.3: APPLYING A CODE FILTER

### SECTION 4.4: APPLYING A USER FILTER

### SECTION 4.5: NORMAL PLAYBACK

### SECTION 4.6: FAST FORWARDING

## CHAPTER 5 IMPLEMENTATION

### SECTION 5.1: MODEL

### SECTION 5.2: VIEW

### SECTION 5.3: CONTROLLER

## CHAPTER 6 DISCUSSION

## CHAPTER 7 CONCLUSION

## REFERENCES
CHAPTER 1

INTRODUCTION

Imagine a developer, new to a company who is assigned to an existing, mature project. Before the developer can add to the project, time must be spent getting familiar with the code and understanding how others have contributed. The new developer must step through many files, trying to interpret how the different methods, variables, and classes fit with one another. The developer is forced to rely on incomplete snapshots from the team's version control system to try and get a feel for how the existing code evolved over time. Getting a sense for how the project has changed is an extremely daunting task.

When many think of software development, they understand it to be a linear set of actions: when a new feature is to be developed a team draws up requirements, extends the system, and then tests the feature to ensure it meets expectations. However this is not what truly happens in practice. What is overlooked is the nonlinear process of understanding the existing codebase, reviewing new and old code added to the system, and then refactoring, revising and reworking these changes. With this nonlinear approach to comprehending the changes made in code comes a need for tools that allow users to understand and contextualize previous changes.

Change in software has been studied quite extensively over the past decade, and this is for a good reason. One of the most significant amounts of time spent by developers, especially by those who have moved to an already mature project, involves sifting through numerous lines of existing code to learn how the system fits together and what parts of code can be reused and
extended [10]. This initial step of understanding how code has undergone changes is often cited as one of the most important tasks when developing [22]. It is important for developers to understand the reasoning behind changes, as well as the context in which the changes were made. The need to understand change in software has led to new systems that are better able to catalog, visualize, and interact with code.

Version control systems have been central to managing changes, analyzing and storing useful snapshots, and supporting team-based projects. With every commit, users contribute information to the group's knowledgebase [4]. A number of studies, however, have shown the limitations of strictly relying on the information in version control systems to understand how code has changed [14, 18]. As these works suggest, using such information to catalogue change is difficult, for commits are often incomplete and inaccurate in terms of describing a programmer’s intentions. A developer may perform many code changes between each commit which cannot be captured by a standard version control system. Further, there are many other interactions developers have with the code that go beyond simple text changes; they may run tests, automatically change code using their current IDE, etc [13].

One possible way to fill in the gaps of what is missing via the standard version control system is to capture every fine-grained edit a programmer makes throughout development. More comprehensive information on how a user has changed code can be useful for tasks as simple as peer-to-peer code reviews and as complex as performing an analysis of a user’s overall efficiency during development [23]. After capturing these fine-grained tasks, users must be able to efficiently interpret all the collected information in order to gain understanding.
This thesis describes CodeSkimmer, a visualization and replay interface built on top of CodingTracker, an Eclipse plug-in created by Stanislav Nagera and Danny Dig to capture low-level edits during development [13]. CodingTracker works non-intrusively in the background to record every user interaction with the programming IDE and saves all these actions to an external file. This file can then be loaded by our interface.

CodeSkimmer offers a novel visual representation of each interaction with the code. Further, our tool allows the entire sequence of changes to be replayed to recreate exactly the actions previously recorded. Social, textual, and temporal filters can be applied to designate areas of interest, and we offer an original algorithm for tracking selected areas of text as it is modified. Finally, our tool offers a novel way of highlighting interesting operations through automatic alteration of playback speed. In this way, our interface attempts to meet many of the aforementioned needs to understand change during software development.

The remainder of the thesis is organized as follows:

**Chapter 2: Related Work** describes related research into using version control systems to understand change, why such systems fall short, and other existing replay interfaces similar to our own.

**Chapter 3: User Interface Design** reference articles from which we have pulled design ideas and explains some of our choices for how things function and how things are organized in our interface.

**Chapter 4: Usage** shows common replay tasks and how they can be performed using the interface.
Chapter 5: Implementation explains how our interface was coded including some of the algorithms used.

Chapter 6: Discussion explains why we think this interface is novel, current limitations of the tool, and possible ways to extend and enhance the interface.

