Visualization of Fine-Grained Code Change History

YoungSeok Yoon
Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
youngseok@cs.cmu.edu

Brad A. Myers, Sebon Koo
Human-Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
bam@cs.cmu.edu, sebonk@andrew.cmu.edu

***Abstract*—Conventional version control systems save code changes at each check-in. Recently, some development environments retain more fine-grain changes. However, providing tools for developers to use those histories is not a trivial task, due to the difficulties in visualizing the history. We present two visualizations of fine-grained code change history, which actively interact with the code editor: a timeline visualization, and a code history diff view. Our timeline and filtering options allow developers to navigate through the history and easily focus on the information they need. The code history diff view shows the history of any particular code fragment, allowing developers to move through the history simply by dragging the marker back and forth through the timeline to instantly see the code that was in the snippet at any point in the past. We augment the usefulness of these visualizations with richer editor commands including selective undo and search, which are all implemented in an Eclipse plug-in called "AZURITE". AZURITE helps developers with answering common questions developers ask about the code change history that have been identified by prior research. In addition, many of users' backtracking tasks can be achieved using AZURITE, which would be tedious or error-prone otherwise.**

Keywords—program comprehension; software visualization; integrated development environments; selective undo

I. INTRODUCTION

Software developers use version control systems (VCSs) such as Subversion and Git to keep the history of how the source code evolved over time. Developers 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, developers can better understand the source code by looking at the evolution histories. This can be useful when reviewing code changes or before modifying any existing codebase written by others. Second, developers 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 developer experimenting with different implementations or from different developers working independently. Finally, the histories are not only useful for the developers, but are also useful for the researchers who are interested in how software is developed over time. Mining software repositories [1] is known to be an effective research methodology and there is even a whole conference on this topic.

In recent years, there has been a growing belief among software engineering researchers that automatically recorded finer-grained change histories are needed in order to avoid the significant information loss between two consecutive snapshots inherent in VCSs [2, 3, 4, 5, 6, 7]. The basic idea is to keep all the small low-level changes such as individual insertion, deletion, and replacement of text. Recently, this approach has been shown to be feasible [4, 5, 8], and there have been attempts to make use of these fine-grained histories in two different ways.

The first way is to help developers understand the code evolution by recording and replaying fine-grained changes in the integrated development environments (IDEs) [6, 9, 10]. One experiment showed that developers can answer software evolution questions more quickly and correctly when provided with a replay tool. The second way is to analyze the history data for research purposes. This approach has also been successfully used to identify programmers' common coding practices such as backtracking [11] and refactoring [12].

However, the potential applications of the fine-grained histories have not been fully explored. There could be some cases where the history can be useful for the developers, while VCSs cannot provide the same benefits. For example, we are developing a selective undo [13] feature for code editors which allows developers to undo a change made a while ago, without affecting the later changes that are irrelevant to the change being undone. There are many cases where VCSs cannot help with this kind of selective revert, which we call "backtracking". For example, the desired code may not be in the repository at all. The revert feature of VCSs could be inadequate even when the code is in the repository, when wanted and unwanted code are intermixed in the current code as often happens [11, 14].

Unfortunately, it is not a trivial task to provide useful tools for developers using these fine-grained code change histories. The main problem is information overload; developers make a huge number of low-level changes while editing source code. Without proper visualization and filtering mechanisms, it is hard for developers to focus on the information they need. This becomes a basic requirement for any richer editing commands, such as various forms of searching, undo and redo, which would be executed on the past changes.

Various factors make it difficult to visualize the change history especially in the code editing context. For example, many existing selective undo user interfaces for graphical editors display a list of all of the low-level editing operations along with human-readable descriptions of the individual operations [13, 15, 16]. However, text editing operations are often so fine-grained and numerous that it is hard for the users to interpret the high level editing intent just by looking at the individual edits. In addition, graphical applications can use a thumbnail to represent a snapshot of the document at a certain point of time, which makes it easier to present the edit history to the user [17, 18, 19, 20, 21]. In contrast, a thumbnail of a large text file does not give much information to the users.

