CLARKSON UNIVERSITY

CSeR - A Code Editor For Tracking & Visualizing Detailed Clone Differences

A Thesis by

Ferosh Jacob

Department of Mathematics and Computer Science

Submitted in partial fulfillment of the requirements

for the degree of

Master of Science

Computer Science

June 2009

Accepted by the Graduate School

Date DEAN

The undersigned have examined the thesis entitled **CSeR - A Code Editor For Tracking & Visualizing Detailed Clone Differences** presented by Ferosh Jacob, a candidate for the degree of Master of Science, Computer Science and hereby certify that it is worthy of acceptance.

_____

Date

EXAMINING COMMITTEE

_____

Christino Tamon

_____

Robert A. Meyer

_____

ADVISOR Daqing Hou

Abstract

Clone support can improve the quality and maintainability of software projects. Although significant research has focused on locating code clones, less work has addressed tracking "copy and paste" operations, even though these operations are a primary source of clone formation.

This thesis presents CSeR, a code editor that records clones created through copy-and-paste operations and tracks the changes made to a new clone. CSeR visualizes those changes with distinct colors so that programmers can understand how two related pieces of code have diverged.

CSeR computes changes incrementally by comparing the Abstract Syntax Trees (ASTs) of a pair of clones as edits are made. This incremental, AST-based approach makes CSeR more precise than tools that compare files only after the fact.

An empirical study was conducted using 37 test cases collected from industrial and research projects to evaluate the robustness and usefulness of the tool. In total, CSeR identified 533 changes, which were categorized into 20 different types. A comparative study with related tools demonstrates the distinctive features of CSeR. Finally, this thesis discusses potential extensions that could broaden the scope of CSeR in future work.

Acknowledgements

I would like to extend my heartfelt thanks to my advisor, Daqing Hou, who has guided me throughout the project and always helped me stay focused. I also like to thank the committee members Christino Tamon, Robert A. Meyer for their comments and corrections. I am grateful to my parents, room mates (Pravin Shukla, Pavan Vimal, Punith), friends (Kalyan, Anshuman) for their love and support during the project.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

For many programmers, copying and pasting code is unavoidable. It is therefore surprising that existing programming editors provide little support for copy and paste beyond ordinary text editing and refactoring. For example, Eclipse performs automatic refactoring when a programmer copies and pastes a class. Refactoring alone, however, is not enough.

Each copy-and-paste action carries information that can help programmers understand and maintain related code. Kim [15] observes, "In fact programmers employ their memory of C&P history as they make changes to code or decide when to restructure code." She continues, "However, a programmer's recollection of C&P history can be short-lived, somewhat inaccurate, and difficult to transfer from person to person."

Copy-and-paste operations therefore play an important role in software development. To provide better support for developers, we must analyze those operations more carefully. In particular, we need to understand what information is lost when the IDE does not track copy-and-paste history.

1.1 Problem: Loss of Relationship

Developers often copy and paste code within a class, or copy an entire class, because there is meaningful similarity between the source and the target. That similarity is evident to the developer at the time of the copy-and-paste operation, but it may not be evident to

another developer, or even to the same developer at a later time.

In other words, copied and pasted code contains an implicit relationship. If the IDE does not track the copy-and-paste operation, that relationship is lost. Even when the relationship is known, developers still need detailed information about how the clones differ.

1.2 Problem: Missing Detailed Information

Detailed information includes how clones differ, where new statements were inserted, and which statements were deleted, moved, or updated. This fine-grained information can:

1. Improve understanding between related clones
2. Guide similar copy-and-paste operations in the future

These differences can support a better understanding of the code. The problem is more difficult when clones are located in different files, because the developer must leave the editor context and use a separate tool, such as the Eclipse Compare Editor, or manually compare the files. The limitations of the Compare Editor are discussed in the validation chapter.

1.3 Example Scenario

Consider Figure 1.1. SetFilterWizardPage is a wizard page used when creating a new source folder to include or exclude file patterns from the build path. A brief study of this class's history suggests that it was previously implemented as a dialog and later replaced by a wizard page. The earlier dialog was not deleted; it was maintained so that external clients could set file patterns in a source folder independently of creating or editing the source folder. Most likely, the new SetFilterWizardPage class was copied from ExclusionInclusionDialog.

Problem: Loss of Relationship Consider a situation in which a developer modifies SetFilterWizardPage. The same change may also be required in ExclusionInclusionDialog.

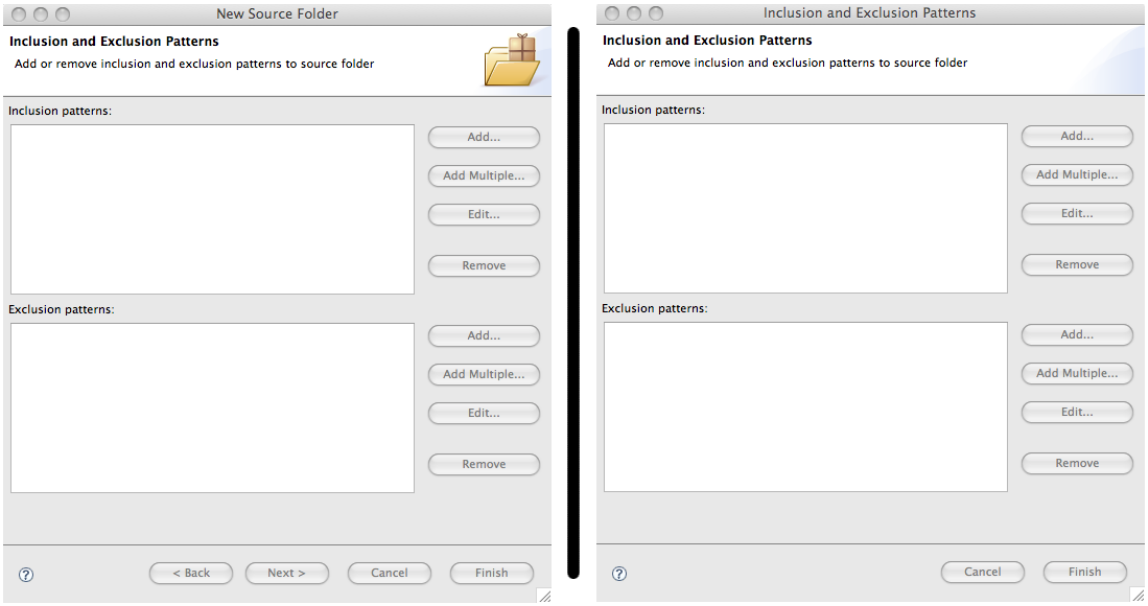

Figure 1.1: SetFilterWizardPage and ExclusionInclusionDialog

For example, a change for BIDI (bidirectional support for internationalization) was added to SetFilterWizardPage and then added to ExclusionInclusionDialog as well. A similar case occurred for a bug fix related to new source-folder creation.

The question is how a programmer can know which classes are affected by a bug fix or change. What relationship do these classes have? Even if the Eclipse team decides to remove the duplicates, they still need to know which classes were created in a similar fashion and how similar those classes are.

Problem: Loss of Detailed Information Even when the relationship is known, finding detailed differences between clones can be time-consuming. In the example scenario, part of the outline view of the two classes is shown in Figure 1.2. A developer may want to know which methods in SetFilterWizardPage were removed, which methods correspond to methods in ExclusionInclusionDialog, and which methods are new in ExclusionInclusionDialog. These questions are not limited to methods; they can also apply at the expression level.

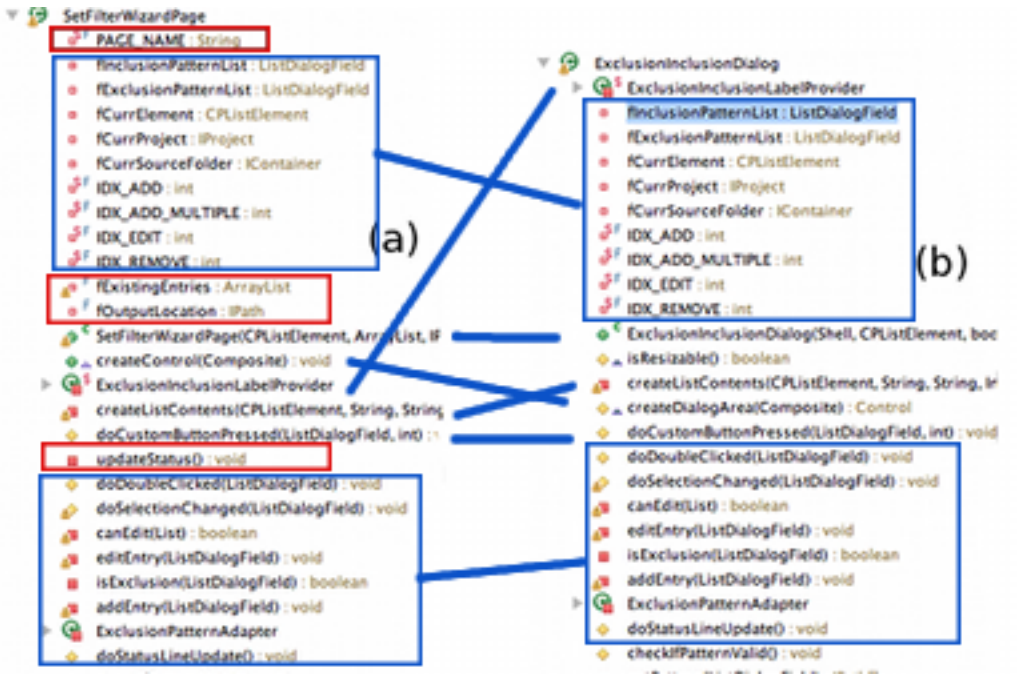

Figure 1.2: Outline View of SetFilterWizardPage and ExclusionInclusionDialog

1.4 Tracking and Visualizing Detailed Code Differences

CSeR addresses these problems by calculating changes with an incremental AST-based algorithm. The changes are shown directly in the editor and are calculated as the user edits the code. For the example considered here, once ExclusionInclusionDialog is created in an environment that supports CSeR, the code appears as shown in Figure 1.3.

The remainder of this thesis is organized as follows. Chapter 2 introduces the basic definitions and background information. Chapter 3 explains the design of CSeR in detail. Chapter 4 validates the tool. Chapters 5 and 6 discuss related work and present the conclusion and future work.

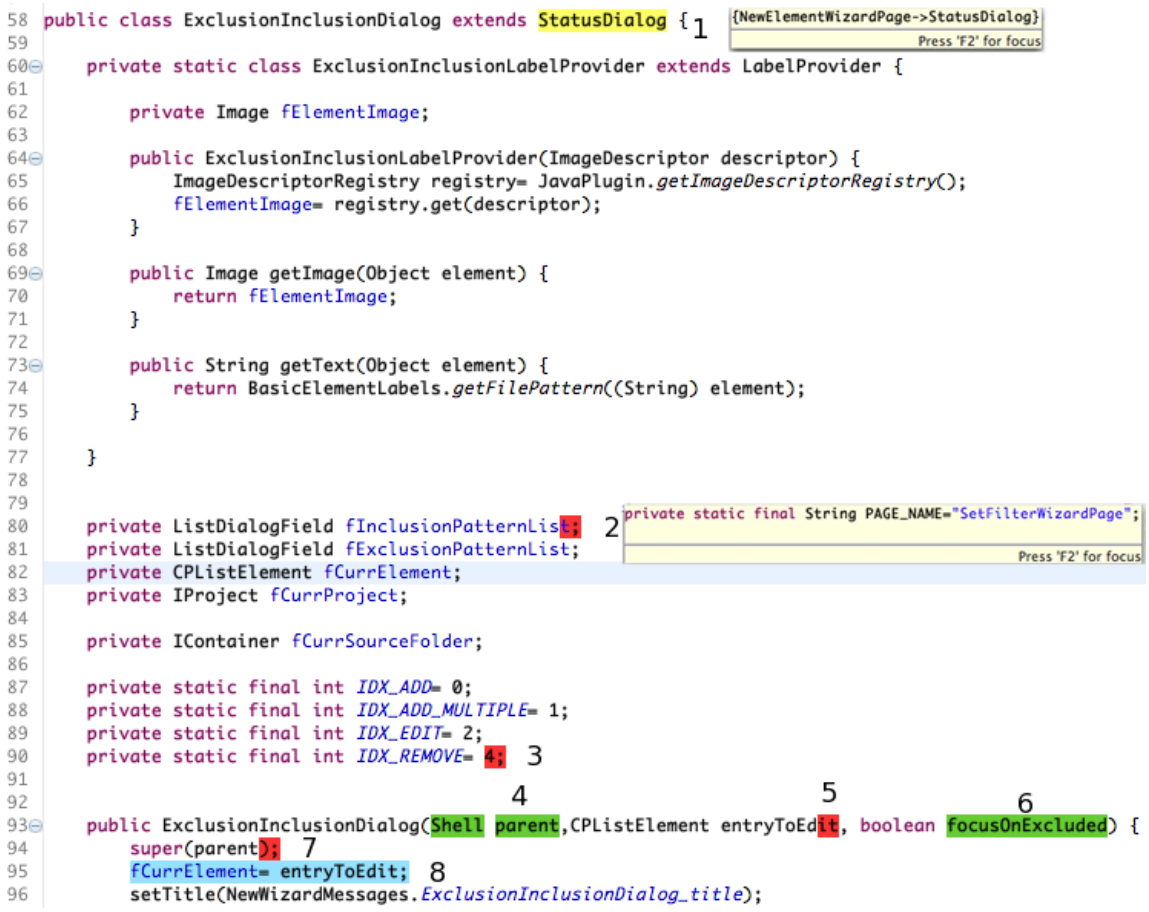

Figure 1.3: CSeR Showing SetFilterWizardPage from ExclusionInclusionDialog

Chapter 2

Background and Definitions

2.1 Context

Unless otherwise specified, this thesis assumes that a programmer copied a file and pasted it in another location, either with the same name or with a different name. The copied file is called the original file, and the pasted file is called the current file. In most scenarios, the current file is also the file being edited.