Chapter 7: Conclusion
CHAPTER 2

RELATED WORK

A number of studies over the past decade have declared that one of the most time-intensive tasks performed by developers is understanding the code changes of others. The following group of studies attempts to address this issue by pulling information from version control systems then processing it for display purposes to other developers.

Some of the earlier methods for visualizing changes in code over time involve analyzing the commits of developers to version control systems. These systems ultimately evolved into the diff viewers that are more common today, and involve analyzing the line numbers and text differences between consecutive commits to determine which lines correspond to one another. With this information researchers were able to graphically display changes at various levels of granularity [3, 4].

A paper from SIGSOFT 2012 showed a tool used for seeing how a file changed throughout its lifetime [21]. Their tool CHRONOS, shown in Figure 2.1, uses a process they name History Slicing to visualize specific selections of interesting code over its commit history. This allowed for users to focus more specifically on code in which they are interested, rather than being inundated with unnecessary data.
Figure 2.1 - A snapshot of CHRONOS showing four different levels of zoom. It is easy to quickly zoom in on areas of interest, but data is limited to the snapshots provided by version control systems.

Our project tool is similar in that we also try to enable the user to focus on specific areas of code. However, because we have access to every recorded interaction with the code, we are able to provide more fine-grained information to the user and show the code changed between commits.

A study performed by Thomas Fritz and Gail C. Murphy explored the kinds of questions developers regularly ask and then discussed the lack of support by existing tools for answering these questions [6]. They attempted to use Information Fragments pulled out of version control systems to identify which users regularly modified different areas of. Their paper focused largely on declaring who "owns" which parts of code. Similar to this idea, researchers from the University of Berne, Switzerland used a different algorithm to extract data from a CVS log and automatically apply an ownership model [7]. They declared that the last developer to commit a line of code "owns" that line, and a developer who owns the most lines of a file owns that file. Using this metric they generated a very interesting timeline, shown in Figure 2.2, which displays the evolution of a codebase over time.
Figure 2.2 - A timeline view of ownership over a codebase labeled with interesting shifts. A general idea of evolution can be determined but it is difficult to understand anything to a finer granularity.

While this can be a useful way to represent code evolution and is good for understanding, from a broad perspective, how users have interacted with certain parts of software, the coarseness of this representation makes it difficult for programmers to quickly gain an understanding of why changes are being made to lines of code and why code ownership has shifted.

These works focus attention on the information that can be pulled from version control systems. However, throughout this period of research there were also many academics discussing the limitations of such approaches. In 2005, Romain Robbes and Michele Lanza wrote a paper surveying the uses and capabilities of current versioning systems, and during this discussion highlighted many of the shortcomings [18]. More recently, a team further promoted this idea, charging that version control systems are ultimately incomplete, imprecise, and impossible to use for measuring the evolution of code. Rather than pushing change to the background, these researchers argued that change should be made first-class citizens [14].

In response to these ideas, a myriad of tools were created with the goal of capturing and effectively using more fine-grained changes that are performed on a set of code. One such tool
was created by Romain Robbes and Michele Lanza following their original paper on the limitations of version control systems. Their team created SpyWare: A Change-Aware Development Toolset. This tool captured snapshots of a project’s code organized as an Abstract Syntax Tree (AST). This method was well-suited for inferring contextual changes made to the program. However, because of this organization method, the system was also dependent on compilable code, meaning that valuable data would still be lost whenever users introduced compilation bugs [20].

Another replay tool, OperationRecorder, was created as an Eclipse plug-in and is very similar to SpyWare. The tool captures all of a users changes by pulling information out of the Eclipse undo history and stores this information in a MySQL database. All these changes are again organized as an AST in order to properly locate where changes were performed in the code; this contextual information is very important but, again, limits the recorder to compilable code. Further, this tool does not include any kind of replay mechanism which is the core component of our project [15].

