Visualizing and Navigating Source Code History

Alex Bradley*
Department of Computer Science
University of British Columbia

ABSTRACT

We present the Source Code History Navigator, a tool for visualizing the evolution of source code and comparing many revisions of a source code file. Our tool provides an overview of the entire history of a source file through a "history flow"-style visualization in which revisions are plotted as vertical pixel stripes, with each pixel line in a stripe corresponding to a line in the revision and coloured according to some property of interest (e.g., authorship of the line, differences from previous and next revisions, line type, or code age.) Stripes can be expanded to obtain a detail view showing the entire text of the revision. Many such text viewers can be opened side-by-side simultaneously, and their scrollbars can be locked together, facilitating easy comparison of differences across multiple revisions. We present a detailed usage scenario to demonstrate the applicability of our tool.

Index Terms: D.2.6 [Software Engineering]: Programming Environments—Programmer workbench; D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—Version control

1 INTRODUCTION

Software developers working on large project teams frequently need to explore the history of the source code with which they are working. They may wish to gain understanding of important design decisions that led to the current state of the code, to discover which developers contributed to the evolution of a given feature, or to determine when, and by whom, malfunctioning code was introduced. A common way to approach these tasks is to consult previous revisions of relevant source code files stored in a version control system such as CVS, Subversion or Perforce.

Using current integrated development environments (IDEs) and revision control GUIs, it is easy for a user to obtain a high-level overview of the revision control history of a file, in the form of a list of revisions giving revision numbers, dates, authors and check-in notes. Current tools also make it easy to compare the text of two source code files to see what changes occurred. However, the developer may need to compare several past revisions to gain the desired understanding of the code's evolution. Using current tools, this may require a time-consuming and error-prone process of requesting many pairwise comparisons between revisions.

Developers may also want to gain a "big picture" overview of how the text of a source code file changed over time through all its revisions. Research prototypes such as CVSscan [19] and the Visual Code Navigator [11] address this task by providing compact graphical "stripe" visualizations of code history. Such tools, however, are not integrated into current IDEs, and do not provide strong support for detailed textual comparison of many revisions.

We present a tool called the *Source Code History Navigator* (SCHN) to address the problems described above. SCHN allows developers to visualize and navigate the evolution of an individual source code file. "Stripe" visualizations in the manner of CVSscan provide a high-level overview of the entire history of the file. Each stripe can be expanded into a viewer showing the text of a revision. Many such viewers arranged side-by-side permit easy comparison of many revisions of a file. SCHN is implemented as an Eclipse plugin, providing tight integration with a major IDE.

The structure of this paper is as follows. Section 2 reviews work related to our approach. Section 3 describes the features of SCHN in detail, and section 4 gives details of the implementation approach used for its development. Section 5 provides a usage scenario demonstrating the applicability of our tool. Section 6 discusses the strengths and weaknesses of our tool and lessons learned during its development. Section 7 provides directions for future work, and section 8 concludes.

2 RELATED WORK

We consider related work in the areas of text comparison and software evolution visualization (including one paper in the category of document evolution visualization.) The contribution of our tool lies in applying some of the strengths of current text comparison GUIs to many-revision text comparisons and in integrating that text comparison capability into a pixel-stripe visualization of source code history. Our focus+context approach to combining those levels of overview and detail is somewhat inspired by the Table Lens [14], in which textual details of spreadsheet rows are interspersed with compact graphical representations of spreadsheet data.

2.1 Text Comparison GUIs

Many GUI tools exist for text comparison, including Eclipse's built-in comparison tools, KDiff3 [5], Beyond Compare [15], Perforce Visual Client/P4Merge [13], and Araxis Merge [2]. These tools typically provide two- or three-way file comparison with strong support for highlighting differences in colour, highlighting character-by-character differences in a line, linked scrolling of the files being compared, and visualizing deletions. Three-way comparison is often targeted at the special case of merging revisions (comparing an ancestor revision to two conflicting changes.)

2.2 Software and Document Evolution Visualization

The idea of showing code in a compact graphical representation by converting code lines to pixel lines is well known in the software visualization literature. The early SeeSoft system created by Ball and Eick [3] explored the idea of representing large codebases by showing many files in such a condensed graphical representation according to various colouring schemes. Subsequent systems such as Augur [6] and Tarantula [9] built on this idea. Tarantula shows codebases coloured according to test suite coverage. Augur shows multiple code files coloured according to properties such as code age; it also moves towards software evolution visualization, as it allows users to explore changes in the code over time using a slider.

Showing code evolution through a horizontal sequence of vertical pixel stripes also has roots in multiple previous papers. Viégas et al. [17] created "history flows", with lines coloured by authorship, to visualize Wikipedia edit history. Voinea [18, 19, 20] has explored the domain of software evolution visualization extensively and has

*e-mail: awjb@cs.ubc.ca

produced a variety of visualization techniques for exploring the history of source code files and entire codebases. Voinea's CVSscan tool [19] includes a pixel-stripe revision visualization similar to "history flows", but incorporates different colouring strategies and has more sophisticated filtering techniques to accommodate the potentially large size of code files. CVSscan also includes a "details-on-demand" feature which allows the user to select a single revision from the flow and view the associated source code. The Visual Code Navigator of Lommerse et al. [11] incorporates Voinea's work.

3 Source Code History Navigator

Our proposed solution, the Source Code History Navigator (SCHN), is based on the information visualization paradigms of overview+detail and focus+context. An overview of file history is provided through a graphical display of vertical revision "stripes". Double-clicking on any stripe expands it into a detail view showing the text of the corresponding revision. These detail views can be seen as areas of "focus", with the surrounding unexpanded stripes providing context. The overview+detail paradigm also applies on the level of the individual source code viewer; the corresponding stripe remains visible to the right of the viewer and can be used for quick navigation to any point in the file.

Our solution can also be seen in terms of Tufte's concept of small multiples [16]. Revision stripes are essentially a small-multiples representation of revision history, placing compact representations of revisions side-by-side to show their evolution over time. A sequence of many text viewers placed side by side in the horizontally scrollable SCHN interface can also be seen as a small multiples display; the user can make a rapid horizontal scan of many revisions to detect changes of interest.

3.1 History Flow of Revision Stripes

Following Viégas et al. [17], we adopt the name *history flow* for our graphical representation of source code history. A history flow consists of a sequence of thin vertical bars, each bar corresponding to one revision. The bars are ordered in ascending or descending order of revision date (the ordering can be configured by the user.) In each bar, the horizontal pixel line n pixels distant from the top edge of the bar corresponds to the n-th line of the text of the revision. Each pixel line can be assigned a colour according to one of several colouring schemes, as described in section 3.1.1. Above each revision stripe is a gold cylinder icon. As this is the same icon used to represent individual revisions in the Eclipse IDE's History view, it serves as a hint to Eclipse users that each stripe represents a revision. Hovering over this icon with the mouse will reveal a tooltip displaying the revision number, author, date and check-in note (see Figure 1.) Black marker lines divide the revision icons into groups of five, allowing the user to use the line of icons as a simple measurement device.

