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

©Ferosh Jacob 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 be useful for the quality and maintenance of software projects. Significant research has been done in locating code clones but not much effort has been put in tracking "Copy and Paste" operations even though these operations are the primary source for clone formation. We design and implement a code editor named CSeR, which keeps record of clones created by "Copy and Paste", and tracks and visualizes the changes made to a new clone with distinct colors. The changes are calculated by comparing the Abstract Syntax Trees (ASTs) for a pair of clones incrementally as edits are made to the code. This incremental approach makes CSeR unique and more accurate than other similar tools. An empirical study was conducted with 37 test cases collected from industry and research projects to test the robustness and usefulness of the tool. A total of 533 changes were identified and categorized into 20 different types. A comparative study with related tools is included, which demonstrates the uniqueness of CSeR. Finally, some potential extensions that can widen CSeR's scope of applications are explained as 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 copy and paste is inevitable. It comes with a little surprise that the existing programming editors do not offer much support for copy and paste other than the usual text editors and refactoring. For example, Eclipse does an automatic re-factoring when programmer does a "copy and paste" of a class. But re-factoring alone is not enough. There are some information which is linked with every copy and paste action and can be leveraged to help programmers. 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." So without any doubt, copy-and-paste operations play a vital role in the software development process. We need to analyze those operations more carefully so that we can provide some additional help to the developers. In order to know the necessity for tracking CnP operations, we should know what we are missing if we do not have those operations.

1.1 Problem : Loss Of Relationship

Developers copy and paste code within the class or a class as whole because there is a level of similarity between the classes. Those similarity was evident for that developer while doing the CnP operation but won't be for another developer or even for the same developer

at a later time. So in short between the classes or the code which is copied and pasted there is an invisible relationship. This relationship is lost if the IDE won't track the CnP operations. Even if you have the relationship some how, developer still need to know the detailed differences between the clones.

1.2 Problem : Missing Detailed Information

Detailed Information means how clones differ, Where are the new statements inserted and what are statements deleted, moved. Those fine grained information is useful as it can

1. Facilitate understanding between clones
2. Guide for similar CnP operations in future

These detailed differences could shed light into better understanding of code. The case would be more harder if the clones are from different files because the developer has to leave the editor context and use some tools like Eclipse Compare-Editor to see the differences, or manually compare the files. The issues while using Compare-Editor is explained in Validation chapter.

1.3 Example Scenario

Consider the figure 1.1, it is a wizard page while creating a new source folder to include or exclude file patterns in build path. From a brief study into the history of this class we understood that it was earlier implemented as a dialog instead of wizard and now eclipse team has replaced that with a wizard page. The earlier copy was never deleted and it was maintained so it could be used for external clients in case they want to set up the file patterns in a source folder as an independent action from creating or editing source folder. Most probably the new class SetFilterWizardPage was copied from ExclusionInclusionDialog.

Problem - Loss of Relationship Consider a situation where a user makes some modification in SetFilterWizardPage. Most probably he has to make the similar change in the