Fluorite is an Eclipse plug-in published in 2011 that captures fine-grained code in a very similar fashion to our own CodingTracker. Instead of using this information to display back to users, however, the researchers who created Fluorite used the data to analyze shortcomings in current development environments. Their research aims to update coding interfaces to reduce the number of unnecessary keystrokes and typos that occur during implementation. They found that an overly large percentage of developer keystrokes were spent using the backspace key and navigating through code using the keyboard arrows. This branch of information could be interesting to include into our visualization at a later point in time, but we are instead focusing on providing the user an understanding of code evolution and change [23].
A fourth project we found to be extremely close to our work was discussed in two papers from 2012 [8, 9]. The two projects were named Syde and Replay and were created as Eclipse plug-ins to record and replay code changes in an evolving code base. Syde is responsible for recording the fine-grained information regarding the system's evolution. It captures changes at each build cycle and sends these to a central server. However, it works only on Java systems as it models a set of code as an AST representing the various packages, classes, methods, and fields of a Java program. Once changes have been recorded, Replay is then able to interpret the data and present the user with an interface for code playback. There are several different views that enable a user to see the operations organized in different ways: based on time, artifacts (classes, packages, etc), or author (the user that made the change). A snapshot of Replay can be seen in Figure 2.3.

![Figure 2.3 - A snapshot of Replay. The labeled areas are 1) The Replay View. 2) The Replay Editor. 3) The Compare Editor. 4) The Customized Outline View. 5) The Toolbar.](image-url)
However, their replay tool does not include the same features as CodeSkimmer; changes are not recreated in real time for the user. Rather the replayer color-codes the backgrounds of target areas to represent how certain lines have been altered based on selected changes. While this may be useful for getting a general overview of what has happened, it does not recreate the coding experience that our tool emulates.

One very nice feature of Syde is that it works in real-time as users are working on files. The tools give users notifications if their changes are conflicting with others, and displays up-to-date visualizations of work that others have done. Two such views are shown in Figure 2.4.

![Figure 2.4 - The list view and graph view for real-time conflict detection in Syde. This could be a very useful feature for concurrent team development where multiple users may be editing the same file at the same time.](image)

Recently, many approaches to code replay have been researched and offered up to the Computer Science community. Our approach combines some of the ideas of others while adding unique visualizations, variable playback, and novel methods for code selection to aid in change comprehension.
CHAPTER 3

USER INTERFACE DESIGN

This chapter discusses many of the design decisions made while creating CodeSkimmer. We will explain the reasoning behind many of these choices as well as cite several references that inspired pieces of the overall design. Figure 3.1 is a snapshot of the major components of the tool. We have divided this chapter into three sections which correspond to the labeled areas of the figure. Section 3.1 explains the timeline visualization, Section 3.2 covers the tabbed filter and information panel, and Section 3.3 describes the set of playback controls and our automatic variation in playback speed.

Figure 3.1 - A basic view of our Timeline Replayer labeled as follows: 1) Timeline Visualization. 2) Filter and Information Panel. 3) Playback Controls
SECTION 3.1: TIMELINE VISUALIZATION

The timeline is the central visualization for our tool. Each change operation is represented as a vertical bar, color-coded based on the type of change that was made. The spacing between bars and the bar width are a constant number of pixels while the height of the bar is proportional to the number of characters changed during that operation. There are two different highlight bars: one which shows the user's selection that can be made with a mouse click and the other which displays the current position of the playback head. Finally, as different filters are applied to the timeline view, the colors of certain bars are faded.

Each bar is representative of a single operation performed within the Eclipse IDE. Our tool mainly focuses on how the actual code has changed, so we group different text changes together and allow all other operations to fall into an "other" category. The text change operations include adding new text to a file, replacing text in a file, and removing text from a
Meanwhile other operations could include things like starting Eclipse, opening a new file, and saving a file. Each type of operation is assigned a different color. We used green to denote an add text operation, blue to denote a replace operation, and red to denote a remove text operation, as these colors are easy to see and generally consistent with other tools [1, 21].

We implemented the bars to align horizontally because time is typically represented along a horizontal axis, and although in this stage of the tool, the horizontal spacing does not directly relate to any unit of time, we wanted to give users a feel consistent with other playback tools. Many of the other previous replay tools simply listed operations vertically which we felt conflicts with the typical design of playback tools [8, 16]. The bars are equidistant from one another in order to minimize the space needed to show a full sequence of operations. A typical project will have a very large set of operations with many large breaks between groups of changes; using this sequence view in place of a time view is one way to reduce the total horizontal space required.

