uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

Marconi Lanna

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.C.S.

GRADE / DEGREE

School of Information, Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Spotting the Difference – A Source Code Comparison Tool

TITRE DE LA THÈSE / TITLE OF THESIS

D. Amyot

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

A. Boukerche

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

D. Deugo

T. Lethbridge

Gary W. Slater

Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

Spotting the Difference

A Source Code Comparison Tool

by

Marconi Lanna

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements for the degree of

Master of Computer Science

under the auspices of the Ottawa-Carleton Institute for Computer Science

School of Information Technology and Engineering
Faculty of Engineering
University of Ottawa

Your file Votre référence
ISBN: 978-0-494-59456-8
Our file Notre référence
ISBN: 978-0-494-59456-8

Às mulheres da minha vida: Eliane, Marina, e Daniela.

E você também, Celo.

To the very first person who, with a bunch of rocks, invented computing.

And to that old, dusty 80286 and its monochromatic screen.

Abstract

Source Code Management (SCM) is a valuable tool in most software development projects, whichever their size. SCM provides the ability to store, retrieve, and restore previous versions of files. *File comparison tools* complement SCM systems by offering the capability to compare files and versions, highlighting their differences.

Most file comparison tools are built around a two-pane interface, with files displayed side by side. Such interfaces may be inefficient in their use of screen space — wasting horizontal real estate — and ineffective, for duplicating text makes it difficult to read, while placing most of the comparison burden on the user.

In this work, we introduce an innovative metaphor for file comparison interfaces. Based on a single-pane interface, common text is displayed only once, with differences intelligently merged into a single text stream, making reading and comparing more natural and intuitive.

To further improve usability, additional features were developed: difference classification — *additions*, *deletions*, and *modifications* — using finer levels of granularity than is usually found in typical tools; a set of special artifacts to compare *modifications*; and intelligent white space handling.

A formal usability study conducted among sixteen participants using real-world code samples demonstrated the interface adequacy. Participants were, on average, 60% faster performing source code comparison tasks, while answer quality improved, on our weighted scale, by almost 80%. According to preference questionnaires, the proposed tool conquered unanimous participant preference.

Acknowledgments

This thesis would never be possible without the help of my family, friends, and colleagues.

First and foremost, I would like to thank my wife for her infinite support and not so infinite patience — *Eu nunca teria feito nada disto sem você*, — my mom, who always gave me encouragement and motivation, my brother Marcelo, my little sister Marina, and my beloved in-laws, Artur, Joeli, and Vanessa.

I am immensely grateful to my supervisor, Professor Daniel Amyot. I never worked with someone for so long without hearing or having a single complain. Many, many thanks.

Many friends contributed helpful feedback and advice. Professor Timothy Lethbridge helped us with many usability questions. Alejandro, Gunter, Jason, Jean-Philippe, and Patrícia were kind enough to experiment with early versions of the tool. Professor Azzedine Boukerche offered me assistance during my first year.

Finally, I want to express my gratitude to my examiners, Professors Tim Lethbridge and Dwight Deugo, and all volunteers who agreed to participate on the usability study.

Thank you all.

Marconi Lanna

Ottawa, Ontario, July 2009

Table of Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

Acronym	Definition
GUI	Graphical User Interface
IDE	Integrated Development Environment
LCD	Liquid Crystal Display
LCS	Longest Common Subsequence
SCM	Source Code Management
UML	Unified Modeling Language

Chapter 1

Introduction

Code is read much more often than code is written[1]. Rarely, though, is code read for amusement or poetry. Code is read to be understood, and usually code needs to be understood when code has to be maintained.

Software maintenance leads to code changes, modifications which themselves have to be read, understood, and reviewed. Communicating those changes among a team can be particularly difficult for projects in which developers may be working on the same files concurrently.

While *Source Code Management* (SCM) is widely employed to control and trace modifications, allowing developers to store and retrieve arbitrary sets of changes from a repository, little attention has been given in recent years to *File Comparison Tools*, a companion piece of software used to inspect differences between files.

This work presents a specialized source code comparison tool based on a set of metaphors and features aimed at improving ease of use, intuitiveness, and efficiency. A comprehensive usability study conducted among sixteen participants using real-world code samples has demonstrated the feasibility and adequacy of the proposed interface.

1.1 Motivation

Software projects are living beings. Requirement changes, bug fixes, compliance to new standards or laws, updated systems and platforms are some of the reasons software constantly requires updating [34]. The more complex a software project, the more likely frequent changes are to occur and the larger the maintenance team is supposed to be.

[1] This citation can be attributed to multiple authors.

Developers working on the same set of files need to be concerned with duplicated or conflicting changes. Redefined semantics, classes, or members may compel a developer to update code she is maintaining, even when working on distinct files, to conform with changes made by others. On largely distributed projects, such as in most open source software projects, *patches* submitted by third parties need to be reviewed before being committed to an SCM repository. Proper mechanisms for communicating changes among software developers are essential, as is the ability to glance at new versions of code and quickly spot differences.

Take, for instance, the testimony given by two senior executives of a leading SCM software vendor justifying why their legacy code is not updated to comply with the company's own code conventions:

> "*While we like pretty code, we like clean merges even better. Changes to variable names, whitespace, line breaks, and so forth can be more of an obstacle to merging than logic changes.*" [55]

Effective file comparison tools help mitigate this kind of problem, giving software developers a better understanding of source code changes.

One of the first widely used file comparison tool was developed in 1974 by Douglas McIlroy for the Unix operating system. `diff`, a command-line tool, "*reports differences between two files, expressed as a minimal list of line changes*" [27]. The standard `diff` output (Figure 1.1), thus, does not show lines common to both files necessary to understand changes in context. Differences are computed and displayed line-by-line, being hard to identify particular changes within lines.

Most contemporary file comparison tools, though, have *Graphical User Interfaces* (GUI) and a set of advanced features such as synchronized display of files side by side, underlining of individual changes within a line, syntax highlighting, and integration with Source Code Management systems and *Integrated Development Environments* (IDE) tools (Figure 1.2).

Despite the improvements developed in the last decades, file comparison tools still are cumbersome to use, yielding sub-optimal results. Amongst the most common problems, we may cite:

- Displaying both versions at the same time, side by side, represents a waste of screen real estate and may lead to horizontal scrolling, even on large wide-screen displays. Differently

```
$ diff 1.old.java 1.new.java
4c4,6
< public abstract class NaivePrime{
---
> public class NaivePrime{
>
>       private NaivePrime(){}
9c11
<       public static boolean isPrime(int n){
---
>       public static boolean isPrime(long n){
15c17,25
<               for (int i = 2; i < n; i++){
---
>               if (n == 2)
>                       return true;
>
>               if (n % 2 == 0)
>                       return false;
>
>               long sqrt = (long)Math.sqrt(n);
>
>               for (long i = 3; i <= sqrt; i += 2){
28,29d37
<                       String message = " is composite.";
<
31,33c39
<                               message = " is prime.";
<
<                       System.out.println(i + message);
---
>                               System.out.println(i);
```

Figure 1.1: **Sample `diff` Output**

from vertical scrolling, horizontal scrolling is very inefficient and unpleasant and, when possible, should be avoided [42].

- Reading does not follow a single flow of text. Pieces of text may appear on one side of the screen, the other, duplicated on both sides, or differently on both. A user has to keep track of two *reading points* at the same time.

- With changes split throughout the sides of the screen, it is difficult to make direct comparisons since one's eyes have to scroll back and forth across the interface, constantly losing focus.

The system proposed in this thesis attempts to address those shortcomings, offering a more intuitive, ease to learn, and effective user interface model.

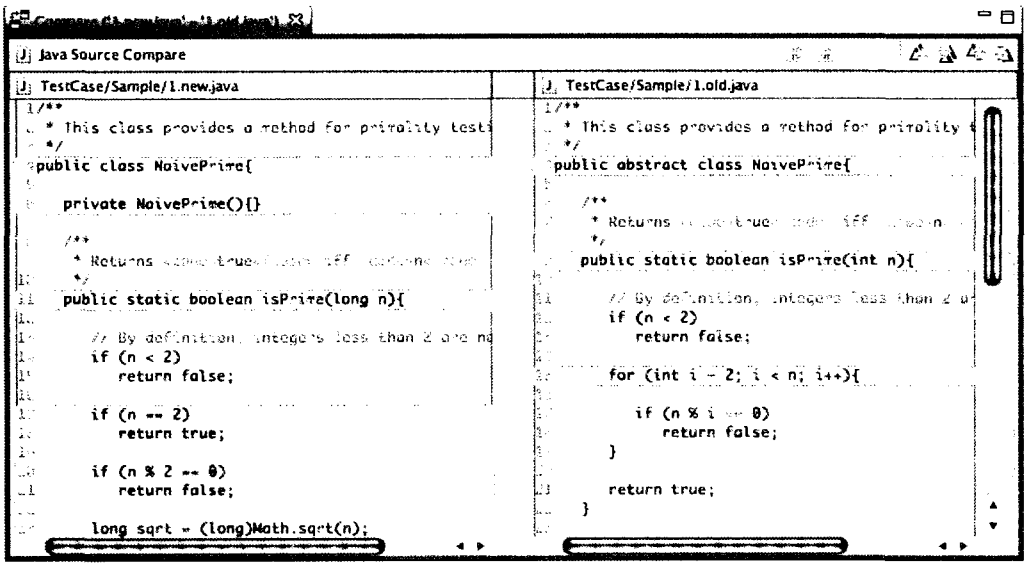

Figure 1.2: **Eclipse Compare Editor:** A typical file comparison tool.

1.2 Research Hypothesis and Proposed Interface

In this thesis we postulate that the two-pane interface is an inefficient and ineffective metaphor to represent file differences (Section 3.1). Furthermore, a file comparison user interface model which offers improved ease of use and efficiency is proposed and validated. The proposed interface (Figure 1.3) is based on the following principles:

Single-pane Interface: Differences between files should be consolidated and displayed in a single pane, facilitating reading and comprehension. By not displaying two pieces of text side by side, screen real estate usage is maximized, reducing eye movement across the screen and virtually eliminating horizontal scrolling.

Difference Classification: Individual differences should not only be highlighted but also classified into *additions*, *deletions*, and *modifications*, providing a natural and intuitive metaphor to interpret changes.

Special Interface Artifacts: Displaying *modifications* presents an interesting challenge: two pieces of text, the original and the modification, have to be shown to represent a single change, which is in evident contrast to the single text stream view. Special interface elements have to be employed to overcome this problem without breaking the first principle.

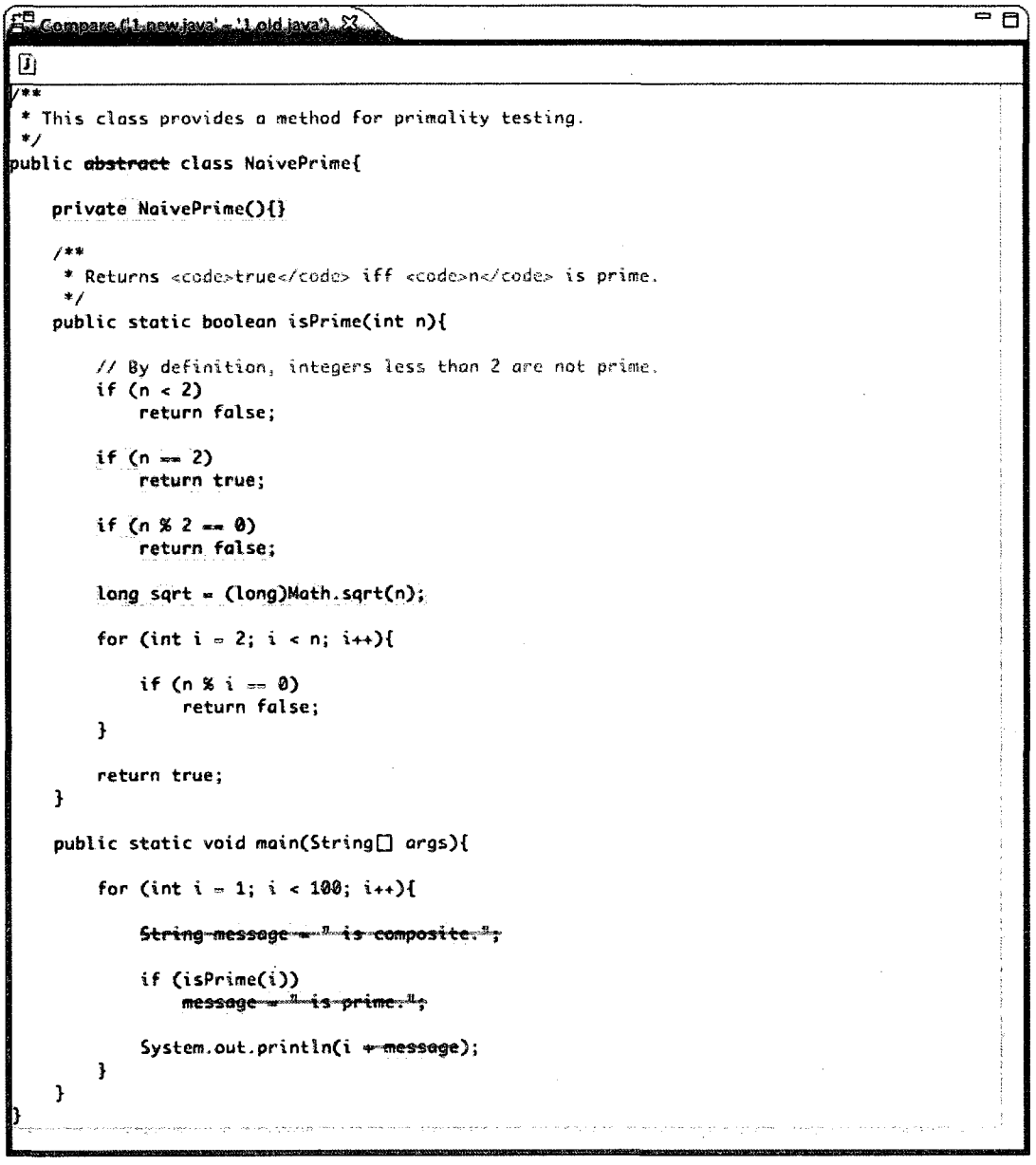

Figure 1.3: **Proposed Tool:** A sample comparison displayed using the proposed tool.

Finer Granularity: Multiple changes in a single line can be difficult to understand. Complexity can be reduced by breaking large differences into smaller, individual pieces.

1.3 Thesis Contributions

To validate the principles discussed in section 1.2, a fully functional, working prototype was implemented.

The most distinctive characteristic of the proposed tool is the use of a single-pane interface to display differences in accordance with the single text view principle. Differences are computed and displayed using *token* granularity. A single line of text may contain different changes, and of different types. Additions, deletions, and modifications are highlighted using different colors. Two complementary artifacts, *tooltips* and *hot keys* (not shown), were developed for displaying modifications without duplicating text on the interface.

Tooltips allow the user to quickly glance at a particular modification by putting the mouse pointer over it; a pop-up window then displays the original text. On the other hand, hot keys, when pressed, switch between both versions of the text in place. In any case, the original text is always displayed near the modified text, in evident contrast to the traditional interfaces where both pieces of text are on different sides of the screen, far from each other.

A formal usability study conducted among sixteen participants using real-world code samples confirmed the effectiveness of the proposed interface, showing average speed improvements of 60% while also increasing answer quality on our weighted scale by almost 80%.

1.4 Background Information

File comparison tools are pieces of software used to compute and display differences between files. Although general enough to compare arbitrary pieces of text, those tools are mostly used in association with Source Code Management systems to review source code changes and resolve eventual conflicts.

Commonly, comparisons are performed against two files, traditionally called the *left* and *right* sides. There is no implicit or explicit precedence relation between the files. In this work, by convention, the left side is considered to be the modified version and the right side, the original one.

Comparisons can also involve three files, usually to resolve conflicts caused by concurrent development. In those cases, the third file is called the *ancestor* and is, by definition, the source from which the other two were derived.

1.4.1 The Longest Common Subsequence

To determine the differences — or, ideally, the *minimal* set of differences — between files, file comparison tools usually compute the *Longest Common Subsequence* (LCS) [1].

A *sequence* $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is said to be a *subsequence* of $X = \langle x_1, x_2, \ldots, x_m \rangle$ if there exists a *strictly increasing* sequence $I = \langle i_1, i_2, \ldots, i_k \rangle$ of indexes of X such that for all $j = 1, 2, \ldots, k$, we have $x_{i_j} = z_j$. Z is said to be a *common subsequence* of X and Y if Z is a subsequence of both X and Y. The *longest-common-subsequence* problem can be stated as follows: given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, find the maximum-length common subsequence of X and Y [9].

Please note that the LCS is *not* unique. Given sequences $X = \langle 1, 2, 3 \rangle$ and $Y = \langle 2, 1, 3 \rangle$, both $Z = \langle 1, 3 \rangle$ and $W = \langle 2, 3 \rangle$ are longest common subsequences of X and Y. The LCS, *per se*, does not compute the minimal set of differences[2]; those are presumed to be all elements not in the LCS.

1.4.2 Files and Differences

To compute the longest common subsequence against source code files, the sequences can be formed from the file *lines*, *words* or *tokens*, or even individual *characters* (Section 4.3.1). Traditionally, most comparisons tools compare files line by line (Chapter 2). For brevity, we refer to *nodes* in this text.

It is convenient to use a compact notation to represent files and differences. File content is represented as a sequence of lower case letters — each letter representing a node — displayed horizontally, as in `abc`. New nodes are represented with a previously unused letter, as in `abcd`. Removed nodes are simply omitted: `ab`. Some nodes are neither removed nor inserted, but have their content altered; those are represented by upper case letters: `aBc`.

We will frequently refer to differences in more specific terms, respectively *additions*, *deletions*, and *modifications* (Section 3.3.2). Intuitively, for a file `abc` modified into `aCd`, `b` is a *deletion*, the pair (`c`, `C`) is a *modification*, and `d` is an *addition*. Collectively, additions, deletions, and modifications may be called *changes*, to distinguish them from plain *differences*.

1.4.3 Alternatives to the LCS

Although algorithms to compute the LCS, or some variation form, have been vastly employed by most file comparison tools, some existing alternatives try to improve on the traditional line matching algorithms by introducing features such as detecting moved lines, telling if lines were

[2]Since the LCS is not unique, it would be more appropriate to refer to "*an* LCS", and "*a* minimal set of differences."

modified or replaced by different lines, or using a programming language's syntactical structure to compute the differences.

A complete discussion of difference algorithms is beyond the scope of this work. For samples of recent work on this area, please refer to [5, 32].

1.5 Related Work

Academic research and specific literature in the field of file comparison interfaces is, for the most part, scarce. However, code comparisons are not limited to source text. Graphical models, such as UML diagrams, and visual maps can also be used to represent changes to a code base.

Atkins [3] discusses *ve*, or Version Editor, a source code editing tool integrated with version control systems. The tool interface, which can emulate both the *vi* and *emacs* editors, highlights additions and deletions using, respectively, bold and underlines. The tool is capable of showing, for each line, SCM metadata information such as author, rationale, and date of modification. The author estimates that productivity gains due the tool represented savings of $270 million over ten years.

Voinea *et al.* [52] introduces *CVSscan*, a code evolution visualization tool that arranges code changes into a temporal map. Versions of a source file are represented in vertical columns, with the horizontal dimension used to represent time. Lines of code are represented as single pixels on the screen, using colors to mean *unmodified* (green), *modified* (yellow), *deleted* (red), and *inserted* (blue). Actual source text comparisons can be made by "*sweeping*" the mouse across the interface. Differences are displayed using a "*two-layered code view*" which closely resembles a two-pane comparison interface (Section 2.1).

Seeman *et al.* [50] and Ohst *et al.* [47] describe tools that use UML diagrams to represent changes in object-oriented software systems as graphical models. Both tools are limited to comparing classes and members, providing no means to visualize changes to the code text.

Chawathe *et al.* [7] presents *htmldiff* [26], a tool to capture and display Web page updates. Changes are represented by bullets of different colors and shapes meaning *insertion*, *deletion*, *update*, *move*, and *move+update*.

On the topic of file comparison tools, Mens [36] provides an overview of merge techniques, categorizing them into orthogonal dimensions: two- and three-way merging (Section 2.1); textual, syntactic, semantic, or structural (Section 7.3); state- and change-based; reuse and evo-

lution. The author also discusses techniques for conflict detection and resolution, difference algorithms, and granularity (Section 3.3.4).

1.6 Sample Test Case

To introduce participants to file comparison tools in the usability experiment (Chapter 5), a sample, handwritten test case was created (Figures 1.4 and 1.5). The sample test case was used to explain how comparisons were to be performed, presenting to the participants a sensible set of additions, deletions, and modifications. No measurements were done using the sample test case.

Figure 1.3 on page 5 shows this comparison as represented by the proposed interface. In the next chapter, *Comparison Tools Survey*, all screenshots were taken using the sample test case.

1.7 Thesis Outline

This thesis is organized into seven chapters — of which this was the first — plus ten appendices:

Chapter 2, *Comparison Tools Survey*, covers the features offered by some popular file comparison tools;

Chapter 3, *Spotting the Difference*, discusses some of the deficiencies perceived with current file comparison offerings while proposing improvements;

Chapter 4, *Architecture and Implementation*, briefly reviews the prototype development;

Chapter 5, *Usability Evaluation*, details the usability experiment and analyzes its main results;

Chapter 6, *Lessons from the Usability Study*, examines the main insights acquired from the usability experiment;

Chapter 7, *Conclusion*, summarizes thesis contributions and discusses future work;

```
 1 /**
 2  * This class provides a method for primality testing.
 3  */
 4 public abstract class NaivePrime{
 5
 6    /**
 7     * Returns <code>true</code> iff <code>n</code> is prime.
 8     */
 9    public static boolean isPrime(int n){
10
11       // By definition, integers less than 2 are not prime.
12       if (n < 2)
13          return false;
14
15       for (int i = 2; i < n; i++){
16
17          if (n % i == 0)
18             return false;
19       }
20
21       return true;
22    }
23
24    public static void main(String[] args){
25
26       for (int i = 1; i < 100; i++){
27
28          String message = " is composite.";
29
30          if (isPrime(i))
31             message = " is prime.";
32
33          System.out.println(i + message);
34       }
35    }
36 }
```

Figure 1.4: **Test Case, Original**

```
 1 /**
 2  * This class provides a method for primality testing.
 3  */
 4 public class NaivePrime{
 5
 6    private NaivePrime(){}
 7
 8    /**
 9     * Returns <code>true</code> iff <code>n</code> is prime.
10     */
11    public static boolean isPrime(long n){
12
13       // By definition, integers less than 2 are not prime.
14       if (n < 2)
15          return false;
16
17       if (n == 2)
18          return true;
19
20       if (n % 2 == 0)
21          return false;
22
23       long sqrt = (long)Math.sqrt(n);
24
25       for (long i = 3; i <= sqrt; i += 2){
26
27          if (n % i == 0)
28             return false;
29       }
30
31       return true;
32    }
33
34    public static void main(String[] args){
35
36       for (int i = 1; i < 100; i++){
37
38          if (isPrime(i))
39             System.out.println(i);
40       }
41    }
42 }
```

Figure 1.5: **Test Case, Modified**

Appendix A, *Test Cases*, reproduces the source code files used in the usability experiment;

Appendix B, *List of Differences*, enumerates all differences participants were expected to report in the usability experiment;

Appendix C, *Experimental Data*, lists, in tables, the raw data collected during the experiment, including participants answers;

Appendix D, *Statistical Information*, provides basic statistical information about the data gathered in the experiment;

Appendix E, *Outlier Data*, reproduces the main time charts including outlier data;

Appendix F, *Experiment Script*, is a transcription of the protocol followed during the experiment;

Appendices G through J provide transcriptions of all forms and questionnaires used in the experiment.