CSeR treats copying a file as a special case of copying a code block in which no specific position is provided. Therefore, features that apply to files also apply to code blocks, and vice versa. Positions in the current file may be called current positions, while positions in the original file may be called original positions.

2.2 Change

A change is usually an AST-level change from the original file to the current file. Adding spaces within a line or between lines does not trigger a change. Adding, deleting, or updating comments is also not considered a change because CSeR does not track comment changes. More technically, CSeR relies on the default implementation of ASTMatcher in JDT.

2.3 Correspondence

In CSeR terminology, two AST nodes correspond when they were identical during the copy-and-paste operation that formed the clone, and any later difference between them is due to user editing.

2.4 User-edit

A user edit is any editing action, ranging from typing or deleting a single character to inserting, replacing, or deleting a block of code or an entire file.

2.5 Insert, Delete, Update operations in CSeR

CSeR's edit operations are defined within the context of the AST. Adding a new AST node is considered an insert, removing an AST node is considered a delete, and modifying an existing AST node is considered an update.

These operations differ from ordinary text insert, delete, and update operations. For example, inserting a single character is not necessarily an insert in CSeR. In most cases, it is an update, because the smallest node that can be inserted must be at least three characters long, such as a pre- or post-increment expression with a one-character variable name.

2.6 Position

In Eclipse terminology, positions describe text ranges in a document. Positions are adapted as changes are applied to that document. A text range is specified by an offset and a length. Positions can also be marked as deleted. Deleted positions no longer represent valid text ranges in the managed document.

2.7 Position Updaters

Region updaters are listeners that respond to position-change events. Whenever an event changes regions in a document, CSeR's database must be updated with the new information. Region updaters perform this update for CSeR.

2.8 Anchor AST

The anchor AST is usually the immediate parent of the AST node being considered. This information is crucial for statements and body declarations. The anchor AST of a statement is usually a Block, while the anchor AST of a body declaration is usually a TypeDeclaration. CSeR uses this information to distinguish between operations at the statement level and operations at the body-declaration level.

2.9 Statements

This thesis uses the same definition of Statement nodes as JDT. Statements are generally well-formed AST nodes that are direct children of a body node. Statements can occur inside a method, inside a static block in a class, or inside another block of statements.

2.10 Body Declarations

This thesis uses the same definition of BodyDeclaration nodes as JDT. Body declarations are generally well-formed AST nodes that are direct children of a TypeDeclaration node. Type declarations can occur as inner classes inside a method, or as field or method declarations inside a class.

2.11 Nodes

Nodes represent a finer level of division below the statement node. It is practically impossible to visit every node that can compose a statement, so the implementation uses a

smaller set of nodes that programmers commonly change. In practice, these nodes can be expressions such as variable names and string literals, single-variable declarations such as method parameters, or type parameters used in method calls.

2.12 Block

A Block is the AST node that corresponds to a block of statements. It is usually a sequence of statements enclosed in { and }.

2.13 Scope

The meaning of scope depends on the context. For statements, the scope is the Block under consideration. For nodes, the scope is the node under consideration. For methods and fields, the scope is the entire class.

2.14 Action

An action is the way a user performs an edit. Different programmers use different editing styles: some type instead of copying and pasting small code blocks, some delete with the Delete key, some use Backspace, and others cut text to delete it. Here, the question is how the programmer made the change, not what change was made.

2.15 Goal

A goal is the intended code change. The goal is independent of how the change was made. Examples include changing a variable name, changing a class name, or adding another parameter to a method.

2.16 Infix Expressions

Infix expressions have the form operand-operator-operand. The operator can be an arithmetic, Boolean, assignment, or dot operator. Both operands are expressions.

2.17 JDT

Java Development Tools[1] provides the Eclipse plug-ins that enable Java application development in the Eclipse IDE. JDT also provides APIs for processing Java ASTs and related features. CSeR makes extensive use of these APIs.

2.18 CSeREditor

CSeREditor is a superset of the Eclipse JavaEditor and the CSeR tools. For consistent CSeR results, all relevant operations should be performed in CSeREditor.

2.19 PositionList

PositionList is a data structure containing positions. Positions are ordered in this structure, so relationships such as "after," "before," and "smallest" can be defined within it.

2.20 Session

The meaning of session depends on the context. In the context of the CSeR application, a session begins when Eclipse starts and ends when Eclipse closes. In the context of a statement, a session refers to the block in which the user is working. For example, when a user edits variables and method calls inside a single method block, the user is working in one session. When the user creates another block, such as a for loop or if statement, a new session begins.

[1] http://www.eclipse.org/jdt

2.21 Levenshtein distance

Levenshtein distance is a metric for measuring the difference between two sequences. For example, the distance between “kitten” and “sitten” is 1, while the distance between “kitten” and “sitting” is 3, because at least three edits are required.

2.22 Document

An IDocument represents extensible content and allows clients to set and manipulate that content. On each document change, all registered document listeners are informed exactly once.

Chapter 3

Design

3.1 Requirements

CSeR is a clone-tracking tool. In this work, clones are assumed to be formed through copy-and-paste operations. A user may copy and paste either a block of code or an entire file, creating two forms of clone formation. Copying a file is treated as a special case of copying a block of code.

At a high level, CSeR begins with two identical blocks of code. One or both blocks may later change as a result of user edits. The goal of CSeR is to track those changes and present them to the user in a clear and readable way.

One assumption used throughout the editing scenarios is that CSeR does not respond until the code reaches a consistent state. A consistent state is a state in which the code has a well-formed AST. Such a state can exist even when the code has resolution errors. In other words, CSeR remains active when syntax is valid, even if names cannot yet be resolved.

The difference between resolution and syntax errors is shown in Figure 3.1. In the figure, the error marked "1" is a resolution error, where "IEditorPart" cannot be resolved to a type. The error marked "2" is a syntax error: "Syntax error, insert "}" to complete Method Body."

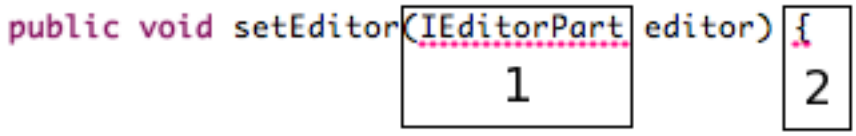

Figure 3.1: Difference between Resolution errors and Syntax errors

Figure 3.2: Consistent Tracking Independent of Actions

We analyzed the requirements as a set of use cases, listed below. The use cases, implementation decisions, and later test cases are based on the Eclipse[1] IDE. Most editing scenarios were collected from our experiments; [17, 14] also contributed to the list.

3.1.1 UC001 Consistent Tracking

This use case requires consistent tracking for the same goal, as defined in Section 2.15, even when the goal is reached through different actions, as defined in Section 2.14. Common actions are listed in Table 3.1.

Table 3.1: **Set of common Actions**

No	Common Actions
1	Inserting Characters at the text caret
2	Deleting text selection
3	Overwriting text selection
4	Backspacing over characters
5	Undoing
6	Pasting
7	Refactoring Tools
8	Code Completion

Stated briefly, CSeR should treat the same change consistently regardless of how the user makes it. For example, in Figure 3.2, "b" is updated to "c". In an editor such as Eclipse, this can be done in at least two ways:

[1]http://www.Eclipse.org

1. Type “c” over the character “b”.

2. Delete character “b” and then type “c”.

This use case therefore states that change tracking should be independent of the editing action. For the example above, CSeR should show the same change, as in Figure 3.2, regardless of how the user made it. This use case applies to the use cases in Sections 3.1.2, 3.1.3, 3.1.4, 3.1.5, 3.1.6, 3.1.7, 3.1.8, 3.1.9, 3.1.10, and 3.1.11.

3.1.2 UC002 Simple Name

This use case corresponds to changing identifiers inside a class. The common modifications involving identifiers are explained in Table 3.2.

Table 3.2: **Set of Name Modifications**

No	Modification involving identifiers	Example
1	Creating a name	
2	Replacing a name	i → count
3	Updating a name	foreColor → bgColor
4	Removing a name	
5	Splitting a name	fPosition → f. Position

3.1.3 UC003 Statements

This use case corresponds to changing statements 2.9 inside a Block 2.12. Common statement modifications are listed in Table 3.3. Updating a statement is not listed in the table because it is a broad case handled separately. For example, updating the arguments in a method call is also an update to the surrounding statement.

Table 3.3: **Set of Statement Modifications**

No	Modification involving Statements	Example
1	Inserting one or more statements	
2	Removing one or more statements	i → count
3	Merging two statements to make a statement	i=expr1;j=expr2; →i=expr2;
4	Moving a statement	i++;j++; → j++;i++;

3.1.4 UC004 Arguments

This use case corresponds to changing arguments inside a method call. Common argument modifications are explained in Table 3.4.

Table 3.4: **Set of Argument Modifications**

No	Modification involving identifiers	Example
1	Inserting an additional argument	print(3) → print(3, 4)
2	Removing an argument	print(3, 4) → print(3)
3	Merging two arguments to make an argument	print(3, 4) → print(34)

3.1.5 UC005 Parameters

This use case corresponds to changing parameters inside a method declaration node. Common parameter modifications are given in Table 3.5.

Table 3.5: **Set of Parameter Modifications**

No	Modification involving identifiers	Example
1	Inserting an additional parameter	void print(int i) → void print(int i, int j)
2	Removing an argument	void print(int i) → void print()
3	Merging two arguments	void print(int i, int j) → void print(int ij)

3.1.6 UC006 Expressions

This is the most general case because expression changes can take many forms. Our strategy is to support the most common cases. Common expression modifications are explained in Table 3.6.

Table 3.6: **Set of Expression Modifications**

No	Modification involving identifiers	Example
1	Inserting a new expression	k=4; → k=4*getValue(j);
2	Removing an expression	k=var1*var2*var3 → k=var1*var3
3	String literal	k= “Hello World”; → k=“Test World”
4	Number literal	k=4. 345;→k=4. 548;
5	Operator	i=i+3;→ i=i-3;

3.1.7 UC007 Comments

This use case does not mean that the tool must track the text of comments. Instead, it refers to AST changes caused by introducing or removing comments. The most common comment-related modifications are given in Table 3.7.

Table 3.7: **Set of Comment related Modifications**

No	Comment related modifications	Example
1	Commenting out a statement	k=4; → // k=4;
2	Creating annotations	@Override
3	Inside statement	k= a+2; → k=a+3;//2;

3.1.8 UC008 Keywords

Keywords are the reserved words in Java. A complete set of Java keywords is shown in Table 3.9. Common keyword-related modifications are given in Table 3.8.

Table 3.8: **Set of Keyword related Modifications**

No	Keyword related modifications	Example
1	Inserting a modifier	int k; → // private int k;
2	Inserting an optional keyword	"extends"
3	Removing a modifier or optional keyword	

Table 3.9: **Set of Keywords in Java 2**

abstract	double	int	strictfp
boolean	else	interface	super
break	extends	long	switch
byte	final	native	synchronized
case	finally	new	this
catch	float	package	throw
char	for	private	throws
class	goto	protected	transient
const	if	public	try
continue	implements	return	void
default	import	short	volatile
do	instanceof	static	while

3.1.9 UC009 Fields

Fields in classes correspond to method declarations and field declarations. In JDT[2]2.17 terminology, these are called body declarations 2.10. A few cases of field-level edits are discussed in Table 3.10.

Table 3.10: **Set of Field level Modifications**

No	Field level modifications	Example
1	Inserting new field or method	
2	Removing one or more method(s) or field(s)	
3	Modify a field initializer	int k; → int k=0;

3.1.10 UC010 Single Statement with Multiple Changes

These cases include scenarios in which UC002 through UC006 occur together or in some combination. The tool should show the resulting change consistently and sensibly. A few cases of multiple edits are discussed in Table 3.11. This use case applies to the use cases in Sections 3.1.2, 3.1.3, 3.1.4, 3.1.5, and 3.1.6.

Table 3.11: **Single Statement Multiple Changes**

No	Multiple Edit modifications	Example
1	Simple Names	k=var1+var2; → k=var3+var4;
2	Simple Name and Number literal	k=4+var1;→ k=234+var2;
3	Parameters and Simple Name	k= a+2+this(); → k=a+2+that(3);//2;

3.1.11 UC011 Delete Operation

For all delete operations, the tool should log the code that was removed. This can be an important source of information when the original programmer reviews the change later or when another programmer reviews it. Delete operations can come from any applicable use case above. This use case applies to the use cases in Sections 3.1.3, 3.1.4, 3.1.5, 3.1.6, 3.1.7, 3.1.8, and 3.1.9.

[2]http://www.Eclipse.org/jdt

3.1.12 UC012 Condition In Conditional Or Loop Statements

In this case, expressions are inserted specifically for "if" statements, although the same idea applies to loop statements. This use case differs from UC003 because the user is not inserting a full statement, but rather an expression and a keyword, as listed in Table 3.12. This use case applies to the use cases in Sections 3.1.3 and 3.1.6.

Table 3.12: **Modification Of Conditions In Statements**

No	Conditional And Loop Statement Modifications	Example
1	Insert Condition checks	k=var1; → if(var1!=null) k=var1;
2	remove Condition checks	if(var2 ≥ 3) k=var1;→ k=var1;

Table 3.13: **Set of all Use Cases**

No	Use Case ID	Name	Related Use Cases
1	UC001	Consistent Tracking	ALL
2	UC002	Simple Name	
3	UC003	Statements	
4	UC004	Arguments	
5	UC005	Parameters	
6	UC006	Expressions	
7	UC007	Comments	
8	UC008	Keywords	
9	UC009	Fields	
10	UC010	Multiple Edit	UC002, UC003, UC004, UC005, UC006
11	UC011	Delete Operation	UC003, UC009
12	UC012	Conditional Statements	UC003, UC006

3.2 Implementation

The requirements above led to the design and implementation of CSeR. The current implementation is an Eclipse plug-in. It includes an editor and a simple view. CSeREditor 2.18 is a superset of the default Eclipse Java editor and the CSeR tools. The editor behaves like the default Java editor until the programmer performs a copy or paste operation, or explicitly activates CSeR.