978-1-4799-0369-6/13/$31.00 ©2013 IEEE

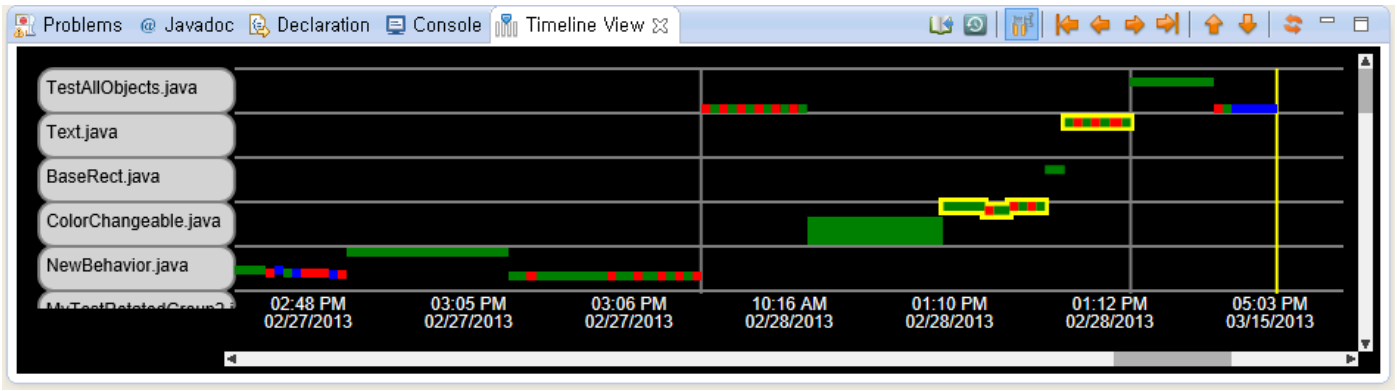

Fig. 1. Timeline Visualization of AZURITE. Each row contains the history of a single file. Each rectangle represents a single edit operation. Rectangles are color-coded by the type of edit (Inserts=green, Deletes=red, Replacements=blue). The horizontal axis represents time, which is currently not linear because the timeline is in compact mode. The vertical location of each rectangle within a row indicates the relative location in the file where the edit was performed. The vertical gray lines (the first two lines from the left) divide sessions, and the vertical yellow line (the first from the right) indicates "now". The view can be arbitrarily zoomed and scrolled, both horizontally and vertically, and the rectangles can be selected with the mouse, which highlights them in yellow.

This paper presents two user interfaces specifically designed to visualize fine-grained code change history while overcoming the problems described above: a *timeline visualization*, and a *code history diff view*. The timeline visualization (see Fig. 1) displays the changes in a two-dimensional space controlled by various filtering mechanisms. New edits in the code are displayed *as they are made*, and users can also load past editing histories. One or more edit operations can be selected in the timeline to execute various editor commands on those selected operations, such as highlighting the relevant code and selective undo of only the selected operations.

The code history diff view (see Fig. 3) shows the history of a particular code fragment. An arbitrary area of code can be selected and the code history diff view can be launched for that specific section of code. Developers can then move through the history back and forth by dragging the marker in the timeline to see the evolution of that fragment.

These two visualizations closely interact with each other and also with the code editor. The flexibility of these two visualizations make it easy to answer the history related questions frequently asked by developers [22, 23]. Moreover, the editor commands built on top of these visualizations make it possible to help developers achieve certain tasks, which could not be done with any existing tools. In order to show the feasibility of these visualizations, we implemented them in an Eclipse plug-in called AZURITE[1], as described next.

II. TIMELINE VISUALIZATION

A. Basic Features

The timeline visualization of AZURITE is shown in Fig. 1. Unlike most other tools that display the edit history in a linear list [6, 9], here the edit history is displayed in a two-dimensional space. The horizontal axis represents time, and the time keys are shown along the x-axis. Each row contains the edit history of one file.

Individual changes 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 tool captures more editor commands, but we only display these three primitive edit types because all editing operations that change the code result in one of these three types, and we wanted to minimize the information overload as much as possible. Other filtering options could be trivially added, for example to show only the deletes. The horizontal location and width of a rectangle represents the time and duration of the edit performed. The vertical location and height of a rectangle within the row represents the relative location of the edit within the file. 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, so that the user can see all the files and the entire history of all edits, or the specific details of one editing session.

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 most recently 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.

Note that unlike the undo stack, the edit history contains *all* the edits that have ever been performed, in chronological order. Any undo operations 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 shown as a tooltip which is shown on mouse hover. The tooltip (see Fig. 2) contains the exact time when the edit has made, and the text that was inserted and/or deleted by that edit.