Chapter 2

Comparison Tools Survey

File comparison tools are popular tools available for a variety of systems and platforms and are used by both developers and non-developers. They cover a broad range of functionalities, from general text comparison to specialized code editing.

In this chapter, we examine the main features offered by a representative selection of file comparison tools. Firstly, we discuss features expected to be offered by modern file comparison tools.

2.1 File Comparison Features

The following features were observed when evaluating the selected comparison tools:

Interface Metaphor: How the tool displays the files for comparison on the screen. Most tools use a *two-pane* interface with files displayed side by side, although some widely used tools are still based on *textual* interfaces.

Vertical Alignment: Tools that display files side by side should, preferably, keep both sides vertically aligned. While most tools employ sophisticated *synchronized scrolling* mechanisms, some would simply pad the text with *blank lines.*

Highlighting Granularity: The granularity with which differences are highlighted. Common options include *whole lines*, *words* or *tokens*, and *individual characters.* For tools that provide the option, the finest level of granularity was considered.

Difference Navigation: Whether the tool provides a mechanism to navigate between differences. The most common options are *previous* and *next* buttons, or *direct access*, usually

represented by a thumbnail view of the differences.

Syntax Highlighting: Indicates whether the tool supports some level of syntax highlighting, preferably for the Java programming language.

Ignore White Space/Case: Indicates whether the tool ignores differences in white space and case during comparisons. Usually, a user-selectable option.

Merge Support: Indicates whether the tool allows differences to be copied, or *merged*, from one file to the other.

Three-way Comparisons: Indicates whether the tool supports comparing a pair of files simultaneously with a common ancestor.

2.2 Comparison Tools

Nine file comparison tools were selected for this survey. The sample was chosen amongst popular IDEs and stand-alone tools, open-source and proprietary, covering the most significant development platforms: Java, Apple Mac OS X, Unix, and Microsoft Windows.

While it is by no means an exhaustive list, we believe this to be a very representative set of the features commonly found on most file comparison tools.

```
$ diff 1.old.java 1.new.java
4c4,6
< public abstract class NaivePrime{
---
> public class NaivePrime{
>
>       private NaivePrime(){}
9c11
<       public static boolean isPrime(int n){
---
>       public static boolean isPrime(long n){
15c17,25
<               for (int i = 2; i < n; i++){
---
>               if (n == 2)
>                       return true;
>
>               if (n % 2 == 0)
>                       return false;
>
>               long sqrt = (long)Math.sqrt(n);
>
>               for (long i = 3; i <= sqrt; i += 2){
28,29d37
<                       String message = " is composite.";
<
31,33c39
<                               message = " is prime.";
<
<                       System.out.println(i + message);
---
>                               System.out.println(i);
```

Figure 2.1: **GNU diffutils**

2.2.1 diff

diff - compare files line by line

GNU diffutils *man page*

diff is one of the first file comparison tools. It was originally developed by Douglas McIlroy for the Unix operating system in the early 1970s [27]. **diff** is an implementation of the Longest Common Subsequence algorithm which takes two text files as input and compares them line by line.

By default, **diff**'s output (Figure 2.1) represents the set of lines which do not belong to the LCS. Lines are marked as "*from FILE1*" or "*from FILE2*" [19], which can be interpreted as additions and deletions.

Although it might not be directly comparable to more advanced graphical tools, **diff** is still widely used and was included for historical reasons. For this survey, the GNU diffutils implementation [23] was used.

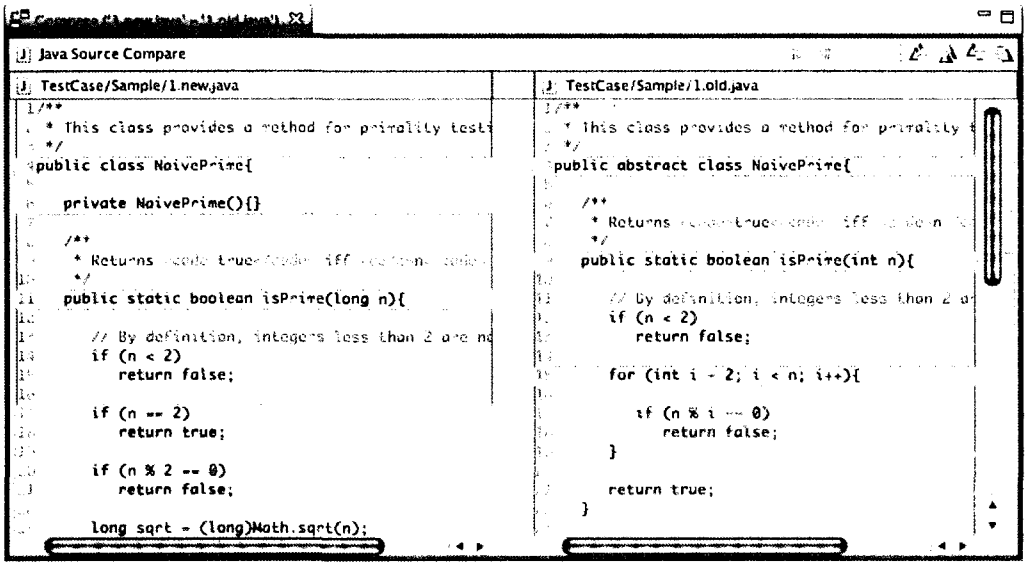

Figure 2.2: **Eclipse**

2.2.2 Eclipse

Eclipse is a project and a development platform mostly known for its aptly named Eclipse IDE, very popular amongst Java developers [18].

While reviewing the IDE and all its features is outside the scope of this survey, Eclipse's *Compare Editor* [12] is a modern, advanced graphical file comparison tool[1], providing a two-pane interface with support for merging, three-way comparisons, and syntax highlighting for multiple programming languages (Figure 2.2).

Unique amongst comparison tools is its *Structure Compare* feature, which outlines differences using a tree of high level elements, such as classes, constructors, and methods. Although most tools support file merging, Eclipse is one of the few tools to allow text to be edited directly in the comparison, offering even advanced editing features such as code completion and access to class documentation[2].

[1]Strictly speaking, the Eclipse platform provides a comparison framework on top of which comparison tools are implemented. The distinction between platform, framework and tools will not be made.

[2]Version 3.5, *Galileo*

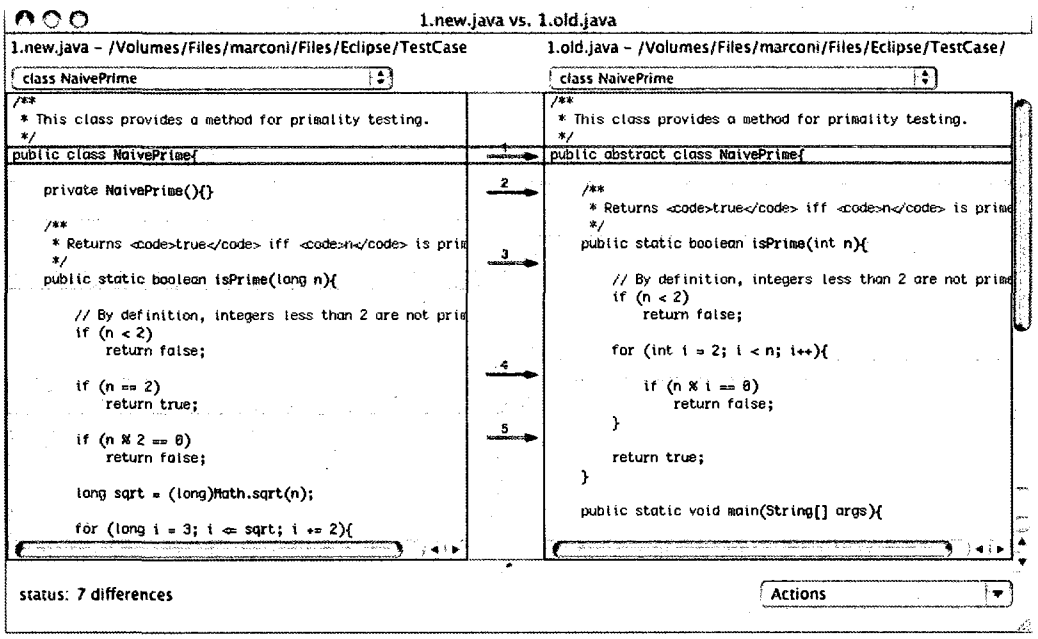

Figure 2.3: **FileMerge**

2.2.3 FileMerge

FileMerge [16] (Figure 2.3) is a stand-alone tool bundled with Apple's Xcode Development Tools, the only officially supported development environment for native applications on the Mac OS X platform. FileMerge's features are comparable to most other tools, offering a two-pane interface with support for merging and three-way comparisons.

Contrary to Apple fashion, the interface presents some idiosyncrasies. Direct access to differences is cumbersome, as it shares the same space with — and gets blocked by — the vertical scrollbar. In addition, given the interface has no toolbar or buttons, *next and previous* navigation is be done exclusively through keyboard shortcuts or via menu.

Unique to FileMerge is its ability to directly access classes and methods using a drop-down menu. Although similar in nature, this feature is not as advanced as Eclipse's Structure Compare.

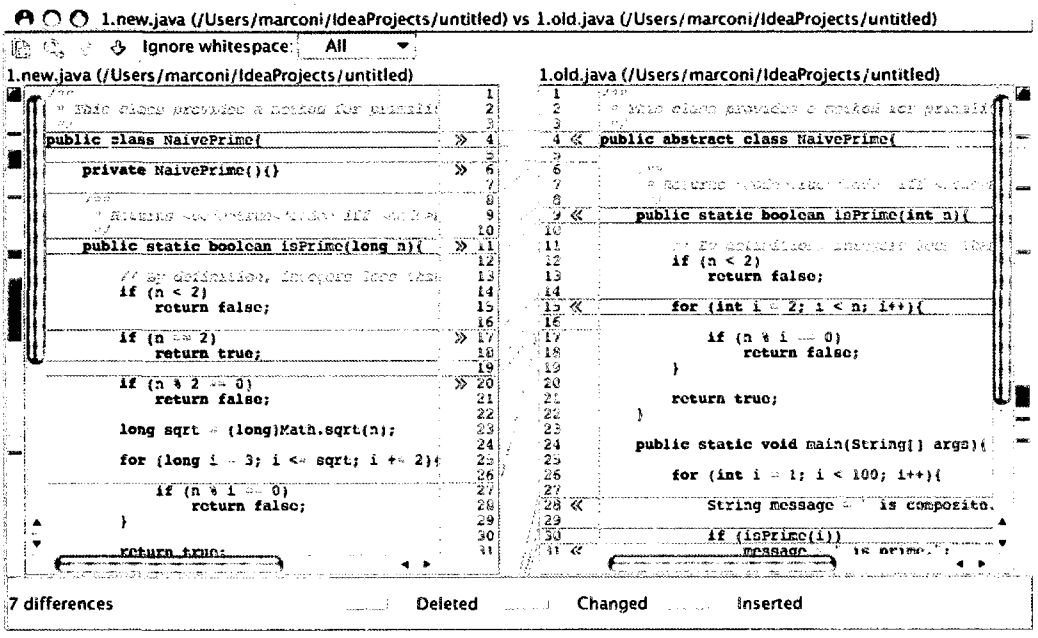

Figure 2.4: **IntelliJ IDEA**

2.2.4 IntelliJ IDEA

IntelliJ IDEA [28] (Figure 2.4) is a commercial IDE oriented mostly towards Java development. Its two-pane comparison interface compares favorably to most other tools, using colors to classify changes into "*inserted*", "*deleted*", and "*changed*". The tool supports syntax highlighting, merging, and three-way comparisons.

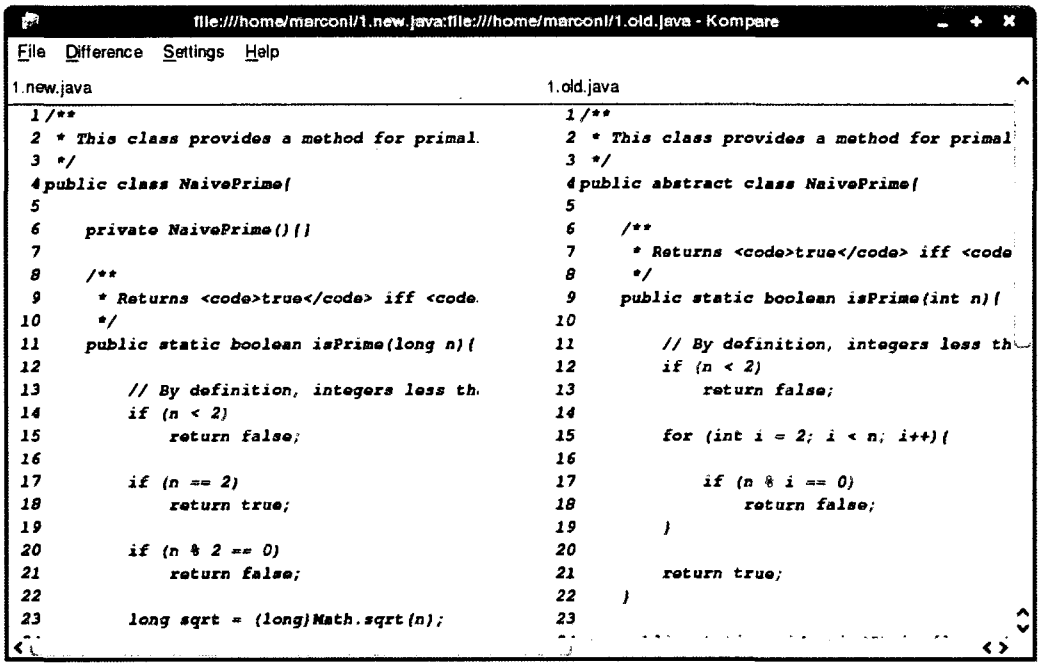

Figure 2.5: **Kompare**

2.2.5 Kompare

Kompare [33] (Figure 2.5) is a graphical front-end for the `diff` utility, developed for Unix systems running the K Desktop Environment (KDE). The two-pane interface uses colors to represent "*added*", "*removed*", and "*changed*".

The tool lacks features offered by most other tools, such as three-way comparisons and syntax highlighting. The tool provides single character highlighting, although this feature did not work properly on most of our evaluations. Therefore, it was considered to offer line highlighting only.

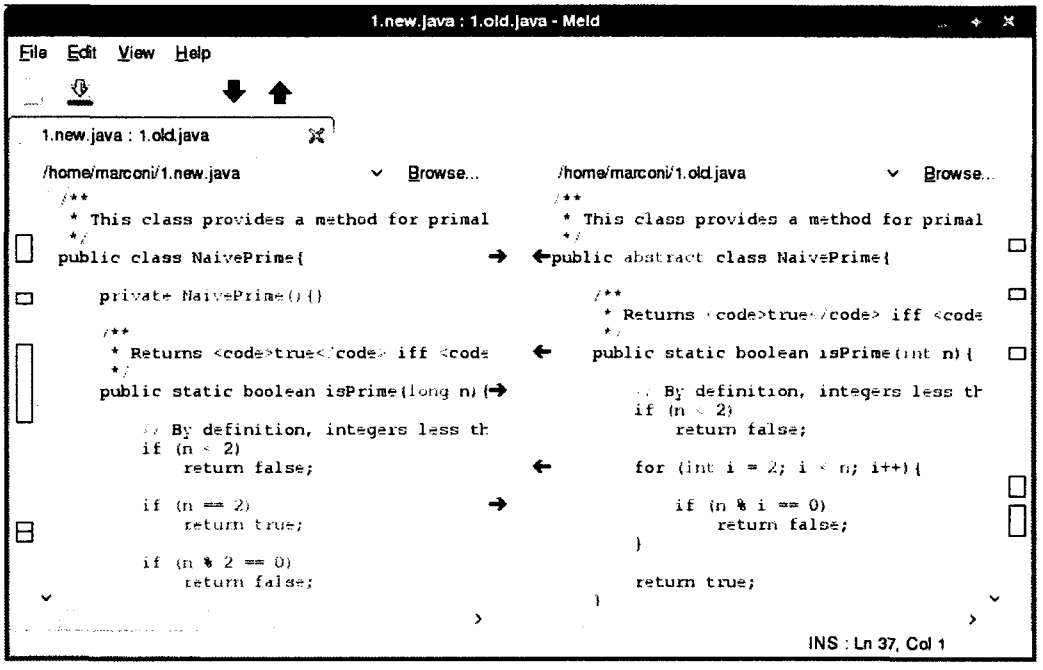

Figure 2.6: Meld

2.2.6 Meld

Meld [35] (Figure 2.6) is an open-source, stand-alone file comparison tool for Unix systems using the GNOME environment. Although the tool presents a pleasant and feature-complete interface, it does not support syntax highlighting and white space ignoring is limited to blank lines.

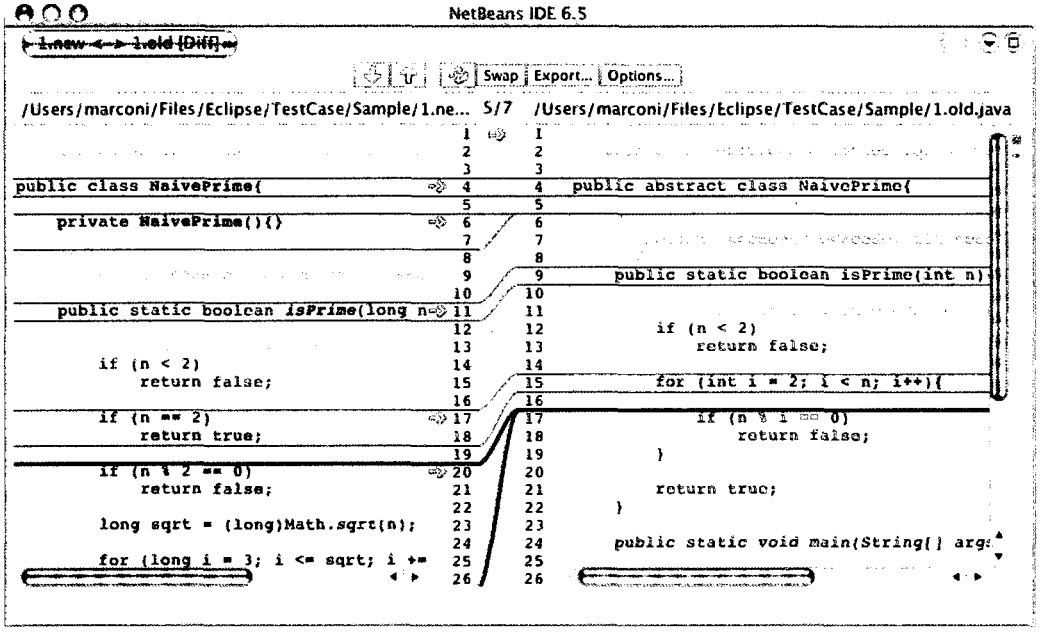

Figure 2.7: **NetBeans**

2.2.7 NetBeans

Sun Microsystems' NetBeans [41] (Figure 2.7) is a popular, open-source IDE targeting mostly Java development. Its two-pane comparison interface uses colors to classify differences and provides most features offered by other tools.

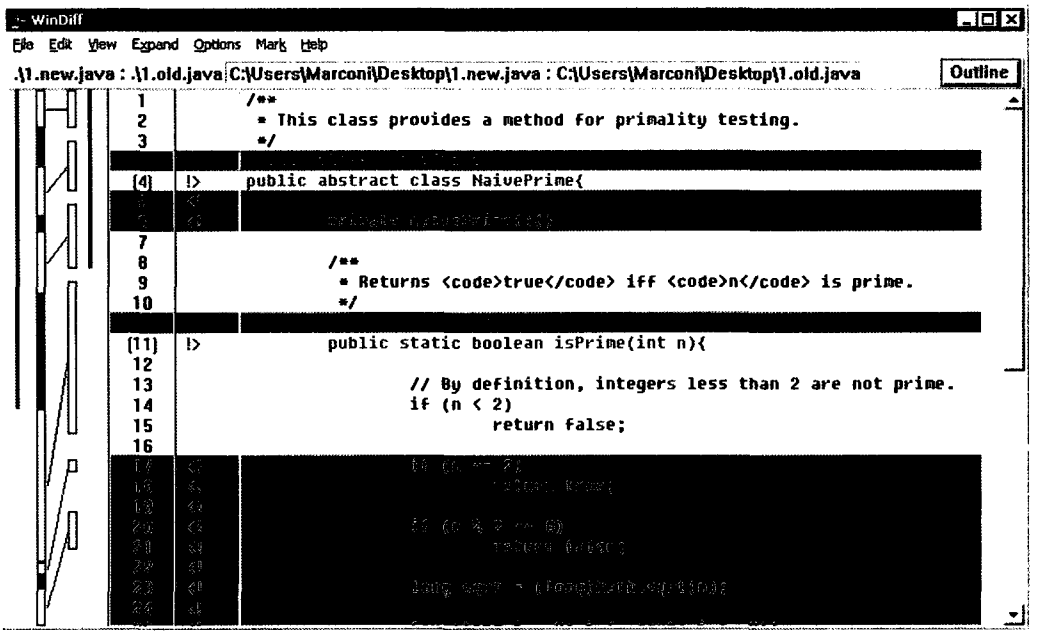

Figure 2.8: **WinDiff**

2.2.8 WinDiff

Microsoft's WinDiff [37] (Figure 2.8) is the file comparison tool distributed with the Visual Studio suite of software development tools for Windows. Even though the tool continues to be included even in the latest version of Visual Studio (2008), it seems to not have been updated in years, a reminiscent of Windows 3.1 days.

Its interface is unusual amongst the tools we analyzed, resembling more a textual than a graphical interface. Differences are represented using background colors: red represents differences from the left file, and yellow represents differences from the right file [38].

Given its lack of advanced features and awkward interface, the tool was included in this comparison only for completeness.

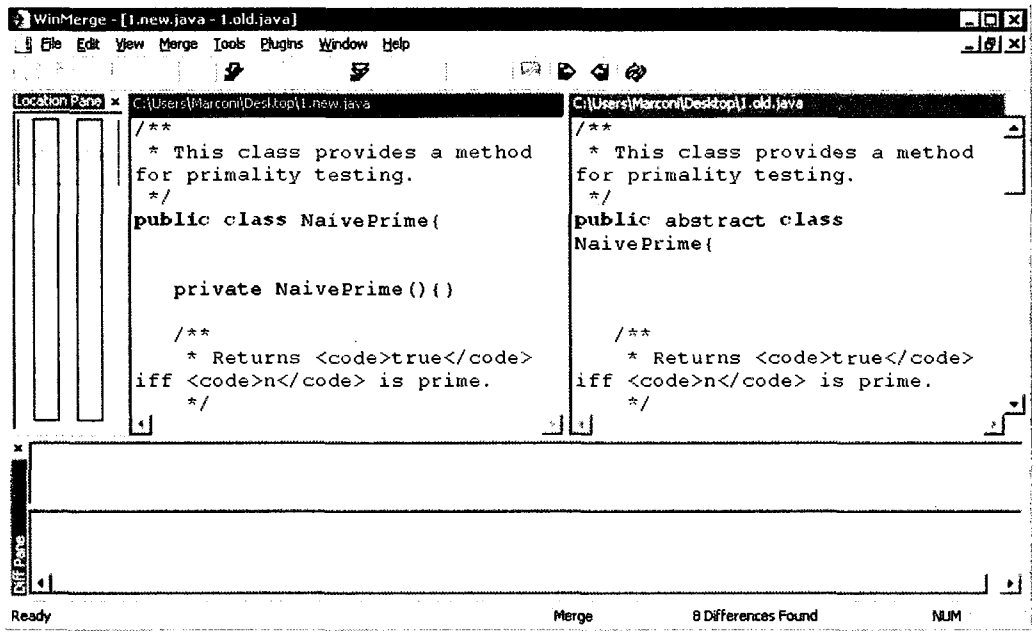

Figure 2.9: **WinMerge**

2.2.9 WinMerge

WinMerge [56] (Figure 2.9) is an open-source, stand-alone file comparison tool for the Windows platform. The tool offers a complete and advanced set of features, and supports plugins for extended functionality, such as ignoring code comments or extracting textual content from binary files.