The plug-in uses the JDT APIs and the plug-ins that ship with the default Eclipse

installation. The only additional external library is used to compute Levenshtein distance 2.21; it comes from the Apache[3] library. The simple view lists all clones collected through copy and paste. Clones can be opened in a difference view to compare them.

3.2.1 Trap "Copy and Paste"

To manage clones created by copy and paste, CSeR must intercept copy-and-paste operations performed in the IDE. In the current implementation, the default JDT copy-and-paste operation is modified to include the tracking feature.

Code 1 The Hello World code in Java.

```
package com. sample;

public class HelloWorldApp {

 public static void main(String[] args) {

  System. out. println("Hello World");

 }
}
```

What information is deduced when code is copied and pasted inside the editor? When the code in 1 is copied and pasted, CSeR parses the AST into smaller AST nodes, as shown below. As mentioned earlier, CSeR must maintain correspondence between two ASTs. To do this, it compares nodes and determines which nodes are identical, similar, or different.

Most user changes are made gradually. This allows CSeR to reduce the comparison to a smaller scope 2.13. Instead of comparing the entire AST after every edit, CSeR usually needs to find only the AST node relevant to the edit and compare it with the corresponding node in the original file 2.1.

CSeR prefers to compare small AST nodes because the comparison is faster. However, identifying the relevant small AST node requires tracking positions. As shown in 3.2.1,

[3]http://wwww.apache.org

each AST node is defined with a position 2.6. Because AST information is heavy compared with a lightweight position consisting of an offset and a length, CSeR stores and updates positions. Even when saving changes to a particular AST node in a file, the change is linked internally to the position of the AST node.

In short, when code is copied and pasted inside the IDE, positions from both the copied file and the pasted file are added to the CSeR database. For example, consider copying and pasting the class in 1. The positions saved for "HelloWorldApp" are shown in 3.2.1. A corresponding set of positions is also associated with the pasted file, "TestWorldApp."

The criteria for calculating positions are explained in detail in Sections 3.2.4, 3.2.6, and 3.2.5. It may seem surprising that positions must be calculated separately for two files that are clones. The reason is that they are not always exact textual clones after the IDE processes the paste. For example, when a Java file named "HelloWorldApp.java" is copied to "TestWorldApp.java," the editor changes occurrences of the class name "HelloWorldApp" inside TestWorldApp.java to "TestWorldApp." That automatic refactoring changes the positions.

1. package com. sample; [offset: 0, length: 19]
2. com[offset: 8, length: 3]
3. sample[offset: 12, length: 6]
4. ```
   public class HelloWorldApp {
   public static void main(  String[] args){
   System. out. println(‘‘Hello World’’);
   }
   ```
   [offset: 21, length: 130]
5. HelloWorldApp[offset: 34, length: 19]
6. ```
   public static void main(String[] args){
   System. out. println(‘‘Hello World’’);
   ```

```
}
```

[offset: 60, length: 89]

7. main[offset: 79, length: 4]
8. String[] args[offset: 84, length: 13]
9. String[offset: 84, length: 6]
10. args[offset: 93, length: 4]
11.
```
  {
 System. out. println(‘‘Hello World’’);
}
```

[offset: 99, length: 50]

12. System. out. println(“Hello World”); [offset: 109, length: 34]
13. System. out. println(“Hello World”)[offset: 109, length: 33]
14. System[offset: 109, length: 6]
15. out[offset: 116, length: 3]
16. println[offset: 120, length: 7]
17. “Hello World”[offset: 128, length: 13]

3.2.2 Correspondence

As mentioned above, positions from both the pasted file and the copied file are added to the database. CSeR tracks the changes that occur in the current file relative to the original file. Positions from the current and original files are stored in a bidirectional map. In other words, given a position in the current file, CSeR can find the corresponding position in the original file. Using the JDT API, a position can be converted to an AST node when such a node exists.

Consider a pair of clones, O (Original) and C (Current) defined in CSeR. Let p_1, p_2 *be positions such that* $p_1 \in O$ *and* $p_2 \in C$. *Cor is a function defined from O to C such that* $Cor(p_1) = p_2$ *and similarly* Cor' *is a function defined from C to O such that* $Cor'(p_2) = p_1$

The implementation of the correspondence relationship is shown in Figure 3.3. CheckPosition is a data structure that contains two positions: one from the current file and one from the original file. CheckPositions stores multiple CheckPosition values. Only selected positions are shown in the figure for clarity.

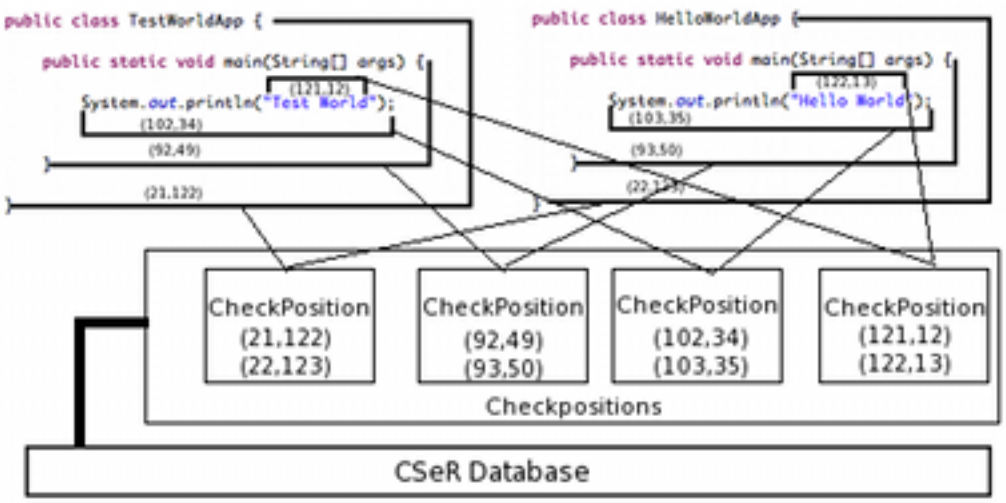

Figure 3.3: Implementation details of Correspondence Relationship

Why are there position updaters?2.7 Position updaters are listeners that respond to position-change events. Whenever an event changes positions, CSeR's database must be updated with the new information. Position updaters perform this task for CSeR.

This can be explained with the earlier example. Consider the operations shown in 3.2.2 being performed in the newly created file "TestWorldApp." All operations are assumed to occur in CSeREditor. After the operations, the class appears as shown in Code 2. The resulting position changes are explained in Table 3.14.

Operations

1. Select the text “Hello World” inside the TestWorldApp file.

2. Replace “Hello World” with “Test World”.

Code 2 The Test World code in Java.

```
package com. sample;

public class TestWorldApp {

 public static void main(String[] args) {

  System. out. println("Test World");

 }
}
```

Table 3.14: **Corresponding positions of HelloWorldApp and TestWorldApp**

No	Position Description	Original	Current	Recent
1	Entire package name - package com. sample	0, 19	0, 19	0, 19
2	First part package - com	8, 3	8, 3	8, 3
3	Second part package - sample	12, 6	12, 6	12, 6
4	Class declaration - public class Hello..	21, 124	21, 123	21, 122
5	Class name - HelloWorldApp	34, 13	34, 12	34, 12
6	Method declaration - public static voi..	54, 89	53, 89	53, 88
7	Method name - main	73, 4	72, 4	72, 4
8	Parameters - String[] args	78, 13	77, 13	77, 13
9	Parameter Type name - String	78, 6	77, 6	77, 6
10	Parameter name - args	87, 4	86, 4	86, 4
11	Method block - { System. out. println(..	93, 50	92, 50	92, 49
12	Expression statement - Syst..World”);	103, 34	102, 34	102, 33
13	Method invocation - Syst..elloWorld”)	103, 33	102, 33	102, 32
14	Simple name - System	103, 6	102, 6	102, 6
15	Simple name - out	110, 3	109, 3	109, 3
16	Simple name -println	114, 7	113, 7	113, 7
17	String literal - “Hello World”	122, 13	121, 13	121, 12

Table 3.14 shows the corresponding positions. The second column gives a short description of the AST node from the original file. The other three columns give positions. The third column specifies the position of the node in the original file, while the fourth and fifth columns give the position of the same node in the current file at different editing points. The

current position is before any edit, and the recent position is after the operation specified in 3.2.2. A few inferences from the table are given below.

Inferences from Table 3.14

1. The position of package name node ‘com. sample‘ remains unchanged in both the current and original files.

2. The current position in 2.1, before any edits, for the class declaration differs from the position in the original file by one character. This is due to the internal refactoring that took place inside TestWorldApp during the copy-and-paste operation, which changed occurrences of “HelloWorldApp” to “TestWorldApp.” Since the names differ by one character and there is only one occurrence of “HelloWorldApp” inside the class declaration, the length of the class declaration in the current file differs from the original file by one character.

3. The current position for the class declaration after the edit also differs from the class declaration before the edit by one character. This is because the operation removed “Hello World” (11 characters) and inserted “Test World” (10 characters), which is equivalent to removing one character.

3.2.3 Trap-Edit Approach

Every file opened inside the IDE editor is represented by a document 2.22 object. The document contains positions. Whenever the document changes, document listeners are activated and receive specific information about the change, including the position where the change occurred and the nature of the change, such as insert, delete, or replace.

Algorithm 1 describes how to find the smallest AST containing the change when the user edits inside a clone. The algorithm does not define “Compare” because its implementation varies by node type. For example, statements cannot be compared in the same way as identifiers. The functions and external library calls used in the algorithm are explained in 3.15.

Algorithm 1 TRAPEDIT - Calculate the editing AST

Require: positionOfEdit positionsOfCurrentFile positionsOfOriginalFile
Ensure: positionsOfCurrentFile, positionsOfOriginalFile not empty

```
1: for all position in positionsOfCurrentFile do
2:    if strictlycontains(position, positionofEdit then
3:       unFilteredPositions. add(position)
4:    end if
5: end for
6: ASTPosition ⇐ smallestPosition(unFilteredPositions)
7: currentNode ⇐ getASTNode(currentFile, ASTPosition)
8: originalPosition ⇐ findImageInCurrentFile(ASTPosition)
9: originalNode ⇐ getASTNode(parentFile, parentPosition)
10: if getType(currentNode) is Bodydeclaration then
11:    X ⇐ 1
12: else {getType(currentNode) is Block}
13:    X ⇐ 2
14: else
15:    X ⇐ 3
16: end if
17: Compare(currentNode, originalNode, X)
```

Understanding the TRAPEDIT Algorithm Some editing scenarios are explained below to clarify how the algorithm works using the information in Figure 3.3. Only selected positions are shown for clarity.

1. Consider adding a space before the "Test World" string. The position of the edit is "121, 1." Lines 1-5 of the algorithm add all four containing positions to unFilteredList because each one contains the edit position. In the next step, line 6 calculates the smallest position. This position is not "121, 12," but "121, 13," because the position updaters in CSeREditor have already updated the database.

   Position updaters update all occurrences of "121, 12" to "121, 13" throughout the CSeR database. Eclipse calls position updaters before document listeners, so all positions observed by document listeners are already updated. Once the smallest position is known, CSeR calculates the current node, "Test World." Then, using Table 3.14, it calculates the original position and the original node.

   Once both nodes are known, CSeR checks the node type. In this case, the type is

Table 3.15: **Functions Used In TRAPEDIT Algorithm**

Function Name	Input	Output	Description
Contains	Position1, Position2	true or false	Is Position2 inside Position1
smallestPosition	List of positions	Position	Find the smallest Position
getASTnode	File, Position	AST node	Find AST node from Position
FindImageInCurrentFile	Position	Position	Find corresponding original position
getType	AST node	Type	Return the type of the AST node

neither a block nor a type declaration, so control flows to the default case where the type is "StringLiteral." A message is calculated using the current node and original node, and then displayed in the editor using annotation markers. More detail about this comparison is provided in Section 3.2.6. Figure 3.4, change 1, shows the representation of this change in CSeREditor.

2. Consider pasting "doA();" after the print statement inside the method. Lines 1-5 of the algorithm add two positions to unFilteredList: the block node and the class declaration node. These are the only nodes that contain the edit position. The smallest one, the block node "92, 49," is selected. The current block node and the original block node are then calculated as described above and passed to the appropriate comparator. More detail about this comparison is provided in Section 3.2.4. Figure 3.4, change 2, shows the representation of this change in CSeREditor.

3. Consider pasting a declaration for "doA" after the "main" method. Lines 1-5 of the algorithm add only one position to unFilteredList: the class declaration node. It is the only node that contains the edit position, so it is also the smallest node. The class declarations of TestWorldApp and HelloWorldApp are then passed to the appropriate comparator. More detail about this comparison is provided in Section 3.2.5. Figure 3.4, change 3, shows the representation of this change in CSeREditor. Adding a field

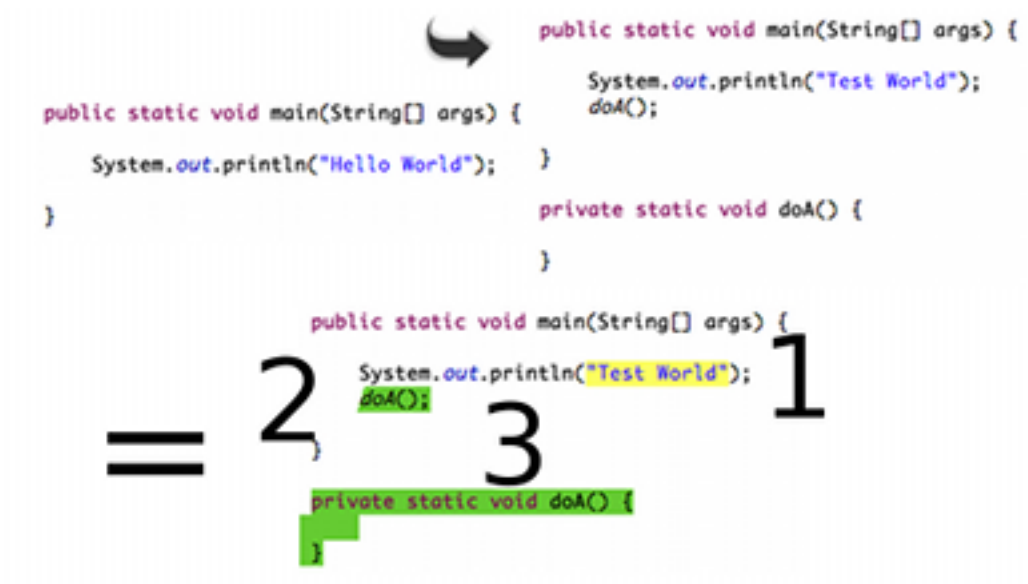

Figure 3.4: Understanding TRAPEDIT Algorithm with TestWorldApp

is handled similarly.