B. Layout Modes

AZURITE's timeline visualization supports two layout options: *real-time mode* and *compact mode*. In real-time mode,

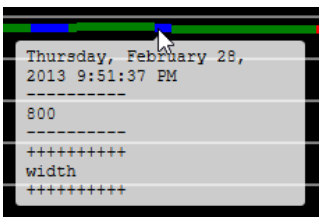

Fig. 2. An example tooltip. The detailed timestamp is shown at the top. The deleted text is shown between the lines with minus signs (-), and the inserted text is shown between plus signs (+). The blue rectangle under the cursor represents an edit near the top of the file that replaced a hard-coded constant "800" with a variable named "width".

[1] AZURITE is a blue mineral, and here stands for Adding Zest to Undoing and Restoring Improves Textual Exploration. The plug-in and detailed information about AZURITE can be found at: `http://www.cs.cmu.edu/~azurite/`.

the rectangles are horizontally located proportionally to the actual time that they were made. This is a trivial option in terms of implementation, but it turned out there is a significant problem with this approach. There are many gaps between the changes because developers use only about 20% of their time actually editing code [24], which makes it difficult to navigate through the edit history in the timeline.

To resolve this problem, AZURITE provides a compact mode, which is used by default. 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. Fig. 1 shows the compact mode.

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.

C. Selecting Changes and Invoking Commands

The user 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 (see Fig. 1). 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.

The first command in the menu is "selective undo" which undoes only the selected changes while keeping the other changes unaffected. Note that AZURITE allows rectangles to be selected across multiple files and undone, which is a significant advantage over conventional undo which only works on a single file. Another command is "undo everything after the selection," a convenient way to revert the whole file at once. Note that this "revert" is put into the timeline like any other operation, so users can easily change their mind and undo it.

Other commands vary depending on the number of selected edits. When there is exactly one selected operation, users can choose "jump to this location" to open the relevant file in the code editor and move the cursor to the location where the operation was performed. The same command can be invoked by double-clicking a rectangle in the timeline. To perform this operation correctly, AZURITE must take into account any later-performed operations that might have changed the code and its location in the file, as explained below (Sec. IV.A).

When multiple operations are selected, users can choose "show all files edited together," which shows all the files that were edited in the same timeframe when the selected operations were performed. In the future, we will investigate to what extent it makes sense to provide a "jump to locations" command to allow users to focus the code editors on *multiple* blocks of code at once, since this is not directly supported by any code editor today. We will also add further commands to this menu, as described below in Sec. VI.

D. Storing and Viewing 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. At the right-most edge of each session, a gray vertical line is shown to indicate the boundary between the two adjacent 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.

E. Filtering and Searching Changes

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 together, and (4) show all files in the history.

Users can also *search* the edit history to find the information they need. The history search feature is invoked from the code editor menus, and the search results are shown in the timeline as selected operations. Currently, 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 [11]. 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. This search is also used by the code history diff view (see Sec. III). 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 enough 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, we used our selective undo feature to calculate the snapshot at each point and check if the snapshot contains the desired code or not. 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 are considered.

F. Implementation

AZURITE's timeline visualization is written in HTML-5 / CSS / JavaScript, and it communicates with the backend through the embedded browser interface of Eclipse. We used this approach for one of our previous tools [25] and it has several merits over using IDE-specific APIs such as SWT and JFace. First, since web development is very popular, the visualization toolkits tend to be more mature than the IDE-specific toolkits. Also, using web development technologies theoretically makes the timeline reusable across multiple IDEs.

The drawing part is written using the Scalable Vector Graphics (SVG) format [26] and the JavaScript package D3.js [27], which means that we were able to implement the zooming and scrolling without much extra effort.

G. Performance Evaluation

The timeline should not significantly affect the response time of the code editor. According to our field study data[2], the average number of edits per week was 8,480, assuming 40 hours of work a week. Thus, we measured the response time of several important operations of our timeline with 500 and 10,000 rectangles, which approximates two hours and one

[2] The data was collected for 295 hours of coding activities from 5 professional developers using FLUORITE [5].