Unique to WinMerge is its *quad-pane* interface with two horizontal panes at the bottom of the interface to display the current difference, corroborating our perception that two-pane interfaces are inefficient in their use of screen real estate (Section 3.2).

Amongst two-pane tools, WinMerge was the only tool not to support synchronized scrolling, resorting to blank line padding to keep both sides at the same height. The tool lacks a proper token parser, and only *words* separated by space or punctuation can be highlighted. Nevertheless, it was the only tool to support highlighting with single character granularity.

2.3 Feature Summary

Tables 2.1 and 2.2 summarize the features offered by the tools analyzed. Some features might be offered only as a user-selectable option.

Tool	Version	Metaphor	Alignment	Granularity	Navigation
diff	2.8.1	Textual	N/A	Line only	N/A
Eclipse	3.4.2	Two-pane	Sync	Token	Prev/Next, Direct
FileMerge	2.4	Two-pane	Sync	Token	Prev/Next, Direct
IDEA	8.1	Two-pane	Sync	Token	Prev/Next, Direct
Kompare	3.4	Two-pane	Sync	Line only	Prev/Next
Meld	1.2.1	Two-pane	Sync	Token	Prev/Next
NetBeans	6.5	Two-pane	Sync	Token	Prev/Next, Direct
WinDiff	5.1	GUIfied	N/A	Line only	Prev/Next, Direct
WinMerge	2.12.2	Quad-pane	Blank lines	Word, Character	Prev/Next, Direct

Table 2.1: **Comparison of File Comparison Tools**

Tool	Merge	Three-way	Syntax Highlight.	Ignore Space	Ignore Case
diff	No	No	No	Yes	Yes
Eclipse	Yes	Yes	Yes	Yes	Yes
FileMerge	Yes	Yes	Yes	Yes	Yes
IDEA	Yes	Yes	Yes	Yes	Yes
Kompare	Yes	No	No	Yes	Yes
Meld	Yes	Yes	No	Blank lines	No
NetBeans	Yes	Yes	Yes	Yes	Yes
WinDiff	No	No	No	Yes	Yes
WinMerge	Yes	No	Yes	Yes	Yes

Table 2.2: **Comparison of File Comparison Tools (continued)**

2.4 Chapter Summary

In this chapter we explored common features offered by notable file comparison tools. The next chapter reconsiders those features and the negative impact they can have on the user experience, building upon those limitations to introduce an improved file comparison interface metaphor.

Chapter 3

Spotting the Difference

> **compare** *estimate, measure, or note the similarity or dissimilarity between.*
>
> New Oxford American Dictionary, 2nd Edition

The previous chapter showed that most file comparison tools have a consistent set of features and similar user interfaces. With a few exceptions, it can be said that the typical file comparison tool has a two-pane interface, with synchronized vertical scrolling and mechanisms to navigate between differences; differences are highlighted at a line level, with fine-grained differences within a line further emphasized.

In this chapter, we analyze in more depth the features offered by file comparison tools, exploring their shortcomings and using this knowledge to design an improved file comparison interface.

3.1 Research Hypothesis Restated

The main hypothesis investigated in this thesis is that the ubiquitous two-pane interface metaphor is inefficient and ineffective to represent differences between files. Inefficient for its waste of screen real estate, especially in the critical horizontal dimension [42]. Ineffective for it makes reading and comparing changes difficult since text is duplicated and split across the screen.

To address those design flaws, a new interface metaphor is proposed: differences between files are consolidated and presented to the user into a single text view. We call it the *single-pane* interface. In the next sections, it is discussed how our investigation led to this simplified, more effective design.

3.2 Display Design

According to Wickens *et al.* [54]:

> "*Displays are human-made artifacts designed to support the perception of relevant system variables and facilitate the further processing of that information. The display acts as a medium between some aspects of the actual information in a system and the operator's perception and awareness of what the system is doing, what needs to be done, and how the system functions.*"

The authors describe thirteen principles of display design, of which we reproduce the following. It is easy to see how the file comparison tools analyzed in the previous chapter violate most of these principles.

3.2.1 Principles of Display Design

Principle 1: Make Displays Legible

> "*Legibility is critical to the design of good displays. Legible displays are necessary, although not sufficient, for creating usable displays.*"

Most tools make heavy use of lines surrounding blocks of text, connecting differences across the screen. Those lines can be confusing (Section 6.2.2), cluttering the interface and making it difficult to read. The proposed interface completely dispenses the use of such artifacts.

Principle 5: Discriminability

> "*Similarity causes confusion, use discriminable elements. Similar appearing signals are likely to be confused. The designer should delete unnecessary similar features and highlight dissimilar ones.*"

Some tools do not make the distinction between *additions*, *deletions*, and *modifications*, classifying all changes as *differences*, and leaving to the user the burden of interpreting their meaning. Classifying changes is one of the fundamental features of the proposed interface.

Principle 6: Principle of Pictorial Realism

> "*A display should look like the variable that it represents. If the display contains multiple elements, these can be configured in a manner that looks like how they are configured in the environment that is represented.*"

It is easy to argue that, for most people, a series of text changes do not look like two pieces of text displayed side by side. The proposed interface shows all pieces of text in the place they are most likely supposed to belong, highlighting which pieces were inserted, removed, or altered.

Principle 8: Minimizing Information Access Cost

> "*There is typically a cost in time or effort to 'move' selective attention from one display location to another to access information. Good designs are those that minimize the net cost by keeping frequently accessed sources in a location in which the cost of travelling between them is small.*"

Of all principles underlined here, this is probably the one that best describes the essence of the proposed interface. Information which is supposed to be compared should be arranged as close as possible. Two-pane interfaces completely break this principle, putting related information on separated sides of the screen. A user is always forced to move attention from one side to the other, constantly losing focus.

Principle 9: Proximity Compatibility Principle

> "*Sometimes, two or more sources of information are related to the same task and must be mentally integrated to complete the task; that is, divided attention between the two information sources for the one task is necessary. Good display design should provide the two sources with close display proximity so that their information access cost will be low.*"

Since, by design, two-pane interfaces violate Principle 8, they struggle to maintain reasonable levels of information proximity, "*linking* [information sources] *together with lines or configuring them in a pattern*", as described by the authors. Section 3.3.3 describes two mechanisms employed by the proposed interface to further reduce information access costs when it is inevitable to display two information sources at the same time.

Figure 3.1: **Spot the Difference:** Please, do not write on this page.

3.3 The Proposed Interface

Having seen the two-pane interface limitations, we can now suggest some interface advancements.

3.3.1 Single-pane Interface

The single most distinctive feature of the proposed system is the use of a single-pane interface. Files are not displayed side by side, but merged into a single view with differences highlighted.

We believe that using a single-pane interface improves usability by reducing interface clutter (Principle 1), providing a more pictorial data representation (Principle 6), and minimizing information access cost (Principle 8).

Interestingly, one of the main sources of inspiration came from a popular game for kids known as *Spot the Difference* (Figure 3.1, reproduced here under *fair dealing*). In this game, one has to find all differences between two slightly different versions of an image.

If one is willing to cheat, the game can be trivially solved with a simple trick: put one of the images on top of the other and all differences pop before one's eyes (Figure 3.2, on the next page, not to spoil the answer).

To understand figure 3.2, suppose the left image is colored green, and the right image is colored red. Superposing the images, features which are unique to the first image appear in green; features present only on the second image are in red; and where the images overlap, it is black.

If we assume the first image is the modified one and the second image is the original one, it can be said the green features in figure 3.2 were drew over the original image (or *added*) and

Figure 3.2: **Cheating on a Kids Game:** Colors Added for Clarity.

the red features were rubbed out from the original image (*deleted*). Extending the analogy, where green and red blend (as in the very top flower on the branches, the girl's shoes, or the sword cover) the image was *modified*.

The concept behind the single-pane interface is very similar to the trick: by "superposing" the files under comparison, parts that have not changed still look the same, while differences emerge to be easily spotted.

Using a single-pane interface to compare files is, actually, not a new idea. In fact, WinDiff (Section 2.2.8) uses a very primitive single-pane interface, intercalating files and highlighting all but common lines.

More elaborate single-pane comparison interfaces can be found on word processors such as Microsoft Word (Figure 3.3), Apple Pages (Figure 3.4), or OpenOffice Writer. Usually called "*Track Changes*", or similar, those features, when enabled, display all changes made to a document, including even metadata changes such as font and page formatting. Some of those tools are general enough to be used for source code comparisons and were an important source of inspiration for our interface.

3.3.2 Difference Classification

While some comparison tools do classify changes to improve discriminability (Principle 5), classifying changes into *additions*, *deletions*, and *modifications* is one of the core features of the proposed interface, given it lacks the spatial information provided by two-pane interfaces.

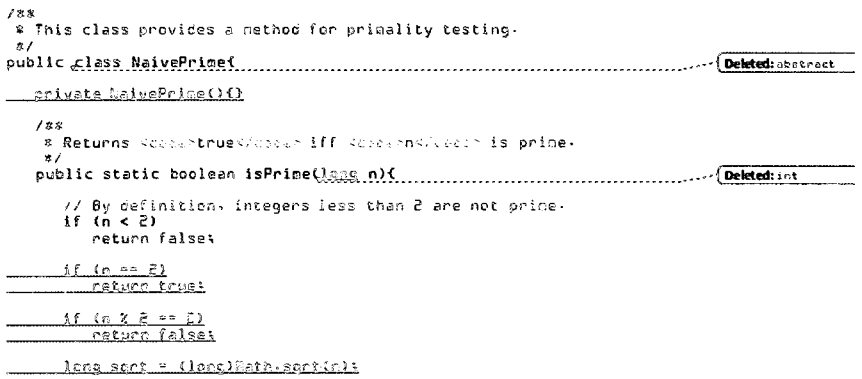

Figure 3.3: **Microsoft Word's Track Changes**

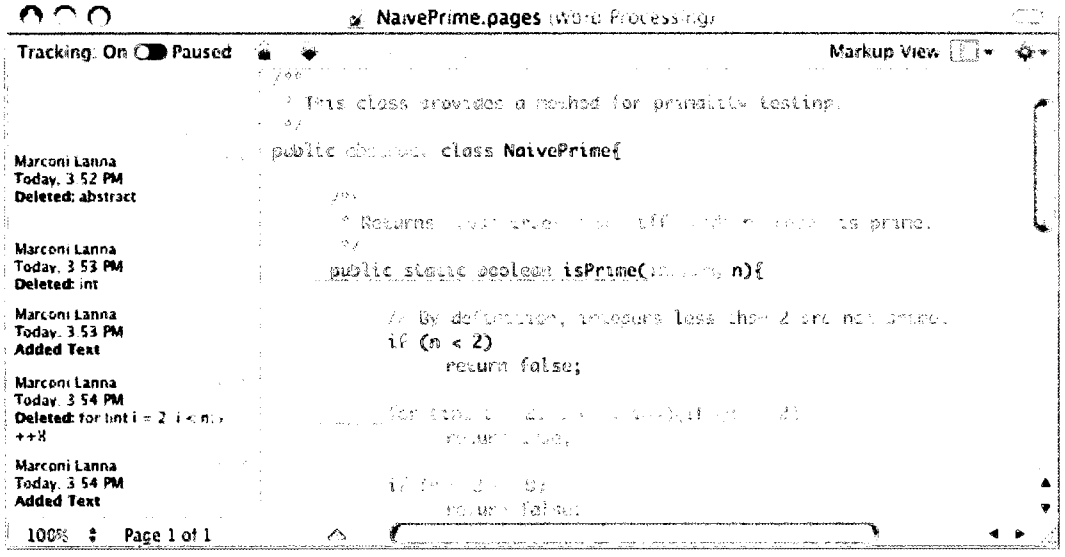

Figure 3.4: **Apple Pages' Track Text Changes**

Additions and Deletions

Additions and deletions are trivially understood. For the sake of the argument, assume nodes are either entirely removed or entirely inserted. Inserted nodes appear only on the *modified* version of a file and are called *additions*. Similarly, removed nodes are present only on the *original* version of a file and are called *deletions*. So, for instance, if file `abc` is changed into `acd`, we say node `b` is a *deletion* and node `d` is an *addition*.

The interface highlights additions in green and deletions in red, with strikeouts.

Modifications

Modifications are an abstraction, a more intuitive way of representing consecutive pairs of additions and deletions.

Suppose file abc is compared to file adc. Although it could be said that node b was removed and node d was inserted[1], usually it would be more intuitive to think about node b being altered into node d[2]. The pair b,d is called a *modification.*

In the interface, modifications are highlighted in orange.

3.3.3 Displaying Modifications

Modifications are particularly challenging to represent, since there are two sources of information, the original and the modified text, that need to be visualized at the same time (Principle 9). To display modifications, two complementary interface mechanisms were implemented: tooltips and hot keys.

By default, the interface always displays the modified version of the text, with one of the mechanisms being used to displayed the original text. Both mechanisms have their advantages, being more or less suitable for different scenarios. They were designed to complement, and not replace, each other.

Tooltips

The first mechanism implemented to display modifications were the tooltips, a pop-up window displayed when the mouse cursor hovers over a modification (Figure 3.5). The original text is displayed in the small window, close to its modified version, allowing the user to easily compare both versions without having to move the eyes across the screen.

While the tooltip mechanism does not eliminate information duplication, it limits duplication to a single change at a time, at most (Principle 1), while greatly reducing information access cost (Principle 8).

[1] See, for instance, Figure 3.3

[2] d may, in fact, not be a modification of b. It might be that node b was deleted and a new, unrelated node d was inserted, coincidently, between nodes a and c. We do not aspire to this level of enlightenment in this work.

```
/**
 * Returns <code>true</code> iff <code>n</code> is prime.
 */
public static boolean isPrime(long n){

    // By definition, integers l      han 2 are not prime.
    if (n < 2)                    int
        return false;

    if (n == 2)
        return true;

    if (n % 2 == 0)
        return false;

    long sqrt = (long)Math.sqrt(n);

    for (long i = 3; i <= sqrt; i += 2){

        if (n % i == 0)
            return false;
    }
```

Figure 3.5: **Tooltips**

```
/**
 * Returns <code>true</code> iff <code>n</code>
 */
public static boolean isPrime(int n){

    // By definition, integers less than 2 are
    if (n < 2)
        return false;

    if (n == 2)
        return true;

    if (n % 2 == 0)
        return false;

    long sqrt = (long)Math.sqrt(n);

    for (int i = 2; i < n; i++){

        if (n % i == 0)
            return false;
```

```
/**
 * Returns <code>true</code> iff <code>n</code>
 */
public static boolean isPrime(long n){

    // By definition, integers less than 2 are
    if (n < 2)
        return false;

    if (n == 2)
        return true;

    if (n % 2 == 0)
        return false;

    long sqrt = (long)Math.sqrt(n);

    for (long i = 3; i <= sqrt; i += 2){

        if (n % i == 0)
            return false;
```

Figure 3.6: **Hot Keys:** pressed (left) and released (right).

Hot keys

Tooltips are very useful for visualizing a single modification, but they do not scale well when, say, a line has many modifications. For displaying multiple modifications at once, a second mechanism was implemented: hot keys (Figure 3.6).

By pressing and holding a pre-defined key, all modifications displayed on the screen are replaced with their original text. The modified text reappears as soon as the user releases the

key. Additions and deletions are *not* reversed in the process.

Hot keys have the added benefit of stimulating the motion detection capabilities of the human brain.

3.3.4 Granularity

Most tools use two levels of highlighting — lines and tokens — which, in our opinion, increases interface clutter and reduces legibility. Using only token granularity to display differences improves readability.

Most importantly, token granularity is used to cleverly classify differences, leading to improved understandability. Suppose a line `abcd` is modified into `bCde`. Most tools would display the whole line as a modification, further highlighting tokens `a` and `c` on one side, and `C` and `e` on the other.

Differently, the proposed interface classifies and displays `a` as a *deletion*, the pair `c` and `C` as a *modification*, and `e` as an *addition*. Interpreting changes at this finer level of granularity gives more intuitive results, and is a feature not usually found on most file comparison tools.

3.4 File Comparison Features Revisited

The proposed features can be summarized by revisiting the criteria outlined in Section 2.1:

Interface Metaphor: Two-pane interfaces can be inefficient and ineffective interface metaphors. The proposed model adopts a single-pane interface to display differences.

Vertical Alignment: Since files are not displayed side by side, it is not necessary to maintain vertical alignment.

Highlighting Granularity: Experimentation has showed that single character granularity can be too fine-grained, producing a large number of differences. Line granularity, on the other hand, is too coarse-grained, demanding the user to read two whole lines to identify what was actually changed. Therefore, token granularity was chosen. Differently from most other tools, whole lines are not highlighted, avoiding interface clutter and allowing for fine-grained difference classification.

Difference Navigation: Initially, difference navigation was not implemented. For further discussion, refer to Section 6.3.1.

Syntax Highlighting: Although it was not strictly necessary for the study, syntax highlighting was implemented to improve readability.

Ignore White Space/Case: During experimentation, white space handling showed itself to be an essential feature. Section 4.3.4 provides a detailed discussion about challenges and solutions. Although it would have been trivial, we did not see the need to implement case ignoring.

Merge Support and Three-way Comparisons: These features were considered outside the scope of this work.

3.5 Chapter Summary

In this chapter we showed how to improve file comparison usability and proposed new interface metaphors: single-pane interface, finer level of difference highlighting and classification, and special artifacts to display modifications.

The next chapter discusses the design and implementation of the prototype used in the usability experiment.

Chapter 4

Architecture and Implementation

In this chapter we describe the architecture, design decisions, and implementation challenges faced while developing the proposed tool.

We named the prototype "*Vision*", a play with the word *revision* — which literally means "*see again*", a satirical reference to two-pane interfaces.

4.1 The Platform

One of our first design decisions in the early development stages was to implement the tool as a plug-in for the Eclipse platform. We can name a few benefits that motivated this decision.

Firstly, the Eclipse platform provides a vast selection of services such as file comparison, lexical analyzers, syntax highlighting, rich text widgets, text hovers, and integration with Source Code Management systems. The availability of those services greatly simplified the implementation and reduced development time.

Secondly, implementing our prototype on top of the same technologies used by the reference tool (Section 5.1) gave us a level playing field for comparing the tools. It would have been more difficult to determine the effectiveness of the proposed interface if we could not otherwise isolate external factors such as, for instance, the difference engine.

Finally, being a plug-in for a popular development environment should give the tool some visibility and acceptance should it eventually be publicly released. It should also be mentioned that most participants of the usability experiment were already acquainted with the Eclipse IDE and, therefore, our tool presented them with a familiar interface look-and-feel.

4.2 Design and Architecture

The system design and architecture was inspired, and occasionally even restrained, by the platform itself. Most of the initial code came from reverse engineering Eclipse's own file comparators, mainly `org.eclipse.compare.contentmergeviewer.ContentMergeViewer`. The system design and architecture had to follow numerous conventions regarding interfaces to be implemented and classes to be extended [8, 13, 14, 15, 20].

The system main classes are represented in the following UML diagram (Figure 4.1):

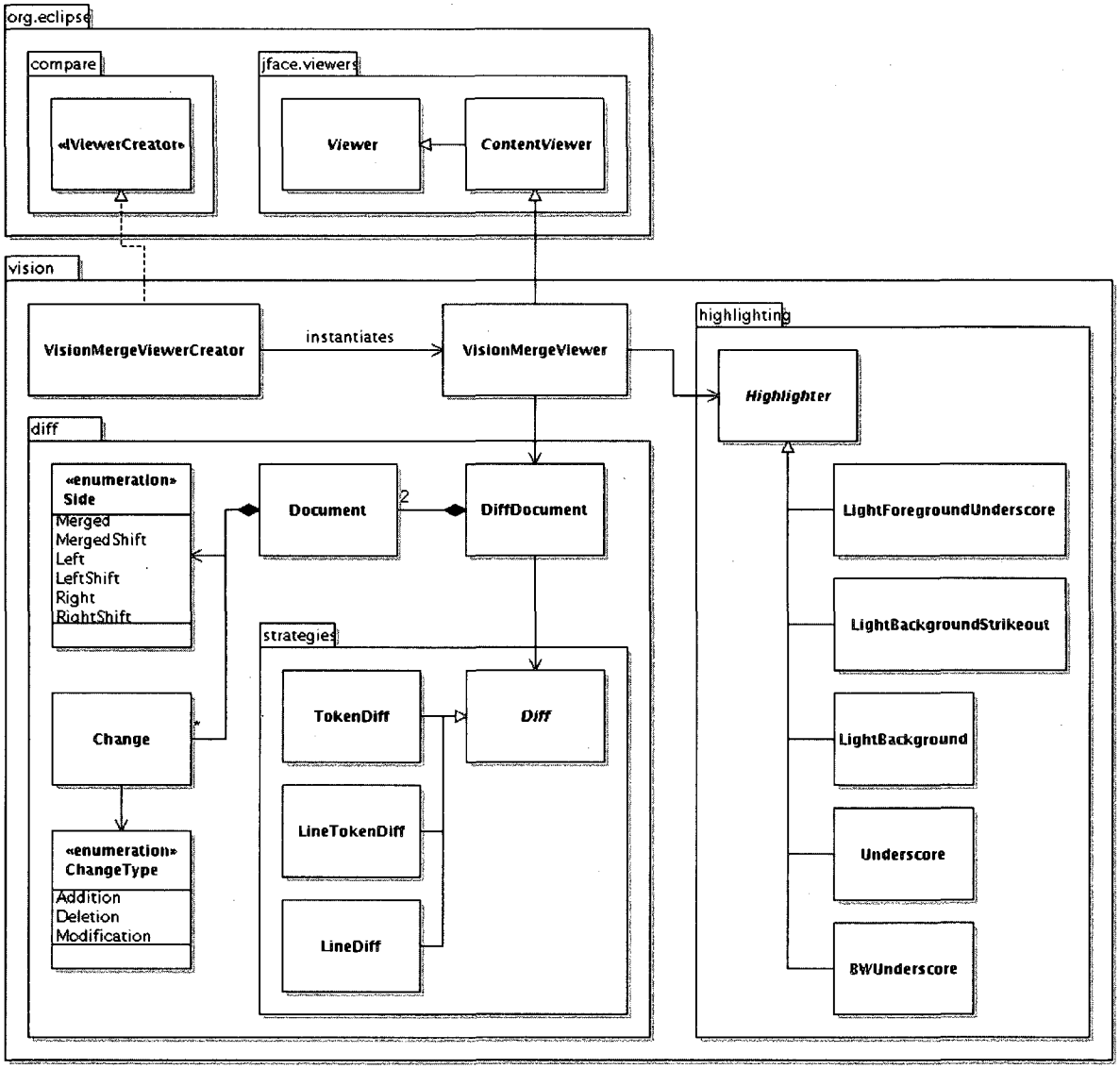

Figure 4.1: **Vision UML Class Diagram:** some classes omitted for clarity.

The starting point of the system is the `VisionMergeViewerCreator` class, required by the platform to extend the `org.eclipse.compare.IViewerCreator` interface, and whose sole purpose is to instantiate the `VisionMergeViewer` class.

`VisionMergeViewer`, the main system class, extends the abstract class `org.eclipse.jface.viewers.ContentViewer`. It is responsible for initializing other system classes and platform services. The main input to this class, the pair of files to be compared, is provided by the platform. Since the tool integrates with the *Team* capabilities offered by the platform, input may come from any of the following:

- Files from the file system;
- Versions from local history;
- Revisions from a supported Source Code Management repository.

After pre-processing the input, `VisionMergeViewer` creates an instance of the `DiffDocument` class, passing the files to be compared as parameters to its constructor.