Using the TRAPEDIT algorithm, every edit operation results in an anchor AST, which can be a TypeDeclaration, a Block, or another node. When the anchor AST is a TypeDeclaration, CSeR compares body declarations. When the anchor AST is a Block, CSeR compares statements. In all other cases, CSeR considers expression nodes.

The comparison approaches for statements, body declarations, and expressions are explained in the following sections.

3.2.4 Statements

By design, when the anchor AST, the smallest AST containing the edit position, is a Block node, CSeR compares statements. Returning to the HelloWorldApp example, consider adding a new statement, "int i=0;," as the first line inside the main method.

Insertion can be performed by pasting the entire line, typing character by character, or combining the two. As a design decision, CSeR does not show any change until the AST is complete. For example, while the user is typing "int i=0;," CSeR does not calculate changes

until the final semicolon is typed. Position updaters are not AST-aware, however, so they continue to maintain position correspondence for every edit.

Once the AST is complete, CSeR starts processing. It first fetches the block from the current file, finds the corresponding AST (Listing 11) in the other clone, and then compares the two. If they do not match, CSeR shows a change. In the TestWorldApp example, the block in the current file is shown in Code 3. The question in this section is therefore how to compare two AST nodes of type Block: 3 and 11.

Code 3 AnchorAST after inserting a new statement in TestWorldApp.

```
{
int i=0;
System. out. println("Test World");
}
```

For Block comparison, CSeR breaks each Block into non-overlapping statements and expressions, and then compares them individually.

How to find non-overlapping statements and expressions in a Block Using statements exactly as defined by the JLS3 [4] grammar can produce overlapping statements, which the current design does not support. CSeR therefore uses visitors that visit statement nodes and return a list of non-overlapping statements and expressions. The nodes visited by the Statement visitor are listed in Table 3.17. For example, only the expressions are visited for a "ForStatement," because the statements inside the "for" loop are visited by other visitor nodes. Figure 3.5 shows how a simple for loop is parsed by the Statement visitor.

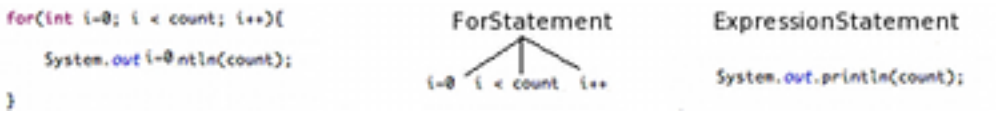

Figure 3.5: Statement visitor visiting a for loop

The output of TestWorldApp and HelloWorldApp after the StatementVisitor is shown in Table 3.16. Again, CSeR does not convert the positions to AST nodes for storage; it

[4] http://java.sun.com/docs/books/jls/third_edition/html/j3IX.html

Table 3.16: **Statements given from the visitor for TestWorldApp**

Original File	Current File
System. out. println("Test World");	int i=0; System. out. println("Test World");

keeps them as positions for further processing. The table shows AST nodes only for clarity.

Table 3.17: **Statements visited by Statement Visitor**

No	Statement Type	Statement	Expression
1	ReturnStatement	√	
2	ThrowStatement	√	
3	BreakStatement	√	
4	ContinueStatement	√	
6	ExpressionStatement	√	
7	AssertStatement	√	
8	VariableDeclarationStatement	√	
9	TypeDeclarationStatement	√	
10	ConstructorInvocation	√	
11	SuperConstructorInvocation	√	
12	ForStatement		√√√
13	EnhancedForStatement		√√
14	WhileStatement		√
15	DoStatement		√
16	SwitchStatement		√
17	LabeledStatement		√

Initial deleted and inserted node positions *In Figure 3.6, let O and C represent sets of positions such that $p_1{}', p_2{}', \cdots, p_N{}'$ are positions in the current file and $p_1, p_2, \cdots, p_M$ are positions in the original file. Let P_c and P_o be the positions of the current and original Blocks, respectively. $P_c \subset p'$ and $P_o \subset p$. An initial inserted node position is any position p_i for which $Cor'(p_i')$, the inverse correspondence function, is not defined. Similarly, an initial deleted node position is any position p_j for which $Cor(p_j)$, the correspondence function, is not defined.*

Algorithm CMPSTATEMENTS 2 describes the statement comparison algorithm. Lines 1-2 fetch the statement positions from both the current file and the original file. Line 3

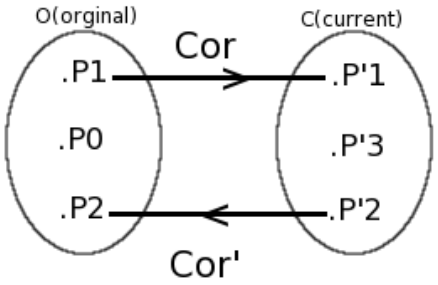

Figure 3.6: Correspondence Relation

finds the corresponding position of each original position. Line 4 computes the difference between those positions and the original positions. Finally, "finalPosition" may contain two types of positions: those that exist in the original positions but not in the current positions, which are deleted positions, and those that exist in the current positions but not in the original positions, which are newly inserted positions. CSeR marks these as initial inserted and deleted nodes. The functions used in CMPSTATEMENTS are explained in Table 3.18.

Why are inserted and deleted positions in Algorithm CMPSTATEMENTS not final? Initial inserted and deleted node positions are positions without correspondence. As shown in Figure 3.6, $P_3{}'$ and P_0 do not have corresponding positions on the other side. This can happen when a node is inserted or removed, but it can also happen in other situations. For example, it can happen during an undo operation.

Consider a statement that is removed and then typed back as the same statement. When an AST node is removed from a file, the position linked to it is removed as well. The associated position updaters are also removed. Typing or pasting that AST back does not restore either the original position or the position updaters. Therefore, a previously deleted

Table 3.18: **Functions Used In CMPSTATEMENTS Algorithm**

Function Name	Input	Description
$getStatementPositions$	ASTnode	find statements inside ASTnode
$findImagesInCurrentFile$	PositionList	Find corresponding PositionList
$diff$	two PositionLists	$(List1 \cup List2) - (List1 \cap List2)$
$IsExist$	Position	returns whether the position really exist

Algorithm 2 CMPSTATEMENTS - Comparing statements

Require: currentBlock originalBlock
Ensure: currentBlock, originalBlock not null

```
1: currentPositions ⇐ getStatementPositions(currentBlock)
2: originalPositions ⇐ getStatementPositions(originalBlock)
3: orgPosWithCorr ⇐ findImagesInCurrentFile(originalPositions)
4: finalPositions ⇐ diff(originalPositions, orgPoswithCorr)
5: for all position in finalPositions do
6:    if IsExist(position) then
7:       print "position may be deleted"
8:       tmpDeletedPositions. add(position)
9:    else
10:      print "position is new"
11:      tmpInsertedPositions. add(position)
12:   end if
13: end for
14: PROCESSINSDELNODES(tmpDeletedPositions, tmpInsertedPositions)
```

node can later appear as an inserted node.

When deleted nodes are inserted back, inserted and deleted nodes must be processed further to determine whether they are final inserted or deleted nodes. Algorithm PROCESSINSDELNODES performs this step for CSeR.

PROCESSINSDELNODES starts by assuming that the final deleted and inserted positions are the same as the temporary deleted and inserted positions. It then iterates over insertedPositions and, for each inserted position, iterates over deletedPositions. For each pair, it checks whether the inserted node and deleted node match (line 5). The matching algorithm, Algorithm 4, uses the default ASTMatcher shipped with JDT. Once a match is found, the nodes may be either identical or similar.

What are "identical" statements? To define identical nodes in CSeR, we introduce two additional terms for statement comparison: upper statement and lower statement. The upper statement is the statement immediately above the current statement, and the lower statement is the statement immediately below it. For example, "int i=0;" is the upper statement for "System.out.println("Test World");," and the lower statement for that print statement is null. Similarly, the upper statement of "int i=0;" is null.

Algorithm 3 PROCESSINSDELNODES- Process Insert/Delete Nodes

Require: tmpInsertedPositions tmpDeletedPositions
Ensure: tmpInsertedPositions, tmpDeletedPositions is not empty

```
1: finalInsertedPositions ⇐ tmpInsertedPositions
2: finalDeletedPositions ⇐ tmpDeletedPositions
3: for all position1 in tmpInsertedPositions do
4:   for all position2 in tmpDeletedPositions do
5:     if AREMATCH(position1, position2) then
6:       print "positions may be moved or the undo"
7:       if AREIDENTICAL(position1, position2) then
8:         print "position1 is same as position2- Undo Operation"
9:       else
10:        print "position1 is moved to position2"
11:        remove ⇐ true
12:      end if
13:    else {ISUPDATE(position1, position2)}
14:      print "position2 is updated to position1"
15:      remove ⇐ true
16:    end if
17:    if remove then
18:      finalDeletedPositions. remove(position2)
19:      finalInsertedPositions. remove(position1)
20:    end if
21:  end for
22: end for
```

CSeR considers a statement in the current file to be identical to a statement in the original file if the statements match and their neighbors also match. Matching neighbors means that the upper statement of the current statement matches the upper statement of the original statement, and the same condition holds for the lower statements. This is explained in Algorithm 5. In short, statement-level comparison considers the order in which statements occur. Undo operations are a common case of identical nodes.

CSeR assumes that a statement in the current file with no correspondence is identical to another statement in the original file with no correspondence if the statements match and have matching neighbors.

What are "moved statements"? If a newly inserted statement in the current file matches a deleted statement in the original file but is not identical, then the statements

Algorithm 4 AREMATCH - Utility for AST node match

Require: Position1, Position2, File1, File2
Ensure: Position1, Position2, File1, File2 not $NULL$

```
1: ASTnode1 ⇐ getASTNode(File1, Position1)
2: ASTnode2 ⇐ getASTNode(File2, Position2)
3: if defaultJDTASTMatch(ASTnode1, ASTnode2) then
4:    print "Nodes match"
5: else
6:    print "Nodes do not match"
7: end if
```

Algorithm 5 AREIDENTITCAL - Checking Identical node, Undo Operations

Require: Position1, Position2, PositionList1, PositionList2
Ensure: Position1, PositionList1 ∈ originalFile AND Position2, PositionList2 ∈ current-File

```
1: if AREMATCH(before(position1), before(position2)) then
2:    if AREMATCH(after(position1), after(position2)) then
3:       print "Nodes are identical"
4:    end if
5: else
6:    print "Nodes are not identical"
7: end if
```

occur in different orders in the current and original files. CSeR treats this as a move operation. Algorithm PROCESSINSDELNODES 3 detects moved statements for CSeR.

What are "updated" statements? If a newly inserted statement in the current file has the same type as a deleted statement, and its Levenshtein distance 2.21 is below the configured value, but the two statements do not match, then CSeR defines it as an updated statement. Algorithm PROCESSINSDELNODES 3 detects updated statements for CSeR.

An example with statement level changes This example is taken from the JDT UI [5] project. The classes JavaDocContext and JavaContext appear to be very similar. We used CSeR to modify a method named "canEvaluate" in JavaDocContext so that it matched the corresponding method in JavaContext. The changes that appear in CSeR are shown in Figure 3.7. The change marked "3" shows the insertion of a new "IfStatement," the change

[5] http://www.eclipse.org/jdt/ui/index.php

Algorithm 6 ISUPDATE - Checking the current Position is an update of Original Position

Require: Position1, Position2, currentFile, OriginalFile
Ensure: Position1, Position2 not $NULL$

```
1: ASTnode1 ⇐ getASTNode(Position1, currentFile)
2: ASTnode2 ⇐ getASTNode(Position2, originalFile)
3: if getType(ASTnode1)== getType(ASTnode2) then
4:    if LevenshteinDistance(ASTnode1, ASTnode2) ≤ CONFIGVALUE then
5:       print "Position1 is an update of Position2 "
6:    end if
7: else
8:    print "Position1 is not an update of Position2"
9: end if
```

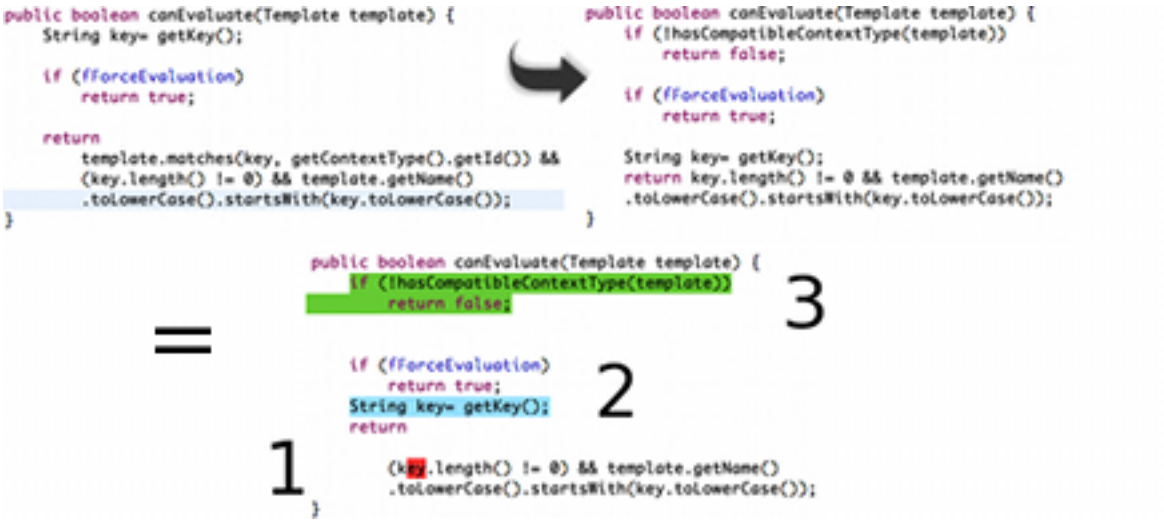

Figure 3.7: Statement changes in CSeR Editor using "canEvaluate" method in JavaContext and JavaDocContext

marked "2" shows a move operation, and the change marked "1" shows a delete operation.