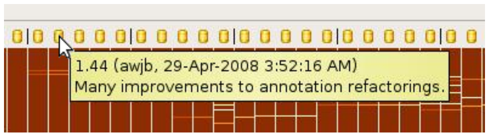

1.44 (awjb, 29-Apr-2008 3:52:16 AM)
Many improvements to annotation refactorings.

Figure 1: Closeup of the top of a SCHN history flow. Mousing over a revision icon reveals a tooltip with revision information. Black markers every five revisions allow the user to use the line of icons as a simple measurement device.

3.1.1 Colouring Schemes

SCHN currently provides the following revision stripe colouring schemes. The user can switch between schemes using a menu.

Differences from next and previous revisions. In this scheme, the differences between each revision and the revisions immediately preceding and following it are computed. Lines where a revision differs from its neighbours are marked in light grey. When the user mouses over a revision stripe, the differences between that revision and its neighbours are displayed using the three-colour "Dark2" qualitative colour palette from ColorBrewer [7]. Lines where the revision differs from only its previous neighbour, or only its next neighbour, are coloured blue or green. Lines where the revision differs from both neighbours are coloured orange. An example of this colouring scheme can be seen in Figure 2.

Line authorship. In this scheme, each author who has submitted a revision is assigned a colour from the nine-colour "Set1" qualitative colour palette from ColorBrewer. (If there are more than nine distinct authors, the same colour will be assigned to multiple authors.) Each line in a revision stripe is then assigned the colour representing the last author who modified that line. An example of this colouring scheme is shown in Figure 3.

Line type. In this scheme, lines are coloured according to Java line type using a simple transformation of the standard Eclipse colouring scheme. (SCHN currently only processes Java files, although it can in principle be extended to handle any type of source code file.) The colour assigned to a line is the colour that would be assigned to the first non-whitespace character in the line using Eclipse Java syntax highlighting. For example, lines beginning with keywords such as `import`, `class`, `public`, `if`, etc. are coloured purple, Javadoc comments are coloured light blue, non-Javadoc comments are coloured green, lines beginning with method calls are coloured black, etc. An example of this colouring scheme is shown in Figure 4.

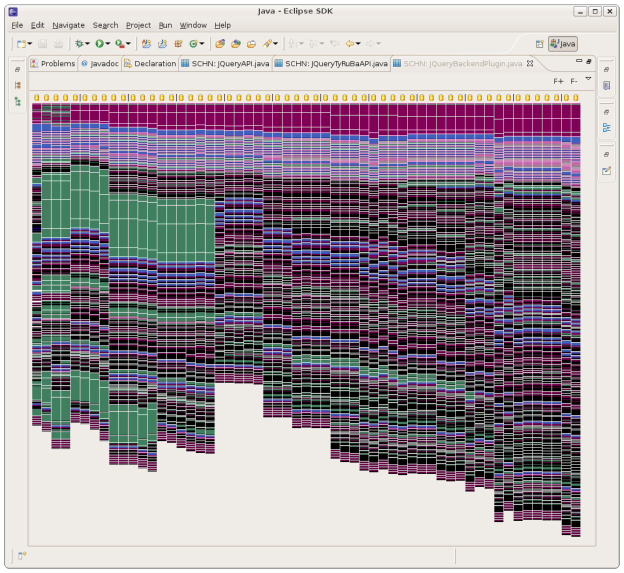

Java - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Java
Problems
Javadoc
Declaration
SCHN: JQueryAPI.java
SCHN: JQueryTyRuBaAPI.java
SCHN: JQueryBackendPlugin.java
F+ F-

Figure 4: A SCHN history flow in which revision stripes are coloured by Java line type.

Code age. In this colouring scheme, each line in a revision stripe is coloured according to the number of revisions that have elapsed since that line was last modified. The colours are selected from the eight-colour YlOrBr sequential colour palette from ColorBrewer. A line that was modified in the current revision is coloured off-white. A line that was modified n revisions ago ($n \leq 7$) is assigned the n-th subsequent colour in the palette. Lines that were modified seven or more revisions ago are assigned the last colour in the palette (dark brown.) Intuitively, this scheme can be described

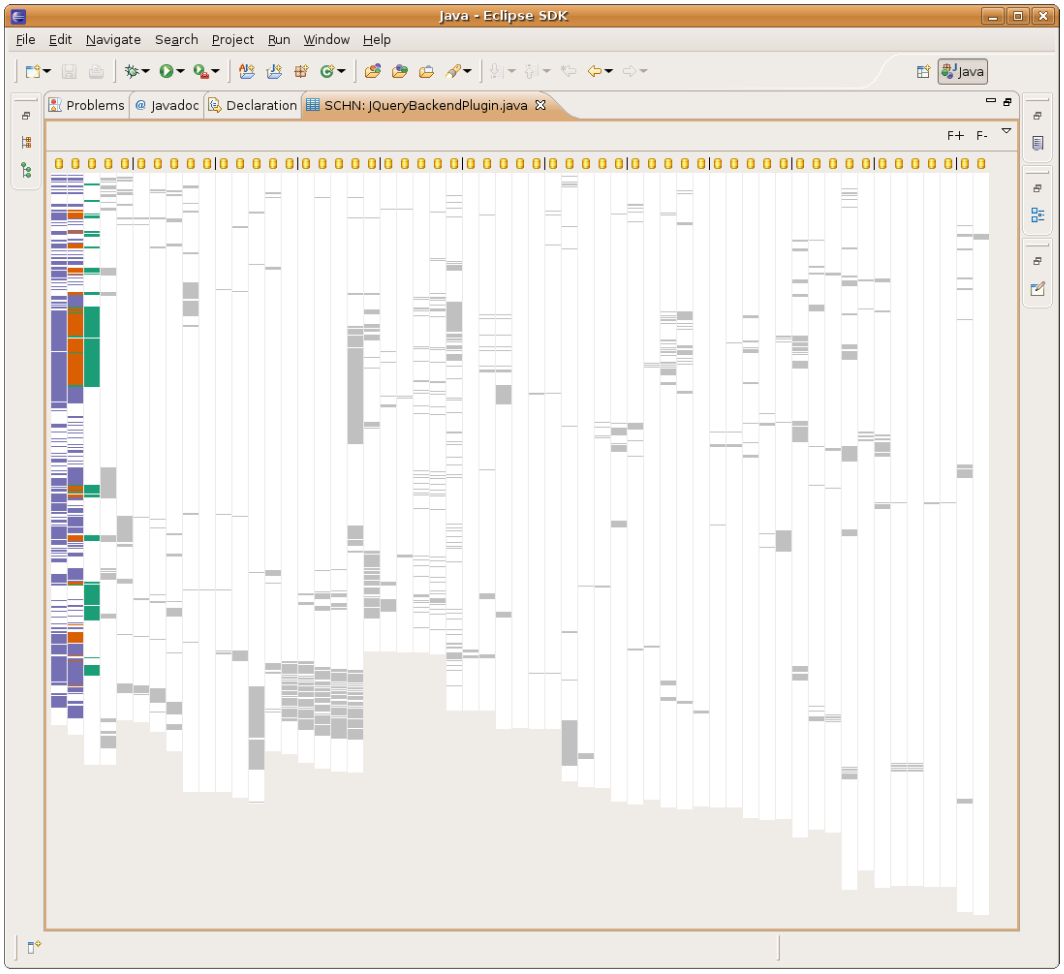

Java - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Java
Problems
Javadoc
Declaration
SCHN: JQueryBackendPlugin.java
F+ F-

Figure 2: A SCHN history flow coloured to show the differences between revisions and their neighbours. The second revision has been selected through mouse-over. Differences between the first and second revision are shown in blue; differences between the second and third revisions are shown in green. Lines where the second revision differs from both the third and first revisions are coloured orange. Differences in other revisions are shown in light grey.

as one in which code blocks start out "white hot", then "cool down" over time through progressively darker shades of orange to brown. An example of this colouring scheme is shown in Figure 5.