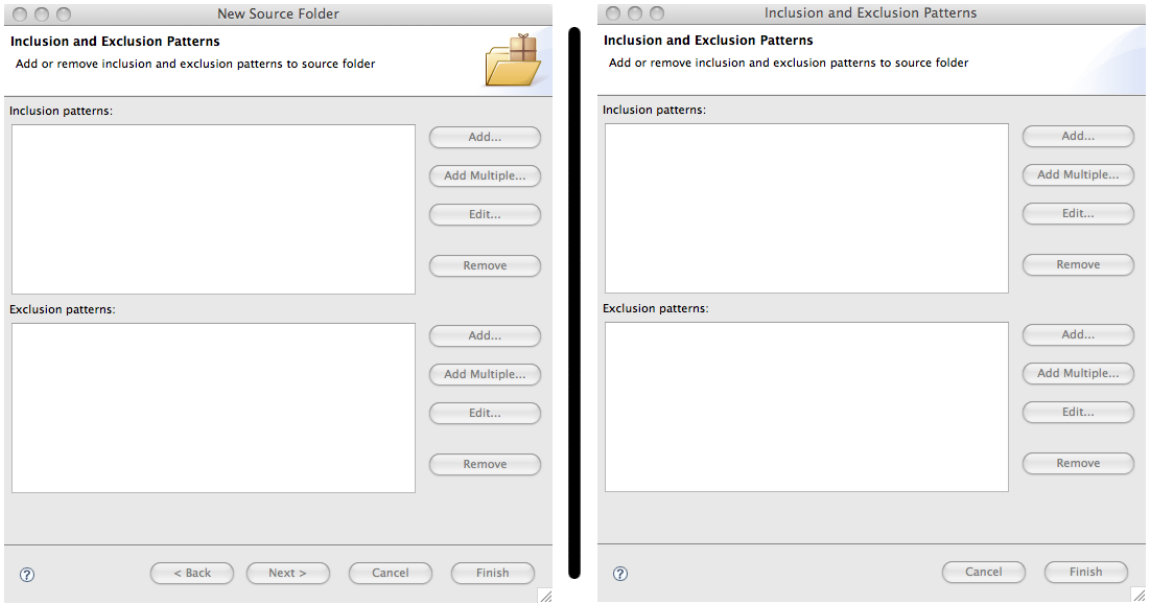

Figure 1.1: SetFilterWizardPage and ExclusionInclusionDialog

ExclusionInclusionDialog. As an example, some change has been added in SetFilterWizardPage for BIDI(Bidirectional Support for internationalization) and it was included in the ExclusionInclusionDialog too, similar is the case for a bug fix related to new source-folder creation. So the question is how will the programmer know about what all classes gets affected when there is a bug fix or change? What sort of relation both the classes have? Even if Eclipse team decides to remove these duplicates, they should know what all classes are created in similar fashion and how much is the similarity. How to find those classes?

Problem - Loss of Detailed Information Even if the relationship is known it would be time consuming to find the detailed information or differences between the clones. Consider the example scenario, part of outline view of the classes are given in figure 1.2. User might be interested to know which all methods in SetFilterWizardPage are removed? which all methods have corresponding method in ExclusionInclusionDialog? which all methods are new in ExclusionInclusionDialog? All these same questions are not just restricted to method level it could be asked even to an expression level.