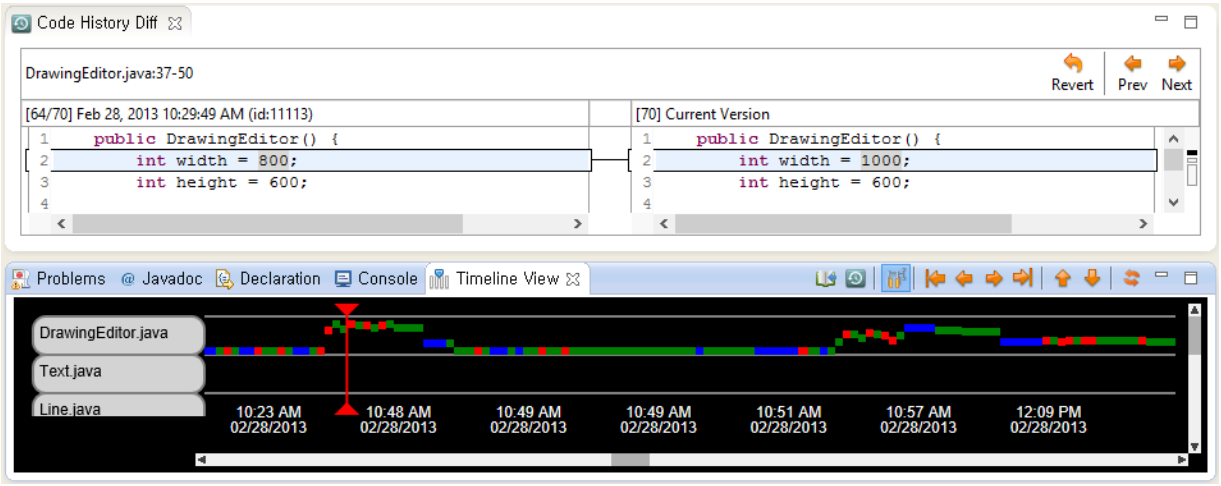

Fig. 3. Code history diff view of AZURITE. The previous version of the selected code from 10:29am is shown in the left panel, and the most recent version of the code is shown in the right panel. Users can move through the history by either clicking the Prev/Next navigation buttons at the top right, or dragging the vertical red marker shown in the timeline, which instantly updates the code on the left panel and the diffs. Multiple code history diff views can be shown at the same time.

week of work, respectively. The time was measured on a PC running Windows 8 and Internet Explorer 10 with a 2.60GHz CPU[3]. The results are summarized in Table 1.

Overall, the compact mode is much slower than real-time mode because of the calculations to remove the gaps. The only operation that is automatically performed while the developer is editing code is Add Rectangle, which only takes a negligible time (35ms) even in compact mode with 10,000 rectangles, which means that the timeline is non-intrusive. The other three operations in Table 1 are called only when the user interacts with the visualization. Horizontal scrolling takes significantly more time than vertical scrolling because it needs to re-calculate the time guides at the bottom. A Layout routine recalculates the positions of all rectangles, and it is called when the file filtering option is changed. Although Layout takes more than 12 seconds for 10,000 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 our tool. In addition, this problem could be mitigated by optimizing the performance by pre-caching some values and grouping multiple operations into one big rectangle for old history items.

III. CODE HISTORY DIFF VIEW

A. Basic Features

The code history diff view is shown in Fig. 3. Users can select an arbitrary code snippet from the regular Eclipse code editor windows and launch this view, which is a code-compare view with two juxtaposed panels. The left-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. The right-hand panel always shows the current version (i.e., the most recent version) of the code snippet. The diffs between the two code snippets are marked. The left side code can go all the way back to when the code did not even exist, assuming that this is in the history.

Users can move back and forth through the history in two ways. First, they can drag the marker within the timeline with mouse to see how the code looked at the time at which the marker is pointing. The marker is a red vertical line, as shown in Fig. 3, which appears in the timeline when a code history diff view is shown. This design was inspired by the time marker in video editors. The code snippet shown on the left panel changes instantaneously as the marker is moved, and diffs are recalculated as well. Alternatively, users can use the navigation buttons above the code 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. In addition, this view can be used to look for some deleted code and copy the code to reuse it in the current code. 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.

To implement this view, AZURITE first searches for all edits performed on the code snippet, and then reconstructs all the intermediate snapshots using the selective undo feature.

B. Scope of Code Snippets

While most other tools only provide file-based history or method-based history at best, 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.

One limitation of our code history diff view is that it can only handle a single contiguous block of code. One 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

TABLE I. SUMMARY OF MEASURED RESPONSE TIMES (IN MS)

	Compact mode		Real-time mode	
	# of rectangles		# of rectangles	
Operation	500	10,000	500	10,000
Add Rect	3	35	3	29
H-Scroll	45	174	26	68
V-Scroll	6	11	6	14
Layout	140	12,383	23	234

[3] The browser information is relevant because the timeline runs in an embedded browser. Different browsers are used by different platforms.

are multiple code snippets coupled together around a certain feature, which has been referred to as a working set [24]. 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.