3.2.5 Body Declarations

In the TRAPEDIT algorithm, when the anchor AST is a TypeDeclaration node, CSeR must compare body declarations 2.10. Consider introducing a change in TestWorldApp. For Code 4, the anchor AST is the entire TestWorldApp class.

As with statements, insertion can be performed by pasting a code block or typing it. CSeR does not show any change until the AST is complete. For example, while typing "private int i=0;," CSeR does not calculate a change until the final semicolon is typed.

Position updaters still operate for every edit, regardless of whether the edit occurs in a TypeDeclaration, a Block, or another node.

Once the AST is complete, CSeR starts processing. It first fetches the class declaration from the current file, finds the corresponding AST from the original file, and compares the two. If they do not match, CSeR shows a change.

Code 4 The Test World with new field.

```
package com. sample;

public class TestWorldApp {

 public static void main(String[] args) {

  System. out. println("Test World");

 }
 private int i=0;//inserted new line.
}
```

Once CSeR has the type declaration, it must find the body declarations that occur inside the class but outside methods. It then performs a field-level comparison to find the difference. An important distinction from statement comparison is the order of fields. The order of fields does not change the meaning of a Java class. Another difference is how fields are calculated in the two classes. Here too, CSeR needs non-overlapping field positions.

How to find body declarations According to JLS3, body declarations in JDT can be any of the nodes listed in Table 3.19. CSeR uses visitors that visit these nodes. The node from the current file and the corresponding node from the clone are both passed through the visitor, which returns the list of body declarations. The output of TestWorldApp is shown in Code 5.

How are body declarations compared? As mentioned earlier, body declarations 2.10 differ from statements because they do not necessarily have to be neighbors to be identical. The same CMPSTATEMENTS algorithm can be used to find the initial inserted and

Table 3.19: **Body declarations specified by JLS3 grammar**

No	Statement Type
1	ClassDeclaration
2	InterfaceDeclaration
3	EnumDeclaration
4	MethodDeclaration
5	ConstructorDeclaration
6	FieldDeclaration
7	Initializer
8	EnumConstantDeclaration
9	AnnotationTypeDeclaration
10	AnnotationTypeMemberDeclaration

Code 5 Body declarations given from the visitor for TestWorldApp

1. ```
   public static void main(String[] args){
   System. out. println("Test World");
   }
   ```

2. private int i=0;

deleted fields, but after those fields are calculated, the PROCESSINSDEL algorithm must be slightly modified by removing the neighbor-checking step. Apart from this change, body declarations follow the same processing as statements.

Algorithm 3 (PROCESSINSDELNODES) is slightly modified to manage body declarations and is named PROCESSINSDELBDNODES 7.

An example with BodyDeclaration changes This example is also taken from the JDT UI [6] project. The classes NewAnnotationWizardPage and NewClassWizardPage appear to be very similar. We used CSeR to modify the body declarations inside NewClassAnnotationWizardPage to convert it to NewAnnotationWizardPage. The changes that appear in CSeR are shown in Figure 3.8. The change marked "2" shows the insertion of a new field, and the change marked "1" shows a delete operation. The deleted nodes are shown when the mouse hovers over the marker.

[6] http://www.eclipse.org/jdt/ui/index.php
   ```

Algorithm 7 INSDELBDDLS - Comparing Body Declarations

Require: tmpInsertedPositions tmpDeletedPositions
Ensure: tmpInsertedPositions, tmpDeletedPositions is not empty

```
1: finalInsertedPositions ⇐ tmpInsertedPositions
2: finalDeletedPositions ⇐ tmpDeletedPositions
3: for all position1 in tmpInsertedPositions do
4:    for all position2 in tmpDeletedPositions do
5:       if AREMATCH(position1, position2)  then
6:          print  "position1 is same as position2- Undo Operation"
7:          finalDeletedPositions. remove(position2)
8:          finalInsertedPositions. remove(position1)
9:       end if
10:   end for
11: end for
```

Figure 3.8: BodyDeclaration changes in CSeR Editor using NewClassWizardPage and NewAnnotationWizardPage

Figure 3.9: Node changes in CSeR Editor using FieldTag and FileNameTag

3.2.6 Nodes

This is the default case when the anchorAST is neither TypeDeclaration nor Block.

A careful analysis shows that in this case, the anchor AST is one of the values listed in Table 3.20. For Code 6, it is a StringLiteral node. In this case, insertion can again be performed either by pasting a code block or by typing it. CSeR assumes that the developer eventually brings the AST back to a correct state.

Once the AST is complete, CSeR starts processing. It first fetches the node from the current file, here "Test World," and then finds the corresponding AST from the other clone, here "Hello World." Finally, it compares the two. Since they are not the same, CSeR shows a change.

What if the user is editing in the margin of an ASTNode? In this case, the TRAPEDIT algorithm returns the parent of the edited node. In the example above, it returns "System.out.println("Test World")." CSeR then splits this node into smaller units, in the same way it divides a Block into statements, and processes each unit separately.

Code 6 The Test World with an update.

```
package com. sample;

public class TestWorldApp {

 public static void main(String[] args) {

  System. out. println("Test World");
  // String literal changed from Hello World to Test World


 }
 private int i=0;//inserted new line.
}
```

How to find nodes As mentioned earlier, CSeR uses visitors for the nodes specified in Table 3.20. Passing the nodes through these visitors produces the smaller non-overlapping units that must be compared.

Table 3.20: **Nodes as a result of Test Cases by JLS3**

No	Node Type
1	ImportDeclaration
2	MethodInvocation
3	PackageDeclaration
4	SimpleName
5	SingleVariableDeclaration
6	StringLiteral
7	NumberLiteral
8	VariableDeclarationFragment
9	ArrayInitializer

How are nodes compared? Node comparison is implemented in the same way as Block comparison, with one difference in the implementation of the $diff$ function. Consider the example above, where the user modifies the boundary of 'System.out.println("Test World").' There is already a change at "Test World." If CSeR used the same $diff$ implementation, that node would not appear in the list of initial deleted or inserted nodes because it already has correspondence. Therefore, the important change in node comparison is the modified

diff implementation. Algorithm 8 shows how this is done. In the algorithm, the function *undid* returns all nodes that have correspondence on both sides.

Algorithm 8 NODEDIFF - diff Implementation for nodes

Require: currentPositions originalPosWithCorr originalPositions

```
1: initialdeleteInsertPositions ⇐ diff(currentPositions,orgPoswithCorr)
2: positionWithCorrespondence ⇐ undid(currentPositions,orgPosWithCorr)
3: for all position in positionWithCorrespondence do
4:    if ∄match(getASTNode(currentPositions(position),getASTNode(originalPositions(position))))
      then
5:       print "position has to included "
6:       initialdeleteInsertPositions. add(position)
7:    end if
8: end for
9: RETURN initialdeletePositions
```

An example with Node changes This example is also taken from the Javalobby Community Platform project [7]. The classes FieldTag and FileNameTag appear to be very similar. We used CSeR to modify the only different node inside FieldTag to convert it to FileNameTag. The changes that appear in CSeR are shown in Figure 3.9. The change marked "1" shows the node update, and the old node is shown when the mouse hovers over the marker.

3.3 Sample Scenarios

3.3.1 Conditional Statements

Five different cases of conditional statements are shown below.

3.3.2 Arguments

Parameters and arguments are treated the same way. An example of a parameter change is shown in Figure 3.15.

[7] http://www.ohloh.net/p/ui/gotjava

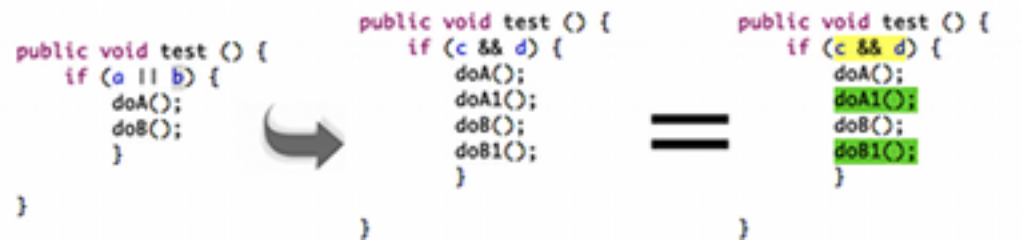

Figure 3.10: Conditional Statement Case 1 : Update Expression

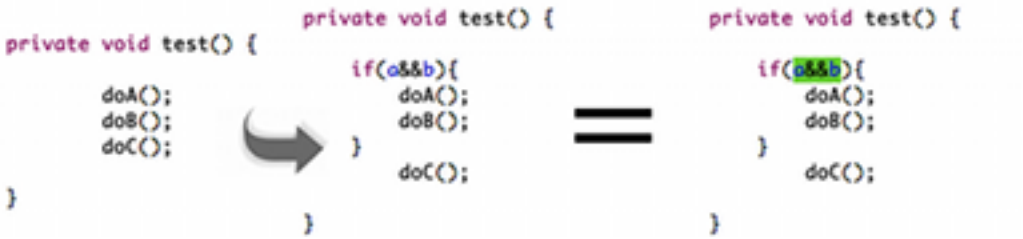

Figure 3.11: Conditional Statement Case 2 : Insert condition

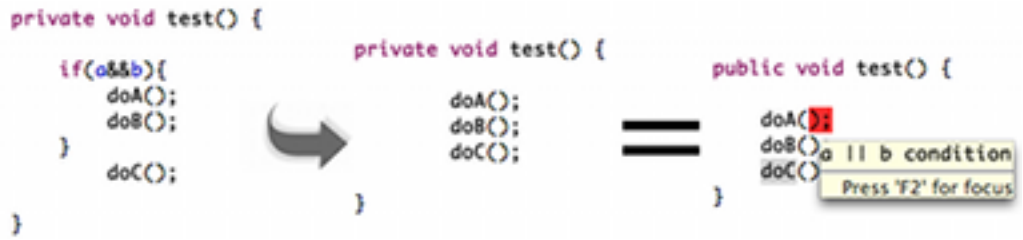

Figure 3.12: Conditional Statement Case 3 : Remove Condition

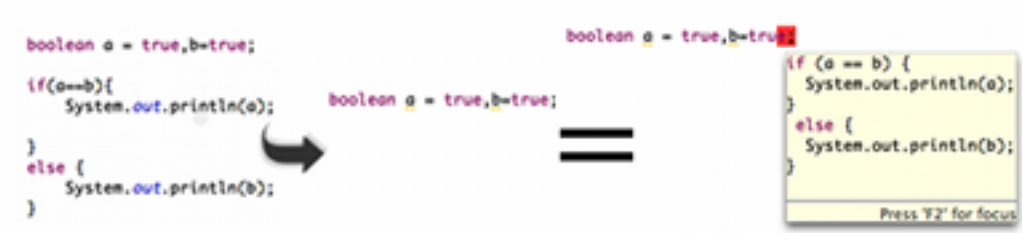

Figure 3.13: Conditional Statement Case 4 : Delete An If Statement

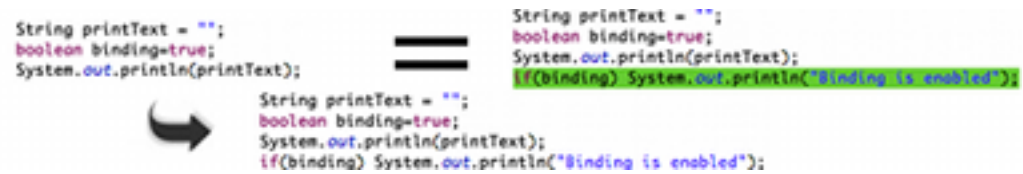

Figure 3.14: Conditional Statement Case 5 : Inserting A Complete If Statement

Figure 3.15: Inserting new arguments

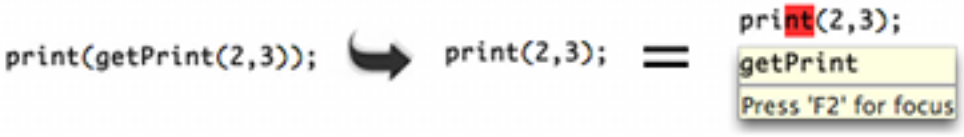

Figure 3.16: Flattening arguments

3.3.3 Array Initializer

An example of modifying an array initializer statement is shown in Figure 3.17.

3.3.4 Comments

An example of commenting code is shown in Figure 3.18.

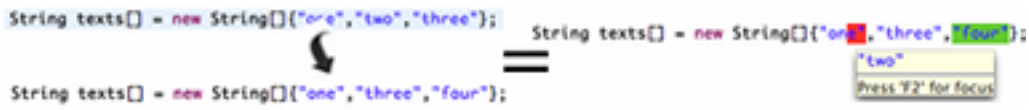

Figure 3.17: Modifying an array initializer

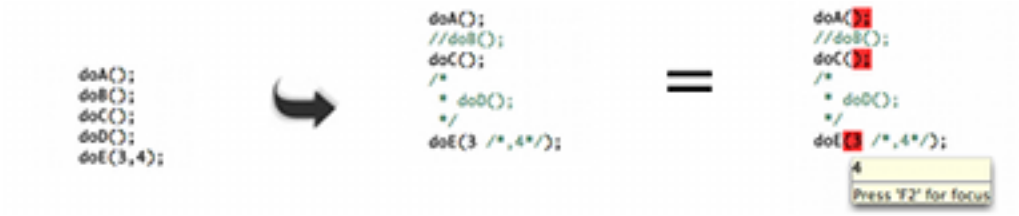

Figure 3.18: Commenting source code

Chapter 4

Validation

4.1 Robustness

This section analyzes the robustness of CSeR. Because CSeR adds features to an existing editor, we must ensure that the existing features of the editor are not affected. Eclipse provides many editor features, so the most practical way to validate compatibility is to use CSeR together with other Eclipse tools. We performed almost 50 test cases to ensure that existing features were not affected.