The vertical height of a bar represents the size of an operation as measured by the number of characters altered during that specific change. The height is scaled at rendering time, and the largest operation is given a height of 50 pixels while the smallest operation is given a height of 5 pixels. Several of the tools we researched had similar approaches to characterizing an amount of change [11, 19, 21]. We included this feature to allow users to quickly focus on certain changes that were significant in size and to quickly distinguish between very different text operations - such as typing a single character and pasting a large block of text. A larger bar simply means that more characters were altered.
We included two different ways of highlighting a certain operation, shown by the yellow and light red bars. The light red bar shows the position of the current playback point; as changes are recreated for the user, the bar progresses to the right to show that an action has been performed. All operations left of the light red bar are operations that were already performed during playback while all operations to the right of the light red bar are future operations that have not yet been recreated. Meanwhile the yellow bar highlights the current selection made by the user via a mouse click. We colored this bar quite differently so that it stands out clearly and can be quickly found by the user.

Lastly, some of the colored bars are faded while others remain opaque. As users apply different filters to the set of operations, the visualization gets updated. Any operation that is marked as interesting to the user, so matches the set of filters, remains solid in color while any operation that is not interesting to the user is faded. Another option we discussed was to hide any bars that were of no interest to the end user, but because one of the central goals of our interface was to allow the user to gain contextual understanding for change, we felt that leaving all operations visible was a stronger option. In this way, one can quickly see how a certain set of changes fits with the overall evolution of the code.
SECTION 3.2: FILTER AND INFORMATION PANEL

Figure 3.3 - The different tabs of the filter and information panel. The first two tabs allow a user to apply the different supported filters while the final tab provides the user with detailed information about a specific operation. Using a tabbed interface saves space and groups similar functionality.

The tabbed filter and information panel lies to the right of the timeline visualization. This area was created using tabbed panels in order to save space and cluster pieces of the interface with similar functions. Here, users can generate new filters to match certain operations, which are made to alter the speed of playback and update the timeline display. The implementation of these filters are discussed at greater length in Chapter 5. The final tab also allows users to see summarized text information about a selected operation.

In general we tried to keep this area as simple and straightforward as possible. We did not focus as much on how the information is displayed as the majority of user attention will be on the timeline visualization and on the Eclipse Editor as operations are being replayed.
SECTION 3.3: PLAYBACK

Figure 3.4 - The set of controls that allow users to load or unload code changes and then play them back.

The final major piece of our tool involves the playback of a set of recorded operations. Pictured in Figure 3.4 is the set of buttons that allows the user to control playback and to load or unload operation files. As with several of the other design choices, we aimed to keep our interface consistent with the common models to which users are already accustomed. Instead of using text buttons, we used common icons to represent the different functions - icons used by a majority of video, audio, or other recording players.

Playback involves recreating recorded user operations and applying them in the Eclipse IDE. As operations are replayed, CodeSkimmer sends commands to the Eclipse IDE that reconstruct exactly what the original developer performed. A user of our tool can see, in real time, text being changed, files being opened or saved, etc - just like watching a movie of the original code being edited. Because our tool recreates these actions just as they were originally performed, the playback results in an exact replica of each file that was edited using Eclipse. This is different from other representations that have been offered before, which use highlighting to simply mark areas of the code that underwent change [8].

Filters are our method for enabling users to specify operation characteristics of interest and have those matching operations emphasized. One way that these operations get emphasized was discussed in Section 3.1 - the bar representation in the timeline view remains opaque while others are faded. A matched operation will also be emphasized during playback. To highlight
operations of interest as they are replayed, we alter their playback speed. The idea for this was drawn from research on Video Skimming [16]. Operations that do not match any of the filters are played as quickly as possible with no delays between consecutive actions, but those operations that do match a filter are slowed to the speed at which they were originally recorded. The implementation of this variable playback speed is discussed further in Chapter 5. Our goal is to allow users to only focus their attention on changes of interest while still providing them a general idea of the context in which those changes were made.
SECTION 4.1: LOADING AND UNLOADING OPERATION FILES

Loading the serialized operation files generated by CodingTracker is a straightforward process. The user simply clicks the Load button via the toolbar.

This opens a standard File Selection Dialog where the user navigates to and selects the appropriate file.

The user is then prompted to enter a username which will be applied to the set of operations. This is needed because usernames are not collected by CodingTracker during recording and must be added to the operations retroactively.
Once the username has been entered, the operations appear in the Timeline Visualization and can be replayed using the playback buttons. There is no limit to the number of operation files that can be loaded - each file will be inserted into the full list based on timestamp.

Operations can be unloaded by pressing the Reset button. This completely clears the workspace of all current projects, removes all filters that have been applied, and clears the list of operations. Essentially it completely resets the replay environment.