IV. Underlying Data Structures

A. Dynamic Segment Management

To support these visualizations, a full code editing history must be kept. At least the following information must be kept to determine the position and size of each rectangle and for displaying the tooltip: the *type* of each edit operation (i.e. Insert, Delete, or Replace), the *offset* representing the location where the operation was performed within the file, the *length* of the inserted or deleted text, the actual *text* that was inserted or deleted, and the precise *timestamp* indicating when the edit was performed. AZURITE collects these data using our FLUORITE [5] logging tool, which is included with the AZURITE plug-in.

However, this information cannot be directly used to support many of the most useful commands. For example, when performing selective undo on some past operations, the implementation must be able to tell *where* those operations were performed in the *current* state of the code. This is not trivial because the offsets change whenever some text is inserted or removed above the location in the file of the edit. Thus, before a selective undo operation can be performed, all offsets of operations above it in the file need to be adjusted dynamically. Additionally, these dynamic positions are required to support searching for all of the edits performed in a certain area of code.

The basic idea here is similar to the *dynamic pointers* introduced in the collaborative editing context [28]. Our approach differs in that we keep track of dynamic *segments* instead of pointers to single positions. A segment is composed of offset and length. The dynamic segment calculation mechanism used in AZURITE is illustrated in Fig. 4.

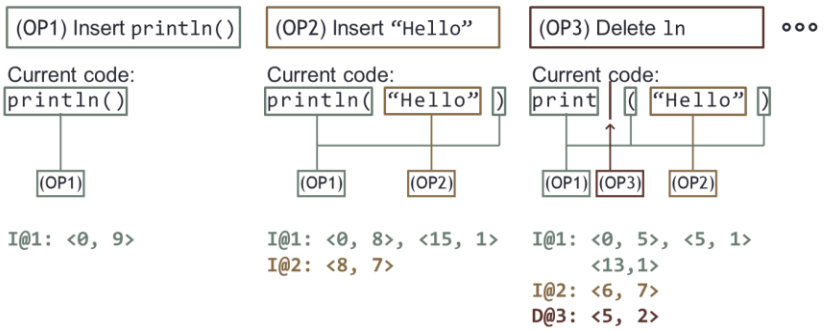

Fig. 4. Illustration of dynamic segment management. Each dynamic segment is denoted as <*offset*, *length*>. OP1 inserts `println()`, OP2 inserts `"Hello"` within the parentheses, and then OP3 deletes `ln` from the method name `println`, in temporal order. Below the code is illustrated how the existing dynamic segments are updated or split as new operations are performed.

B. Detecting Changes Made Outside of the IDE

Source code can be changed 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 AZURITE would be out of sync when the file is reopened later. Moreover, there is even a possibility of IDE crash, cutting off the history file.

Since we are dealing with individual incremental changes instead of full snapshots, a single missing item in the edit history can confuse the entire history. To avoid this problem, AZURITE 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, AZURITE 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. Similarly, when reading the history of past sessions, AZURITE compares the final snapshot of each file in the past sessions, and the known initial snapshot of that file in the next session to check the validity of the history. This process is done using the Google-diff-match-patch open-source library [29, 30]. The high-level architecture of the entire system is shown in Fig. 5.

C. Granularity of Changes

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 [5], 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 size of the scope that could be visualized in the code snippets (Sec. III.B) would be determined by the granularity of changes recorded in the logs.

V. Example Use Cases

This section lists some example use cases where these visualizations and information filtering mechanisms can be used to solve real-world problems that previous research shows that software developers face. This provides evidence that these tools will be useful.

A. Answering History-Related Questions Developers Ask

Prior research has identified many hard-to-answer questions developers ask as part of their development activities [23, 31, 32]. These include the following history-related questions (quoted directly from [23]), which can be easily answered using AZURITE's visualizations. Note that answering these history questions can also be an effective strategy for answering high-

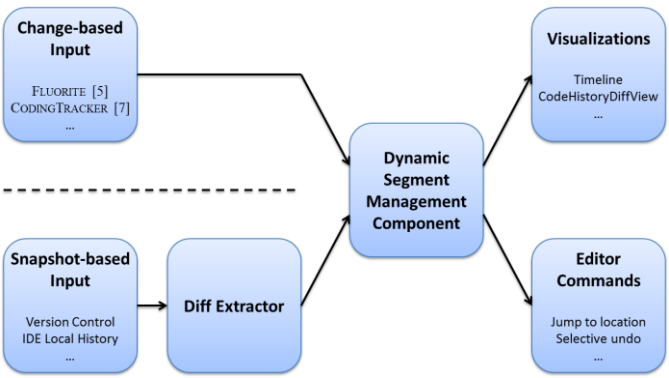

Fig. 5. The high-level architecture of the entire system. Each node represents a component in the system, and the arrows indicate the data flow. Different input sources can be used to provide the same set of visualizations. In case of snapshot-based input sources, diffs between two consecutive snapshots should be extracted first.

er-level rationale questions, as pointed out in [23].