These experiments indicate that the tool works in normal scenarios. However, a tool is robust only if it can also handle edge cases. To evaluate those cases, we must analyze different kinds of user edits, as explained in the next section.

4.1.1 User Editing

As explained in the requirements chapter, editing a source file involves both actions and goals. The same goal can be achieved through different actions. A tool is more robust when it depends on the goal rather than on the particular action used to reach that goal. Because CSeR compares code during user edits, actions still matter in practice. This section therefore considers different types of user edits. [17] contributed significantly to this classification.

Table 4.1 shows different kinds of user edits and the level of CSeR support for each. A

Table 4.1: **Analysis Of User Edits in CSeR**

No	Type	Goal Description	Action Description	Implemented	References
1	Names	Creating a name	Paste or Type	√	Fig 3.15
2		Replacing part	Paste or Type	√	Fig 3.3
3		Correcting typos	Backspace and Type	√	Fig 3.3
4		Replacing name	Backspace, Type or Paste	√	Fig 3.10
5		Removing name	Backspace, Delete or Type	√	Fig 3.12
6		Splitting a name	Type in between	√	Section 4.1.2
7		Renaming	Using tools	×	
8	Lists	Creating a new list	Type or Paste	√	
9		Inserting a new element	Type or Paste	√	Fig 3.15
10		Removing an element	Delete, Type or Backspace	√	Fig 3.17
11		Moving an element	Cut and Paste or Copy Paste and Delete	√	Fig 3.7
12		Removing entire list	Backspace or Delete	√	Fig 3.13
13		Flattening a list inside a list	Backspace or Delete	√	Fig 3.16
14	Expressions	Inserting a new expression	Type or Paste	√	Fig 3.11
15		Updating an expression	Type or Paste	√	Fig 3.10
16		Removing an expression	Delete, Type or Backspace	√	Fig 3.12
17		Moving an expression	Cut and Paste or Copy Paste and Delete	√	
18	Comments	Comment code	Type Line or Block comment	√	Fig 3.18
19		Creating annotations		×	
20		Inside expressions	Type Block comments	√	Fig 3.18
21	Keywords	Insert keyword		×	
22		Update keyword		×	
23		modify keyword		×	

name refers to anything that is not a Java keyword, such as a method name, class name, or variable name. Lists correspond to structures that appear between delimiters, such as {} around statement lists and () around parameter lists. List elements are delimited by characters such as semicolons between statements and commas between parameters. Actions 2.14 and goals 2.15 are explained in the definitions chapter.

4.1.2 Special Scenarios

Type Change Edit As a design detail, two important operations occur during every keystroke: AST comparison and position updating. Position updating keeps the CSeR database aligned so that every tracked position corresponds to an AST node. The complexity of AST comparison depends on the size of the AST, while the complexity of position updating depends on the number of tracked positions. Since the number of tracked positions can be several hundred, updating a single tracked position must remain lightweight. Therefore, CSeR does not perform AST parsing during position updates, which makes position updating AST-unaware. In short, position updating is AST-unaware, while AST

comparison is AST-aware.

In Figure 4.1, the user types the method "doB();" within the bounds of "doA();." Because position updating is AST-unaware, it treats the edit as an update to the method and includes the position of "doB();" along with "doA();." The CSeR database then contains $(x, 6) \leftrightarrow (x, 12)$, where x is an offset value. As mentioned earlier, CSeR expects every tracked position to produce an AST for comparison. In this case, the second position does not produce a single AST because it contains two AST nodes instead of one. This is called a type-change edit, because the edit changes one method into two methods or otherwise changes the type of the AST node.

This issue can be solved by removing the corrupted position from the CSeR database. There are two places where this can be done. The first is during position updating. To do this without sacrificing performance, CSeR must find the position updaters involved in the edit and check whether their positions still produce AST nodes. If not, the position should be removed from the CSeR database. The second place is during AST comparison: if a tracked position does not produce an AST, it should be removed from the database.

In the example above, after the user finishes editing, the tracked position corresponding to "doA();" is removed. When CSeR calculates the deleted and inserted nodes, it finds two inserted nodes and one deleted node. Since "doA();" appears in both places and has identical neighbors, the statements are considered the same and no marker is added for that node. An insert marker is added only for "doB();."

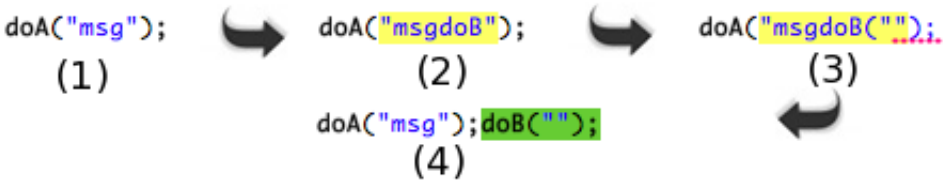

Figure 4.1: Type Change Edit

Unexpected Consistent State CSeR does not compare until a valid AST exists, but a valid AST can be reached before the user reaches the intended final state. Consider Figure

4.2. The user wants to insert "int j=0" between the two given lines. As shown in the figure, a consistent state is reached at step 4, with the two statements "int i=0;" and "int j=i++." On comparison, CSeR finds that the second statement is new and that the statement "i++" was deleted. It therefore adds a red marker for the deletion and a green marker for the new statement. Once the user finishes typing, CSeR recalculates the changes and displays the result shown in step 5.

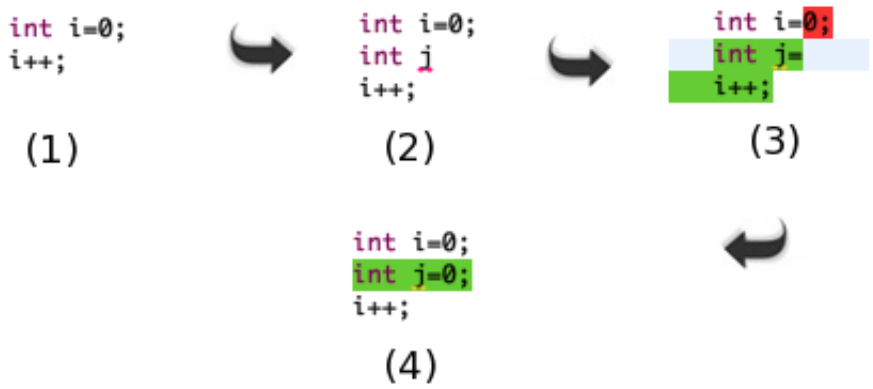

Figure 4.2: Unexpected Consistent State

4.2 Comparison with Existing Tools

This section analyzes existing tools and shows why tools that appear to replace CSeR cannot completely or accurately address the problem considered here. Most of the tools analyzed are source-differencing tools. The general distinction is that CSeR is not simply another tool for finding differences between two source files at one point in time. Instead, it calculates differences between two files that were identical at some earlier point, and it calculates those differences incrementally as edits are made.

4.2.1 Text Based Tools- diff, Compare-Editor, Version Editor

Many text-based tools are available. Most use the diff algorithm, which is based on solving the LCS (Longest Common Subsequence) problem. Because this approach was developed

for text files, it has clear disadvantages when applied to Java source code. The algorithm cannot distinguish source code from comments. It also does not understand that the order of fields and methods does not change the meaning of a Java class.

Another disadvantage occurs when two methods are interchanged. Suppose the first method is five lines long and the second is ten lines long. A diff-based tool may sacrifice the shorter method and show correspondence between the longer methods in both classes. In short, correspondence between smaller methods or fields can be lost in move operations.

The Eclipse Compare Editor uses an objectified version of diff[1]. Even though it can identify Java tokens and show token-level differences, it still inherits limitations from its text-based algorithm. Moved methods and fields may not be identified correctly, and the tool may show correspondence between two methods even when a better correspondence exists elsewhere.

For example, consider JavaContext and JavaDocContext. Analysis of the two classes shows that some correspondences are wrong. The method "evaluate" on the right has a corresponding method "evaluate" on the left, but the Compare Editor treats it as a deleted method. Meanwhile, "getCharacterBeforeStart" is removed from JavaContext and has no correspondence on the other side. Figure 4.3 shows the incorrect correspondence produced by the Compare Editor. This happens because the "evaluate" method is moved in the second class.

Version-Editor [1] provides tight integration between revision history and the editor. It makes version data available in context, showing changes made so far, newly added lines, creation time, change status, and other details. The changes are calculated against versions in the repository. However, the comparison is line-based, so the limitations of text-based tools still apply.

[1] http://en.wikipedia.org/wiki/Diff

4.2.2 AST Based Tools- Breakaway and ChangeDistiller

Breakaway [5] supports generalization tasks. It identifies detailed structural correspondence between classes and prints the calculated correspondence to the console. ChangeDistiller [8] is intended to find changes between multiple versions of code. It compares trees using a change-detection algorithm for hierarchically structured information, producing an edit script that can convert the original version of a tree into the modified one.

Both tools compare the ASTs of two source files. However, neither tool exactly addresses the problem considered here. ChangeDistiller is designed to compare different versions of a single source file, so it expects a higher level of similarity, such as the same class name. It is therefore less useful for comparing two different classes. Breakaway, on the other hand, is used for generalization and expects classes to be similar. Both tools differ from ordinary source-comparison tools because their comparison is AST-based.

"Level mismatch" [5] is a common error in tree-comparison algorithms. It occurs when a statement node is not found to correspond to another statement even though the statements exist at different AST levels. For example, one statement may be directly inside a method declaration while the other is inside an if condition within a method declaration. Figures 3.11 and 3.12 show how CSeR handles such cases. CSeR can keep the correspondence between statements even when their AST levels change.

Compared with both tools, CSeR has an advantage because it begins tracking changes when the classes are identical and then tracks changes continuously. The changes are tracked based on the user edit and the AST being edited. For example, if the parameters in a method declaration are reordered or modified so that the signatures no longer correspond, ChangeDistiller may fail to find the correspondence because it does not consider the state-

Figure 4.3: Wrong Correspondence in Eclipse Compare Editor

ments inside the method when finding correspondence between methods. This can produce poor results.

Breakaway may fail to consider neighbors. For example, when two classes each have two fields of the same type, Breakaway may show incorrect correspondence because it does not consider the order or neighbors of each field. Breakaway may also fail to consider statements inside a method when comparing method declarations. CSeR, by contrast, considers neighbors when comparing statements. In CSeR, a statement in the current file whose correspondence is undefined is considered the same as a statement in the original file only if the statements match and have matching neighbors.

Breakaway applies a greedy algorithm and takes the first corresponding element whose similarity is greater than its threshold value. Conceptual disconnect errors, described in Section 5.1.2 of [5], occur when common elements such as String, =, and "" provide enough similarity to exceed the threshold. Because Breakaway does not search for the best match, it can produce "Unordered mismatch" and "Ordered mismatch" errors [5]. These errors are not relevant to CSeR because of its incremental design.

ChangeDistiller extends the algorithm by Chawathe et al. [3] for extracting changes in hierarchically structured data. It is limited in finding an appropriate number of move operations, particularly for parameter-ordering and statement-ordering changes. As noted in the limitations section of [8], parameter changes can affect the ability to find overall class correspondence. Figure 3.15 shows how CSeR handles these changes. Parameter changes are calculated when the user edits inside the parameters, while statement changes are calculated when the user edits inside a method, so these two kinds of changes are independent.

4.2.3 Clone Detectors

A small experiment was conducted to study clones reported by clone detectors. The clone detector used in the study was CCFinder[2], with its default configuration. All files used in the examples are relevant because they are later used in the CSeR experiment described in

[2] www.ccfinder.net

Section 4.3.1. The results are shown in Table 4.2. Here we consider five clone groups. LOC refers to lines of code. Missed LOC refers to lines that should have correspondence but were missed by CCFinder. #Clones is the number of clones detected by CCFinder. Missed LOC does not include lines that appear in only one file, nor does it include spaces or comments; it counts corresponding lines missed by CCFinder.

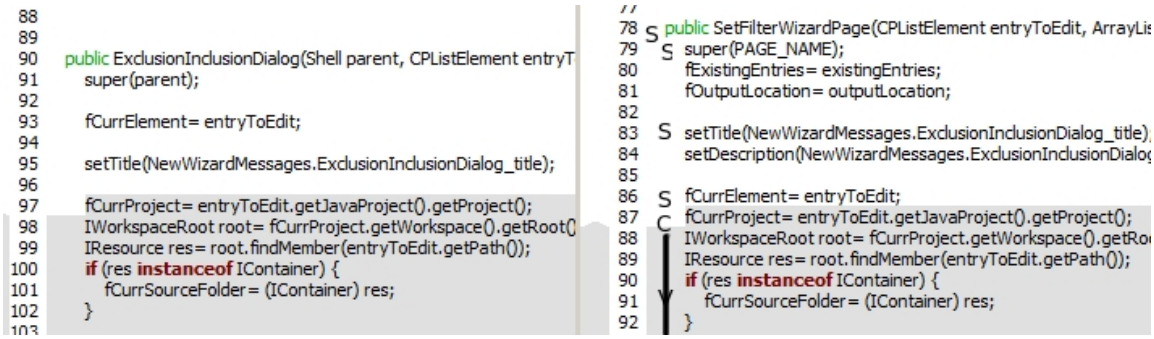

Figure 4.4: Calculation of Missed LOC

The calculation of "Missed LOC" is shown in Figure 4.4. The figure is taken from the CCFinder view. For the file SetFilterWizardPage, the clones are from ExclusionInclusionDialog. Lines marked "S" are treated as skipped lines, while lines marked "C" are corresponding lines calculated by CCFinder. The other lines exist only in one of the files. In the figure, comparison is performed with respect to the first file on each row.

The ideal result for the experiment would be a clone detector finding a single clone without any missed lines. We did not observe that result in the experiment, which involved five clone groups and 14 files. In most cases, CCFinder found clones for only part of the class, as shown by the #Clones value in Table 4.2. Even when only one clone was found, as in clone group 5, there were skipped lines, showing that the clone did not include the entire class.