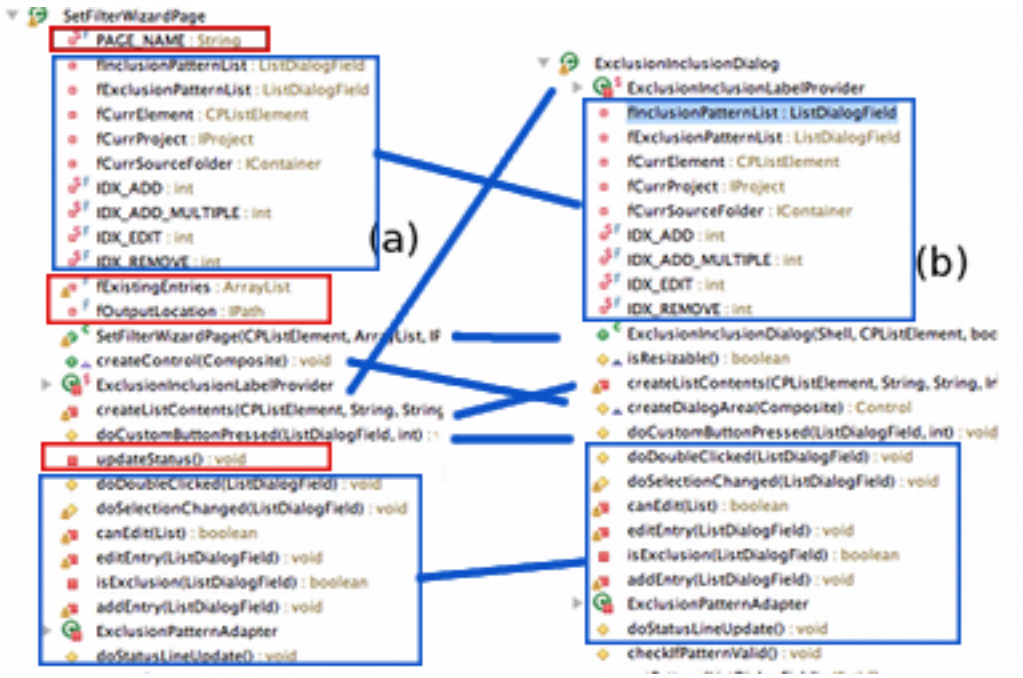

Figure 1.2: Outline View Of SetFilterWizardPage And ExclusionInclusionDialog

1.4 Tracking And Visualizing Detailed Code Differences

CSeR solves these problems efficiently, CSeR calculates changes based on AST(Abstract Syntax Tree) based incremental algorithm. The changes are shown in the editor and is calculated during the user edit. For the example considered here, Once the ExclusionInclusionDialog is created in a environment which supports CSeR, code will look like as in Figure 1.3.

The rest of the thesis is arranged as follows, Second chapter includes the basic definitions and background information, Third chapter explains the design in detail, Fourth chapter validates the tool and in the next two chapters related tools are explained and concluded.