1) When, how was this code changed or inserted?
2) How has it changed over time?
3) 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.

4) What recent changes have been made?

Developers can load the history of the past editing session to see what recent changes have been made to the project. 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.

5) 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, using the marker position shown in the timeline, the user can select by time across all the edited files the range of operations involved in that change. Then the other files edited together within that timeframe can easily be determined by invoking the "show all files edited together" command from the context menu.

6) How did this ever work?

Although it would not directly answer this question, AZURITE can help provide some clues by allowing developers 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.

B. Selective Undo Scenarios

In our previous study, we identified several problems developers face while trying to go back to a previous state, which we call "backtracking" [11]. Based on those observations, we list some scenarios where the visualizations and selective undo feature can help with backtracking tasks. Since these scenarios often occur in between version control commits, version control systems would probably not help in these situations.

1) Reverting LayoutManager to a Previously Used One

When programming graphical user interfaces (GUIs), one often ends up having to experiment with different layout managers to get the UI to look as desired. Imagine the following scenario: A developer is implementing a GUI dialog in Swing. She first writes the code with `GridBagLayout`, and then changes to a simpler `BoxLayout` manager because of the complex parameters of `GridBagLayout`. 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 can be done 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 everything to this point" 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 had the developer commented out the `GridBagLayout` code instead of deleting it. However, we observed that even the developers who explicitly said they regularly commented out code, occasionally deleted code which turned out to be needed later [11].

2) Restoring Deleted Code in General

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

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 relatively 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. This works even 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 regular text search would be less useful.

4) Aborting or Undoing a Manual Refactoring[4]

Murphy-Hill et al. discovered that developers perform refactoring manually about 90% of the time [33]. Aborting a manual refactoring in the middle, or undoing it sometime later could be tedious, because there can be many steps of changes and even multiple files involved to achieve a certain 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, they can find the other files that were edited together using AZURITE's filtering feature and 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.

C. Other Benefits

Developers often need to remember the previously attended locations in code when resuming from an interruption or when switching between tasks [34]. The timeline can be used as automatic bookmarks of recently edited locations in such a situation. Using the timeline, developers 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

[4] This example was not observed from our study, but derived from the first author's personal experience of using AZURITE during his own work.

the timeline. Unlike manual bookmarks, this works well even when there are a great many files in the developer's workspace.

VI. Future Work

A. Supporting a Team Development Environment

One limitation of the current version of Azurite is that it only handles a single developer'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?" [31]. In order to achieve this, it would be best if the edit history was somehow kept in the version control system with code, and Azurite would need to be able to deal with the code merging situations.

B. Providing New History Search Features

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 we support the most common kinds of search through the history already (as discussed above), we envision that novel kinds of history search would be helpful. Our goal is to enable users to express whatever they remember about the previous edits or situations that they want to select in the history. In particular, we plan 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,
- a specific or a sequence of edit operations happened (e.g., copy-and-paste from the web, an Extract Method refactoring),
- a particular point in real time ("last Thursday"), or a range of time ("last week"),
- a set of semantically equivalent (or similar) code edits were made in various locations (crosscutting concerns),
- a particular task was being completed (e.g., as tracked by task management systems such as Mylyn [35]), or
- any combination of these.

Using these options, one could search for "all edits since the last commit related to `println` statements," for example.

C. Providing Richer Commands Executed on Past Operations

With the capability of filtering and selecting past changes provided by the timeline, we are able to provide more editor commands that can be executed on some past operations. We demonstrated some commands such as selective undo in Section II.C, but we also plan to provide more commands including the following:

- highlight all of the affected code in the code editor,
- redo (apply the same change to other similar code fragments),
- add a checkpoint / bookmark / annotation (so that users can revert back to this point at any time, or make a note about a set of changes or a point in time),
- commit to the repository (without committing the other changes), or
- find and select code or edit operations which are semantically related to the selected ones. For example, if the current edit changes the name of a variable, find all the other edits of that variable name.

VII. Related Work

Dwell-and-spring [36] is a recent selective undo mechanism for direct manipulation. It provides an interface for undoing any press-drag-release interaction, while Azurite works on editing operations.