In short, clone detectors fail to provide a complete high-level picture. Even the correspondence provided for code blocks is incomplete or inaccurate. Some correspondences are left out by clone detectors. This small experiment shows that class correspondence can become worse when method declarations and fields move, which is not an uncommon situation.

Table 4.2: **Analysis Of Clones Using CCFinder**

No	File Name	LOC	Missed LOC	#Clones
1	**SetFilterWizardPage**	340		
	ExclusionInclusionDialog	327	54	5
2	**NewClassWizardPage**	292		
	NewAnnotationWizardPage	142	24	2
	NewEnumWizardPage	146	5	2
	NewInterfaceWizardPage	133	5	2
3	**NewClassCreationWizard**	96		
	NewAnnotationCreationWizard	96	0	2
	NewEnumCreationWizard	96	6	2
	NewInterfaceCreationWizard	96	6	2
4	**CleanUpPreferencePage**	60		
	CodeFormatterPreferencePage	62	0	1
5	**CodeStylePreferencePage**	129		
	CodeAssistPreferencePage	98	6	1

4.3 Demonstration of Usefulness

The usefulness of a tool is measured by the difference it makes in practice. In the context of CSeR, we claim that the tool can serve as a useful source-code documentation aid and as a guide for similar activities in the future. The previous sections discussed problems with existing tools and showed why those tools cannot correctly or completely address the issue. This section shows how CSeR addresses the issue, how it manages clones, how the information is presented, and how programmers can benefit from it. We identified test cases and applied CSeR to them. These test cases also demonstrate the robustness of the tool in real-world scenarios.

4.3.1 Experiment Setup

First, we identify classes or code that may have been created by copy and paste. Then we recreate the scenario with CSeR. Identifying appropriate candidates is not easy. Our approach uses the clone detector CCFinder [3] [13] as a starting point, followed by manual inspection of classes in the same package or related packages. If two classes appear similar,

[3] http://www.ccfinder.net

we screen them further to determine whether they may have been created by copy and paste. To do this, we compare the positions and spacing of statements and comments, first at the class level, then at the method level, and finally at the statement level.

One case from the JLCP [4] project illustrates the issue. A class file had two authors, and the programmer who copied the file from another file with a different author forgot to remove the old comments. Once similar files are identified, they must be paired. It is sometimes difficult to identify which file was copied from which, or which file should be treated as the original and which as the current file. Because CSeR supports two-way correspondence, this decision does not affect the experiment.

The current CSeR implementation supports only pairwise comparison, so related classes must be split into pairs. CSeR always shows changes with respect to another file. If three classes are similar, we split them into two pairs. For example, if Class A and Class B are similar, we copy Class A and try to construct Class B from the copied version. The changes shown by CSeR are tracked and analyzed in Section 4.3.4.

In short, we are trying to recreate the situation the programmer went through while modifying the clones.

The test cases were collected from three sources. The first source was the Eclipse project, specifically the JDT UI, JDT Core, and SWT projects. Eclipse is a well-known open-source project and provides a good source of carefully written Java code. The second source was Java Lobby Community Platform (JLCP), a web-based business project that provides another practical test case. The final source was clone examples from the literature. Literature clones often include extreme cases, diverse domains, common cases, or generalized cases. A tool that passes these cases is more likely to help programmers in practice.

To give a clearer picture of the experiments, two case studies are explained in detail.

[4] http://sourceforge.net/projects/gotjava

4.3.2 Case Study 1 SetFilterWizardPage And ExclusionInclusionDialog

ExclusionInclusionDialog and SetFilterWizardPage are an interesting case because they belong to different type hierarchies, yet they are very similar. The parent class of ExclusionDialog is StatusDialog, while the parent class of SetFilterWizardPage is NewElementWizardPage. Details of the changes are shown in Tables 4.3 and 4.4, and part of the code is shown in Figure 4.5. As a rule of thumb, a red marker indicates a delete operation, a green marker indicates an insert operation, and a yellow marker indicates an update operation.

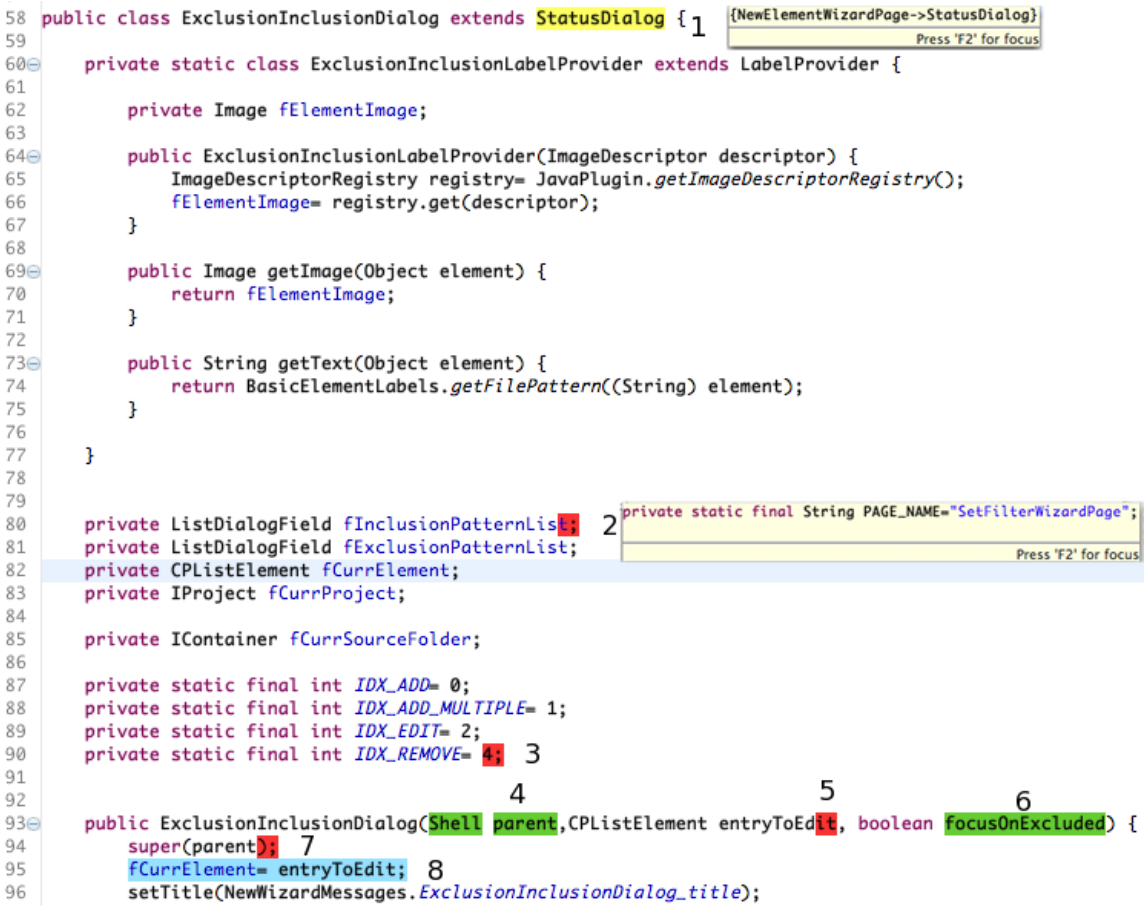

Figure 4.5: CSeR Showing SetFilterWizardPage from ExclusionInclusionDialog

In Figure 4.5, the differences start with the parent class. SetFilterWizardPage is a child of NewElementWizardPage, while ExclusionInclusionDialog is a child of StatusDialog. This change is marked "1" and categorized as "other" in the update-change type in Table 4.4.

Changes “2” and “3” show the deletion of three fields from the class. The deleted fields can be seen by hovering near the marker; the figure shows this for change “2.”

Similarly, change “7” corresponds to the deletion of two statements. As mentioned earlier, when a delete occurs, CSeR always looks for a place to attach the marker for the delete operation. In this case, two statements after “super(parent)” are deleted, so CSeR places a marker after that statement. Change “8” corresponds to a move operation: the “fCurrElement=entryToEdit” statement is moved one statement earlier.

Changes “4,” “5,” and “6” correspond to parameter changes. Change “5” indicates the deletion of two parameters, “ArrayList existingEntries” and “IPath outputLocation,” while “4” and “6” are marked in green to show newly inserted parameters. The inner class in the figure does not show any changes because it has no change from the original file other than position, and the position of an inner class declaration does not change the meaning of the class.

4.3.3 Case Study 2 NewClassCreationWizard And Clones

A quick comparison of the wizards used to create classes, interfaces, enums, and annotations shows that they share a substantial amount of code. This case study includes NewClassCreationWizard, NewEnumCreationWizard, NewAnnotationCreationWizard, and NewInterfaceCreationWizard. Changes for part of the code in the editor are shown in Figure 4.6. The changes are analyzed in Table 4.4.

The figure is shown with respect to NewAnnotationCreationWizard. There are only five changes. Changes “1,” “2,” and “5” are variable-type changes, while “3” and “4” are variable-name changes. Every update change carries a message showing the original node, and this message is shown with change “5.”

The same kind of change occurs for both NewInterfaceCreationWizard and NewEnumCreationWizard. NewEnumWizardPage and NewInterfaceWizardPage replace NewAnnotationWizardPage, respectively, and the variable names are changed in a similar fashion. This supports the argument that programmers often copy and paste templates rather than

Figure 4.6: CSeR Showing NewAnnotationCreationWizard from NewClassCreationWizard

code that will remain identical.

4.3.4 Result Analysis

The results of the experiments are summarized in Tables 4.3, 4.4, 4.5, and 4.6. One change excluded from the analysis in all cases is the class-file name. For example, when NewClassCreationWizard is copied and pasted as NewAnnotationCreationWizard, the second class goes through Eclipse refactoring, which changes occurrences of NewClassCreationWizard to NewAnnotationCreationWizard. Because this is an obvious change, it is not highlighted in the code and is not included in the data below. After analyzing the changes in different classes, we created categories for each change type. These categories are explained below.

Division of Changes The division of changes is a study of changes observed with CSeR; it is not directly tied to the design of CSeR. In general, we do not consider overlapping changes. If there is a new method, we count only one change: the inserted method decla-

ration. Even though the statements inside the method are also new, they are not counted separately. The same rule applies to statements: if a statement is new, the expressions inside it are not counted separately.

The categories are based on experience gained while analyzing the changes. For example, a broad category such as expression update could include variable-name changes, literal changes, and method-invocation name changes. Because we observed many variable-name changes and other specific cases, we treated them separately. Similarly, literal changes could be split into string, number, and character literal changes, but the test cases motivated treating them together.

Update Changes As shown in Table 4.6, most of the changes were updates. The largest portions of update changes were variable-name and type updates (V and T). An update is categorized this way when the variable name or type is changed partially or completely. For example, if the statement "Expression statement = null" is changed to "Expression expression=null," it is considered a variable name change (V). If it is changed to "Body statement =null," it is considered a type change (T). Changing the return type of a method is also considered a type change.

L stands for literal, and includes string, number, and character literal changes. Method changes can occur either in a method definition or in a method invocation. In some cases, such as Case Study 1, a method must be renamed. When that happens, the places where the method is called inside the class must also change. Both the declaration and invocation changes are considered method changes.

```
Category fileCat = FileCenterManager.getInstance().createCategory(name, description, parentCategory);
License license = FileCenterManager.getInstance().createLicense(licenseTitle, licenseURL, isFreeLicense);
      =
License license = FileCenterManager.getInstance().createLicense(licenseTitle, licenseURL, isFreeLicense);
```

Figure 4.7: Statement Update change

Special cases in V include "return null → return variableName" and "variableName → methodName." In a case such as Figure 4.7, it is better to consider the change a statement

update rather than several simple-name updates. Statement updates are explained below. Other updates include cases where only an "else block" or "else if" is deleted while the rest of the statement is maintained. These cases are included in the other (O) category.

Statement Updates From Figure 4.7, it may not initially be obvious that the statement should be considered an update. After considering the context, however, there is sufficient reason to treat it that way. If we counted individual changes in the example, there would be more than six updates involving changes in names, types, and method names. Instead, we treat these changes as a single statement update.

Insert & Delete Changes S (Statement), M (Method Declaration), and F (Field Declaration) are self-explanatory categories. A statement can be an if statement or a method invocation. C (Class Declaration) includes anonymous inner classes and other inner classes. P refers to parameters in a method declaration. E (Expression) has a broader scope: it includes adding new members to an array initializer, adding only an "if" condition, adding new arguments to a method invocation, or adding a new expression to an existing expression. Delete applies to all the same cases.

Move Changes S (Statement), M (Method Declaration), and C (Class Declaration) are defined the same way as above, except that the operation is a move.