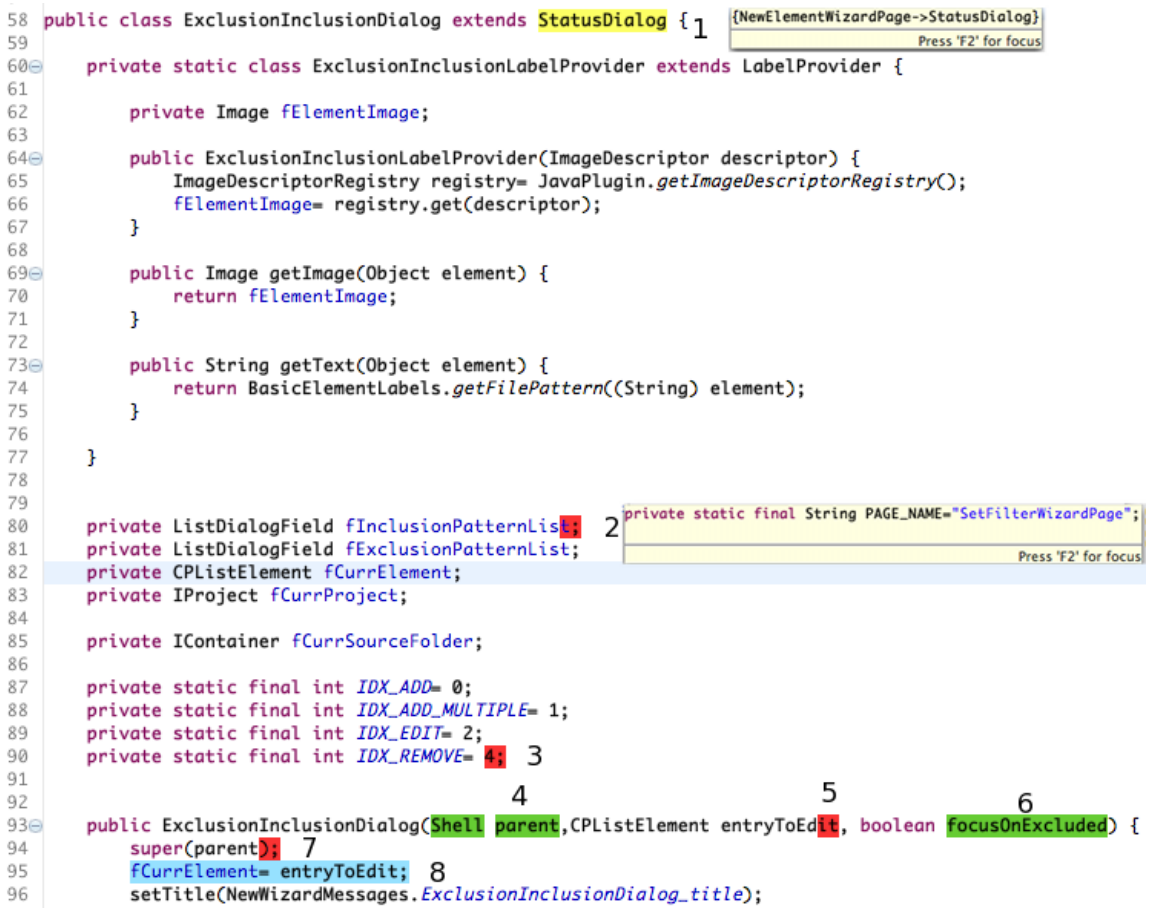

Figure 1.3: CSeR Showing SetFilterWizardPage from ExclusionInclusionDialog

Chapter 2

Background and Definitions

2.1 Context

If not specifically mentioned it is always assumed that programmer copied a file and pasted in some location with a different or same name. The copied file is called original file and pasted file is called current file. Current file is also the editing file in almost all scenarios. CSeR treats copying files as a special case of the copying code block where position is not specified, hence all those features applicable to files are also applicable to code block and vice versa. Position in current file may be referred as current positions and in original file as original positions.

2.2 Change

Change is usually the AST change. Change from the original file to the current file. Obviously adding space with-in the line or between lines wont trigger changes. Even adding/deleting/updating comments are also not considered as changes as we don't track changes for comments. More technically it is the default implementation of ASTMatcher in JDT.

2.3 Correspondence

In CSeR's terminology when it says two AST's are corresponding it means during the copy paste operation which led to the formation of clone they were identical and any difference now is due to the user-edit.

2.4 User-edit

This refers to anything starting from typing/deleting a character to insert/replace/delete block of code or a file itself.

2.5 Insert, Delete, Update operations in CSeR

This CSeR's edit operations are defined with in the context of AST, An addition of a new AST node will be considered as Insert, while removing one is considered as Delete and modifying an existing node is Update. It is different from the usual Insert/Delete/Update operations as inserting a character need not necessarily be an Insert in CSeR context, it will most probably be an update in CSeR, because the smallest node that can be inserted has to be at least 3 characters long and it is (i++) Pre/Post Expressions with a single letter variable name.

2.6 Position

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

2.7 Position Updaters

Regionupdaters are listeners which listens to the event of position change. Whenever there is an event which change regions, CSeR's database has to be updated with the new information. The Regionupdaters do this for CSeR.

2.8 Anchor AST

Usually it is the immediate parent of the AST we are referring to. This become a crucial information for Statements and BodyDeclarations since the anchorAST of statement would be always Block and for Bodydeclaration would be Typedeclaration. CSeR make use of this information to distinguish between the operations in statements and body declarations level.

2.9 Statements

Same definition as the JDT's definition of Statement node. Statements are generally well formed AST's which are the direct children of the Body node. Statements can occur anywhere inside a method in a static block inside a class and even block of statements inside a method

2.10 Body Declarations

Same definition as the JDT's definition of Body Declaration node. Bodydeclarations are generally well formed AST's which are the direct children of the Type Declaration node. Typedeclarations can occur anywhere inside a method as an inner class or inside the class as field or method declarations.