To compute the differences between the files, `DiffDocument` invokes a static method of the abstract class `Diff`, which itself delegates to one of its concrete implementations: `TokenDiff`, `LineTokenDiff`, or `LineDiff`. `Diff` then returns an iterator to a list of `org.eclipse.compare.rangedifferencer.RangeDifference` objects computed by `RangeDifferencer`, from the same package.

`DiffDocument` uses this set of raw differences to compute a pair of `Documents`. Each `Document` is composed of a version of the merged text from the input files and a list of `Changes` describing the differences between them. Section 4.3 discusses in more detail the process briefly depicted in this paragraph and the previous one.

The pair of `Documents` is then used by `VisionMergeViewer` to render the user interface. Text is actually displayed on the screen by `org.eclipse.jface.text.source.SourceViewer`, configured by the `org.eclipse.jdt.ui.text.JavaSourceViewerConfiguration` class.

Difference highlighting is performed by one of the concrete `Highlighter` implementations. Most are combinations of foreground or background highlighting colors, combined or not with strikeouts and underscores. Available options can be selected at runtime. One particular implementation, `BWUnderscore` (Figure 4.2), uses only underscores and strikeouts without colors to represent the different types of changes. It was intended mainly at producing black and

```
/**
 * This class provides a method for primality testing.
 */
public abstract class NaivePrime{

    private NaivePrime(){}

    /**
     * Returns <code>true</code> iff <code>n</code> is prime.
     */
    public static boolean isPrime(long n){

        // By definition, integers less than 2 are not prime.
        if (n < 2)
            return false;

        if (n == 2)
            return true;
```

Figure 4.2: **BWUnderscore**

white printings, but could also be useful for color-blind persons, although it was not possible to evaluate it for this purpose.

4.3 Making a Difference

This section describe how the merged document, `Document`, and its set of `Changes` is computed from the pair of files being compared.

4.3.1 Difference Computation

Actual file comparison is performed by `RangeDifferencer`, a utility class provided by the framework implementing the file comparison algorithm described in [39]. `RangeDifferencer` takes two `org.eclipse.compare.contentmergeviewer.ITokenComparators` as input and returns the Longest Common Subsequence (LCS), represented by an array of `RangeDifferences`.

Different `ITokenComparators` can be used to manipulate the comparison strategy. Comparison strategies are encapsulated by the `vision.diff.strategies` package. Three strategies were implemented, all specific to Java source code. Support for additional programming languages — or general text files — can be easily implemented by extending the `Diff` class.

The first strategy implemented, `JavaDiff`, compares the input token by token, as defined by `org.eclipse.jdt.internal.ui.compare.JavaTokenComparator`[1]. This strategy deviates

[1]The platform discourages the use of *internal* packages in production systems. Notwithstanding, it was considered harmless for a prototype while simplifying its development.

from conventional line-by-line comparisons, which are more efficient to compute. Nevertheless, the strategy ended up being reasonably fast to compute, at least on modern personal computers.

The finer level of granularity provided the `JavaDiff` usually led to clearer, more comprehensible results than the conventional line-by-line strategy. However, this strategy suffered some severe complications when dealing with complex sets of changes, specially those described in Section 6.3.5, *Line Reordering.*

Consequently, we decided to revert to a more traditional approach (Algorithm 4.1). Firstly, differences are computed on a line-by-line basis (line 2). Then, for a range of consecutive differing lines, differences were computed recursively using token granularity (line 9). This strategy is implemented by `LineTokenDiff`.

A third strategy, `LineDiff`, which computes differences on a line basis only, was implemented after the usability experiment to support the features described in Section 6.3.5.

Algorithm 4.1: DifferenceComputation

Input: A pair of files to be compared, `left` and `right`

Output: A list of difference ranges, `differences`

```
1  differences ← ∅
2  aux ← computeLCS(left, right, LineStrategy)
3  while range ← aux.next do
4      if range.rightLength = 0 then
           // Empty right side: the entire line(s) was added
5          differences.add(range)
6      else if range.leftLength = 0 then
           // Empty left side: the entire line(s) was deleted
7          differences.add(range)
8      else
           // No empty sides: process recursively using token granularity
9          aux2 ← computeLCS(range.left, range.right, TokenStrategy)
10         while subrange ← aux2.next do
11             differences.add(subrange)
12 return differences
```

4.3.2 Difference Classification

The Longest Common Subsequence as computed by `RangeDifferencer`, independently of the comparison strategy used, is not sufficient for the purposes of our interface. Differences have to be filtered and interpreted before computing the `Document` pair and their `Changes`.

The main problem is how to infer, from a raw set of differences, *additions*, *deletions*, and *modifications*. Take, for instance, a line of code `a = b` modified into `a = c + d`. It can be said that:

1. `b` was modified into `c + d`;
2. `b` was modified into `c` and `+ d` was added;
3. `c +` was added and `b` was modified into `d`;
4. `b` was modified into `+`, `c` and `d` were added;
5. `b` was deleted and `c + d` was added;
6. And similar permutations.

Given the problem does not tolerate a formal, unique solution, a set of heuristics was developed to approximate an answer (Algorithm 4.2).

Differences are initially separated into three groups for classification. First, differences which appear only in the modified version of the file are classified as *additions* (lines 3–4). Analogously, differences which appear only in the original version are classified as *deletions* (lines 5–6).

The third group is composed of the differences which appear on both sides. Unfortunately, it would not be adequate to trivially classify those differences as *modifications*: the ranges may have an uneven number of differences coming from each side and experimentation has shown that, usually, one token or line of code is *not* modified into two tokens or lines of code.

The `LineTokenDiff` difference computation strategy described in the last section handles such cases with appreciable elegance, refining a block of differing lines into a new set of finer grained differences. Those differences are then recursively classified as additions, deletions, and modifications.

Algorithm 4.2: DifferenceClassification

Input: A list of difference ranges, `differences`

Output: A list of classified changes, `changes`

```
1  changes ← ∅
2  while range ← differences.next do
3      if range.rightLength = 0 then
           // Empty right side: the content on the left was added
4          changeType ← Addition
5      else if range.leftLength = 0 then
           // Empty left side: the content on the right was deleted
6          changeType ← Deletion
7      else
           // No empty sides: the content on both sides was modified
8          changeType ← Modification
9      i ← 0
10     while difference ← range.next do
11         i ← i + 1
12         if changeType = Modification then
13             if i > range.rightLength then
                   /* No more differences on the right side: remaining
                      differences on the left are considered additions      */
14                 changeType ← Addition
15             else if i > range.leftLength then
                   /* No more differences on the left side: remaining
                      differences on the right are considered deletions     */
16                 changeType ← Deletion
17         changes.add(new Change(difference, changeType))
18 return changes
```

For the remaining cases with uneven numbers of differences from each side, differences are matched to one another, *in order*, and classified as *modifications*. Exceeding differences, to one side or the other, are classified as additions or deletions, respectively (lines 13–16).

This arrangement produced overall good results, while still being simple to implement and understand.

4.3.3 Merged Document

The merged document, used by the user interface to display differences on the screen, is computed directly from the files being compared and their differences.

All text belonging to the Longest Common Subsequence is copied verbatim into the merged document, as well as all differences classified as additions or deletions (Section 4.3.2). For modifications, only the modified text is copied into the merged document, while the original text is saved in an auxiliary data structure used to display the tooltips.

To implement the hot-key feature efficiently, a mirror copy of the merged document is produced by reversing modification order: the original text is copied into the document, while the modified version is saved in parallel. Additions and deletions are *not* reversed in the mirror document.

4.3.4 White Space Handling

For comparison purposes, the interface *always* ignores differences in white space. However, while white space could easily be ignored when computing differences, highlighting white space showed itself to be a more challenging problem.

Highlighting all white space differences (Figure 4.3, taken from an earlier prototype) produced cumbersome, not to say meaningless, results.

On the other hand, ignoring all white space (Figure 4.4) leads to many small differences separated by a few spaces. A balanced solution had to be reached.

Many strategies were tried, like ignoring all white space at the beginning and end of lines, ignoring all unaccompanied white space, or ignoring only consecutive white space. Through experimentation, the strategy that yield the best results was to ignore line *and* difference leading and trailing white space, while highlighting inter-token white space within differences; the results can be appreciated in all screenshots throughout this thesis.

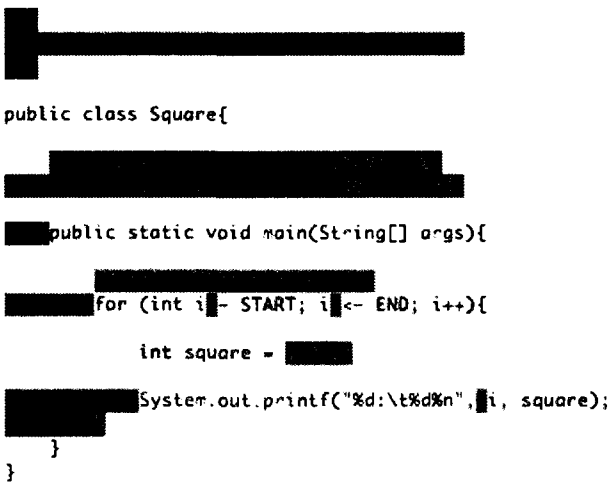

Figure 4.3: **Highlighting All White Space Differences**

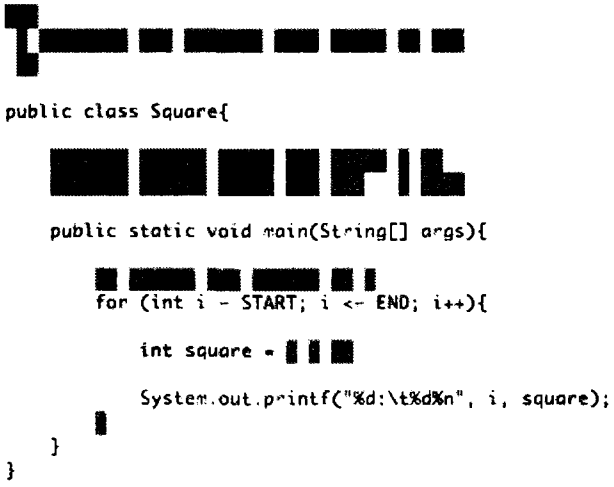

Figure 4.4: **Ignoring White Space Differences**

4.4 Chapter Summary

This chapter gave an overview of the system design and architecture, showing how it integrates and makes use of the services offered by the platform. Implementation challenges and heuristics to compute and classify differences and the merged document were discussed.

In the next chapter we show how the prototype behaved during the usability experiment, compared to the reference tool.

Chapter 5

Usability Evaluation

To validate the proposed interface model, we conducted a usability study with sixteen participants[1] using six real-world test cases. In this chapter, we describe the usability experiment and discuss its main results.

The experiment described here together with the documents reproduced in Appendices G, H, I, and J were reviewed and approved by University of Ottawa Health Sciences and Science Research Ethics Board, certificate **H 07-08-02**.

5.1 Methodology

The main experiment consisted in performing six comparison tasks against the selected test cases using two tools: the proposed tool, as described in Chapter 3, and a reference tool.

For the reference tool, the Eclipse IDE was selected because of its popularity amongst Java developers [18], advanced set of features (Section 2.2.2), and similarity to the proposed tool, given both tools are implemented on top of the same framework (Section 4.1). It is our belief that any other comparison tool with a similar set of features would deliver equivalent results in this experiment.

All participants used both tools to perform the experiment, half the comparisons each, alternating between the tools at each comparison. The first participant started the experiment using the reference tool, the second using the proposed tool, and so forth. Test cases were presented always in the same order (Section 5.1.1), regardless of which tool was used first.

[1] When referring to a participant in singular, the pronoun *she* will always be used, regardless of participant gender.

Therefore, each test case was compared using each tool half the time.

Initially, the participants were introduced to both tools using a sample test case (Section 1.6) to demonstrate how comparisons are made, how features are used, and how the output is to be interpreted. Then, participants were asked to perform one of the comparison tasks and, in a second step, explain the differences between the files. The first step was timed, while the second was not. Participant answers were recorded on a spreadsheet. No feedback was given to participants during the experiment.

For the complete experiment script, please refer to Appendix F.

5.1.1 Test Cases

Six test cases were selected among popular open-source Java projects, which gave us a diversified spectrum of coding styles and changes:

1. Google Collections Library [24];
2. Project GlassFish [22];
3. The Eclipse Project [11];
4. The Jython Project [31];
5. Spring Framework [51];
6. JUnit Testing Framework [30].

The test cases were selected in a roughly arbitrary manner, to help prevent bias. First, the source code repository of a project was randomly browsed, looking for files having approximately between 100 and 200 lines of code. When a suitable candidate was found, we descended its revision history till there were about seven to 30 individual changes. Those parameters were selected to give us a good balance of code size and complexity while avoiding extensively lengthy and difficult comparisons.

The test cases were then subjectively ordered by complexity and length, ranging from small and simple to large and complex, and numbered from 1 to 6. Presenting the test cases in increasing order of complexity — rather than in random order — allowed participants to address any learning curve they might have.

Participants were not told about the nature of the test cases.

Appendix A reproduces the complete source listing of all test cases. Appendix B lists all differences participants were supposed to report.

5.1.2 Answer Grading

Participant answers usually do not fall into just two categories, *right* or *wrong*. Subtleties have to be considered when judging participant answers. During the experiment, the following criteria were adopted:

Right: The participant described the difference with reasonable accuracy;

Partial: The participant partially described the difference;

Omission: The participant failed to notice the difference;

Error: The participant described the difference incorrectly, or described something that was not considered to be a difference.

When evaluating participant or tool performance, it is useful to have a single unit of measurement. For this purpose, we suggest using a *weighted score* scale, defined as[2]:

$$WeightedScore = (0 \times Right) + (0.5 \times Partial) + (1 \times Omission) + (2 \times Error)$$

5.1.3 Environment Configuration

For the experiment, we used the "*Eclipse IDE for Java Developers*" distribution, version 3.4.1 Ganymede [12], on an Apple Macintosh computer running Mac OS X 10.5.5 Leopard connected to a standard 17-inch LCD display, native resolution of 1280×1024 pixels, 75Hz vertical refresh rate, and a stock two-button mouse with a vertical scrolling-wheel.

The Eclipse IDE was running with default settings, except for the following: On Preferences, General, Compare/Patch, General the Open structure compare automatically option was **deselected**, while the Ignore white space option was **selected**. The first option was deselected to reduce interface clutter, while the second was selected to reduce the number of spurious changes reported by the reference tool, bringing its output closer to the output of the proposed tool.

[2]Although this particular choice of relative weights is somewhat arbitrary, no reasonable choice of positive factors would reverse the results discussed in section 5.4.

The Java perspective was used with all of its views closed, except for the Package Explorer view, which was minimized. The workbench window was maximized, and the Hide Toolbar option was selected. The Mac OS X *Dock* had the hiding option turned on. All those measures were taken to avoid distractions and maximize the screen area allocated to the editor window used for file comparisons.

5.2 Participants

For this study, we were able to recruit sixteen participants with various levels of experience with the Java programming language and file comparison tools (Section 5.2.1). While most participants were graduate students, some of them were professional software developers working in the industry.

5.2.1 Self Assessment Form

Below we reproduce participant's answers to the Self Assessment Form (Appendix I).

The first two questions asked the participants about their experience with the Java programming language and the Eclipse development environment (Figure 5.1).

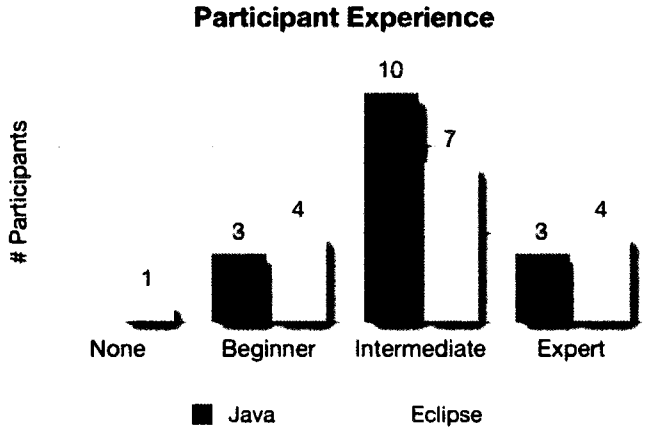

Figure 5.1: **Participant Experience**

For this experiment, we wanted participants with a broad variety of skills, ranging from inexperienced users to experts. All participants claimed to have at least beginner-level knowledge of the Java programming language, meeting the experiment's only prerequisite. Most participants considered themselves to be intermediate users of both Java and Eclipse, with a

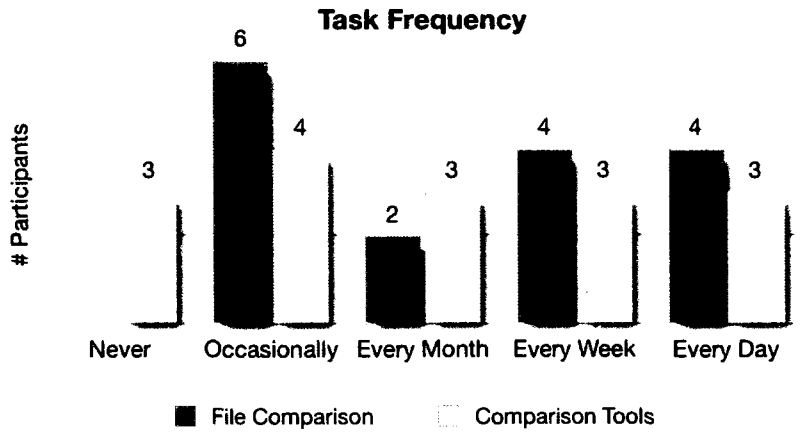

Figure 5.2: **Task Frequency**

smaller but significant number of beginners and experts. Only one participant said to have no experience using Eclipse, which was acceptable for this study.

The next two questions asked participants about the frequency they perform file comparison tasks and how often do they use a specialized file comparison tool (Figure 5.2). Half the participants claimed to compare files at least once a week, whereas most others would do it only occasionally. File comparison tools are used roughly most of the time, even though three participants claimed never to use them.

5.3 Experimental Results

5.3.1 Participant Performance

In this section we analyze the individual performance of participants, without regard to the tools used. Looking at participants individually, we can see there was a significant variance among them regarding time spent to perform tasks and number of mistakes made.

To perform all comparison tasks, participants were as fast as 3 minutes and 27 seconds or as slow as 22 minutes and 51 seconds, a span of over 660% (Figure 5.3). Looking at Figure 5.3, though, reveals that participants were evenly distributed over the range from about 200 to 650 seconds, with only one participant clearly outside this range, *Participant 16.*

Since Participant 16 was more than twice as slow as the second slowest participant, we decided to remove the respective data from our performance analyses. Otherwise, it would unbalance all comparisons, distorting the experiment results against one tool or the other

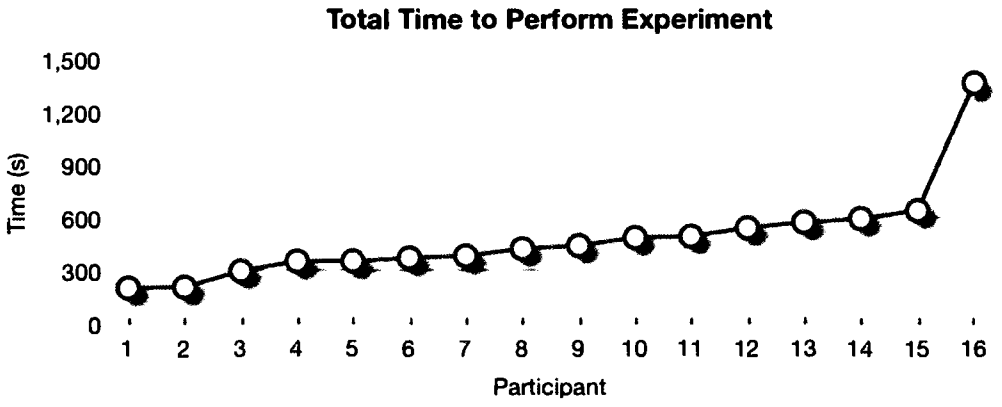

Figure 5.3: **Participant Time:** Ordered by time. Participant numbers anonymized.

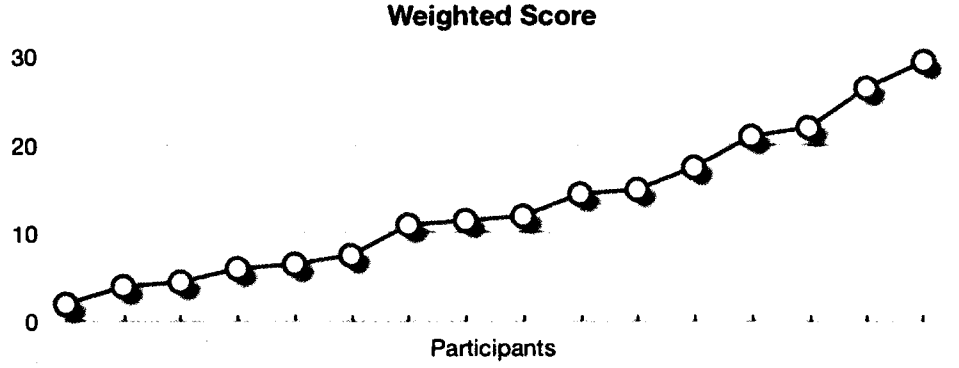

Figure 5.4: **Weighted Score:** Ordered by score. Participant numbers not shown.

at each comparison[3]. For reference, Appendix E reproduces the main time charts including Participant 16 data.

Individual participant performance was even more divergent when comparing the number of mistakes done during the experiment (Figure 5.4). Weighted scores ranged from 2 to 29.5, a span of almost 15 times. Despite the variance, the distribution was smooth, with no outliers. All data was therefore considered, *including* Participant 16.

In Figure 5.5 we plot a scatter diagram combining both metrics, time and score (Participant 16 not represented). Linear regression analysis[4] shows that there is no clear correlation between time and score, with a coefficient of determination $R^2 = 0.010$.

[3] As a matter of fact, keeping the data would, overall, favor the proposed tool.

[4] $y = -0.0065x + 15.89$

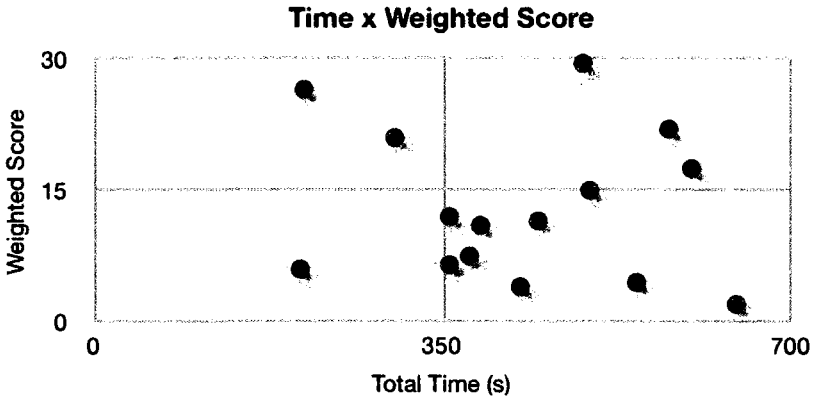

Figure 5.5: **Time × Weighted Score**

Finally, it is important to assess how evenly participant performance was distributed among those who started the experiment using the reference tool (*Group 1*) and those who started using the proposed tool (*Group 2*). Given participants were randomly assigned to groups — by order of arrival — ideally we should have similar levels of performance for both groups.

Unfortunately, participants in Group 1 performed notably better than participants in Group 2, with an average total time to perform the experiment of 362 seconds, versus 487 seconds for Group 2. Furthermore, Group 1 committed less mistakes with an average weighted score of 11.0, versus 15.4 for Group 2.

5.3.2 Task Performance

In this section we show the time each participant took to perform the comparison tasks, grouped by comparison tool and, for better visualization, ordered by participant time (Figures 5.6–5.11).