SECTION 4.2: VIEWING INFORMATION FOR A SELECTED OPERATION

The user is able to view detailed information about a specific operation. To do so, the user manually clicks on the operation of interest and the choice is highlighted by a yellow bar.
By navigating to the Replay Info tab to the right of the Timeline Visualization, the user can then see information about the operation. In this example we can see information about what text is replaced by the operation, what text is added, the position of this change, the total length of the change, and the timestamp for when the action was originally performed.

SECTION 4.3: APPLYING A CODE FILTER

One type of filter that can be applied to a set of operations is the Code Filter. These filters are added by highlighting and capturing an area of text in the completed (e.g. up-to-date) version of a file. The user opens up the file of interest using the Eclipse Project Explorer
The area of interest is then highlighted with the mouse. In this case we selected the entire `compute()` method.

```java
public class Derived1 extends Base {
    private static final String FIVE_MESSAGE = "DERIVED 5";
    private static final String TEN_MESSAGE = "DERIVED 10";
    private static final String THOUSAND_MESSAGE = "Derived 1000";

    public Derived1(int x, int y) {
        super(x, y);
    }

    public void compute(int z) {
        switch (z) {
            case 5:
                System.out.println(FIVE_MESSAGE);
                break;
            case 10:
                System.out.println(TEN_MESSAGE);
                break;
            case 1000:
                System.out.println(THOUSAND_MESSAGE);
                break;
        }
    }
}
```

The user then navigates to the Code Filter tab and uses the Capture Selection button to add a new code filter. The added filter information is displayed in the text area.
Because the checkbox is selected to also apply all Code Filters to the Timeline View, the visualization is automatically updated. The boxed areas below show operations that meet the filter criteria. The other, faded operations are left for contextual reasons but do not generate a match with our filter. Notice that filtered operations do not need to be contiguous.

SECTION 4.4: APPLYING A USER FILTER

User filters can be added in a simple way. When a user loads in a new operations file, he or she provides a username that is applied to the entire set of operations. In the User Filter tab, a drop down list is automatically populated with a list of every unique username currently loaded. To add a filter, the user selects the appropriate username from the drop-down list and presses the Select User button. The current list of user filters is shown in the text area below these buttons. User filters can also optionally be applied to the timeline visualization, just like the code filters.
SECTION 4.5: NORMAL PLAYBACK

One of the key features of our Replayer is the ability to playback loaded code changes. After a set of operations has been loaded, the user can initialize playback by selecting the Skip Back button. This clears the workspace of all files so that there will be no conflicts when trying to apply the loaded operations.

Once the workspace is ready for playback, the Play and Step Forward options are enabled. To play back the set of changes, the user must simply click play. In this mode, all changes will be recreated without delay unless there is an applied filter. Any operations that match the applied filter will play back at the originally recorded pace. This speed of play is automatically altered at runtime to allow for the user to focus on interesting changes while quickly surpassing others.

The current operation to be recreated is highlighted in the visualization by a light red line. This way the user can follow along to see the current playback state.

During playback, the user has the option to completely stop playback or to pause it. When playback is paused, the user can use the Step button to progress a single operation at a time. This can be useful for reviewing specific changes of interest.
SECTION 4.6: FAST FORWARDING

There are times that the user may want to quickly advance to a specific operation of his or her choice. To do so the user can use the Fast Forward option. First the user makes a selection on the timeline using the mouse.

Now when the user presses the Fast Forward button, playback will progress fast and ignore any filter matches.

Playback will automatically stop exactly one operation before the selection. This way the user can step forward and see the selected change whenever he or she is ready.
CHAPTER 5
IMPLEMENTATION

CodeSkimmer was built as a Java plug-in for the Eclipse IDE, specifically version 3.6 SR1 RCP, and leverages the Eclipse Standard Widget Toolkit (SWT) to create each part of the view. The layout for the plug-in generally follows the standard Model View Controller (MVC) paradigm, so we have structured this chapter according to each of the three paradigm elements. It should be noted that a basis for our replay tool was first built by Stanislav Nagera which was able to recreate the actions recorded by CodingTracker. Our work redesigned the visualization, allowed for the application of filters, and added the automatic speed changes during replay.

