A Tool for Automated Line Mapping Across File Versions

Aaron Tir
Faculty of Science,
University of Windsor
110139556
tir@uwindsor.ca

Victor Rodriguez
Faculty of Science,
University of Windsor
110138746
rodrig44@uwindsor.ca

Ali Asghar
Faculty of Science,
University of Windsor
110140646
asghar4@uwindsor.ca

Fineas Muntean
Faculty of Science,
University of Windsor
110135802
munteanf@uwindsor.ca

Shatab Shameer Zaman
Faculty of Science,
University of Windsor
110159301
zaman74@uwindsor.ca

Abstract – This project presents an automated tool for mapping line-level changes between different versions of a java source file. The tool identifies correspondences, insertions, deletions, and moved lines using a normalization process combined with a longest common subsequence (LCS)–based matching algorithm. Its performance was evaluated using a newly constructed dataset consisting of 25 file pairs and 1350 manually validated line mappings, supplemented by the dataset provided in the course. The results demonstrate that the technique effectively captures most structural and textual modifications, particularly those involving whitespace, comments, and localized edits. A visualization design is also proposed to support clear interpretation of linemapping information through side-by-side comparisons and color-coded change indicators.

Keywords: line mapping, version control, LCS, software maintenance, refactoring, visualization

I. Introduction

This project presents a line-mapping tool designed to automatically identify how individual lines of code evolve between two versions of the same file. The goal is to support software maintenance, refactoring understanding, historical analysis, and change tracking. The work fulfills three requirements: (1) develop an automated technique to map lines across versions, (2) evaluate the technique on a provided dataset and a newly collected dataset of 25 file pairs, and (3) design a visualization for line-mapping information.

II. Data Collection

2.1. Overview of data collection

A dataset of 25 file pairs was created to evaluate the line-mapping tool. Each pair contains two versions of the same java source file, with the second version including realistic edits such as insertions, deletions, reordered blocks, comment changes, and whitespace modifications. Both versions of each file were manually reviewed, and ground-truth mappings were recorded for every line, producing 1350 validated correspondences used in the evaluation.

2.2. Selection of file pairs

The file pairs were chosen to represent a range of code structures and modification types. Selection focused on diversity in programming constructs (e.g., functions, conditionals, loops), variation in change patterns (insertions, deletions, mixed edits, reordering), and realism consistent with typical version-control updates. Files were drawn from algorithm implementations and simplified code examples, ensuring each pair reflected an actual revision.

2.3. Construction of line mappings

Ground-truth line mappings were created by manually comparing each pair of files. The process involved aligning corresponding regions, checking textual similarity, and categorizing each line as matched, inserted, deleted, or moved/modified. Ambiguous cases—such as repeated or slightly edited lines—were verified using surrounding context. This produced the final set of 1350 reference mappings used to assess the tool's accuracy.

III. Technique Description and Evaluation

3.1. Method overview

The technique maps lines between two versions of a file using two steps: line normalization and LCS-based alignment.

Normalization removes trailing whitespace and comment segments (// and single-line /* … */) so that formatting differences do not affect comparison. After normalization, the algorithm constructs a dynamic-programming table to compute the longest common subsequence (LCS) of lines between the versions. Backtracking over the table yields a mapping that classifies each line as a match, insertion, or deletion, represented as (orig_line, new_line) tuples, where -1 indicates absence in the given version. The results are exported as XML for downstream use.

3.2. Experimental setup

Evaluation was performed on a dataset of 25 file pairs containing 1350 manually validated line mappings. Each pair was processed by the tool, and the resulting mappings were compared directly to ground-truth labels. Additional tests were performed using several examples from the instructor-provided dataset to ensure consistency across diverse file structures and modification types. The mappings are written into an XML file, as shown in figure 1 below:

```
<LOCATION ORIG=”4” NEW=”6”/>
//Line mapped from line 4 to line 6
<LOCATION ORIG=”-1” NEW=”5”/>
//Line added to line 5
<LOCATION ORIG=”7” NEW=”-1”/>
//Line deleted from line 7
```

Figure 1. Example XML output.

3.3. Accuracy on 1350 line mappings

Out of 1350 ground-truth mappings, the tool correctly identified 1172 lines, yielding an accuracy of 86.8%. Most mismatches occurred in files with whitespace or repeated lines, which are challenging to handle for LCS-based methods. The technique performed strongly on whitespace edits, comment changes, and localized modifications.

3.4 Figures, tables, and evaluation data

Table 1 summarizes the overall accuracy of the line-mapping tool on the dataset of 25 file pairs. A total of 1350 manually validated ground-truth line correspondences were used to evaluate performance. The tool correctly mapped 1172 of these lines, resulting in an overall accuracy of 86.8%.

Table 1.Overall accuracy of the line-mapping tool.