Since comparisons 1 and 2 were the first participant contact with the tools, they were expected to take relatively more time on average, even though those were the simplest test cases. Comparisons 3 to 6 were performed roughly in increasingly average time, as expected.

Statistical hypothesis testing using the *one-tailed Welch's t test* [53] — two samples of different sizes, unequal variances, and null hypothesis that one mean is greater than or equal to the other — showed that test cases 4 and 6 achieved 99.9% confidence level, while test cases 2 and 1 had, respectively, 95% and 90% confidence levels (Table D.1). Test case 5, the only the proposed tool was slightly slower than the reference tool, was not statistically significant. Combining the significance tests using Fisher's method [17] resulted in a *p-value* of 3×10^{-6}.

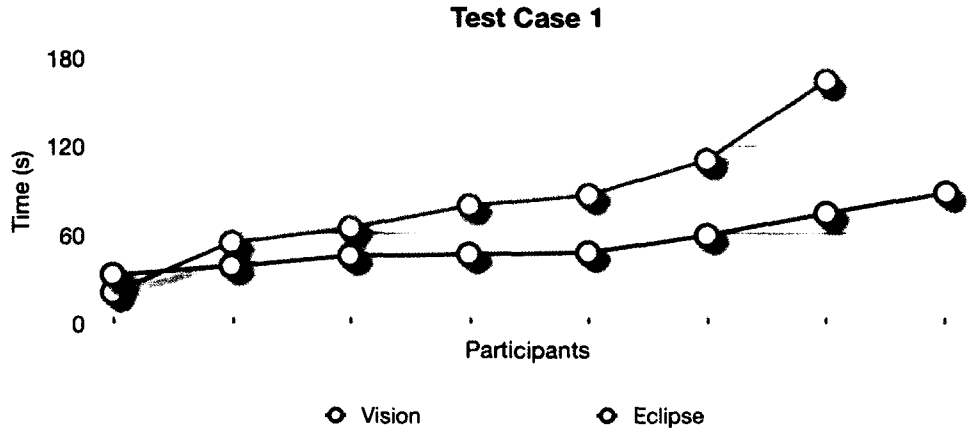

Figure 5.6: **Time to Perform 1st Comparison Task**

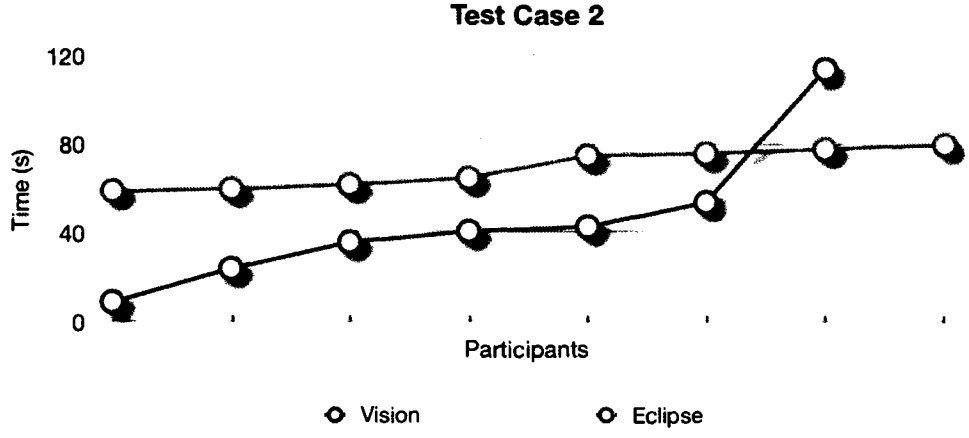

Figure 5.7: **Time to Perform 2nd Comparison Task**

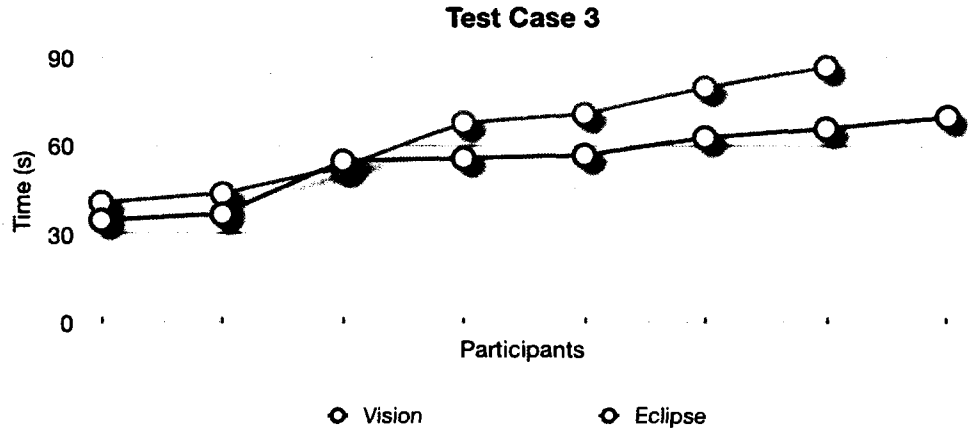

Figure 5.8: **Time to Perform 3rd Comparison Task**

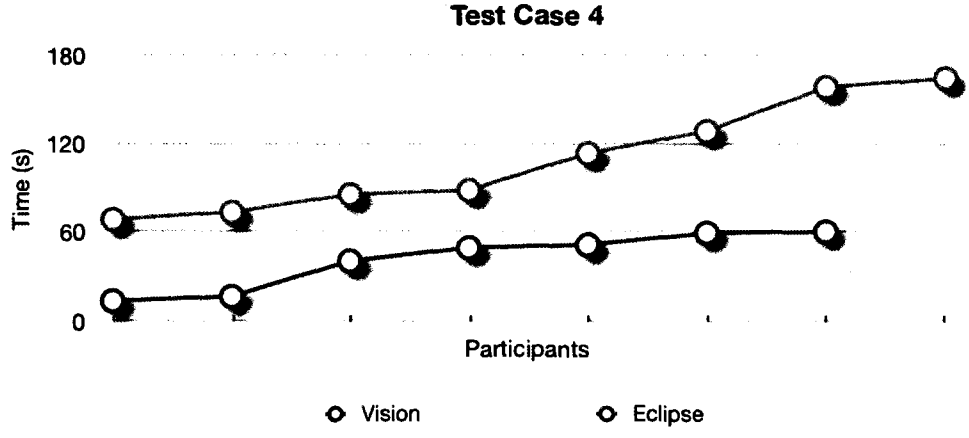

Figure 5.9: **Time to Perform 4th Comparison Task**

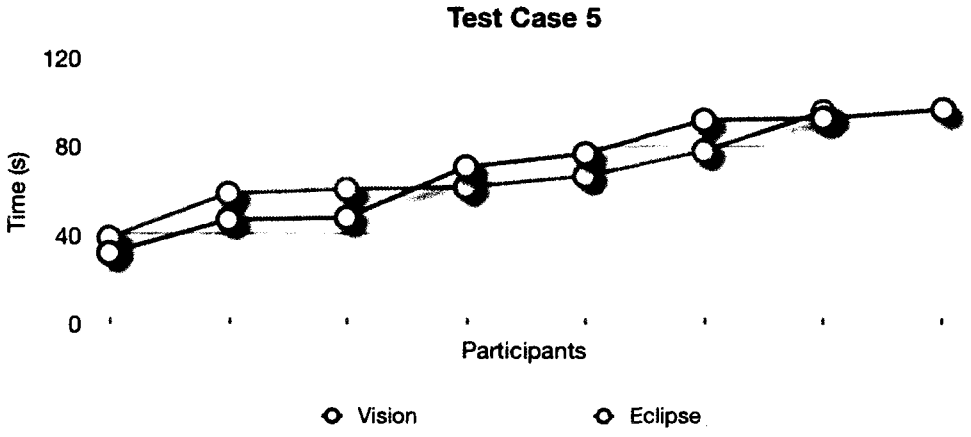

Figure 5.10: **Time to Perform 5th Comparison Task**

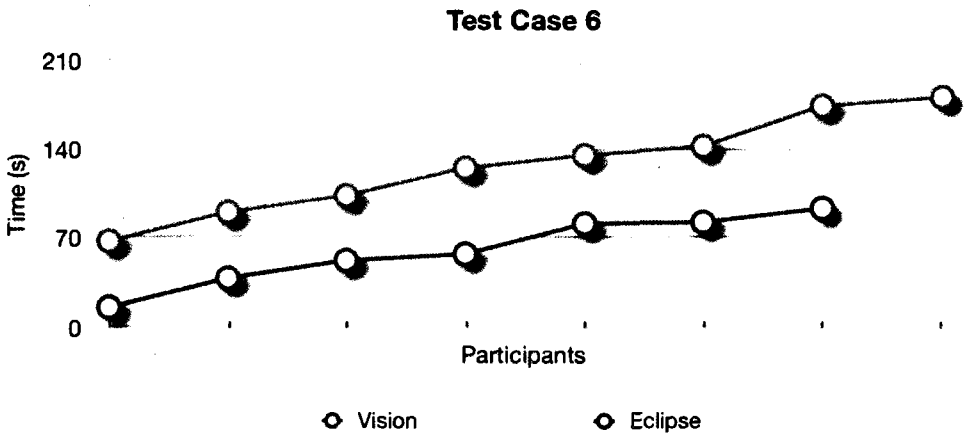

Figure 5.11: **Time to Perform 6th Comparison Task**

5.3.3 Participant Answers

Figures 5.12 to 5.16 show the total number of mistakes made by all participants for each comparison task, grouped by comparison tool.

Again, comparisons 1 and 2 performed relatively worse than what would be expected given their complexity level. Comparisons 3 to 6 had strictly increasing average weighted scores, in agreement with our estimations.

Statistical significance — again using the one-tailed Welch's *t test* — regarding the total number of incorrect answers was obtained only for test case 4, at the 95% confidence level (Table D.3). The combined statistical significance of all experiments according to Fisher's method was $p = 4.3\%$.

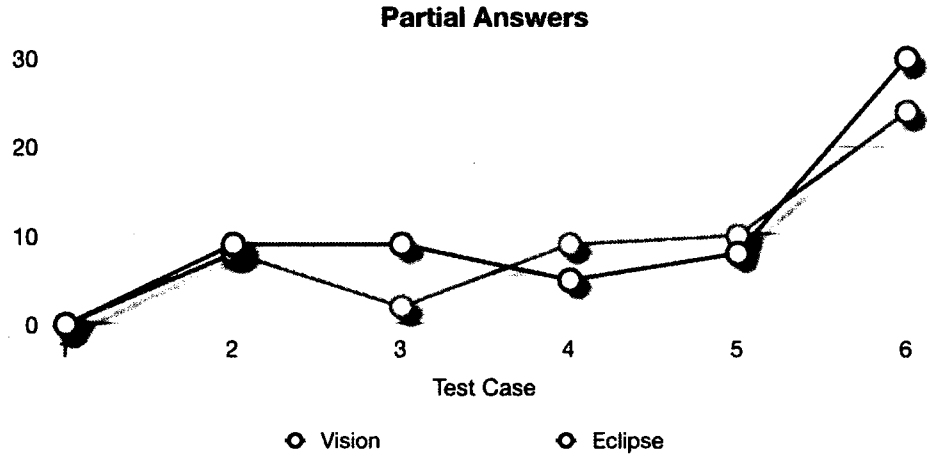

Figure 5.12: **Partial Answers**

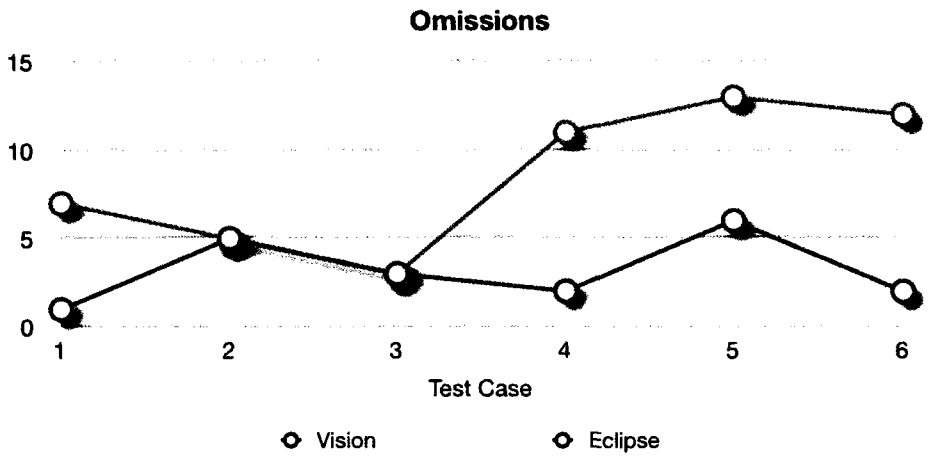

Figure 5.13: **Omissions**

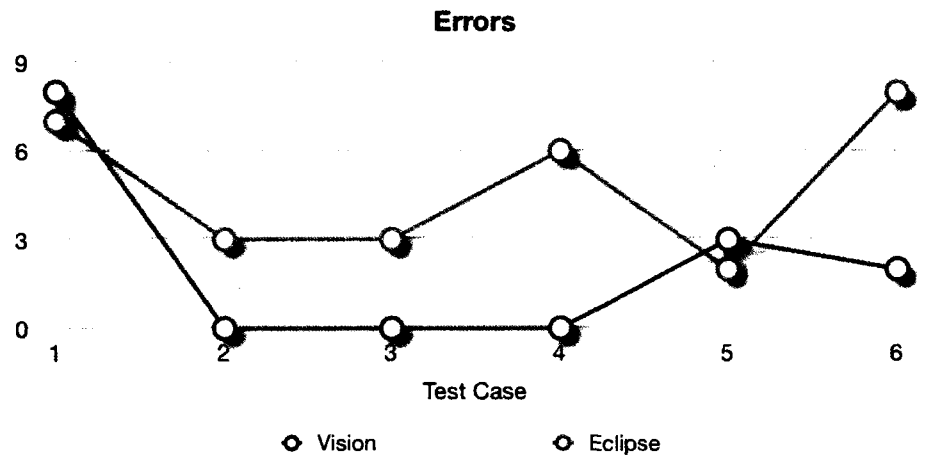

Figure 5.14: **Errors**

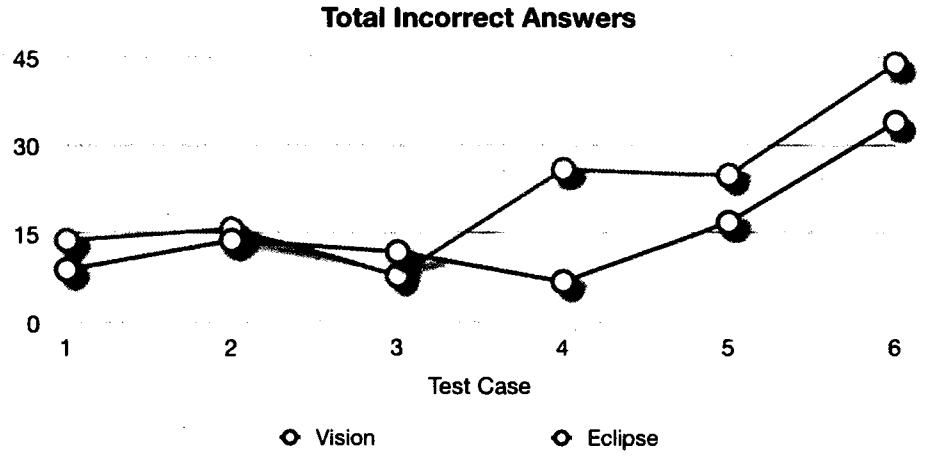

Figure 5.15: **Total Incorrect Answers**

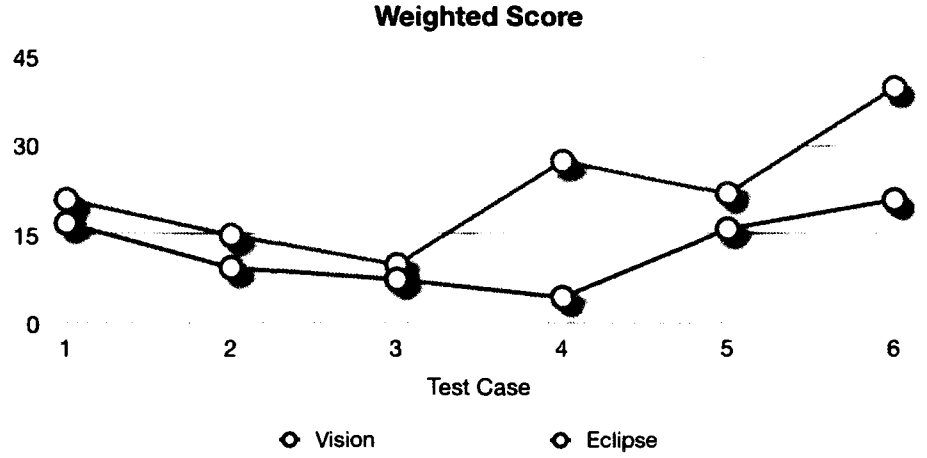

Figure 5.16: **Weighted Score**

5.4 Experiment Summary

Figure 5.17 consolidates all time measurements on a single chart where we can see that the proposed tool performed better than the reference tool for most tasks, with an average speed-up of 60% (Figure 5.18).

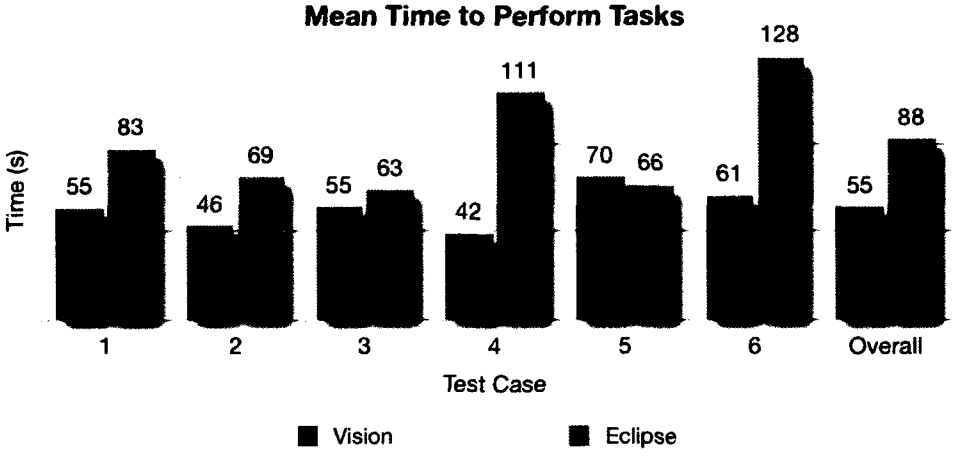

Figure 5.17: **Mean Time to Perform Tasks**

The unbalance between Groups 1 and 2 can be easily seen in comparisons 3 (Figure 5.8) and 5 (Figure 5.10), where the fastest group using the reference tool performed almost as fast or slightly better than the slowest group using the proposed tool.

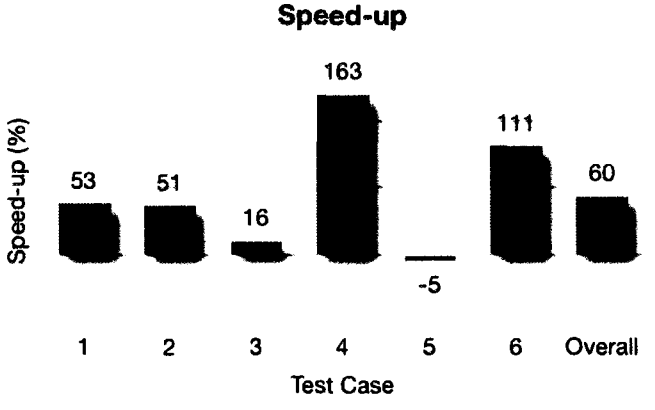

Figure 5.18: **Speed-up**

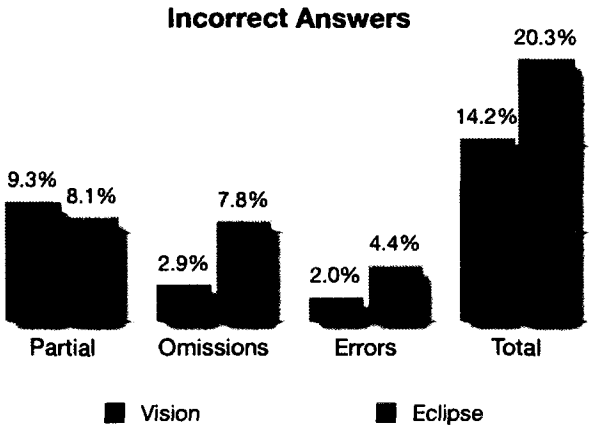

Figure 5.19: **Incorrect Answers:** As a percentage of all answers.

Figure 5.19 shows that, generally, the proposed tool also performed better than the reference tool regarding number of incorrect answers, with an average weighted score improvement[5] of almost 80% (Figure 5.20).

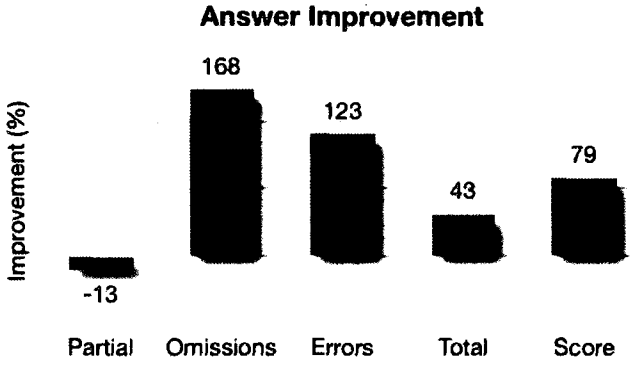

Figure 5.20: **Answer Improvement**

At first it may seem, though, that the proposed tool had worse partial answer results than the reference tool. According to figure 5.12, this is observed mainly in comparisons 3 and 6. However, looking at the charts in figures 5.13 and 5.14 we can clearly see that, for those same comparisons, the increase in number of partial answers is accompanied by a significant decrease in the number of omissions and errors. In other words, some incorrect answers might have migrated to more trivial levels which, in itself, is a satisfactory improvement.

[5]Defined as: $Eclipse/Vision - 100\%$

5.5 Preference Questionnaire

Finally, we look at the subjective experimental results and analyze the participants answers to the preference questionnaire (Appendix J).

First we asked participants which of the tools was easier to learn, easier to use, more efficient, and more intuitive (Figure 5.21)[6]. Most participants considered the proposed tool more or much more easy to learn, easy to use, efficient, and intuitive, while just a few participants said both tools were about equally easy to learn and intuitive.

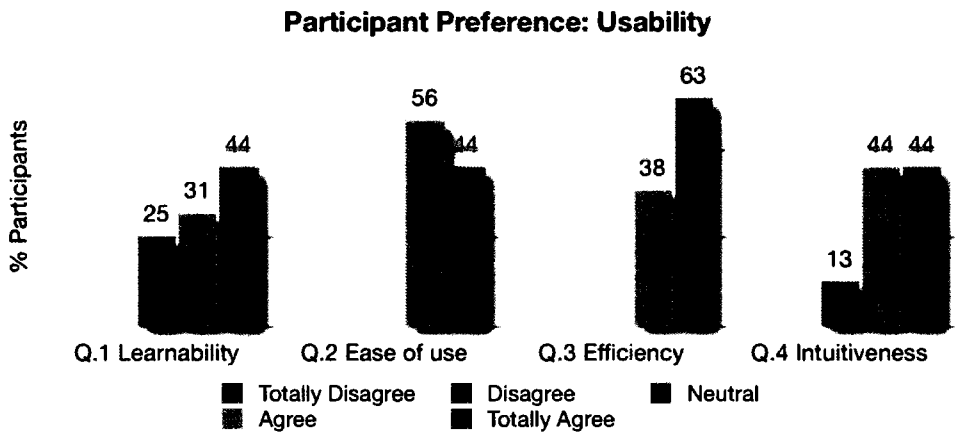

Figure 5.21: **Usability Criteria:** Is the proposed tool better regarding ...?