3.2 Revision Source Code Viewers

Double-clicking on a revision stripe in the history flow expands it into a source code viewer. (See Figure 6 for an example.) The viewer is scrolled to display the line corresponding to the pixel on which the user double-clicked. The background of each line in the source code viewer is coloured according to a transformation of the colouring scheme that is applied to the stripes. The transformation applied (for the sake of text legibility) is an alpha blend against a white background ($\alpha = 0.3$). (For the neighbour-differences colouring, the same mouseover behaviour applies to the text viewers as to the stripes.) Eclipse Java syntax highlighting is applied to the source code in the viewer. The number, author and date of the revision are displayed in a label above the viewer, and the revision check-in note is displayed in a text field below the viewer. (This text field can be suppressed if the user desires more vertical viewing space.)

The corresponding revision stripe remains visible to the right of an expanded source code viewer and acts as an overview of the file and a navigation aid. A black rectangle in the revision stripe shows the location in the overall revision text of the area currently displayed in the text viewer. Clicking or dragging in the revision stripe scrolls the revision text viewer accordingly.

The horizontal and vertical scrollbars of all opened revision text viewers can be locked together for easier comparison of multiple revisions. When scrollbars are locked, scrolling one viewer vertically will automatically scroll all other viewers so that matching lines (according to the computed differences between revisions) are side-by-side. Scrolling one viewer horizontally will scroll all other viewers by the same offset.

A resize handle to the right of a source code viewer-stripe pair allows the viewer to be resized horizontally to suit the user's convenience. Double-clicking on the handle resizes the viewer to the length of the longest line in the revision.

The user interface provides options to expand and collapse all viewers and to hide stripes that do not currently have viewers open. It is also possible to increase and decrease the font size of the text in the viewers. Selecting a small font size (e.g., 4pt) can be seen as providing a mid-level overview, more condensed than full-size text but less condensed than the one-line-per-pixel stripe overview representation.

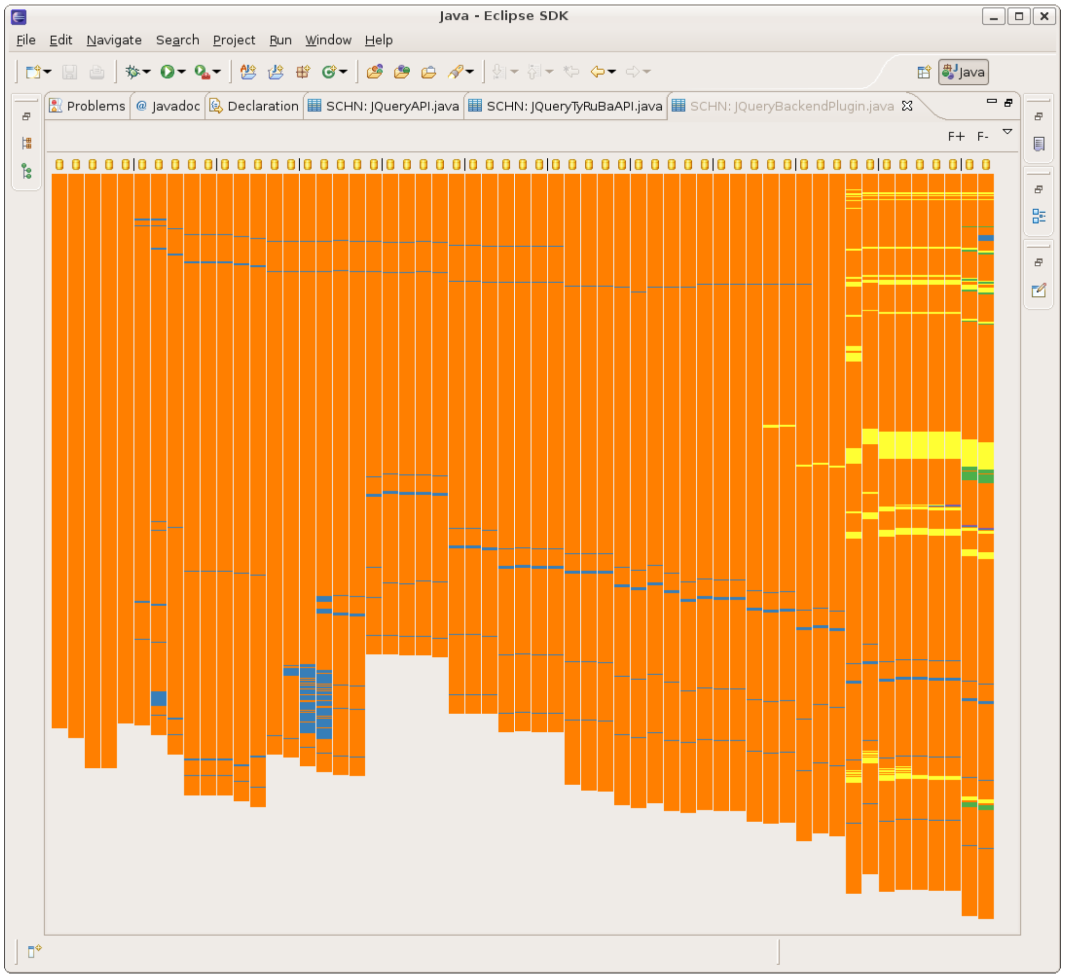

Java - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Java
Problems
Javadoc
Declaration
SCHN: JQueryAPI.java
SCHN: JQueryTyRuBaAPI.java
SCHN: JQueryBackendPlugin.java
F+ F-

Figure 3: A SCHN history flow in which revision stripes are coloured to show the last author who revised each line.

4 Implementation Approach

SCHN is implemented as an Eclipse plugin using the Java SWT graphics library. SCHN uses an Eclipse extension point to add an “Explore history…” menu item to the context menu for Java files, allowing the user to select Java files in the IDE’s Package Explorer view for analysis with SCHN. Clicking this menu item displays a new Eclipse view showing the SCHN user interface.