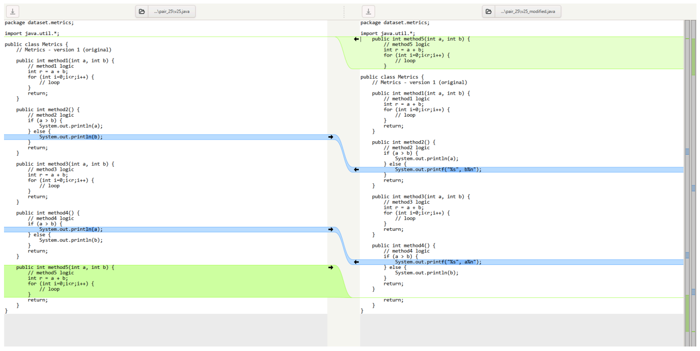
Total # of lines	Lines correctly mapped	Accuracy (%)
1350	1172	86.8%

Inaccuracies primarily occur in edge cases involving repeated lines, whitespace, and opening or closing brackets of functions. For example, the tool may incorrectly map a closing bracket to a different function or interpret it as an added line. These errors are largely due to the limitations of the LCS algorithm, which can create ambiguities when distinguishing between highly similar sequences of lines.

IV. Presentation of Line Mapping Information

To help users understand how individual lines evolve between two file versions, a visualization was designed to present line-level correspondences, insertions, deletions, and moved lines in a clear and interactive format. The visualization displays the original file on the left and the modified file on the right, with color-coded highlighting and connecting curves showing how each line maps across versions.

4.1. Figure 2. Line-mapping visualization generated for one of the evaluated file pairs.



```
package dataset.metrics;

import java.util.*;

public class Metrics {
    // Metrics - version 1 (original)

    public int method1(int a, int b) {
        // method1 logic
        int r = a + b;
        for (int i=0;i<r;i++) {
            // loop
        }
        return;
    }

    public int method2() {
        // method2 logic
        if (a > b) {
            System.out.println(a);
        } else {
            System.out.println(b);
        }
        return;
    }

    public int method3(int a, int b) {
        // method3 logic
        int r = a + b;
        for (int i=0;i<r;i++) {
            // loop
        }
        return;
    }

    public int method4() {
        // method4 logic
        if (a > b) {
            System.out.println(a);
        } else {
            System.out.println(b);
        }
        return;
    }

    public int method5(int a, int b) {
        // method5 logic
        int r = a + b;
        for (int i=0;i<r;i++) {
            // loop
        }
        return;
    }
}
```

```
package dataset.metrics;

import java.util.*;
    public int method5(int a, int b) {
        // method5 logic
        int r = a + b;
        for (int i=0;i<r;i++) {
            // loop
        }

public class Metrics {
    // Metrics - version 1 (original)

    public int method1(int a, int b) {
        // method1 logic
        int r = a + b;
        for (int i=0;i<r;i++) {
            // loop
        }
        return;
    }

    public int method2() {
        // method2 logic
        if (a > b) {
            System.out.println(a);
        } else {
            System.out.printf("%s", b%n");
        }
        return;
    }

    public int method3(int a, int b) {
        // method3 logic
        int r = a + b;
        for (int i=0;i<r;i++) {
            // loop
        }
        return;
    }

    public int method4() {
        // method4 logic
        if (a > b) {
            System.out.printf("%s", a%n");
        } else {
            System.out.println(b);
        }
        return;
    }

        return;
    }
}
```

4.2. Interpretation of visualization

The visualization uses a side-by-side layout, where each file version is shown in its original ordering. Lines that correspond across versions are connected using curved arrows. Color highlights indicate the type of change:

• Blue lines represent matched lines. These lines share the same normalized content, and the connecting curves indicate their positions in both versions.

• Green regions indicate inserted or deleted blocks. When a block appears only on one side, the absence of a matching curve shows that the line is new or removed.

• Curved arrows illustrate the mapping direction. A smooth path between two blue lines indicates that the line in the original file maps directly to its counterpart in the modified file.

Lines containing small textual modifications (e.g., formatting changes. or altered string literals) are also matched if their normalized form remains similar. This makes it easier for developers to identify what changed semantically versus what was only reformatted

V. Conclusion

This project developed and evaluated an automated tool for mapping line-level

changes between different versions of a java
source file. By combining normalization of
source lines with an LCS based algorithm, the
tool generates accurate correspondence and
classifies insertions and deletions in a structured
XML format suitable for analysis and visualization.
Evaluation on a dataset of 25 file pairs containing
1350 manually validated mappings demonstrated an
accuracy of 86.8%, with most remaining errors occurring
in cases involving reordered or highly similar lines.
A visualization design was also proposed to support clear
inspection of line-evolution patterns. Overall, the tool
provides an effective foundation for understanding code
changes and can be extended in future work to handle
more complex transformations, such as block moves or
semantic edits.