It is interesting to observe that the most noticeable tendency towards the proposed tool can be observed in the efficiency criterion, corroborating our empirical observations.

The second set of questions (Figure 5.22) asked participants how well they liked the proposed features: single-pane interface, highlighting granularity, difference classification, and modification-displaying artifacts. Again, most participants believed the proposed features represent a significant improvement over conventional file comparison tools. Difference classification was, undoubtedly, the feature that gathered the most positive remarks.

The next question (Q.9) asked which of the artifacts, tooltips and hot keys, if any, was the most useful. As can be seen in Figure 5.23, there was no clear preference towards any alternative, with most participants preferring to use both. This is a fairly reasonable result: the artifacts were designed to be complementary rather than mutually exclusive.

Finally, the last question (Q.11)[7] asked participants which tool they would choose if given

[6] $Q.x$ refers to the question number in Appendix J.

[7] Q.10 was annulled.

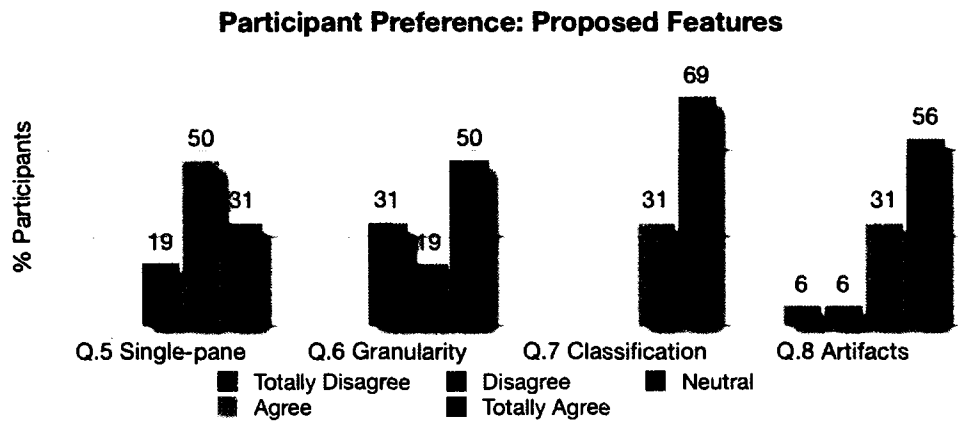

Figure 5.22: **Proposed Features:** Is the .... feature an improvement?

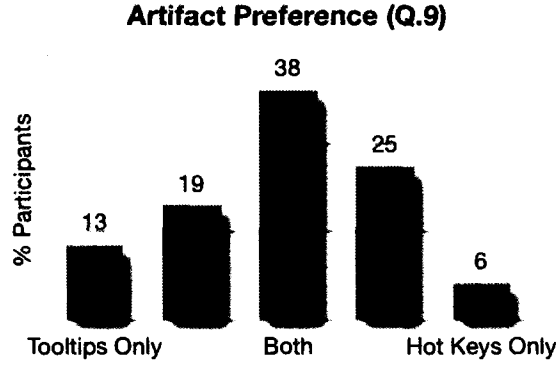

Figure 5.23: **Modification Visualization Preference**

the option. 63% of the participants said they would use the proposed tool mostly, while 38% answered they would use the proposed tool only.

The null hypothesis that participants did not favor one tool or artifact over the other was rejected at the 99.9% confidence level for all questions in the preference questionnaire, except Q.9 (Table D.4) — confirming both conclusions that the proposed tool was preferred to the reference tool and that participants would rather use both artifacts concurrently.

5.6 Chapter Summary

In this chapter we described the usability experiment methodology and setup, and reviewed the data collected through observation and questionnaires. Generally, the proposed tool performed better than the reference tool, improving both performance and answer quality. The

experimental evidence is strongly supported by participant impressions after the experiment, and hypothesis testing showed most results to be statistically significant.

In the next chapter we continue our analysis, looking at the most common problems observed during the experiment.

Chapter 6

Lessons from the Usability Study

During the usability study, we were able to obtain more detailed information than just time measurements or subjective answers. In this chapter we closely investigate those observations which could not be recorded on spreadsheets or questionnaires, looking at general usability problems related to both tools.

6.1 General Remarks

The usability experiment contained, in total, 82 differences participants were supposed to report (Appendix B). Of those, 31 were correctly described by all participants (Appendix C) and, together with the set of 17 differences that had at most one incorrect answer, can be considered trivial. The number of incorrect answers, 226, represents 17% of the total number of answers.

Considered in isolation, the proposed tool had 18 additional trivial questions and a total of 93 (14%) incorrect answers, while the reference tool had 5 additional trivial questions and 133 (20%) incorrect answers (Figure 5.19, on page 59).

In the following sections, we discuss the most commonly observed problems that were responsible for the majority of the incorrect answers.

6.2 The Reference Tool

Even though it may not have been the concern of this study, we would like to start discussing some usability problems found on the reference tool. This section, by its very nature, is going to be brief.

6.2.1 Automatic Scroll to First Difference

Paradoxically, the first usability problem we could observed is actually a feature aimed at *improving* usability: The reference tool, when opening a new comparison, automatically scrolls to the first difference on a file.

Although at first very convenient, in practice this feature seems to confuse the participants more than it can help them. This was clearly evident in Test Case 2 (Sections A.3 and A.4), where the first difference does not occur before lines 61–62. As far as we could observe, most, if not all, participants scrolled the screen back to the first line before proceeding.

6.2.2 Pair Matching

One of the challenges of implementing a two-pane file comparison interface is the creation of a visual connection between the documents to represent the conceptual relationship of a change. The reference tool goes to great lengths to maintain the link between visual and conceptual models, drawing lines and boxes around the text (Section 3.2, Principle 9). What we could observe during the experiment, though, was that this approach did not scale well for small, close changes, particularly those involving line deletions.

See, for instance, Figure 6.1 below, taken from Test Case 1 (Sections A.1 and A.2), differences 5–6 (Appendix B, page 123). It can be difficult to establish what the first line is connecting to. A few participants got confused with the number of lines crossing the screen, associating the large block on the right — which was deleted — with the second line of text on the left — which, actually, was not even changed.

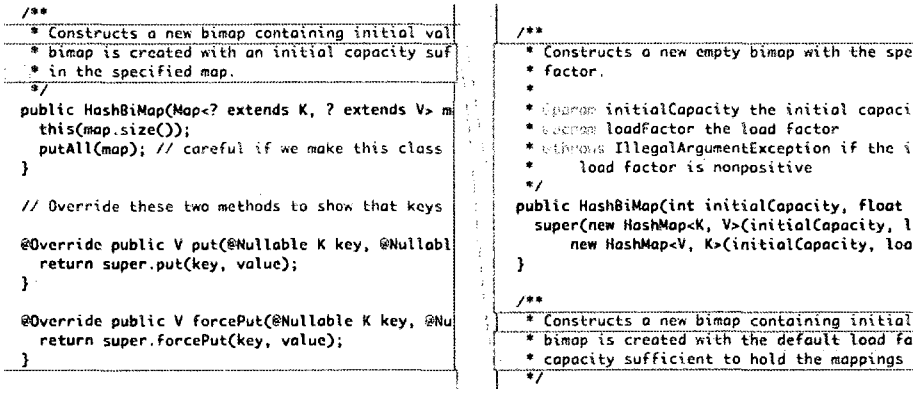

Figure 6.1: **Pair Matching**

6.2.3 Differences on the Far Right

As predicted, the two-pane interface would inevitably lead to horizontal scrolling. Surprisingly, though, the problem we observed most frequently was not horizontal scrolling itself; ironically, it was *not* scrolling horizontally. Some participants would not scroll the screen horizontally even when a line visibly continued off of the limits of the screen, inexcusably missing an otherwise fairly trivial difference.

Consider, for instance, Test Case 5 (Sections A.9 and A.10), difference 15 (page 126), which had one of the worst scores of all differences, second only to reordered lines (section 6.3.5). Of eight participants, only two were able to spot this change using the reference tool. In contrast, everyone using the proposed tool was able to correctly identify that difference.

6.2.4 Vertical Alignment

For most cases, the reference tool does a good job of keeping both sides of the screen vertically synchronized. However, as can be easily observed in Figure 6.1, it is not possible to maintain vertical alignment across the whole screen.

The problem is more evident with large blocks of line insertions or deletions. Usually, only the very top differences will be aligned; the bottom of the screen often gets itself badly misaligned. The reference tool will usually correct the alignment as the user scrolls down the screen, but only if she does it line by line. Users who prefer to scroll the screen a page at a time would still frequently experience this problem.

6.2.5 Vertical Scrolling

Since, on the reference tool, there are two independent vertical scroll bars, it is not unusual for one of the bars to reach the end of its course before the other. The mouse wheel would, then, have not effect on the second side, causing some confusion amongst participants. This was a minor issue, though, and had no observable negative impact on the answers.

6.2.6 Dangling Text and Line Reordering

Dangling text and line reordering were problems that affected both tools equally bad. To avoid unnecessary repetition, we postpone the discussion to sections 6.3.2 and 6.3.5.

6.3 The Proposed Tool

During the usability study, despite the performance and accuracy improvements demonstrated by the experimental results, we could observe some areas where the proposed tool showed a few limitations. In this section we discuss the usability problems we could observe, while also proposing refinements and eventual solutions.

6.3.1 Short Differences

The proposed tool, by design, highlights differences using token granularity, in contrast to whole lines or blocks. While this design decision helped reduce interface clutter, leading to improved clarity and readability, it also introduced a minor problem: short differences, usually single-character tokens, can be difficult to spot. This behavior was observed both during the comparisons and spontaneously reported by participants at the end of the experiment, yet no participant failed to report such differences.

Proposed Solution

Fortunately, this problem is easily solved. The difference navigation feature described in Section 2.1 provides a simple, yet elegant, solution. For inspiration, the reference tool offers us two complimentary mechanisms (Figure 6.2).

The first, represented by the buttons on the top right corner, are *next* and *previous* buttons to easily navigate amongst differences. The second, represented by the white squares on the far right, is a vertical ruler with marks for changes, a snapshot representation of differences

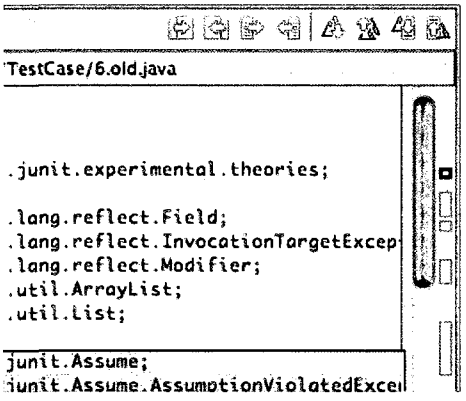

Figure 6.2: **Short Differences**

throughout the entire file. Clicking on a square scrolls directly to that particular difference.

Systematically using any or both of those navigational aids prevents even the smallest changes from getting missed.

6.3.2 Dangling Text

The dangling text problem is an ambiguous, yet strictly correct, arrangement which affected both the proposed and the reference tools.

```
@Override
protected void validateZeroArgConstructor(List<Throwable> errors) {
    // constructor can have args
}

@Override
protected void validateTestMethods(List<Throwable> errors) {
    for (FrameworkMethod each : computeTestMethods())
        each.validatePublicVoid(false, errors);
}

@Override
protected List<FrameworkMethod> computeTestMethods() {
    List<FrameworkMethod> testMethods= super.computeTestMethods();
    List<FrameworkMethod> theoryMethods= getTestClass().getAnnotatedMethods(Theory.class);
    testMethods.removeAll(theoryMethods);
    testMethods.addAll(theoryMethods);
    return testMethods;
}
```

Figure 6.3: **Dangling Text**

Take, for instance, Figure 6.3 from Test Case 6 (Sections A.11 and A.12), differences 12 and 13 (page 127). When asked, many participants said the third `@Override` annotation was added to the `computeTestMethods` method. A more careful inspection, though, reveals the method was already annotated: notice the first `@Override` annotation is *not* highlighted. The actual insertions were the `validateZeroArgConstructor` and `validateTestMethods` methods, both with their respective `@Override` annotations.

Proposed Solution

Dangling text is a non-deterministic problem. Given the original file `ab` and its modification `acab`, two sets of changes are possible: `ACab` and `aCAb`. Without additional clues, both are equally probable and correct, despite one being more intuitive than the other. Section 7.3, *Future Work*, discusses a possible strategy to mitigate this kind of problem.

6.3.3 Token Granularity

Even though it can be said both tools were affected by the token granularity problem, this problem only had a negative impact on participants using the proposed tool and, even then, under a single particular circumstance.

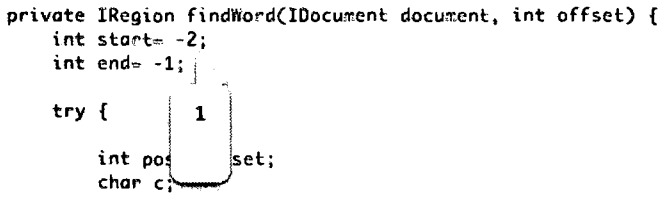

Figure 6.4: **Token Granularity**

Take, for instance, Test Case 3 (Sections A.5 and A.6), difference 9 (page 124). As can be seen in Figure 6.4, the number "2" is highlighted in orange, while the tooltip shows "1". Some participants using the proposed tool with tooltips answered the number "1" was changed to "-2", despite the "−" sign not being highlighted.

What surprised us, though, was that participants using the hot keys — or the reference tool, for that matter — were not affected by the glitch. All participants instinctively gave the correct answer, since the number "1" would always follow the "−" sign on the screen. For those who opted for the tooltips, though, the number "1" was displayed isolated from its context, and not all were able to identify the correct answer.

Proposed Solution

For the particular example discussed here, the lexical analyzer used by both tools complies literally with the Java language specification [25], which states that a decimal numeral — or *integer literal* — always represents a *positive* integer; the minus sign is the arithmetic negation unary operator [2], and is not part of the integer literal.

For our purposes, strict compliance with the language specification is not a requirement, and a more "humane" parser could have been used.

6.3.4 Difference Classification Heuristics

One of the main problems faced while implementing the proposed tool was to deduce from two pieces of text a set of changes, extrapolating from the differences a semantic meaning.

The heuristics we implemented (Section 4.3.2) is simple and easy to understand, yet generally yield satisfactory results. However, during the usability experiment we could observe that some participants were inclined to interpret the tool output much too literally, even when it represents unreasonable results.

```
/*
 * @see org.eclipse.jface.text.ITextHover#getHoverInfo(org.eclipse.jface.text.ITextViewer, org.eclipse
 * @deprecated As of 3.4, replaced by {@link ITextHoverExtension2#getHoverInfo2(ITextViewer, IRegion)}
 */
public String getHoverInfo(ITextViewer textViewer, IRegion hoverRegion) {
```

Figure 6.5: **Difference Classification Heuristics**

```
/**
 * {@inheritDoc}
 *
 * @deprecated As of 3.4, replaced by {@link ITextHoverExtension2#getHoverInfo2(ITextViewer, IRegion)}
 */
public String getHoverInfo(ITextViewer textViewer, IRegion hoverRegion) {
```

Figure 6.6: **Difference Classification Heuristics:** Hot key pressed.

Take, for instance, Figures 6.5 and 6.6, from Test Case 3, difference 4 (page 124): a one-line ordinary comment was changed to a two-line (not counting the blank line) Javadoc comment. While it would be more appropriate to highlight the entire block as *modified*, the tool interpreted the first line as modified and the second line as *added*. Although strictly correct, this interpretation is mostly non-intuitive, and generated some confusion amongst participants.

6.3.5 Line Reordering

The last problem we analyze, line reordering, was the one which had the worse rate of errors, affecting both tools and all participants.

Line reordering happens when a entire line changes its relative position in the text, accompanied or not by modifications. For the tests cases used, this was most evident in the import statements in Test Case 5, difference 3 (page 125), and Test Case 6, differences 2–6 (page 126).

None of the tools offers any special provisions to handle such situations. In the best case, a moved line (changed or not) will be represented as pairs of lines additions and deletions. In the worst case, changes get mixed up, a scenario which can be very demanding to understand.

Even though both tools were greatly affected by this problem, it can be said, through observation, that the proposed tool performed worse than the reference tool.

Take for instance Comparison 6, differences 2–6. The proposed tool had, for these five differences, a weighted score of 14.5, while the reference tool scored 15.5. Despite the numbers, which may suggest the proposed tool performed slightly better, truth is that four of the six participants with the lowest overall scores used the proposed tool to perform this comparison. Those who managed to give the few correct answers using the proposed tool went to great strengths to understand the differences, consuming large amounts of time and effort. The only person to get all answers in this comparison right was using the reference tool, as were the three persons to correctly answer Test Case 5, difference 3.

Proposed Solution

Line reordering is a very challenging problem. Firstly, moved lines need to be detected, a non-trivial problem since the lines could also have been modified while moved. Secondly, an interface metaphor has to be envisaged to represent this kind of change.

Looking closely at the reference tool, though, we see that it is no better than the proposed tool for handling moved lines. The only reason it performed better in the usability experiment was that the two-pane interface provides a kind of *fall-back* mechanism. When changes cannot be easily interpreted using the tool aids, users can revert to a manual approach, reading each version of the text and deducing the changes by themselves. In this case, the reference tool is no better than, say, opening two text editors and aligning them side by side. Using the proposed tool it is much more difficult to mentally reconstruct the two versions of the text because they were merged into a single view.

The solution we propose, represented in Figure 6.7, is a second, line-oriented visualization mode which allows the user to revert the text back to its original representation, without departing from the single-pane metaphor, and while still providing some visual aids to help users understand the differences.

In this special mode, only one of the versions of the text is presented on the screen at a time; the hot key can still be pressed to switch between the original and the modified version. Differences are computed and highlighted using line granularity. When displaying the modified version only added and modified lines are shown. Conversely, the original version shows only deleted and modified lines. Blank lines are inserted for vertical alignment. Tooltips are disabled

```
import org.junit.Assert;

import org.junit.experimental.theories.PotentialAssignment.CouldNotGenerateValueException;
import org.junit.experimental.theories.internal.Assignments;
import org.junit.experimental.theories.internal.ParameterizedAssertionError;
import org.junit.internal.AssumptionViolatedException;
import org.junit.runners.BlockJUnit4ClassRunner;
import org.junit.runners.model.FrameworkMethod;
import org.junit.runners.model.InitializationError;
import org.junit.runners.model.Statement;
```

Figure 6.7: **Line Reordering**

in this mode since, by assumption, lines do not match one another.

This solution was implemented in our prototype after the usability study, therefore its effectiveness could not be attested. Nevertheless, we believe this approach should at least match the reference tool when dealing with moved lines. While it may not be a complete, definitive solution to the problem, we consider it to be a good compromise given the current constraints.

6.4 Miscellaneous Observations

This section briefly discusses some minor issues observed during the usability experiment which are not subject of further consideration.

Original and Modified Order

At least three people using the reference tool got confused about which side on the screen represented which of the file versions, reversing their answers. To avoid getting all subsequent answers wrong, therefore invalidating the whole test case, participants were reminded after the second such mistake.

Conversely, a single participant had a similar problem with the proposed tool when using the tooltip for the first time, but she soon realized her own mistake and was able to correct herself.

Color Highlighting

By design, it was decided to make use of the platform syntax coloring facilities in addition to our own difference highlighting. At first, this would pose no problems since there was no apparent conflict.

Nevertheless, at least two people mistook the *foreground* green **comment** syntax highlighting for the *background* green **addition** difference highlighting. This only happened on the very first comparison using the proposed tool which, in both cases, was Test Case 2.

Java 5 Features

Even though not a usability problem per se, we could observe a certain level of confusion amongst participants regarding features introduced in version 5 of the Java language [2]. This was most evident in Test Case 4 (Sections A.7 and A.8), where most changes involved updating the code to use *generics*, *enhanced for loops*, and *annotations*, but it could also be observed in Test Case 5 and, to a lesser extent, Test Case 6.

In some cases, participants were not able to correctly describe, using Java terms, a change; most would point to the screen and say "*This* was changed to *that*", which was considered a valid answer as long as the intent were correct.

Enhanced *for* loops showed themselves to be more challenging: some participants tried to match tokens (*i.e*, "*int i = 0* was changed to *Class<?>*"), not realizing the missing semicolons. Those were considered only as partial answers.

6.5 Chapter Summary

Most differences in the usability experiment — 80% for the proposed tool, 65% for the reference tool — had at most one incorrect answer, and could, therefore, be considered trivial.

In this chapter we discussed the general usability problems that affected most of the incorrect answers given by participants. While some problems demand improvements in the underlying technologies used to implement the tools and others are topics of further research, practical solutions were proposed or implemented.

In the next chapter, we conclude this work, summarizing our contributions and suggesting directions for future work.

Chapter 7

Conclusion

7.1 Main Contributions

Be it at quantitative, objective measurements or at qualitative, subjective preference answers, the proposed interface developed in this work showed itself to be a more adequate metaphor to performing file comparison tasks than the traditional, two-pane interface implemented by the reference tool (Sections 5.4 and 5.5).

Implementing a single-pane interface satisfactorily is not a trivial venture. Take, for instance, WinDiff (Section 2.2.8). Microsoft's file comparison tool closely resembles a one-pane interface, yet it was arguably one of the least powerful and user-friendly tools analyzed. Even looking at the more advanced "*Track Changes*" feature offered by most word processors, it is easy to see they lack most refinements offered by the proposed tool.

Classifying differences as *additions*, *deletions*, and *modifications* (Sections 3.3.2 and 4.3.2), one of the critical elements of the interface, was the most well-received feature, with the strongest shift on participant preference according to the questionnaires. Interpreting consecutive pairs of additions–deletions as modifications is an enhancement not usually explored by most file comparison tools. Although most tools highlight tokens inside a block of changed lines, we introduced the idea of using finer levels of granularity to classify changes (Section 3.3.4).

Displaying modifications was particularly challenging, for this clearly conflicts with the essence of using a single stream of text. To overcome the problem, two independent, complementary mechanisms, tooltips and hot keys, were developed (Section 3.3.3). Participants showed no particular preference for one artifact over the other; most of them would rather use both in conjunction.

Legibility was also carefully considered and, after numerous iterations, we came with what we believe is the best approach to handle white space, minimizing interface clutter and improving readability.

7.2 Threats to Validity

During all phases of experiment planning and execution, great care was taken to ensure fairness and minimize bias. When choices could be made, decisions tended to favor the reference tool, as in excluding Participant 16's time data (Section 5.3.1) or increasing the comparison area to its maximum (Section 5.1.3).

By the very nature of the experiment, test cases had to have reasonable length and levels of complexity. Otherwise, participants would quickly get tired or bored, leading to answer degradation and compromising the experiment outcome.

There are no reasons to believe the tool would underperform under lengthy comparisons — arguably, it probably would outperform most other comparison tools. However, output quality may degrade with some complex sets of changes, specially those involving line reordering (Section 6.3.5). Although there was a test case (Sections A.11 and A.12) which predominantly exemplified this issue, participant perception could have been shifted had we used more and most extensive such samples.

Most participants had previous experience with the reference tool, and may already be weary of some of its shortcomings. The proposed tool, on the other hand, had the novelty factor in its favor, and its colorful interface is sure to cause a favorable first impression. A study with participants new to both tools could probably have had a different outcome. However, given the reference tool's popularity amongst Java developers, it would have been difficult to recruit such individuals, who probably would have been familiar with other, similar tools, anyway.

As a direct consequence, participants were aware of the proposed tool — a blind experiment was not attainable. Together with the fact the experiment was performed by the research himself, this might have had some influence on the preference questionnaire answers, despite it being anonymous.