SCHN leverages several Eclipse libraries to provide various features. Java syntax highlighting is provided by Eclipse JDT, and CVS repository access uses Eclipse’s internal CVS library. Differences between revisions are computed using Eclipse’s `RangeDifferencer` library [1], which implements the comparison algorithm of Miller et al. [12].

Eclipse JDT syntax highlighting is also exploited to provide the Java line type revision stripe colouring. This colouring is computed by examining the foreground colours of the initial non-whitespace characters in each line in the corresponding source code viewers. (Source code viewers are always initialized, even when they are not visible.)

As previously mentioned, colour palettes were generated using ColorBrewer [7]. Colours were checked for suitability for the colour-blind using Vischeck [4].

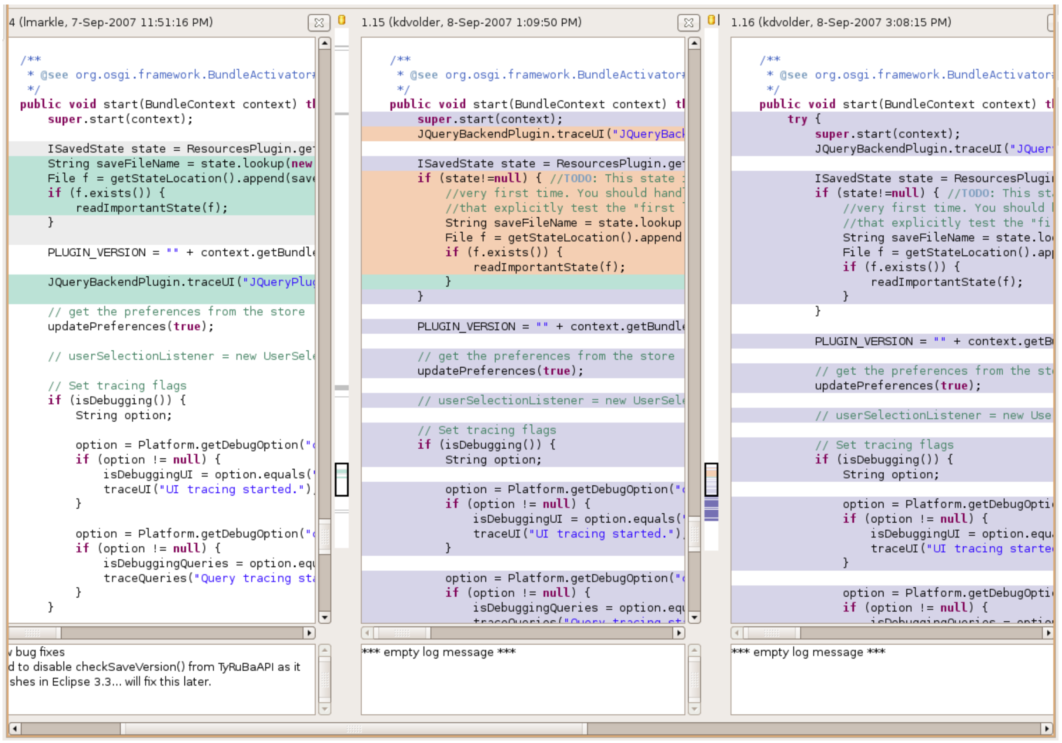
5 Usage Scenario

A usage scenario based on exploring the history of a source code file for the JQuery [8] project demonstrates how SCHN might be useful to a developer. We will call the three actual JQuery developers involved in this example A, B, and C.

Suppose that Alice, a new developer on the JQuery project, is finding that some of the extension points for the JQuery Eclipse plugin do not function as desired. (Extension points allow Eclipse plugin developers to extend plugins provided by others through a well-defined interface.) Knowing that some configuration of extension points takes place in `JQueryBackendPlugin.java`, she wants to explore the history of this file to determine which developers introduced certain extension points.

She begins by opening SCHN on the file and selecting authorship colouring (see Figure 3.) Extension point definitions are located near the end of the file (at the location of the lowest yellow and green lines in the rightmost vertical stripe.) Expanding viewers for the current revision, 1.57, and the previous revision, she can see that the predicates extension point was added by user A (the author assigned the green colour) in the previous revision. She can confirm this fact by expanding a viewer for revision 1.55 and selecting the differences-from-neighbours colouring scheme.

She next notes that another extension point is coloured yellow. She select authorship colouring and follows the yellow-coloured blocks from revision 1.55 back to revision 1.49. By us-



4 (lmarkle, 7-Sep-2007 11:51:16 PM)

```
/**
 * @see org.osgi.framework.BundleActivator#
 */
public void start(BundleContext context) th
    super.start(context);

    ISavedState state = ResourcesPlugin.get
    String saveFileName = state.lookup(new 
    File f = getStateLocation().append(save
    if (f.exists()) {
        readImportantState(f);
    }

    PLUGIN_VERSION = "" + context.getBundle

    JQueryBackendPlugin.traceUI("JQueryPlug

    // get the preferences from the store
    updatePreferences(true);

    // userSelectionListener = new UserSele

    // Set tracing flags
    if (isDebugging()) {
        String option;

        option = Platform.getDebugOption("c
        if (option != null) {
            isDebuggingUI = option.equals("
            traceUI("UI tracing started.");
        }

        option = Platform.getDebugOption("c
        if (option != null) {
            isDebuggingQueries = option.equ
            traceQueries("Query tracing sta
        }
    }
```

v bug fixes
d to disable checkSaveVersion() from TyRuBaAPI as it
shes in Eclipse 3.3... will fix this later.

1.15 (kdvolder, 8-Sep-2007 1:09:50 PM)

```
/**
 * @see org.osgi.framework.BundleActivator#
 */
public void start(BundleContext context) th
    super.start(context);
    JQueryBackendPlugin.traceUI("JQueryBack

    ISavedState state = ResourcesPlugin.get
    if (state!=null) { //TODO: This state i
        //very first time. You should handl
        //that explicitly test the "first t
        String saveFileName = state.lookup
        File f = getStateLocation().append
        if (f.exists()) {
            readImportantState(f);
        }
    }

    PLUGIN_VERSION = "" + context.getBundle

    // get the preferences from the store
    updatePreferences(true);

    // userSelectionListener = new UserSele

    // Set tracing flags
    if (isDebugging()) {
        String option;

        option = Platform.getDebugOption("c
        if (option != null) {
            isDebuggingUI = option.equals("
            traceUI("UI tracing started.");
        }

        option = Platform.getDebugOption("c
        if (option != null) {
            isDebuggingQueries = option.equ
            traceQueries("Query tracing st
```

*** empty log message ***

1.16 (kdvolder, 8-Sep-2007 3:08:15 PM)

```
/**
 * @see org.osgi.framework.BundleActivator#
 */
public void start(BundleContext context) th
    try {
        super.start(context);
        JQueryBackendPlugin.traceUI("JQuery

        ISavedState state = ResourcesPlugin
        if (state!=null) { //TODO: This sta
            //very first time. You should h
            //that explicitly test the "fi
            String saveFileName = state.lo
            File f = getStateLocation().ap
            if (f.exists()) {
                readImportantState(f);
            }
        }

        PLUGIN_VERSION = "" + context.getB

        // get the preferences from the st
        updatePreferences(true);

        // userSelectionListener = new Use

        // Set tracing flags
        if (isDebugging()) {
            String option;

            option = Platform.getDebugOpti
            if (option != null) {
                isDebuggingUI = option.equ
                traceUI("UI tracing starte
            }

            option = Platform.getDebugOpti
            if (option != null) {
                isDebuggingQueries = optio
```

*** empty log message ***

Figure 6: SCHN revision text viewers, coloured according to differences between the center revision and its neighbours.