Some text editors, such as Emacs [37] and DistEdit [38], support regional undo, where the user undoes the operation that affected a specific selected region of text. Regional undo can be achieved in Azurite by searching for all edits for the region of code and invoking selective undo. 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 [39]. In this case, Azurite includes all the partially overlapping dynamic segments (see IV.A) and undoes them, which means that some code right outside user's selection could be reverted.

There are a few systems which track fine-grained code change history similar to Fluorite [5]. OperationRecorder [4] associates each edit with the corresponding abstract syntax tree (AST) node. CodingTracker [7, 8] also tracks fine-grained code editing data, but mostly focuses on analyzing developers' coding behaviors such as refactoring. IDE++ [40] is a system that captures all types of IDE interactions, which are not limited to code edits. However, none of these provide the visualization, history view or undo operations of Azurite.

There are systems that provide history search, which has also been called "history slicing." OperationSliceReplayer [6] uses the AST data kept by OperationRecorder [4] to filter the changes that affected a certain class member. Chronos [41] 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.

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 [42]. 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 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.

There are other edit history visualizations using timelines. Chronos [41] 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 [43] is a visualization for presenting the *social history* of a software project, similar to Facebook's Timeline. Developers 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, but it is primarily designed for helping people recall and share memories, not for providing editor commands as provided by Azurite. The software evolution Storyline [44] is another timeline visualization which focuses on who contributed to the project over time.

Most VCSs provides a way to trace the history of a single file, which is presented as a linear list of relevant changesets along with their commit messages. Xcode 4 has a feature called

Version Editor [45], 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. The local history feature of some IDEs keeps snapshots of each file automatically upon file save (Eclipse [46]) or as the code changes (NetBeans [47]). The local history shows a linear list of saved snapshots of each file, but only with their timestamps without any human-readable descriptions. These approaches are limited in that the history can only be seen at file level, and it can be hard to find the desired snapshots.

VIII. 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. We demonstrate in this paper that these fine-grained histories can also be useful for developers with proper visualizations and several editor commands which tightly integrate with the history. We believe that providing more refined editor commands with more history search options would make developers more comfortable in code editing, fostering more exploration and more reliable backtracking. This approach might also be applied to regular text editors, and possibly even to graphical editors.

Azurite is an open-source Eclipse plug-in. More information about Azurite can be found, and the plug-in can be downloaded, at: http://www.cs.cmu.edu/~azurite/.

Acknowledgments

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

References

[1] 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, 2007, pp. 77-131.

[2] R. Robbes and M. Lanza, "A Change-based Approach to Software Evolution," Electronic Notes in Theoretical Computer Science, vol. 166, 2007, pp. 93-109.

[3] R. Robbes and M. Lanza, "SpyWare: a change-aware development toolset," Proc. Intl. Conf. on Soft. Eng. (ICSE'08), 2008, pp. 847-850.

[4] T. Omori and K. Maruyama, "A change-aware development environment by recording editing operations of source code," Proc. Intl. working Conf. on Mining Soft. Repositories (MSR'08), 2008, pp. 31-34.

[5] Y. Yoon and B. A. Myers, "Capturing and analyzing low-level events from the code editor," In PLATEAU'11, 2011, pp. 25-30.

[6] K. Maruyama, E. Kitsu, T. Omori, and S. Hayashi, "Slicing and replaying code change history," In ASE'12, 2012, pp. 246-249.

[7] 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 ECOOP'12, 2012, pp. 79-103.

[8] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, R. Z. Moghaddam, and R. E. Johnson, "The Need for Richer Refactoring Usage Data," In PLATEAU'11, 2011.

[9] L. Hattori, M. D'Ambros, M. Lanza, and M. Lungu, "Software Evolution Comprehension: Replay to the Rescue," In ICPC'11, 2011, pp. 161-170.

[10] L. Hattori and M. Lanza, "Syde: a tool for collaborative software development," Proc. Intl. Conf. Soft. Eng. (ICSE'10), 2010, pp. 235-238.

[11] Y. Yoon and B. A. Myers, "An exploratory study of backtracking strategies used by developers," In CHASE'12, 2012, pp. 138-144.

[12] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson, "Use, disuse, and misuse of automated refactorings," Proc. Intl. Conference on Software Engineering (ICSE'12), 2012, pp. 233-243.

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

[14] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," Proc. Intl. Conf. on Soft. Eng. (ICSE'11), 2011, pp. 351-360.