SECTION 5.1: MODEL

CodeSkimmer extends the functionality of CodingTracker, an Eclipse plug-in created by Stanislav Nagera and Danny Dig [13]. CodingTracker non-intrusively captures all actions performed by users during their interactions with the Eclipse IDE, then serializes and saves these operations to a file. Our plug-in can then open the file generated by CodingTracker, leverage the same parser to deserialize the information, and gain access to the entire set of recorded operations. Some of the core strengths of CodingTracker is its unobtrusive nature - users do not have to do anything atypical to have their edits recorded - and its simple code capture scheme is not reliant on any inherent code structure. This means that, unlike many other tools, code does not need to be syntactically correct for it to function properly.
Each loaded operation is stored in memory with the information to recreate the associated action and a timestamp denoting when the action originally occurred. During our implementation, we extended each operation to also include the username of the person by whom the action was first performed. The timestamps were used for properly ordering the operations - especially when combining multiple files - as well as simulating real-time playback. To simulate, the playback thread is paused for a period equal to the timestamp difference between consecutive operations.

Filters are an essential part of our tool and allow for the user to specify operations of interest. As represented by Figure 5.1, filters were coded with an abstract parent class to allow for extendibility in the future and require an initialization function - called just before replay - and a boolean check function which, given an operation, returns true if the operation matches the filter or false if it does not. This function gets called just before each operation is replayed. At least one of each filter type must generate a match for the operation to be slowed or highlighted by the timeline visualization. In other words we perform a logical OR within a filter type and a logical AND between filter types. For CodeSkimmer, there are two filter types: username filter and a code selection filter.

Figure 5.1 - The filter hierarchy. Operation Filter is an abstract class and can be easily subclassed to extend the tools' functionality.
The username filter simply checks to see if an operation matches a specific username. It returns true if the usernames match and false if the usernames do not match.

The code selection filter is a bit more complicated because of the fact that code changes over time. A range of interest can therefore fluctuate in both position and length as code is added or deleted. Our selection filter implementation handles this by stepping through the sequence of operations one change at a time, repeatedly updating a user's selection range as needed. This idea is best illustrated by an example:

Assume we have a newly created, empty Java file. At Timestamp 1 a 3-line block of code, Block A, is added to the beginning of the file. At Timestamp 2 another 3-line block of code, Block B, is added to the beginning of the file. Finally, at Timestamp 3, a final 3-line block of code, Block C is also added to the beginning of the file. Thus we have a final file that is 9 lines long and the code blocks in order of beginning of the file to end of the file are C, B, A.
Now a user is interested in seeing all operations that affected any of the first 2 lines of Block A or the last 2 lines of Block B. After Timestamp 3 (when the selection is originally made) this would be the lines in the range [5, 8]. If we simply (and mistakenly) use just the range [5, 8] to check each timestamp for a match, we would immediately fail at Timestamp 1. At this point there are not even 5 lines, but we should return a match as it was during this step that Block A was created! To remedy this, we must start with our initial range of [5, 8] and step backward through our set of operations one timestamp at a time, updating the range while checking for matches.

During Timestamp 3, Block C was added before line 1 and was 3 lines in length, so this timestamp affected the range [1, 3]. This does not overlap our current range of [5, 8] so this operation is not a match. Now we must update our current range of interest by stepping backward. Prior to Timestamp 3, there were 3 fewer lines before both the start of our range and the end of our range, so stepping one operation backward means we must subtract 3 from both our start and end values. This gives us a new range of [2, 5].

During Timestamp 2, Block B was added before line 1 and was 3 lines in length, so again this timestamp affected the range [1, 3]. This time, however, the range of the operation overlaps [2,5], our current range of interest, so we declare that this operation is a match! Now we must again update our current range of interest. Prior to Timestamp 2, there were 3 fewer lines of code before the start and end of our range, so we again subtract 3 from both our start and end values to get a new range of [-1, 2]. Note that we can leave the negative number in our
range even though we know that there can never be code at a negative line number. What matters for this algorithm is the overlap of ranges.

During Timestamp 1, Block A was added at line 1 and was 3 lines in length, so once more this timestamp affected the range \([1, 3]\). Comparing this to our current range of \([-1, 2]\) we again get an overlap so declare this operation as another match. Updating our range of interest one final time we see that 3 lines were added before just the end of our range (and not before the beginning of our range as 1 is not less than -1). We only subtract this number from our end value to get a final range of \([-1, -1]\).