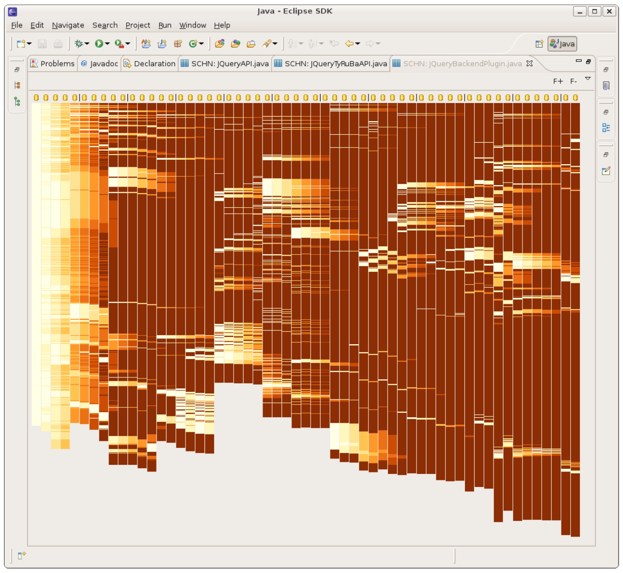

Java - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Java
Problems Javadoc Declaration SCHN: JQueryAPI.java SCHN: JQueryTyRuBaAPI.java SCHN: JQueryBackendPlugin.java

Figure 5: A SCHN history flow in which revision stripes are coloured by code age.

ing differences-from-neighbours colouring, she can confirm that the "include rules" extension point was introduced in revision 1.49 by user B (the author assigned the yellow colour.)

The authorship trail is no longer useful after this point because the remaining extension points are coloured orange, and the author assigned orange (user C) did most of the work on the file. However, Alice would like to find out where the remaining extension points were introduced. She decides to use code age colouring, uses the "expand all" UI feature to expand all text viewers, and uses the "lock scrollbars" feature to ensure all text viewers are scrolled to the location in the file where extension points are defined. She then starts at revision 1.49 and scrolls rapidly to the left. As long as the viewers she sees have a dark brown background colour, she knows that no changes to the code block are occurring and keeps scrolling. As the background colours start to become lighter, she knows a change is approaching and slows down. After scrolling to the left in this manner for many revisions, she discovers that the code block containing extension points was initially introduced by user C in revision 1.13 (see Figure 7.)

At this point, Alice knows which developers were responsible for adding all the extension points in the plugin. She can consult them to determine how to fix the problems with the extension points of concern.

6 Discussion

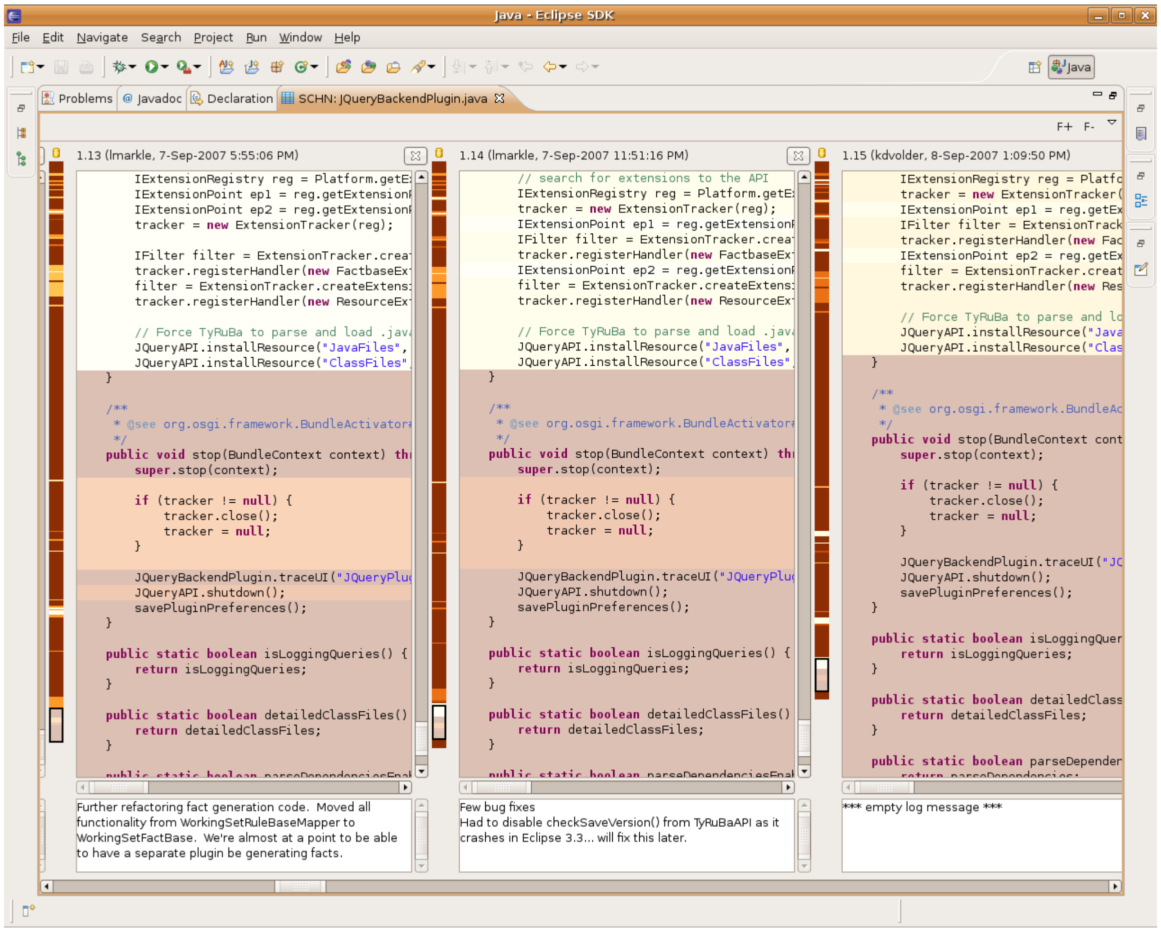
In this section, we evaluate the strengths and limitations of our tool and discuss some lessons learned during its development.

6.1 Strengths

We believe the strengths of SCHN include the following:

- **Tight integration with a major IDE.** Since SCHN is implemented as an Eclipse plugin, developers using Eclipse on a CVS codebase can install the plugin and immediately apply it to their code without any extra processing steps.