Table 4.3: **Inserts & Deletes In Clones Collected From Projects**

No	File (LOC)	Insert						Delete					
		S	E	P	M	F	C	S	E	P	M	F	C
	Eclipse												
	CodeAssistPreferencePage (100)												
1	CodeStylePreferencePage (122)			1	2	2							
	CodeFormatterPreferencePage (61)												
2	CleanUpPreferencePage (60)												
	FoldingPreferencePage (60)												
3	MarkOccurrencesPreferencePage (6)												
	JavaContext (767)												
4	JavaDocContext (223)	5				3	1	1			25	5	
	SetFilterWizardPage (340)												
5	ExclusionInclusionDialog (326)	6		2	1			9		2	1	3	1
	NewClassWizardPage (296)												
6	NewAnnotationWizardPage (145)				1			13	2		6	4	
7	NewEnumWizardPage (147)				1			11	1		6	4	
8	NewInterfaceWizardPage (146)							11	1		6	4	
	NewClassCreationWizard (95)												
9	NewAnnotationCreationWizard (95)			1									
10	NewEnumCreationWizard (94)			1									
11	NewInterfaceCreationWizard(94)												
	PrefixExpression (337)												
12	PostfixExpression (320)								4			4	
13	ParenthesizedExpression (189)							2			3	2	1
	ForStatement (361)												
14	LabeledStatement (275)							6	2		2	4	
15	AssertStatement (261)							6	2		2	4	
	HistoryListAction(C[5]) (185)												
16	HistoryListAction(T[6]) (207)												1
	HistoryDropDownAction (C) (109)												
17	HistoryDropDownAction (T) (109)												
	CallHeirarchyImageDescriptor (180)												
18	JavaElementImageDescriptor (300)	6			5	13		5				2	
	TableEditor (260)												
19	TreeEditor (285)	4				2	1	1		1		1	
	JLCP												
	FileDescriptionTag (75)												
20	FileCreationDateTag (75)												
21	FileTitleTag (75)												
	PanelsDAOFactory (67)												
22	LinksDAOFactory (50)												
	UserDAOFactory (110)												
23	MessagingDAOFactory (122)												
24	GroupDAOFactory (102)												
25	BuddyDAOFactory (121)												
	AddFileCategoryAction (105)												
26	AddLicenseAction (111)	3	1			1		2	1				
	ExpirePollAction (74)												
27	DeletePollAction (72)												

Table 4.4: **Updates & Moves In Clones Collected From Projects**

No	File (LOC)	Update					Move		
		V	T	L	M	O	S	M	C
	Eclipse								
	CodeAssistPreferencePage (100)								
1	CodeStylePreferencePage (122)	1	2			3			
	CodeFormatterPreferencePage (61)								
2	CleanUpPreferencePage (60)	2	1	2					
	FoldingPreferencePage (60)								
3	MarkOccurrencesPreferencePage (6)	1	1						
	JavaContext (767)								
4	JavaDocContext (223)				1		1	2	
	SetFilterWizardPage (340)								
5	ExclusionInclusionDialog (326)				2	1	1	1	1
	NewClassWizardPage (296)								
6	NewAnnotationWizardPage (145)	3		1					
7	NewEnumWizardPage (147)	3		1					
8	NewInterfaceWizardPage (146)	3		1					
	NewClassCreationWizard (95)								
9	NewAnnotationCreationWizard (95)	2	3						
10	NewEnumCreationWizard (94)	2	3						
11	NewInterfaceCreationWizard(94)	2	3						
	PrefixExpression (337)								
12	PostfixExpression (320)	2					1		
13	ParenthesizedExpression (189)	13		3	8				
	ForStatement (361)								
14	LabeledStatement (275)	14	5	3	6				
15	AssertStatement (261)	13	6	3	6				
	HistoryListAction (callhierarchy) (185)								
16	HistoryListAction(typehierarchy) (207)		11		3		1	1	
	HistoryDropDownAction (callhierarchy) (109)								
17	HistoryDropDownAction (typehierarchy) (109)	12	9		2			2	
	CallHeirarchyImageDescriptor (180)								
18	JavaElementImageDescriptor (300)		3						
	TableEditor (260)								
19	TreeEditor (285)	21	6		1			1	
	JLCP								
	FileDescription (75)								
20	FileCreationDateTag (75)			1					
21	FileTitleTag (75)			1					
	PanelsDAOFactory (67)								
22	LinksDAOFactory (50)	1	2	2					
	UserDAOFactory (110)								
23	MessagingDAOFactory (122)	9	4	1	2				
24	GroupDAOFactory (102)	9	4	1	2				
25	BuddyDAOFactory (121)	9	4	1	2				
	fAddFileCategoryAction (105)								
26	AddLicenseAction (111)	3		1		1			
	ExpirePollAction (74)								
27	DeletePollAction (72)			1					

Table 4.5: **Clones Collected From Literature**

No	Desc (LOC)	Insert						Delete						Update					Move		
		S	E	P	M	F	C	S	E	P	M	F	C	V	T	L	M	O	S	M	C
	Fig 6 (23)																				
1[15]	Fig 7 (22)	4						2						1	1			1			
	Scenario 1 (10)																				
2[16]	Scenario 1 (12)		1																1		
	Scenario 2 (2)																				
3[16]	Scenario 2 (3)				1												1				
	Fig 3a (4)																				
4[8]	Fig 3b (4)																1				
	Fig 8a (4)																				
5[8]	Fig 8b (5)	1																			
	Fig 11a (5)																				
6[8]	Fig 11b (10)	1																			
	Fig 12 (1)																				
7[8]	3.15			3																	
	Fig 3 code1																				
8[11]	Fig 3 code2	4																			
	Fig 5 code1																				
9[11]	Fig 5 code2	3																			
	Fig 1 (9)																1				
10[9]	Fig 2 (17)	1						1													

Table 4.6: **Summary Of 533 Changes From Collected Clones**

No	Change Distribution	Description	Internal Distribution
1	Update (49 %, 261)	Variable Name (V)	49 %
2		Variable Type (T)	26 %
3		Method (M)	15 %
4		Literal (L)	8 %
5		Other (O)	<2 %
6	Delete (33 %, 177)	Statement (S)	40 %
7		Method Declaration (M)	28 %
8		Field Declaration (F)	21 %
9		Expression (E)	8 %
10		Parameter (P)	< 1 %
11		Class Declaration (C)	< 1 %
12	Insert (16 %, 82)	Statement (S)	46 %
16		Field Declaration (F)	26 %
15		Method Declaration (M)	14 %
13		Parameter (P)	10 %
14		Expression (E)	2 %
17		Class Declaration (C)	2 %
18	Move (2 %, 13)	Method Declaration (M)	54 %
19		Statement (S)	39 %
20		Class Declaration (C)	7 %

Chapter 5

Related Work

Mainstream research on clones has primarily focused on clone detection. Surveys of clone-related research and clone-detection techniques can be found in [18] and [19].

As explained in [12], inconsistent changes between clones are frequent and can be a major source of defects. This implies that cloning can create substantial development and maintenance problems unless existing clones and their evolution are carefully tracked. In short, changes to clones are frequent, and inconsistent clone changes can introduce defects.

In the study described in [10], we proposed the main design elements for proactive clone management, aimed at providing better support for clone evolution. We also described our initial experience prototyping several features that partially implemented this design, including the consistent renaming utility CReN and CSeR. Our design for proactive CnP support was partially inspired by related work, but it differs from that work in important ways. Clone case studies were used to motivate individual features and to identify or refine design requirements. Future lab-based user studies can help evaluate these features and their effectiveness in helping programmers work with clones.

CnP differs by tracking copied-and-pasted clones directly rather than relying on clone-detection tools. As a result, CnP can support clones that clone-detection tools may fail to capture. Moreover, clone-detection tools tend to have low precision and recall [2], which can create expensive batch-processing work for programmers. Proactive support can potentially

ease this problem by distributing the effort over time.

Clonescape [4] and CPC [21] are recent projects aimed at developing proactive clone support. CnP differs from Clonescape and CPC by providing features such as accidental capture of external identifiers, consistent renaming (CReN), and a clone-difference view (CSeR).

CloneTracker [7] proposes a novel representation for clone locations that is independent of physical properties such as character offsets or line ranges. It also supports a form of simultaneous editing by using Levenshtein distances to locate similar lines between clones. However, CloneTracker relies on clone detection. CnP may also perform more accurately in cases where Levenshtein distance is insufficient.

Context Sensitive Cut Copy and Paste (CSCC&P) [14] is a tool implemented on the LAPIS[1] pattern matcher platform. The tool detects violations of several common contextual relationships, including semantic relationships. It is activated by copy or cut actions, but it differs from CSeR in its operation and goals.

Codelink [20] supports both clone-difference views and simultaneous editing. It uses colors to show commonalities between linked clones in blue and differences in yellow, and uses elision to hide identical parts of the clones from view. However, unlike CSeR, Codelink does not distinguish between "new" and "updated" code.

Codelink uses the longest-common subsequence (LCS) algorithm, like the algorithm implemented by the Unix "diff" utility, to determine commonalities and differences within a clone group. Toomim et al. report two main shortcomings of the LCS algorithm: potentially long running time and unintuitive results. CnP's approach to differencing clones can potentially address these problems.

CReN and Rename Refactoring differ mainly in their scope. CReN works in any user-specified region, while Rename Refactoring works only in predefined scopes such as blocks and classes.

The Breakaway and Jigsaw tools automatically determine detailed structural correspon-

[1] http://groups.csail.mit.edu/uid/lAPIs/index.html

dences between two classes and two methods, respectively [5, 6]. The inputs to Breakaway and Jigsaw are classes or methods that may contain different code. The input to CSeR, by contrast, is always initially identical. CSeR *incrementally* tracks changes as they are made to related clones.

Chapter 6

Conclusion & Future Work

Developers must manage clones efficiently within software projects. Significant work has been done in clone detection, but relatively little work has focused on tracking "copy and paste" (CnP) operations, even though those operations play an important role in clone formation. This thesis demonstrated the advantages of tracking CnP operations directly.

We presented an approach for clone management based on tracking CnP operations. Using this approach, detailed differences between clones are available directly inside the editor. Changes are visualized in different colors to make them easier to understand. The comparison is context-sensitive and AST-based, which makes the approach distinctive. We also discussed CSeR (Code Segment Reuse), a proof-of-concept implementation, and described its design.

We conducted a small experiment involving nine pairs of classes to verify CSeR's advantages over clone detectors. These test cases were selected from an industrial project. We also demonstrated CSeR's usefulness and robustness using 37 test cases selected from industrial projects and research publications. The 533 changes identified by CSeR were carefully analyzed and divided into 20 different categories.

Clone groups CSeR currently considers clones in pairs. Clone groups occur when more than two clones are related. For example, when a class is pasted twice, there are three related clones and therefore a clone group. The current CSeR implementation represents

this situation as two clone pairs: the first and second clones, and the first and third clones. Although the second and third clones are related, the current implementation does not identify that relationship.

A better design would connect all related clones as a group. This would allow a developer to view the differences between any two clones in the group. For example, when NewAnnotationCreationWizard and NewEnumCreationWizard are both created from NewClassCreationWizard, a developer should be able to see the difference between NewEnumCreationWizard and NewAnnotationCreationWizard.

Tracking Code and Identifying Templates Consider a scenario in which a developer is working on a large file and suddenly something stops working. The developer may not know which changes caused the problem. CSeR could be extended to track changes in code across two editing time frames. It could also provide a capability to return to the state before an edit.

CSeR already calculates editing regions in clones. Because these edits are based on the AST, CSeR could be integrated with the Eclipse template feature. As a rough idea, when a developer pastes a few lines of code, the developer could activate a CSeR template option. CSeR would calculate the editing regions, pass that information to the template model, and create a keyed template that could be used later.

Version Control Integration and Side-by-Side View While calculating changes, CSeR saves information related to a class file in a text file with the same name, and this file is read whenever someone opens the class file. This mechanism could be extended to version control. If such integration were available, changes could be shared among users. Comparisons could also be performed against earlier versions on the local machine or in the version-control system.

CSeR does not currently have a side-by-side view. It would be useful to integrate CSeR data with the Eclipse Compare Editor so that users could view comparisons in a familiar interface while retaining CSeR's more accurate comparison data.

Bibliography

[1] David L. Atkins. Version sensitive editing: Change history as a programming tool. In *ECOOP 98, SCM-8, LNCS 1439*, pages 146–157. Springer-Verlag, 1998.

[2] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[3] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 493–504, New York, NY, USA, 1996. ACM.

[4] A. Chiu and D. Hirtle. Beyond Clone Detection. Course Project, CS846 Spring 2007, University of Waterloo. `http://www.cs.uwaterloo.ca/~dhirtle/publications/beyond_clone_detection.pdf`. Accessed Jan. 12, 2009.

[5] R. Cottrell, J.J.C. Chang, R.J. Walker, and J. Denzinger. Determining Detailed Structural Correspondence for Generalization Tasks. In *Proceedings of ESEC/FSE'07*, 2007.

[6] Rylan Cottrell, Robert J. Walker, and Jörg Denzinger. Semi-automating Small-Scale Source Code Reuse via Structural Correspondence. In *Proceedings of FSE'08*, pages 214–225, 2008.

[7] E. Duala-Ekoko and M.P. Robillard. Tracking Code Clones in Evolving Software. In *Proceedings of ICSE'07*, 2007.

[8] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. Change distilling:tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[9] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 321–330, New York, NY, USA, 2008. ACM.

[10] Daqing Hou, Patricia Jablonski, and Ferosh Jacob. CnP: Towards an Environment for the Proactive Management of Copy-and-Paste Programming. In *ICPC'09*, 2009.

[11] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. Context-based detection of clone-related bugs. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 55–64, New York, NY, USA, 2007. ACM.

[12] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do Code Clones Matter? In *ICSE'09: Proc. of the 31st International Conference on Software Engineering*, 2009.

[13] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[14] Reid Kerr and Wolfgang Stuerzlinger. Context-sensitive cut, copy, and paste. In *C3S2E '08: Proceedings of the 2008 C3S2E conference*, pages 159–166, New York, NY, USA, 2008. ACM.

[15] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in OOPL. In *Proc. Int'l Symp. Empirical Software Engineering*, pages 83–92. IEEE Press, 2004.

[16] Miryung Kim and David Notkin. Program element matching for multi-version program analyses. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 58–64, New York, NY, USA, 2006. ACM.

[17] Andrew J. Ko, Htet Htet Aung, and Brad A. Myers. Design Requirements for More Flexible Structured Editors. In *Text Editing, CHI '05: Human Factors in Computing*, pages 1557–1560. Press, 2005.

[18] R. Koschke. Survey of Research on Software Clones. In *Dagstuhl Seminar Proceedings*, 2006.

[19] C.K. Roy and J.R. Cordy. A Survey on Software Clone Detection Research. Technical Report 2007-541, Queen's University, 2007. `http://www.cs.queensu.ca/TechReports/Reports/2007-541.pdf`. Accessed Jan. 12, 2009.

[20] M. Toomim, A. Begel, and S.L. Graham. Managing Duplicated Code with Linked Editing. In *Proceedings of VL/HCC'04*, 2004.

[21] V. Weckerle. CPC: An Eclipse Framework for Automated Clone Life Cycle Tracking and Update Anomaly Detection. Master's thesis, Free University of Berlin, 2008. `http://cpc.anetwork.de/thesis/thesis.pdf`. Accessed Jan. 12, 2009.