Our code filter class uses a dictionary structure to keep track of which operations are matches and which operations are not. When the user selects a range of code, a filter is created with the range of text selected and the filename in which the user made his or her selection. During the call to initialize, we iterate in reverse order through our list of operations, check for operation
matches, store matches in our dictionary, and update our range of interest based on code changes. Following is the pseudo code for this algorithm.

```plaintext
// Given: OperationList - the full list of operations to play back
// DoesMatch - the dictionary that maps an operation to true or false
// SelectionRange - the range of our selection after our last operation
// Filename - the name of the file in which we have made our selection
initialize:
   for each Operation in OperationList:
      // initialize our Dictionary and our selection range
      DoesMatch(Operation) = false
      CurRange = SelectionRange
   
   // loop BACKWARD from last operation to first operation
   for each Text Operation in OperationList.Reverse:
      // make sure this operation effects the right file
      if  Operation.FileName != Filename:
         next iteration

      // check if this operation is a match
      if rangesOverlap(Operation.editRange, CurRange):
         DoesMatch(Operation) = true

      // update our current selection range
      RemovedLength = Operation.RemovedText.Length
      AddedLength = Operation.AddedText.Length
      if Operation.RangeStart < CurRange.Start:
         CurRange.Start = CurRange.Start - AddedLength + RemovedLength
      if Operation.RangeEnd < CurRange.End:
         CurRange.End = CurRange.End - AddedLength + RemovedLength
```

**SECTION 5.2: VIEW**

The Timeline Visualization was coded as a custom widget which utilizes a Canvas for drawing the different operation bars. Originally, each operation was its own SWT object within the widget for the purposes of modularity and to tap into some of the built-in functionality of the toolkit. However this proved quite limiting, as it introduced a large amount of memory overhead. The implementation was updated so that an entire set of operations is now drawn onto a single image, greatly enhancing the overall performance of the tool.
The buttons, scrollbars, check boxes, text fields, and labels were all implemented using the built-in widgets of SWT. Each of these widgets can be given multiple Event Listeners which enabled us to assign the proper actions via our controller.

In order to allow for flexibility, all pixel sizes, color values, and images were set as static constants. This way values can be updated to suit future needs or satisfy new preferences.

SECTION 5.3: CONTROLLER

Our controller is what ultimately ties the model and the view together. It assigns the various actions to the view parts and controls the flow of information between the different widgets and the model.

The controller is also responsible for handling the playback of operations. When the user selects to replay a set of operations, a new thread is launched to pass commands to the Eclipse IDE and to handle the timing between consecutive actions. This new thread is important because it allows our tool to continue to handle input from the user during playback.

To handle the automatic variation in playback speed, CodeSkimmer checks each operation to see if it matches a user's filters. If the operation does not match the set of filters, the action is executed and no delay is introduced. If the operation has matched the filters, the action is executed and then the thread sleeps for a time equal to the difference between the current operation timestamp and the next operation timestamp minus the time it took for the current action to execute. This effectively simulates the speed at which actions were originally recorded. The simple pseudocode describing this procedure follows.
// Given: currentOperation - the operation we are currently replaying
// nextOperation - the operation we will replay next
// isFastForwarding - boolean indicating the user has pressed fast forward
runOperation:

// Check if we should play quickly or slowly
fastReplay = True
if (isFastForwarding == False && operationMatchesFilter(currentOperation)):
  fastReplay = False

// execute this operation and keep track of how long it takes to execute
startTime = System.currentTime()
executeUserOperationInUIThread(currentOperation)
endTime = System.currentTime()

// apply delay if we are in slow playback
if (fastReplay == False):
  executedOperationTime = currentUserOperation.getTime()
  nextOperationTime = currentUserOperation.getTime()
  delayPeriod = nextOperationTime - executedOperationTime

  // calculate sleep based on delay and time it took operation to execute
  sleepTime = delayPeriod - (endTime - startTime)
  Thread.sleep(sleepTime)
CHAPTER 6
DISCUSSION

We are excited about this tool and feel there are several key features that set it apart from others in similar research areas. The major attribute that distinguishes CodeSkimmer is the automatic alteration of speed during operation playback. Because so much data is captured, adding ways to organize and sort this information is extremely important. Emphasizing points of interest in this way is, we feel, a truly novel feature. Few tools recreate code changes in front of their users and we have seen none that use the dimension of time as a unique way of emphasizing interesting operations. As stated previously, the idea for this feature was drawn from research on Video Skimming - a way to summarize lengthy videos by changing how quickly different are played. In the same way, we are giving the user a summary of a series of code changes. Our visualization is also a distinctive contribution to the field; the timeline view provides a novel way to quickly interpret a sequence of changes. Finally, our algorithm to track text selection is an efficient way to focus the user on interesting code changes.