- **Revision stripe colourings provide useful insights into code evolution.** All of our colourings provide information about the full revision history of the code that would be difficult to ascertain with traditional techniques such as scanning a textual list of revision summaries or performing pairwise comparisons. The authorship and code age colourings even provide information about *individual* revisions that would be



Java - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Problems
@ Javadoc
Declaration
SCHN: JQueryBackendPlugin.java
F+ F-
Java

1.13 (lmarkle, 7-Sep-2007 5:55:06 PM)

```
        IExtensionRegistry reg = Platform.getE
        IExtensionPoint ep1 = reg.getExtensionP
        IExtensionPoint ep2 = reg.getExtensionP
        tracker = new ExtensionTracker(reg);

        IFilter filter = ExtensionTracker.crea
        tracker.registerHandler(new FactbaseEx
        filter = ExtensionTracker.createExtens
        tracker.registerHandler(new ResourceEx

        // Force TyRuBa to parse and load .java
        JQueryAPI.installResource("JavaFiles",
        JQueryAPI.installResource("ClassFiles"
    }

    /**
     * @see org.osgi.framework.BundleActivator
     */
    public void stop(BundleContext context) thr
        super.stop(context);

        if (tracker != null) {
            tracker.close();
            tracker = null;
        }

        JQueryBackendPlugin.traceUI("JQueryPlug
        JQueryAPI.shutdown();
        savePluginPreferences();
    }

    public static boolean isLoggingQueries() {
        return isLoggingQueries;
    }

    public static boolean detailedClassFiles()
        return detailedClassFiles;
    }
```

Further refactoring fact generation code. Moved all functionality from WorkingSetRuleBaseMapper to WorkingSetFactBase. We're almost at a point to be able to have a separate plugin be generating facts.

1.14 (lmarkle, 7-Sep-2007 11:51:16 PM)

```
        // search for extensions to the API
        IExtensionRegistry reg = Platform.getE
        tracker = new ExtensionTracker(reg);
        IExtensionPoint ep1 = reg.getExtensionP
        IFilter filter = ExtensionTracker.crea
        tracker.registerHandler(new FactbaseEx
        IExtensionPoint ep2 = reg.getExtensionP
        filter = ExtensionTracker.createExtens
        tracker.registerHandler(new ResourceEx

        // Force TyRuBa to parse and load .java
        JQueryAPI.installResource("JavaFiles",
        JQueryAPI.installResource("ClassFiles"
    }

    /**
     * @see org.osgi.framework.BundleActivator
     */
    public void stop(BundleContext context) thr
        super.stop(context);

        if (tracker != null) {
            tracker.close();
            tracker = null;
        }

        JQueryBackendPlugin.traceUI("JQueryPlug
        JQueryAPI.shutdown();
        savePluginPreferences();
    }

    public static boolean isLoggingQueries() {
        return isLoggingQueries;
    }

    public static boolean detailedClassFiles()
        return detailedClassFiles;
    }
```

Few bug fixes
Had to disable checkSaveVersion() from TyRuBaAPI as it crashes in Eclipse 3.3... will fix this later.

1.15 (kdvolder, 8-Sep-2007 1:09:50 PM)

```
        IExtensionRegistry reg = Platfo
        tracker = new ExtensionTracker(
        IExtensionPoint ep1 = reg.getEx
        IFilter filter = ExtensionTrack
        tracker.registerHandler(new Fac
        IExtensionPoint ep2 = reg.getEx
        filter = ExtensionTracker.creat
        tracker.registerHandler(new Res

        // Force TyRuBa to parse and lo
        JQueryAPI.installResource("Java
        JQueryAPI.installResource("Clas
    }

    /**
     * @see org.osgi.framework.BundleAc
     */
    public void stop(BundleContext cont
        super.stop(context);

        if (tracker != null) {
            tracker.close();
            tracker = null;
        }

        JQueryBackendPlugin.traceUI("JQ
        JQueryAPI.shutdown();
        savePluginPreferences();
    }

    public static boolean isLoggingQuer
        return isLoggingQueries;
    }

    public static boolean detailedClass
        return detailedClassFiles;
    }

    public static boolean parseDepender
```

*** empty log message ***

Figure 7: An illustration of our usage scenario. By scrolling through the revisions using the code age colouring, Alice discovers that certain extension points first appeared in revision 1.13.

extremely laborious to calculate manually from pairwise revision comparisons.

- **Revision stripe colouring choices are grounded in previous work.** The authorship, neighbour-differences and line type colourings are based on those used by Voinea et al. [19] and Lommerse et al. [11]. The code age colouring was inspired by SeeSoft (cf. [3], figure 3.)

- **Revision text viewer navigation is linked to the revision stripe on the right**, which serves as an overview of the revision text and a fast navigation tool. Placing the revision stripe on the right builds on the Eclipse IDE's standard practice of providing annotations to the right of a text editor.

- **Allowing the user to lock the horizontal and vertical scrollbars of all revision text viewers is very useful for many-revision comparisons**, as all the revisions displayed can be instantly aligned at a point of interest. Changes can then be found through rapid horizontal scrolling, which effectively creates an animation of the evolution of the file over time in which changes can be spotted through the ability of the human visual system to detect small alterations in a mostly static image.

6.2 Limitations

Some of the limitations of SCHN are as follows:

- **It can be difficult to see the correspondence between individual difference blocks in the neighbour differences view.** Since all the difference blocks for a given pair of neighbouring revision stripes are assigned the same colour on mouseover, it can be difficult to see which correspond to which. It can also be difficult to see where deletions occurred.

- **The scale of the revision stripes is not always appropriate.** Revision stripes are currently constructed according to a simple "one code line = one pixel line" rule. For very long files, this may result in stripes that are too tall to display in their entirety with the available screen real estate. For short files, the revision stripes may only occupy a small fraction of the available screen real estate.

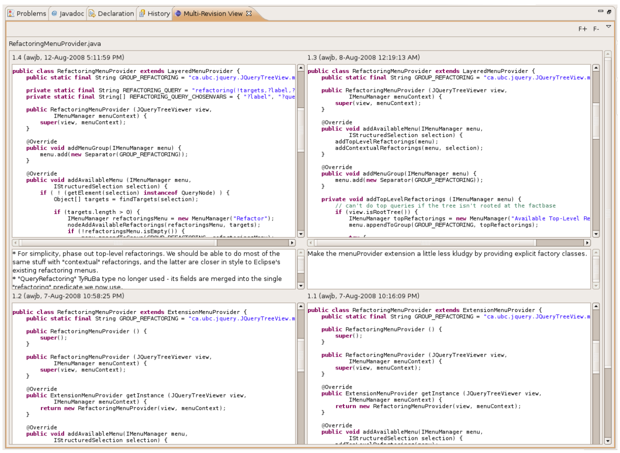
- **Clear legends are not provided for the colours used.** Users must currently rely on their intuition to determine the meaning of the colours used in each colouring. However, it is possible users might misunderstand the intended meaning of the colours. For the authorship view, users must currently work to determine which colour corresponds to which author by tracing coloured lines back to the revision where they start. A legend describing the precise meaning of colours used would facilitate user comprehension of our visualizations.