Although all comparisons were performed using each tool the same number of times, and all participants used each tool for half their comparisons, no single participant was able to see the same comparison on both tools. Arguably, this could be the best way to compare the tools:

a participant would look at both outputs and decide which one gave the best results. However, there is no practical way of performing such experiment without spoiling answers and time measurements; experiment results would be limited to subjective answers, only.

Some problems were identified on the usability experiment itself. Answer classification (Section 5.1.2) is an inherently subjective matter, reliant on examiner discernment. Nevertheless, the proposed tool had a significantly lower total number of incorrect answers than the reference tool, regardless of classification.

While planning the usability experiment, it was thought that measuring only the time participants spent understanding the changes would give a more accurate measurement than asking them to concurrently explain the changes, therefore the two-step procedure described in Section 5.1. However, during the experiment, it was observed that participants would spend more time trying to explain the changes than understanding them in the first place.

Moreover, participants using the proposed tool would normally spend much less time explaining changes than those using the reference tool, increasing even more the performance gap between the tools. There was an effort to try to recover the data from screen recordings, but unfortunately this method showed itself to be too inaccurate to be useful. Sadly, this valuable source of information was lost.

Finally, when pooling such a relatively large group, randomly partitioned into two subgroups, it would be expected to have an even distribution of skills. Unfortunately, one group performed the experiment 35% faster and with 40% better answer quality than the second one. This favored the proposed tool in comparisons 2, 4, and 6, although also favoring the reference tool in comparisons 1, 3, and 5. The pattern can be clearly observed on the charts starting at page 51.

7.3 Future Work

Below we list some areas where the interface could be improved through further research:

Accessibility Issues: Given the interface's reliance on colors, experiments have to be performed to attest color blindness accessibility. One special highlighting strategy (Figure 4.2, on page 39) was implemented using only underlines and strikeouts, and no color. However, it was meant mainly for black and white printing, and no evaluations with color blind people were performed.

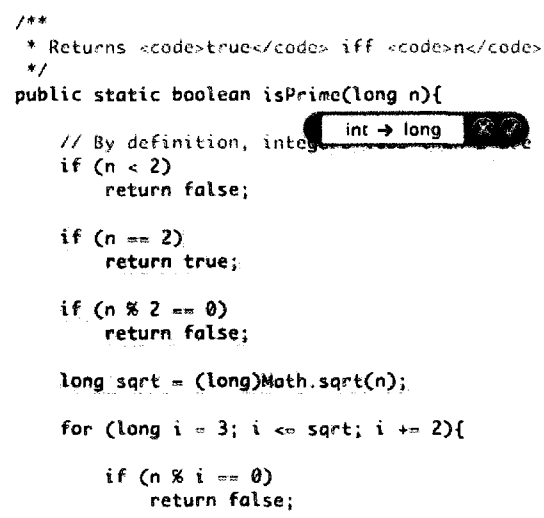

Figure 7.1: **Merging Mock-up**

Syntax Awareness: The interface computes differences first line by line and then, for lines that differ, using a lexical parser to extract tokens. The lexical parser does not need to comply with the Java Language Specification [25], and could be tuned for more intelligible results (Section 6.3.3).

However, results could be greatly improved if this two-step parsing process was replaced by a full syntactic parser. Differences could be computed using higher-level language constructs, such as class members, blocks, and statements. This could solve — or, at least, minimize — the problems discussed in Section 6.3.2.

Merging: One missing feature of the proposed tool is the ability to perform merging. Merging could be easily added to the interface using a simple abstraction: consider each change to be an edit made to the file. Hovering the mouse over a change — or selecting multiple changes at once — would show a pop-up window with buttons to accept or reject the edit. Accepting would just confirm the change, with no practical effect, while rejecting would revert the change back to its original text. A mock-up of such interface is depicted in Figure 7.1.

Three-way Merging: The other important missing feature is the ability to perform three-way merges. This advanced feature is used mostly to solve conflicts caused by concurrent source code modifications.

Three-way merging constitute a more intricate problem. Considering there can be additions, deletions, and modifications from two different sources, a total of 15 combinations, plus no change, is possible and may have to be represented. Some combinations can be particularly awkward to detect and represent, and the techniques to classify and display changes described in this work would have to be revised.

Reordered Lines: Reordered lines was responsible for most incorrect answers on the usability experiment. Further research on how to detect and effectively represent those modifications is needed. The feature developed in Section 6.3.5 should alleviate the problem, but its effectiveness could not be attested on the usability experiment.

Improved Heuristics: The tool could benefit from further research on difference classification heuristics (Section 6.3.4).

Miscellaneous Improvements: For the prototype to be released as a production-quality tool, some miscellaneous improvements — mostly regarding implementation issues — have to be addressed. Those include: support for difference navigation mechanisms (Section 6.3.1), improved lexical parser (Section 6.3.3), user configurable preferences (for instance, choosing the highlighting colors), and removing the dependencies on the platform *internal* packages.

7.4 Final Remarks

File comparison tools are ubiquitous tools, present in most IDEs and available as stand-alone tools for most platforms. They are a perfect complement to Source Code Management systems, themselves a fundamental piece in any software development project.

Our survey, with some of the most popular tools on the market, showed that comparison tools lack feature diversity, for the most part sharing the same set of interface concepts and metaphors. Academic research and innovation in this field has been mostly sparse.

File comparison is less about seeing which lines differ between two files than it is about understanding changes made to a file. The interface proposed is based on simple, intuitive principles, borrowing ideas from features found on tools like word processors.

The proposed interface excelled on most tests and in every usability criteria. Time measurements and answer quality were both greatly improved, numbers which were confirmed by

participant impressions as stated on preference questionnaires and by statistical hypothesis testing. We are confident the interface represents a significant improvement over the typical file comparison tool.

Hopefully, no one will have the impression of playing *Spot the Difference* the next time they compare files.

References

[1] Alfred Aho, John Hopcroft, and Jeffrey Ullman. *Data Structures and Algorithms.* Addison-Wesley Publishing, 1982.

[2] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language.* The Java Series. Addison-Wesley Professional, 4th edition, 2005.

[3] David Atkins. Version sensitive editing: Change history as a programming tool. *System Configuration Management*, pages 146–157, 1998.

[4] Joshua Bloch. *Effective Java.* The Java Series. Addison-Wesley Professional, 2nd edition, 2008.

[5] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. Ldiff: An enhanced line differencing tool. *IEEE 31st International Conference on Software Engineering*, pages 595–598, 2009.

[6] Stuart Card, Allen Newell, and Thomas Moran. *The Psychology of Human-Computer Interaction.* L. Erlbaum Associates Inc., 1983.

[7] Sudarshan Chawathe, Serge Abiteboul, and Jennifer Widom. Representing and querying changes in semistructured data. *Proceedings of the Fourteenth International Conference on Data Engineering*, pages 4–13, 1998.

[8] Eric Clayberg and Dan Rubel. *Eclipse: Building Commercial-Quality Plug-ins.* Addison-Wesley Professional, 2nd edition, 2006.

[9] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[10] Alan Dix, Janet Finley, Gregory Abowd, and Russell Beale. *Human-Computer Interaction.* Prentice-Hall, Inc., 2nd edition, 1998.

[11] The Eclipse Project. DefaultTextHover.java. `http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.jface.text/src/org/eclipse/jface/text/`. Revisions 1.1 and 1.10.

[12] The Eclipse Project. Eclipse IDE for Java Developers. `http://eclipse.org/downloads/`. Versions 3.4.1 and 3.4.2. Retrieved on 2009-04-25.

[13] The Eclipse Project. Eclipse Java Development Tools Plug-in Developer Guide. `http://help.eclipse.org/`. Retrieved on 2009-04-12.

[14] The Eclipse Project. Eclipse Platform Plug-in Developer Guide. `http://help.eclipse.org/`. Retrieved on 2009-04-12.

[15] The Eclipse Project. Eclipse Plug-in Development Environment Guide. `http://help.eclipse.org/`. Retrieved on 2009-04-12.

[16] FileMerge. `http://developer.apple.com/tools/xcode/`. Version 2.4. Retrieved on 2009-04-25.

[17] Fisher's method. `http://en.wikipedia.org/wiki/Fisher%27s_method`. Retrieved on 2009-06-11.

[18] Forrester Research, Inc. IDE usage trends, 2008.

[19] Free Software Foundation. Diffutils man page, 2002.

[20] Erich Gamma and Kent Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins.* Addison Wesley Longman Publishing Co., Inc., 2003.

[21] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman Publishing Co., Inc., 1995.

[22] Project GlassFish. sendfile.java. `https://glassfish.dev.java.net/source/browse/glassfish/mail/src/java/demo/`. Revisions 1.1 and 1.3.

[23] GNU diffutils. `http://gnu.org/software/diffutils/`. Version 2.8.1. Retrieved on 2009-04-25.

[24] Google Collections Library. HashBiMap.java. `http://google-collections.googlecode.com/svn/trunk/src/com/google/common/collect/`. Revisions 16 and 57.

[25] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* The Java Series. Addison-Wesley Professional, 3rd edition, 2005.

[26] HTML Diff. `http://infolab.stanford.edu/c3/demos/htmldiff/`. Retrieved on 2009-04-08.

[27] James Hunt and Malcolm McIlroy. An algorithm for differential file comparison. *Computing Science Technical Report*, (41), July 1976.

[28] IntelliJ IDEA. `http://jetbrains.com/idea/`. Version 8.1. Retrieved on 2009-04-25.

[29] Juliele Jacko. *Human-Computer Interaction Handbook: Fundamentals, Evolving Technologies, and Emerging Applications.* L. Erlbaum Associates Inc., 2002.

[30] JUnit Testing Framework. Theories.java. `http://junit.cvs.sourceforge.net/viewvc/junit/junit/src/main/java/org/junit/experimental/theories/`. Revisions 1.8 and 1.25.

[31] The Jython Project. BytecodeLoader.java. `https://jython.svn.sourceforge.net/svnroot/jython/trunk/jython/src/org/python/core/`. Revisions 4055 and 5479.

[32] Miryung Kim and David Notkin. Discovering and representing systematic code changes. *IEEE 31st International Conference on Software Engineering*, pages 309–319, 2009.

[33] Kompare. `http://caffeinated.me.uk/kompare/`. Version 3.4. Retrieved on 2009-04-25.

[34] Meir Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, September 1980.

[35] Meld. `http://meld.sourceforge.net/`. Version 1.2.1. Retrieved on 2009-04-25.

[36] Tom Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.

[37] Microsoft Corporation. Overview: WinDiff. `http://msdn.microsoft.com/en-us/library/aa242739(VS.60).aspx`. Version 5.1. Retrieved on 2008-08-04.

[38] Microsoft Corporation. Windiff colors. `http://msdn.microsoft.com/en-us/library/aa266120(VS.60).aspx`. Retrieved on 2008-08-04.

[39] Webb Miller and Eugene Myers. A file comparison program. *Software Practice and Experience*, 15(11):1025–1040, 1985.

[40] Eugene Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[41] NetBeans. `http://netbeans.org/`. Version 6.5. Retrieved on 2009-04-25.

[42] Jakob Nielsen. Scrolling and scrollbars. `http://useit.com/alertbox/20050711.html`. Retrieved on 2008-07-09.

[43] Jakob Nielsen. *Usability Engineering.* Academic Press Professional, 1993.

[44] Donald Norman. *The Invisible Computer.* MIT Press, 1998.

[45] Donald Norman. *The Design of Everyday Things.* Basic Books, 2002. Previously published as The Psychology of Everyday Things.

[46] Donald Norman. *Emotional Design.* Basic Books, 2005.

[47] Dirk Ohst, Michael Welle, and Udo Kelter. Difference tools for analysis and design documents. *19th International Conference on Software Maintenance*, pages 13–22, 2003.

[48] Andy Oram and Greg Wilson, editors. *Beautiful Code.* O'Reilly and Associates, 2007.

[49] Robert Sedgewick and Michael Schidlowsky. *Algorithms in Java, Parts 1–4: Fundamentals, Data Structures, Sorting, Searching.* Addison-Wesley Longman Publishing Co., Inc., 3rd edition, 1998.

[50] Jochen Seemann and Jürgen von Gudenberg. Visualization of differences between versions of object-oriented software. *2nd Euromicro Conference on Software Maintenance and Reengineering*, pages 201–204, 1998.

[51] Spring Framework. DefaultImageDatabase.java. `http://springframework.cvs.sourceforge.net/viewvc/springframework/spring/samples/imagedb/src/org/springframework/samples/imagedb/`. Revisions 1.11 and 1.16.

[52] Lucian Voinea, Alexandru Telea, and Jarke van Wijk. CVSscan: visualization of code evolution. *Proceedings of the ACM 2005 Symposium on Software Visualization*, pages 47–56, 2005.

[53] Welch's t test. `http://en.wikipedia.org/wiki/Welch%27s_t_test`. Retrieved on 2009-06-11.

[54] Christopher Wickens, John Lee, Yili Liu, and Sallie Gordon Becker. *An Introduction to Human Factors Engineering.* Pearson Prentice Hall, 2nd edition, 2004.

[55] Laura Wingerd and Christopher Seiwald. *Beautiful Code*, chapter 32: *Code in Motion.* In Oram and Wilson [48], 2007.

[56] WinMerge. `http://winmerge.org/`. Version 2.12.2. Retrieved on 2009-04-25.

Appendices

Appendix A

Test Cases

In this chapter we reproduce the source files used as test cases in the usability experiment. The files come from many established, well-known open-source projects, giving us a broad variety of coding styles and changes:

- Test case 1 (Sections A.1 and A.2) comes from the Google Collections Library [24];
- Test case 2 (Sections A.3 and A.4) is from Sun's GlassFish Application Server [22]. For brevity's sake, the license header was removed.
- Test case 3 (Sections A.5 and A.6) comes from the Eclipse project [11];
- Test case 4 (Sections A.7 and A.8) is from Jython [31], a Java compiler and interpreter for the Python programming language;
- Test case 5 (Sections A.9 and A.10) comes from the Spring Framework [51];
- Test case 6 (Sections A.11 and A.12) is from JUnit [30], the testing framework.

The files were selected roughly at random to avoid bias. First, we looked for files from about 100 to 200 lines, then we went back into the file revision history until there were about seven to 30 individual changes, with varying degrees of complexity. While browsing the file history for changes, we used the reference tool only.

For the complete source listing, please refer to the electronic version, on-line at:

`http://www.site.uottawa.ca/~damyot/students/lanna/`

Appendix B

List of Differences

Below we list all differences participants were expected to report for each comparison task. Line numbers refer to the *new* version of a file, while line numbers in parentheses refer to the *old* version. Please note that this list is slightly subjective and susceptible to the examiner's interpretation.

B.1 Test Case 1

1. **Lines 21–23 (20–21)**: Added three import statements: `java.io.IOException`, `ObjectInputStream`, and `ObjectOutputStream`;
2. **Lines 29–30 (26)**: Added "*A {@code HashBiMap} and its inverse are both serializable.*" to comment;
3. **Lines 36 (32–33)**: Deleted "*and the default load factor (0.75)*" from comment;
4. **Lines 43 (40–41)**: Deleted "*and the default load factor (0.75)*" from comment;
5. **Lines 54–55 (53–66)**: Deleted the `HashBiMap(int initialCapacity, float loadFactor)` method and its comment;
6. **Lines 56–57 (68–69)**: Deleted "*the default load factor (0.75) and*" from comment;
7. **Lines 74–88 (84–85)**: Added the `writeObject(ObjectOutputStream stream)` and `readObject(ObjectInputStream stream)` methods;
8. **Lines 90 (84–85)**: Added the `serialVersionUID` field.

B.2 Test Case 2

1. **Lines 62 (61–62)**: Added `mbp2.attachFile(filename);`
2. **Lines 64–68 (61–62)**: Added multi-line comment;
3. **Lines 69–73 (62)**: Added an anonymous inner class extending `FileDataSource`;
4. **Lines 69, 74–76 (62–64)**: Commented three lines out of the code;
5. **Lines 89–96 (76–77)**: Added multi-line comment;
6. **Lines 107–108 (85–86)**: Added `catch (IOException ioex)`.

B.3 Test Case 3

1. **Line 2 (2)**: Modified copyright years;
2. **Lines 15–20 (14–17)**: Added three import statements: `Assert`, `IAnnotationModel`, and `ISourceViewerExtension2`;
3. **Lines 23–24 (20–22)**: Deleted multi-line comment;
4. **Lines 42–45 (41–42)**: Modified multi-line comment;
5. **Lines 48–50 (44–45)**: Added three lines;
6. **Lines 52 (46)**: Replaced `fSourceViewer.getAnnotationModel()` for `model`;
7. **Lines 56 (50)**: Replaced `fSourceViewer.getAnnotationModel()` for `model`;
8. **Lines 86–92 (79–80)**: Added method `getAnnotationModel(ISourceViewer viewer)`;
9. **Lines 95 (81)**: Changed -1 to -2;
10. **Lines 127 (113)**: Changed $>$ to $>=$.

B.4 Test Case 4

1. **Lines 4–6 (5–7)**: Deleted two import statements: `java.util.ArrayList` and `java.util.Vector`;
2. **Lines 7–8 (8–9)**: Added two import statements: `ClassReader` and `Generic`;

3. **Lines 25 (24)**: Added an unbounded wildcard type to `Class` (twice);
4. **Lines 27 (26)**: Changed the `for` loop for its enhanced-syntax version;
5. **Lines 29 (28)**: Replaced `referents[i]` with `referent`;
6. **Lines 48 (47)**: Added an unbounded wildcard type to `Class`;
7. **Lines 48 (47)**: Replaced `Vector<Class>` with `List<Class<?>>`;
8. **Lines 50 (49)**: Replaced `0` with `referents.size`;
9. **Lines 65 (64)**: Added an unbounded wildcard type to `Class`;
10. **Lines 65–66 (65)**: Deleted the `@SuppressWarnings` annotation;
11. **Line 76 (76)**: Changed `new ArrayList<ClassLoader>()` to `Generic.list()`;
12. **Lines 88 (87–88)**: Added the `@Override` annotation;
13. **Lines 90 (89)**: Added an unbounded wildcard type to `Class`;
14. **Lines 103 (102)**: Added an unbounded wildcard type to `Class`;
15. **Lines 104–115 (102–103)**: Added an `if` block;
16. **Lines 116 (103)**: Added an unbounded wildcard type to `Class`.

B.5 Test Case 5

1. **Line 12 (12)**: Changed `IncorrectResultSizeDataAccessException` import statement to `EmptyResultDataAccessException`;
2. **Lines 14 (14)**: Changed `RowMapper` import statement to `simple.ParameterizedRowMapper`;
3. **Lines 15 (17)**: Changed `support.JdbcDaoSupport` import statement to `simple.SimpleJdbcDaoSupport`;
4. **Lines 20 (19)**: Added `Transactional` import statement;
5. **Lines 27–29 (26–27)**: Added the `<code>` tag to `jdbc.core` and `jdbc.object`;

6. **Lines 36 (34)**: Replaced `JdbcDaoSupport` with `SimpleJdbcDaoSupport`;
7. **Lines 50 (47–48)**: Added the `@Transactional` annotation;
8. **Lines 51 (48)**: Added `<imageDescriptor>` to `List`;
9. **Lines 52 (49)**: Replaced `getJdbcTemplate()` with `getSimpleJdbcTemplate()`;
10. **Lines 54 (51)**: Replaced `RowMapper()` with `ParameterizedRowMapper()`;
11. **Lines 54 (51)**: Added `<imageDescriptor>`;
12. **Lines 55 (52)**: Replaced `Object` with `ImageDescriptor`;
13. **Lines 63 (59–60)**: Added the `@Transactional` annotation;
14. **Lines 69 (65)**: Changed `IncorrectResultSizeDataAccessException` to `EmptyResultDataAccessException`;
15. **Lines 70 (66)**: Deleted last parameter (`, 0`);
16. **Lines 82 (77–78)**: Added the `@Transactional` annotation;
17. **Lines 100 (94)**: Modified comment: "*could*" to "*Could*", and added "..." at the end;
18. **Lines 104 (97–98)**: Added the `@Transactional` annotation.

B.6 Test Case 6

1. **Line 12 (12)**: Changed `Assume` import statement to `Assert`;
2. **Lines 16 (13)**: Changed `Assume.AssumptionViolatedException` import statement to `internal.AssumptionViolatedException`;
3. **Lines 19 (17)**: Changed `internal.runners.InitializationError` import statement to `runners.model.InitializationError`;
4. **Lines 17 (18)**: Changed `internal.runners.JUnit4ClassRunner` import statement to `runners.BlockJUnit4ClassRunner`;

5. **Lines 20 (19)**: Changed `internal.runners.links.Statement` import statement to `runners.model.Statement`;
6. **Lines 18 (20)**: Changed `internal.runners.model.FrameworkMethod` import statement to `runners.model.FrameworkMethod`;
7. **Lines 21–22 (22)**: Deleted annotation `@SuppressWarnings`;
8. **Lines 22 (23)**: Changed `JUnit4ClassRunner` to `BlockJUnit4ClassRunner`;
9. **Lines 29 (29–30)**: Added call to `super.collectInitializationErrors(errors)`;
10. **Lines 34–38 (30–34)**: Refactored the body of the `collectInitializationErrors` into the `validateDataPointFields` method;
11. **Lines 30 (29–30)**: Added call to `validateDataPointFields(errors)`;
12. **Lines 41–44 (36–37)**: Created method `validateZeroArgConstructor(List<Throwable> errors)`;
13. **Lines 46–50 (36–37)**: Created method `validateTestMethods(List<Throwable> errors)`;
14. **Lines 62 (47)**: Changed `childBlock` to `methodBlock`;
15. **Lines 80 (65)**: Deleted `.getJavaClass()`;
16. **Lines 83 (68)**: Changed `Assume` to `Assert`;
17. **Lines 84 (69)**: Added "*assumptions*" to comment;
18. **Lines 109 (94)**: Changed `JUnit4ClassRunner` to `BlockJUnit4ClassRunner`;
19. **Lines 117 (102)**: Changed `childBlock` to `methodBlock`;
20. **Lines 118 (103)**: Changed `childBlock` to `methodBlock`;
21. **Lines 129 (114)**: Changed `getAllArguments` to `getArgumentStrings`;
22. **Lines 137 (122)**: Changed `invoke` to `methodInvoker`;
23. **Lines 143 (128)**: Changed `getConstructor` to `getOnlyConstructor`;
24. **Lines 146 (131)**: Changed `childBlock` to `methodBlock`.

Appendix C

Experimental Data

The raw data obtained from the usability experiment is reproduced below.

To preserve participant's privacy, the numbers listed below have no relationship with the order used during the experiment. And although a number always represents the same participant in all tables, they do not correspond to the numbers used in the charts in Chapter 5.

Tables C.1 and C.2 represent Self Assessment Form (Appendix I) and Preference Questionnaire (Appendix J) answers, respectively.

Tables C.3 through C.8 reproduce the measurements made during the comparisons. They represent Test Case 1 through 6 (Appendix A), respectively.