While many of the included features are exciting, there are also some limitations that are worth mentioning. For one, extremely large operation sets - which would typically be quite common for mature projects - could impact performance as well as inhibit the user's ability to interpret all the information. With the current implementation, Eclipse would struggle to draw all of the operations without exceeding memory limits. This can be solved by introducing new views that allow for zooming at different levels and imposing a max on the number of operations.
that can be viewed at one time. By allowing the user to zoom on specific areas, only a subset of
the operations would need to be drawn. Large sets of operations are also difficult to interpret due
to the overwhelming amount of data. Using zoom views as well as introducing specific time
spacing could prove useful for better enabling the user to understand how various operation sets
fit with one another. One possible way to introduce this would be through use of Ben Bederson's
Jazz Toolkit [2].

Our tool also only centers on the changes in text during development. While this is likely
the major point of focus for users, it could also prove useful to visualize other aspects common to
coding cycles such as showing times when unit tests were run, when builds failed, or when
commits were made to a central repository. Allowing for users to view the different types of
interaction with Eclipse could add new ways of taking in information about a software project or
narrowing down the locations of possible errors.

There are a number of ways that our project can be extended to include greater
functionality. One avenue that we discussed was including other ways to emphasize text
changes during playback. Right now, operations that match user filters are only highlighted
through greater opacity in our Timeline Visualization and through slower playback speeds.
Other ideas could include changing the background color of lines as new text is inserted - a
method that was utilized by the creators of Syde and Replay. This could give users an easier
way to see the types of changes as they occur directly within the Eclipse text editor. Altering the
font size of specific areas of text to contrast it with surrounding code could also give users
another cue for following along and interpreting the types of changes being performed. A final
idea to give emphasis to a specific change could be to play some type of audio alert when the
change is made. This would instigate use of a sense other than sight to help understand operation playback.

Providing the user with different types of views could also improve our tool. Currently users only see the operations in sequence, with no sense of the intensity or spacing. Including views that plot time along the horizontal axis would allow the user to see clusters of operations, see delays during which no actions were taken, and understand new information about the dataset.

This same information could also be represented by graph views. One possible graph would be to show the intensity, the number of operations that occurred during a specified length of time, as a function of time. Users would be able to quickly see points where many changes happened at once. Including our existing filters with these graphs would yield some very interesting potential uses.

It would also be very useful to perform an empirical evaluation of CodeSkimmer by having users attempt to understand code from a real world project. Actions could be recorded using CodingTracker then viewed through our tool, and this data could be compared to that of a standard version control system. The users would then be presented with different tasks. Possible tasks could be answering different questions about how pieces of the code fit together, implementing new classes or methods, fixing known bugs, or simply giving a personal account of which tool allowed them to understand the code more effectively.
CHAPTER 7

CONCLUSION

In this thesis, we presented the concept, design, and implementation of CodeSkimmer, a tool built for understanding change and visualizing fine-grain edits made to code. Our interface allows for fine-grained code changes to be visualized in a time-oriented way and then replayed. The user is able to add user-centric, code-centric, and time-centric filters to identify code of interest and have them highlighted in two major ways. One way highlights the operations of interest in our timeline visualization, and the second way involves the automatic alteration in the speed of playback. This gives users more time to see the effect of interesting changes while less time is wasted displaying changes of little interest. We also presented a unique algorithm, used by our code filters, which tracks ranges of code over time as they are manipulated by changes.

While many of the decisions we made were aimed to produce a tool well-suited for all programmer needs, we recognize that CodeSkimmer is still far from perfect. There are current limitations that reduce the effectiveness of the tool for extremely large operation sets. New ways of visualizing the recorded actions could prove immensely helpful in interpreting the large amount of data collected by CodingTracker. Efforts were made during implementation to allow CodeSkimmer to be extended easily to add new ways of specifying interest and to show interesting changes in different ways.

We hope that this project serves as a solid stepping stone toward reducing the time spent by developers reviewing and understanding code. Currently, this area is being very actively
explored and there are many exciting ideas beginning to surface that could greatly increase the efficiency of development among software engineering teams.
REFERENCES