- **Performance is slow at certain key points.** There can be a significant startup delay while SCHN downloads large numbers of revisions from CVS, and no progress indicator is provided to the user during this time. Certain graphical rendering tasks (e.g., resizing, expanding and collapsing viewers) have a noticeable lag, which would probably be irritating for users.

- **Screen real estate tightly limits the number of revision text viewers that can be displayed simultaneously.** This is an inherent limitation of our approach. Using the normal font size, it is not feasible to show more than three revision text viewers side by side simultaneously on a 1280×1024 display. Using a reduced font size, it may be possible to squeeze in four viewers. We work around this limitation by permitting horizontal scrolling, which is quite effective in permitting at least some types of comparisons between many more revision text viewers than can fit on the screen (see section 6.1.)

6.3 Lessons Learned

One lesson learned during the development of our project was that one of the views we originally proposed did not work well. In our original proposal, we considered displaying revision text viewers in a $2 \times n$ grid (see Figure 8). We implemented this view, but found that its benefit compared to horizontally scrollable side-by-side viewers was outweighed by its disadvantages. The grid view does provide a way to compare four, rather than three, revisions at normal font size simultaneously, albeit with smaller vertical viewer size. However, we found our comparison tasks often necessitated comparing more than three or four revisions quickly, and in contrast to the ease of comparison provided by quickly scrolling through side-by-side views with a linear visual scan (see section 6.1), the grid view required a less comfortable and efficient zigzagging scan pattern. Furthermore, the grid view did not fit cleanly into our dominant UI metaphor of a horizontal flow of revisions in which revision text viewers could be interspersed in a focus+context style. We therefore decided to omit it from the final implementation of our tool.



RefactoringMenuProvider.java

1.4 (awjb, 12-Aug-2008 5:11:59 PM)

```
public class RefactoringMenuProvider extends LayeredMenuProvider {
    public static final String GROUP_REFACTORING = "ca.ubc.jquery.JQueryTreeView.m

    private static final String REFACTORING_QUERY = "refactoring(!targets,?label,?
    private static final String[] REFACTORING_QUERY_CHOSENVARS = { "?label", "?que

    public RefactoringMenuProvider (JQueryTreeViewer view,
            IMenuManager menuContext) {
        super(view, menuContext);
    }

    @Override
    public void addMenuGroup(IMenuManager menu) {
        menu.add(new Separator(GROUP_REFACTORING));
    }

    @Override
    public void addAvailableMenu (IMenuManager menu,
            IStructuredSelection selection) {
        if ( ! (getElement(selection) instanceof QueryNode) ) {
            Object[] targets = findTargets(selection);

            if (targets.length > 0) {
                IMenuManager refactoringsMenu = new MenuManager("Refactor");
                nodeAddAvailableRefactorings(refactoringsMenu, targets);
                if (!refactoringsMenu.isEmpty()) {
```

* For simplicity, phase out top-level refactorings. We should be able to do most of the same stuff with "contextual" refactorings, and the latter are closer in style to Eclipse's existing refactoring menus.
* "QueryRefactoring" TyRuBa type no longer used - its fields are merged into the single "refactoring" predicate we now use.

1.3 (awjb, 8-Aug-2008 12:19:13 AM)

```
public class RefactoringMenuProvider extends LayeredMenuProvider {
    public static final String GROUP_REFACTORING = "ca.ubc.jquery.JQueryTreeView.m

    public RefactoringMenuProvider (JQueryTreeViewer view,
            IMenuManager menuContext) {
        super(view, menuContext);
    }

    @Override
    public void addAvailableMenu(IMenuManager menu,
            IStructuredSelection selection) {
        addTopLevelRefactorings(menu);
        addContextualRefactorings(menu, selection);
    }

    @Override
    public void addMenuGroup(IMenuManager menu) {
        menu.add(new Separator(GROUP_REFACTORING));
    }

    private void addTopLevelRefactorings (IMenuManager menu) {
        // can't do top queries if the tree isn't rooted at the factbase
        if (view.isRootTree()) {
            IMenuManager topRefactorings = new MenuManager("Available Top-Level Re
            menu.appendToGroup(GROUP_REFACTORING, topRefactorings);
```

Make the menuProvider extension a little less kludgy by providing explicit factory classes.

1.2 (awjb, 7-Aug-2008 10:58:25 PM)

```
public class RefactoringMenuProvider extends ExtensionMenuProvider {
    public static final String GROUP_REFACTORING = "ca.ubc.jquery.JQueryTreeView.m

    public RefactoringMenuProvider () {
        super();
    }

    public RefactoringMenuProvider (JQueryTreeViewer view,
            IMenuManager menuContext) {
        super(view, menuContext);
    }

    @Override
    public ExtensionMenuProvider getInstance (JQueryTreeViewer view,
            IMenuManager menuContext) {
        return new RefactoringMenuProvider(view, menuContext);
    }

    @Override
    public void addAvailableMenu(IMenuManager menu,
            IStructuredSelection selection) {
```

1.1 (awjb, 7-Aug-2008 10:16:09 PM)

```
public class RefactoringMenuProvider extends ExtensionMenuProvider {
    public static final String GROUP_REFACTORING = "ca.ubc.jquery.JQueryTreeView.m

    public RefactoringMenuProvider () {
        super();
    }

    public RefactoringMenuProvider (JQueryTreeViewer view,
            IMenuManager menuContext) {
        super(view, menuContext);
    }

    @Override
    public ExtensionMenuProvider getInstance (JQueryTreeViewer view,
            IMenuManager menuContext) {
        return new RefactoringMenuProvider(view, menuContext);
    }

    @Override
    public void addAvailableMenu(IMenuManager menu,
            IStructuredSelection selection) {
```

Figure 8: $2 \times n$ “grid view” concept, which we ultimately decided not to use in our final implementation.

Another lesson learned is that colour selection can be tricky. When designing our differences-from-neighbours colouring, we had a strong intuition that corresponding difference blocks in adjacent revision stripes should be the same colour. We initially produced a design (see Figure 9) in which *all* revision stripes were coloured, using a palette of three colours for one-way differences and orange for two-way differences. (We picked three colours because three was the minimum number of colours we could employ without creating confusion by having the same colour indicate differences in different directions for the same revision stripe.) However, feedback received for this design indicated that this colouring scheme created a strong likelihood that the user would see spurious patterns in the stripes, intuitively perceiving nonexistent connections between distant revisions because of the strong visual cues provided by their identical colours.