[15] B. A. Myers and D. S. Kosbie, "Reusable hierarchical command objects," In CHI'96, 1996, pp. 260-267.

[16] B. A. Myers, "Scripting graphical applications by demonstration," In CHI'98, 1998, pp. 534-541.

[17] D. Kurlander and S. Feiner, "Editable graphical histories," Proc. 1988., IEEE Workshop on Visual Languages 1988), 1988, pp. 127-134.

[18] 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 CHI'02, 2002, pp. 1-8.

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

[20] D. Kurlander and S. Feiner, "A Visual Language for Browsing, Undoing, and Redoing Graphical Interface Commands," Visual languages and visual programming, 1990, p. 257.

[21] M. Chii, M. Yasue, A. Imamiya, and M. Xiaoyang, "Visualizing histories for selective undo and redo," Proc. 3rd Asia Pacific Computer Human Interaction 1998), 1998, pp. 459-464.

[22] R. Holmes and A. Begel, "Deep intellisense: a tool for rehydrating evaporated information," In MSR'08, 2008, pp. 23-26.

[23] T. D. LaToza and B. A. Myers, "Hard-to-answer questions about code," In PLATEAU'10, 2010, pp. 1-6.

[24] A. J. Ko, H. Aung, and B. A. Myers, "Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks," In ICSE'05, 2005, pp. 126-135.

[25] C. Omar, Y. S. Yoon, T. D. LaToza, and B. A. Myers, "Active code completion," In ICSE'12, 2012, pp. 859-869.

[26] World Wide Web Consortium, "Scalable Vector Graphics (SVG) 1.1," 2011; http://www.w3.org/TR/2011/REC-SVG11-20110816/.

[27] M. Bostock, "D3.js - Data-Driven Documents," 2012; http://d3js.org/.

[28] G. D. Abowd and A. J. Dix, "Giving undo attention," Interacting with Computers, vol. 4, 1992, pp. 317-342.

[29] N. Fraser, "google-diff-match-patch - Diff, Match and Patch libraries for Plain Text," 2012; http://code.google.com/p/google-diff-match-patch/.

[30] E. W. Myers, "An O (ND) difference algorithm and its variations," Algorithmica, vol. 1, 1986, pp. 251-266.

[31] T. Fritz and G. C. Murphy, "Using information fragments to answer the questions developers ask," In ICSE'10, 2010, pp. 175-184.

[32] A. J. Ko, R. DeLine, and G. Venolia, "Information Needs in Collocated Software Development Teams," In ICSE'07, 2007, pp. 344-353.

[33] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," In ICSE'09, 2009, pp. 287-297.

[34] C. Parnin and S. Rugaber, "Programmer information needs after memory failure," In ICPC'12, 2012, pp. 123-132.

[35] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," In FSE'06, 2006, pp. 1-11.

[36] C. Appert, O. Chapuis, and E. Pietriga, "Dwell-and-spring: undo for direct manipulation," In CHI'12, 2012, pp. 1957-1966.

[37] Free Software Foundation, "Undo - GNU Emacs Manual," http://www.gnu.org/software/emacs/manual/html_node/emacs/Undo.html.

[38] A. Prakash and M. J. Knister, "A framework for undoing actions in collaborative systems," ACM Trans. Comput.-Hum. Interact., vol. 1, 1994, pp. 295-330.

[39] R. Li and D. Li, "A regional undo mechanism for text editing," Proc. Intl. Workshop on Collaborative Editing Systems (IWCES'03), 2003.

[40] G. Zhongxian, "Capturing and exploiting fine-grained IDE interactions," In ICSE'12, 2012, pp. 1630-1631.

[41] F. Servant and J. A. Jones, "History slicing: assisting code-evolution tasks," In FSE'12, 2012, pp. 1-11.

[42] S. Hayashi, T. Omori, T. Zenmyo, K. Maruyama, and M. Saeki, "Refactoring edit history of source code," In ICSM 2012, pp. 617-620.

[43] A. Kuhn and M. Stocker, "CodeTimeline: Storytelling with versioning data," In ICSE'12, 2012, pp. 1333-1336.

[44] M. Ogawa and K.-L. Ma, "Software evolution storylines," In Proc. SOFTVIS'10, 2010, pp. 35-42.

[45] Apple Inc., "What's New in Xcode 4," https://developer.apple.com/technologies/tools/whats-new.html

[46] Eclipse Foundation, "Eclipse - The Eclipse Foundation open source community website.," http://www.eclipse.org/.

[47] Oracle Corporation, "NetBeans IDE," http://netbeans.org/.