For tables C.3 through C.8 the legend is as follow:

R	Right answer
P	Partial answer
O	Omission
X	Error
E	The reference tool
V	The proposed tool

Participant	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Question 1	c	c	c	c	c	c	d	c	c	b	d	d	c	c	b	b
Question 2	c	c	c	b	c	b	c	d	c	b	d	d	d	b	c	a
Question 3	b	c	e	b	b	b	d	e	d	b	d	e	e	b	c	d
Question 4	a	c	e	b	a	b	d	e	d	c	d	e	a	b	c	b

Table C.1: **Self Assessment Form**

Participant	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Question 1	b	b	a	b	b	a	a	c	c	a	c	a	a	c	b	a
Question 2	a	b	b	b	b	b	a	b	b	a	b	a	a	a	b	a
Question 3	a	b	a	a	b	b	a	b	b	a	a	a	b	a	a	a
Question 4	b	b	a	b	b	a	a	c	a	a	a	b	b	c	b	a
Question 5	a	a	b	c	b	c	a	b	b	a	a	b	b	b	b	c
Question 6	c	a	a	c	c	b	a	a	b	a	a	a	b	c	c	a
Question 7	b	a	b	b	a	a	a	b	a	a	a	a	a	b	a	a
Question 8	b	a	a	b	b	b	a	a	c	a	a	a	a	b	d	a
Question 9	c	b	d	c	c	d	e	d	a	c	c	c	b	a	b	d
Question 10	b	a	b	b	b	c	b	b	b	b	b	b	b	b	b	b
Question 11	e	d	d	d	d	d	e	d	e	e	e	d	d	d	d	e

Table C.2: **Preference Questionnaire**

Participant	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Tool	V	V	V	V	V	V	V	V	E	E	E	E	E	E	E	E
Time (s)	39	33	47	48	46	60	75	89	21	65	55	87	80	111	165	309
Difference 1	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 2	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 3	R	R	R	R	R	R	R	R	R	R	R	R	X	X	R	R
Difference 4	R	R	R	R	R	R	R	R	R	R	R	R	R	X	R	R
Difference 5	R	R	R	R	R	R	R	R	R	O	R	R	R	R	R	X
Difference 6	R	R	R	R	R	R	R	R	R	R	O	R	R	O	R	X
Difference 7	R	R	R	R	R	R	R	R	R	O	R	R	R	R	R	R
Difference 8	O	R	R	R	R	R	R	R	R	O	O	R	R	O	R	R
Errors			1	2	1	2	2									2

Table C.3: **Test Case 1**

Participant	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Tool	E	E	E	E	E	E	E	E	V	V	V	V	V	V	V	V
Time (s)	59	62	65	75	76	78	80	60	9	24	36	43	41	114	54	150
Difference 1	P	P	R	P	P	X	P	R	P	P	P	P	P	P	P	P
Difference 2	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 3	O	R	R	O	O	O	R	R	R	O	O	R	R	R	R	O
Difference 4	X	O	R	P	R	P	R	P	O	O	P	R	R	R	R	R
Difference 5	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 6	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Errors				1												

Table C.4: **Test Case 2**

Participant	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Tool	V	V	V	V	V	V	V	V	E	E	E	E	E	E	E	E
Time (s)	35	37	56	66	55	70	57	63	41	53	44	80	71	68	87	296
Difference 1	R	O	R	R	R	O	R	R	R	R	R	R	R	R	R	P
Difference 2	R	R	R	R	R	R	R	R	R	O	R	R	R	R	R	R
Difference 3	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 4	P	P	R	R	P	P	R	R	R	R	R	R	R	R	R	R
Difference 5	P	P	R	R	O	R	R	R	R	O	R	R	R	R	R	X
Difference 6	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	P
Difference 7	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 8	R	R	R	R	R	R	R	R	R	O	R	R	R	R	R	R
Difference 9	P	P	R	R	P	R	R	R	R	R	R	X	R	R	R	R
Difference 10	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Errors										1						

Table C.5: **Test Case 3**

Participant	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Tool	E	E	E	E	E	E	E	E	V	V	V	V	V	V	V	V
Time (s)	69	74	86	89	159	129	114	165	17	14	50	61	52	41	60	163
Difference 1	R	R	R	P	O	X	X	R	R	R	R	R	R	R	R	R
Difference 2	R	R	R	O	O	R	X	R	R	R	R	R	R	R	R	R
Difference 3	R	R	R	O	R	P	P	R	R	R	R	P	R	R	R	R
Difference 4	P	R	R	R	R	P	R	R	P	R	R	R	P	R	P	R
Difference 5	R	R	R	R	R	P	P	R	O	R	R	R	R	R	R	R
Difference 6	R	R	R	O	R	R	R	R	R	R	R	R	P	R	R	R
Difference 7	X	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 8	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 9	R	R	R	O	R	R	R	R	R	R	R	R	R	R	R	R
Difference 10	R	R	R	P	R	R	R	R	R	R	R	R	R	R	R	R
Difference 11	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 12	R	R	R	O	R	R	R	R	R	R	R	R	R	R	R	R
Difference 13	R	R	R	O	R	R	R	R	R	R	R	R	R	R	R	R
Difference 14	R	R	R	O	R	O	R	R	R	R	O	R	R	R	R	R
Difference 15	R	R	R	P	X	R	R	R	R	R	R	R	R	R	R	R
Difference 16	R	R	R	R	R	O	R	R	R	R	R	R	R	R	R	R
Errors							1									

Table C.6: **Test Case 4**

Participant	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
Tool	V	V	V	V	V	V	V	V	E	E	E	E	E	E	E	E
Time (s)	32	47	48	77	71	97	92	93	61	39	78	67	62	59	96	323
Difference 1	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 2	R	R	R	R	R	R	R	R	P	R	P	R	R	P	R	R
Difference 3	P	P	P	P	P	P	P	P	P	P	R	P	P	R	P	R
Difference 4	R	R	R	R	O	R	R	R	R	R	R	R	R	R	R	R
Difference 5	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 6	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 7	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 8	O	R	R	R	R	R	R	R	R	O	R	R	R	R	R	R
Difference 9	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 10	O	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 11	O	R	R	R	O	O	R	R	R	O	R	R	O	R	R	R
Difference 12	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 13	R	R	R	R	R	R	R	R	R	O	R	R	R	R	R	R
Difference 14	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 15	R	R	R	R	R	R	R	R	O	O	R	O	O	O	O	R
Difference 16	R	R	R	R	R	R	R	R	R	O	R	R	R	R	R	R
Difference 17	R	R	R	R	R	R	R	R	R	P	R	P	R	R	R	R
Difference 18	R	R	R	R	R	R	R	R	R	O	R	R	R	O	R	R
Errors	1	1		1						2						

Table C.7: **Test Case 5**

Participant	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
Tool	E	E	E	E	E	E	E	E	V	V	V	V	V	V	V	V
Time (s)	68	104	126	136	91	143	182	175	58	16	94	39	82	53	83	130
Difference 1	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 2	X	P	R	R	R	R	P	R	P	P	R	P	R	P	P	R
Difference 3	P	P	P	R	R	P	P	O	R	P	P	P	P	P	R	P
Difference 4	P	P	R	P	R	P	P	R	P	P	R	P	P	P	R	P
Difference 5	P	P	P	P	R	P	P	R	R	P	P	P	P	P	P	P
Difference 6	P	P	P	O	R	P	P	R	R	P	R	P	P	P	R	P
Difference 7	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 8	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 9	R	R	R	O	R	R	O	R	R	R	R	R	R	R	R	R
Difference 10	O	O	R	O	R	O	R	R	R	P	R	R	O	R	O	R
Difference 11	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 12	R	R	R	O	R	R	R	R	R	R	R	R	R	R	R	R
Difference 13	R	R	R	O	R	R	R	R	R	R	R	R	R	R	R	R
Difference 14	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 15	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 16	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 17	R	R	R	P	O	O	X	R	R	R	R	R	R	R	R	R
Difference 18	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 19	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 20	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 21	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 22	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 23	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Difference 24	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Errors	1	1		2	1	1				1			1			

Table C.8: **Test Case 6**

Appendix D

Statistical Information

Test Case	**1**		**2**		**3**		**4**		**5**		**6**	
Tool	V	E	V	E	V	E	V	E	V	E	V	E
Maximum	89	165	114	80	70	87	61	165	97	96	94	182
Minimum	33	21	9	59	35	41	14	69	32	39	16	68
Median	47.5	80.0	41.0	70.0	56.5	68.0	50.0	101.5	74.0	62.0	58.0	131.0
Average	54.6	83.4	45.9	69.4	54.9	63.4	42.1	110.6	69.6	66.0	60.7	128.1
Std. Dev.	17.7	42.2	30.9	8.2	11.9	16.5	18.0	35.0	23.0	16.3	25.6	37.0
p-value (%)	6.8		4.9		14.2		<0.1		36.5		0.1	

Table D.1: **Time to Perform the Experiment:** Outlier data excluded.

Test Case	**1**		**2**		**3**		**4**		**5**		**6**	
Tool	V	E	V	E	V	E	V	E	V	E	V	E
Maximum	89	309	150	80	70	296	163	165	97	323	130	182
Minimum	33	21	9	59	35	41	14	69	32	39	16	68
Median	47.5	83.5	42.0	70.0	56.5	69.5	51.0	101.5	74.0	64.5	70.0	131.0
Average	54.6	111.6	58.9	69.4	54.9	92.5	57.3	110.6	69.6	98.1	69.4	128.1
Std. Dev.	17.7	84.4	45.0	8.2	11.9	78.4	43.4	35.0	23.0	86.4	33.2	37.0
p-value (%)	5.2		26.8		11.1		0.9		19.9		0.3	

Table D.2: **Time to Perform the Experiment:** Outlier data included.

Test Case	**1**		**2**		**3**		**4**		**5**		**6**	
Tool	V	E	V	E	V	E	V	E	V	E	V	E
Maximum	2	4	3	4	4	4	2	10	5	10	7	10
Minimum	0	0	1	0	0	0	0	0	1	0	2	1
Median	1.0	1.5	1.5	2.0	1.0	0.0	1.0	2.5	2.0	3.0	4.5	7.0
Average	1.13	1.75	1.75	2.00	1.50	1.00	0.88	3.25	2.13	3.13	4.25	5.50
Std. Dev.	0.78	1.64	0.83	1.22	1.58	1.50	0.78	3.34	1.27	2.80	1.71	2.92
p-value (%)	17.9		32.0		26.4		4.6		19.1		15.9	

Table D.3: **Total Number of Incorrect Answers**

Question	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Maximum	5	5	5	5	5	5	5	5	5	3	5
Minimum	3	4	4	3	3	3	4	2	1	1	4
Median	4.0	4.0	5.0	4.0	4.0	4.5	5.0	5.0	3.0	2.0	4.0
Average	4.19	4.44	4.63	4.31	4.13	4.19	4.69	4.38	2.94	2.00	4.38
Std. Dev.	0.81	0.50	0.48	0.68	0.70	0.88	0.46	0.86	1.09	0.35	0.48
p-value (%)	<0.1	<0.1	<0.1	<0.1	<0.1	<0.1	<0.1	<0.1	41.4	—	<0.1

Table D.4: **Preference Questionnaire:** Regarding Table C.2.

Appendix E

Outlier Data

In this section we reproduce the main time charts including the outlier data removed from the initial analysis. Overall, the removal benefited the reference tool more than the proposed tool. Please note that only time-related data was removed from the analysis. The outlier answers to the comparison tasks and preference questionnaire were still considered.

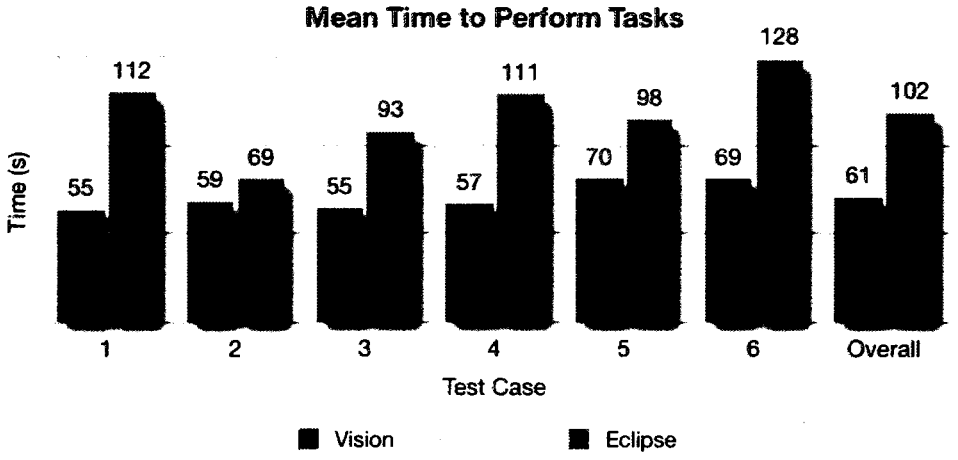

Figure E.1: **Mean Time to Perform Tasks:** Outlier data included.

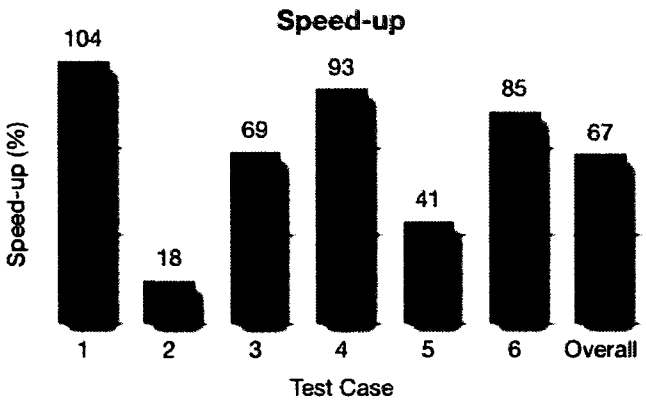

Figure E.2: **Speed-up:** Outlier data included.

Appendix F

Experiment Script

Below we reproduce the protocol which was followed for each participant before the experiments.

1. Briefly explain the experiment and its purpose;
2. Ask participant to read and sign both copies of *Consent Form*;
3. Fill participant number in *Self Assessment Form*, *Preference Questionnaire*, and spreadsheet;
4. Ask participant to answer *Self Assessment Form*. Make sure participant has at least basic knowledge of Java;
5. Explain the proposed tool is a non feature-complete prototype. Only discussed features are the subject of evaluation; judgement shall not be based on expected features (*merging*, *three-way compare*, *etc.*);
6. Explain the tool, not the participant, is being measured. Participant should perform experiment at her own pace, no need to rush;
7. Explain that the participant has to tell what have changed, not what is highlighted. Tools are error-prone and shall not be blindly trusted. Not everything which is highlighted may actually be a change; not all changes are highlighted; a single change may be misrepresented as a set of changes.
   (a) Participant does not need to understand the code nor the purpose of the changes, only what have changed;

(b) Participant does not need to explain every single detail of a change, but has to be specific: "*method X was added*" is OK; "*this line has changed*" is not;

(c) Participant does not need to report changes in white space, line breaks, or empty lines.

8. Open sample comparison using the reference tool;

9. Explain and show how to use the reference tool: Show which side is the new version and which is the old one; show how changes are highlighted and how a set of changes in one side is connected to the other side; explain how to report additions, deletions, and modifications;

10. Open the same sample comparison using the proposed tool;

11. Explain and show how to use the proposed tool: Explain all changes are displayed merged into a single view; show how changes are highlighted and how colors should be interpreted; show how to view modifications using tooltips and hot keys; explain how to report additions, deletions, and modifications;

12. Show how to change the highlighting schema. Explain this is not a feature subject to evaluation, just a preference question for feedback on the alternatives;

13. Record in the spreadsheet which tool is to be used first:

    (a) Participant alternates between the tools at each comparison, using each tool for half the comparisons;

    (b) The first participant start with the reference tool, the second participant with the proposed tool, and so forth;

    (c) Comparison tasks are always performed in the same order, therefore each comparison is performed half the time with the reference tool, half with the proposed tool.

14. Explain no feedback will be given by the examiner during the experiment;

15. Ask participant if she has any questions and if we can proceed with the experiment;

16. Start screen recording tool;

17. Ask participant to compare first pair of files using the assigned tool;

18. For each comparison, record in the spreadsheet time spent understanding the changes;

19. After each comparison, ask participant to explain the changes. Participant can refer to the code to answer questions. Take note of right answers, wrong answers, incomplete answers, and omissions;

20. Ask participant to answer *Preference Questionnaire.*

Appendix G

Recruitment Letter

The following text was sent via e-mail to potential participants.

Hi,

My name is Marconi Lanna and I am a graduate student at the University of Ottawa under the supervision of Prof. Daniel Amyot. I am looking for volunteers to participate in a research project.

I need some people to perform an experiment in which one would compare pairs of files (Java source code) using two different tools and then try to answer a few questions about the comparisons. This would be done using the Eclipse IDE and a specially developed plug-in.

Basic knowledge of the Java programming language is required, to the level of understanding the source code of simple, small classes. A brief explanation of the environment and the tools will be given. Therefore, no experience with the Eclipse IDE or file comparison tools is necessary.

The purpose of the experiment is to evaluate the features offered by the reference and the proposed tools. The outcome of the experiment will be used anonymously in my research.

The experiment should take about 50 minutes and can be scheduled at a time convenient for you.

Participation is strictly voluntary. If you are a student, whether or not you participate in the study will have no effect on your grades or other academic evaluation. Professor Amyot, the thesis supervisor, will have no access to the list of participants nor will know who participates and who does not. All data he will have access to will be anonymous.

If you are willing to participate, please simply reply to this e-mail.

Thanks,

Appendix H

Consent Form

This Consent Form was given to participants before the experiments. Participants were required to read and sign it before performing any tasks.

Consent Form

Invitation to Participate

I am invited to participate in a **University of Ottawa research study entitled " Spotting the Difference: A Source Code Comparison Tool"** conducted by graduate student Marconi Lanna under the supervision of Prof. Daniel Amyot, both from the School of Information Technology and Engineering.

Purpose of the Study

The purpose of the study is to help improve certain features of file comparison tools. Specifically, a single-pane source code comparison tool is proposed as an interface metaphor for reviewing modified versions of a Java source file and understanding the differences between them.

Participation

My participation will consist of comparing eight pairs of files (Java source code) using two different software tools, the Eclipse IDE and a special plug-in, four pairs each, and then explain what I have learned about the comparisons. The researcher will explain how the tools are to be used in the context of the experiment. After the experiment, I will answer an anonymous

questionnaire with general questions about my impressions regarding the experiment.

The time taken to perform the tasks will be measured. However, I understand that **the subject of the evaluation is the performance of the software tools, not mine.** A special software will record the contents of the computer screen during the experiment, but **NO** video or audio recordings of me will be made.

My participation should be done in a single 50-minute session.

Risks

I have received assurance from the researcher that **there are no known risks** associated with this experiment greater than those I might encounter in everyday life.

Benefits

My participation in this study will provide the research with experimental data to evaluate and propose improvements to file comparison tools.

Confidentiality and Anonymity

I have received assurance from the researcher that **all information produced during the session will remain strictly confidential.**

I understand that **the outcome of the experiment will be used only to evaluate the performance of the software tools.**

Anonymity will be protected because neither my name nor any identifiable information will ever be recorded. If needed, data might be tagged with non-traceable numeric IDs.

Conservation of Data

All data produced during the experiment will be kept **anonymously**, and will be **accessed only by the researchers.** The raw data will be kept by the supervisor for a period of 5 years in case of an audit.

Voluntary Participation

I understand that **my participation is strictly voluntary** and if I choose to participate, **I can withdraw from the study** at any time and/or refuse to answer any questions, without suffering any negative consequences.

If I am a student, whether or not I participate in the study will have no effect on my grades or other academic evaluation. Professor Amyot, the thesis supervisor, will have no access to the list of participants nor will know who participates and who does not. All data he will have access to will be anonymous.

If I choose to withdraw, **no data gathered until the time of my withdrawal will be used.**

Acceptance

I, *participant name*, agree to participate in the above research study conducted by Marconi Lanna, under the supervision of Prof. Daniel Amyot, both from the School of Information Technology and Engineering.

If I have any questions about the study, I may contact the researcher by e-mail, mgarc021@uottawa.ca, or his supervisor by phone, (613) 562-5800 ext. 6947, or e-mail, damyot@site.uottawa.ca.

If I have any questions regarding the ethical conduct of this study, I may contact the Protocol Officer for Ethics in Research, University of Ottawa, Tabaret Hall, 550 Cumberland Street, Room 159, Ottawa, ON K1N 6N5, phone (613) 562-5841, e-mail ethics@uottawa.ca

There are two copies of the consent form, one of which is mine to keep.

Appendix I

Self Assessment Form

Participants were asked to answer this self assessment form before performing the experiment. Participants were not questioned about their answers, but only participants which claimed at least a beginner-level knowledge of the Java programming language were invited to continue.

Self Assessment Form

Your answers to this self assessment form will be recorded **anonymously**. Please, do **NOT** write in your name, but **DO** write your participant number.

All questions below should be answered based on your own judgement about yourself and your knowledge of these technologies. You will **NOT** be questioned about your answers. These answers are for reference purposes only and will **NOT** affect the outcome of the experiment.

For each of the questions below, *circle* the answer that **best** matches your opinion.

Question 1

How would you classify your own knowledge of the **Java** programming language?

No knowledge Beginner Intermediate Expert

Question 2

How would you classify your own experience working with the **Eclipse** development environment?

No experience Beginner Intermediate Expert

Question 3

How often do you review changes made by you or by others to source code files?

Never Occasionally Every month Every week Every day

Question 4

How often do you use comparison tools to perform the tasks mentioned on Question 3?

Never Occasionally Every month Every week Every day

Appendix J

Preference Questionnaire

Participants were asked to answer this preference questionnaire after the experiment. Question 10, although still reproduced here for completeness, was annulled.

Preference Questionnaire

This questionnaire is to be answered **anonymously**. Please, do **NOT** write in your name, but **DO** write your participant number.

All questions bellow should be answered based on the features that were discussed and/or showed during the experiment. Please, do **NOT** base your answers on previous knowledge or expected features.

For each of the questions below, *circle* the answer that **best** matches your opinion.

Question 1

Learnability is a measure of how easy it is to learn to use a software product. As an analogy, it is arguably easier for a baby to learn to *crawl* than it is to learn to *walk*.

Given this definition, would you say the **proposed tool** is **easier to learn** than the reference tool?

Strongly Agree Agree Neutral Disagree Strongly Disagree

Question 2

Ease of use is a measure of how easy it is to use a software product **after** its use has been *learned*. Keeping with our analogy, after learned, *walking* is typically easier than *crawling* since

it requires less limbs and is done on a more comfortable position.

Given this definition, would you say the **proposed tool** is **easier to use** than the reference tool?

Strongly Agree Agree Neutral Disagree Strongly Disagree

Question 3

Efficiency is a measure of how quickly tasks can be performed with a software product **after** its use has been *mastered*. Again, *walking* is usually faster than *crawling*.

Given this definition, would you say the **proposed tool** allows you to perform tasks **more efficiently** than the reference tool?

Strongly Agree Agree Neutral Disagree Strongly Disagree

Question 4

Intuitiveness is a measure of how easy it is to understand the output or the interface of a software product.

Given this definition, would you say the **proposed tool** is **more intuitive** than the reference tool?

Strongly Agree Agree Neutral Disagree Strongly Disagree

Questions 5 to 10 below concern the **proposed tool** and its **features**.

Question 5

The use of a **single-pane interface** made it **easier** to understand the differences and perform the comparison tasks.

Strongly Agree Agree Neutral Disagree Strongly Disagree

Question 6

The **highlighting granularity** of the proposed tool (i.e, single-tokens instead of whole-lines) is appropriate to perform the comparison tasks.

Strongly Agree Agree Neutral Disagree Strongly Disagree

Question 7

The **classification of differences** (*additions*, *deletions*, and *modifications*) along with the **use of colors** made it **easier** to understand the differences and perform the comparison tasks.

Strongly Agree Agree Neutral Disagree Strongly Disagree

Question 8

PREMISE: Unlike *additions* or *deletions*, *modifications* require both the original and the changed text to be displayed.

The use of artifacts such as **tooltips** and/or **hot keys** is a **convenient** way to display modifications.

Strongly Agree Agree Neutral Disagree Strongly Disagree

Question 9

For **visualizing modifications**, which artifact would you **prefer** using:

Tooltips only Tooltips mostly Both
Hot keys mostly Hot keys only Neither

Question 10

Which of the highlighting schemas do you think was the **most** pleasant and practical to use:

Background only Background with strikeouts Strikeouts and underlines
No preference

Question 11

If **both tools were available** in your work environment, which tool would you **prefer** using if you had to perform a comparison task?

The **reference** tool only The **reference** tool mostly **Both** tools similarly
The **proposed** tool mostly The **proposed** tool only