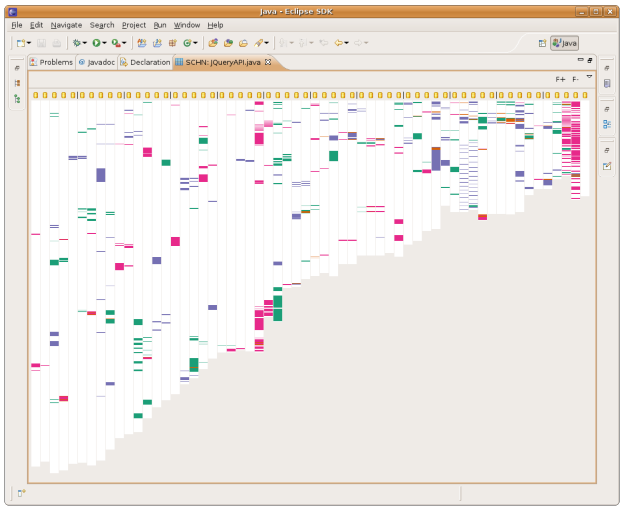


Figure 9: An earlier version of our differences-from-neighbours colouring, which we rejected because of spurious colour correspondences between distant revisions.

We therefore adopted the strategy we currently use, in which all but the moused-over revision and its neighbours have their differences greyed out. In our first attempt at this strategy, we used the standard SWT “dark grey” colour. We discovered, to our chagrin, that this colour was indistinguishable from pink under the Vischeck [4] deuteranope simulation. We therefore decreased the saturation of our grey colour. In our current colouring scheme, the green or blue colours still reduce to dark grey in deuteranope or tritanope colour-blindness simulations (respectively), but are distinguishable from light grey by saturation.

7 Future Work

SCHN provides many opportunities for future exploration. We consider some directions for future work in the areas of better support for the domains of software engineering and version control; better visualization techniques; efficiency improvements; and validation.

7.1 Domain Support

We could make our tool more generally useful by providing syntax highlighting support for a wider variety of programming languages and file types. We could also support version control systems other than CVS, such as Subversion or Perforce, and take into account advanced version control features such as branching.

7.2 Visualization Techniques

We could improve the visualization techniques used in our tool to address some of the limitations outlined in section 6.2:

- To address the problem of identifying corresponding difference blocks, we could implement a feature which allowed users to highlight corresponding neighbour differences for a particular block when they moused over it.

- Scaling techniques for revision stripes could be explored. Scaling small stripes up would be relatively straightforward; however, scaling large stripes down would require merging the colours of some pixel lines. The position-based antialiasing algorithm of Voinea et al. [19] could be considered for this purpose.

- Legends would be fairly simple to provide, possibly through a small popup window which the user could enable if needed.

Our textual file comparison views could be improved by providing character-by-character comparison of corresponding lines, to match the functionality provided by most existing text comparison GUIs. We could also explore different colouring strategies for our revision stripes, such as indicating the bug-fix/feature enhancement requests or task contexts (in the Mylar/Mylyn [10] sense) associated with a line of code. We could allow other developers to provide colouring strategies of their own through an Eclipse extension point.

7.3 Efficiency

The efficiency of SCHN could be improved significantly by caching retrieved CVS revisions so that future SCHN analyses of the same file did not require further unnecessary CVS requests. This improvement would also make SCHN a friendlier player in real-world development environments by reducing the load it places on CVS servers.

Techniques for optimizing the rendering of our UI could also be explored to reduce the lag in performing certain operations.

7.4 Validation

An important direction for future work would be releasing SCHN for real-world use, collecting user feedback, and conducting user studies. These steps would give us insight into any usability flaws in our tool and other features desired by users. We can also test our tool on real-world codebases other than JQuery to observe its usability and performance.

8 Conclusion

We have presented the Source Code History Navigator (SCHN), a tool for navigating and querying the revision history of a source code file. SCHN provides a high-level overview of all code revisions using a history flow of revision stripes, which are coloured according to various schemes, including differences from neighbour revisions, line authorship, line type, and code age. Revision stripes can be expanded to show viewers for the text of a revision, providing a focus+context interaction style. Revision text viewer scrollbar locking and horizontal scrolling through revision viewers provide a powerful technique for many-revision text comparisons. We demonstrate through a usage scenario that SCHN has the potential to permit exploration of code history in ways not well supported by current tools.

Acknowledgements

Early comments and a CVSscan demonstration by Rolf Biehn contributed to the development of this work. Comments and feedback from Tamara Munzner were highly influential in shaping the direction of this project. This work was partly supported by an NSERC CGS M.

References

[1] *Eclipse Platform Plug-in Developer Guide: Class RangeDifferencer*, 2006. http://help.eclipse.org/help32/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/compare/rangedifferencer/RangeDifferencer.html.

[2] Araxis Ltd. Araxis Merge, 2009. http://www.araxis.com/merge-overview.html.

[3] T. Ball and S. G. Eick. Software visualization in the large. *Computer*, 29(4):33–43, 1996.

[4] B. Dougherty and A. Wade. Vischeck, 2009. http://www.vischeck.com/vischeck/.

[5] J. Eibl. KDiff3, March 2009. http://kdiff3.sourceforge.net/.

[6] J. Froehlich and P. Dourish. Unifying artifacts and activities in a visual tool for distributed software development teams. In *ICSE*, pages 387–396, 2004.

[7] M. Harrower and C. A. Brewer. ColorBrewer.org: an online tool for selecting colour schemes for maps. *The Cartographic Journal*, 40(1):27–37, June 2003.

[8] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 178–187, New York, NY, USA, 2003. ACM.

[9] J. A. Jones, M. J. Harrold, and J. T. Stasko. Visualization of test information to assist fault localization. In *ICSE*, pages 467–477, 2002.

[10] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for IDEs. In *AOSD*, pages 159–168, 2005.

[11] G. Lommerse, F. Nossin, L. Voinea, and A. Telea. The visual code navigator: An interactive toolset for source code investigation. In *INFOVIS '05: Proceedings of the 2005 IEEE Symposium on Information Visualization*, page 4, Washington, DC, USA, 2005. IEEE Computer Society.

[12] W. Miller and E. W. Myers. A file comparison program. *Softw., Pract. Exper.*, 15(11):1025–1040, 1985.

[13] Perforce Software. Perforce visual client, 2009. http://www.perforce.com/perforce/products/p4v.html.

[14] R. Rao and S. K. Card. The table lens: merging graphical and symbolic representations in an interactive focus + context visualization for tabular information. In *CHI '94: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 318–322, New York, NY, USA, 1994. ACM.

[15] Scooter Software, Inc. Beyond Compare, 2009. http://www.scootersoftware.com/.

[16] E. R. Tufte. *Envisioning Information*. Graphics Press, Cheshire, CT, USA, 1990.

[17] F. B. Viégas, M. Wattenberg, and K. Dave. Studying cooperation and conflict between authors with history flow visualizations. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 575–582, New York, NY, USA, 2004. ACM.

[18] L. Voinea and A. Telea. Visual querying and analysis of large software repositories. *Empirical Software Engineering*, 14(3):316–340, 2009.

[19] L. Voinea, A. Telea, and J. J. van Wijk. CVSscan: visualization of code evolution. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 47–56, New York, NY, USA, 2005. ACM.

[20] S. L. Voinea. *Software Evolution Visualization*. PhD thesis, Technische Universiteit Eindhoven, 2007.