2.11 Nodes

This is finer level of division after the statement node, It is practically impossible to visit all the nodes that can build a statements so from experiments we have come up with a small list of nodes where programmer usually change, Since it is practically impossible to find a parent class other than the ASTNode we decided to use the name nodes. The nodes can be Expression like a variable name, string literal or SingleVariableDeclaration like parameters in a method declaration or TypeParameter like parameters in a method call.

2.12 Block

ASTNode corresponding to a block of statement, it can be method of a body. It is usually a block of statements enclosed in { and }.

2.13 Scope

The meaning of scope changes with the context, for statements the scope is the Block we consider, while for nodes it is the node we consider and finally for methods and fields scope is entire class.

2.14 Action

Usually this refers to different types of user-edits, different programmers have different style of user-edits, some people prefer typing instead of copy paste small code block, some prefer deleting using delete key, while some back space to delete, few other cut to delete and so on. In short here we are interested in How the programmer make a change and not on What change he make?

2.15 Goal

This refers to the code change, Here we are not interested on how he made that change, Changing variable name, class name, add another parameter in a method are all goals.

2.16 Infix Expressions

Infix Expressions have the form operand operator operand and the operator can be 175 arithmetic, boolean, assignment and dot operators. Both the operands are expression

2.17 JDT

Java Development Tools[1] provides the tool plug-ins which make possible the development of java applications in Eclipse IDE. The JDT also provides set of API for the smooth processing of java AST and related features. The CSeR heavily uses this api.

2.18 CSeREditor

It is a super set of the Eclipse JavaEditor and CSeR tools. All the operations should be performed with and only with CSeREditor for consistent results in CSeR.

2.19 PositionList

It is just a data structure containing the positions. Positions are ordered in this data structure than "after" "before" "smallest" are defined within it.

2.20 Session

Session is usually defined with the context they are mentioned, In the context of CSeR as an application session starts with starting the Eclipse to finally closing it. In the context of

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

statement, session refers to the block user is working, For example when user is working in one method(adding/updating/deleting variables, method calls...) inside a single block he is working in the same session, the moment he creates another block(for loop, If statement) he is in new session.

2.21 Levenshtein distance

It is a metric for measuring the amount of difference between two sequences. Eg kitten to sitten is 1 while between kitten to sitting is 3, since there is no way to do it with fewer than 3 edits

2.22 Document

An IDocument represents extensible content providing support A document allows to set its content and to manipulate it. On each document change, all registered document listeners are informed exactly once.

Chapter 3

Design

3.1 Requirements

CSeR is basically a clone tracking tool. Clones are assumed to be formed only through copy and paste. User can copy and paste a block of code or a file, there by creating two types of clone formation. We realize that, copying files is a special case of the general case, copying block of code.

So as a high level requirement, we have two identical blocks of code and either one of them or both are susceptible to changes depending on the user-edits. The goal of CSeR is to track the changes and present to the user in the most legible way.

An assumption allowed in all editing scenarios is that the tool won't respond until the code reaches a consistent state. Consistent state is any state, a block of code will undergo with a well formed AST, consistent stages are defined even if the code has resolution errors. In other words CSeR will be active if there are no syntax errors but resolution errors . Difference between resolution and syntax errors are shown in Figure 3.1. In the Figure the error marked "1" is a resolution error, which says "IEditorPart" cannot be resolved to a type" and the error marked "2" says "Syntax error, insert "}" to complete Method Body".

We have analyzed the requirements into a set of use cases, and have listed them below.

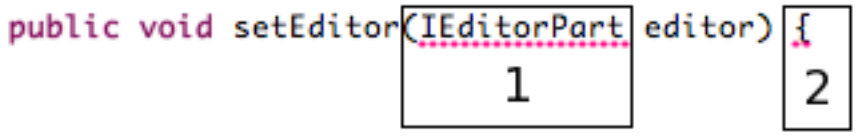

Figure 3.1: Difference between Resolution errors and Syntax errors

Figure 3.2: Consistent Tracking Independent of Actions

All the use cases, implementation decisions and later test cases are based on Eclipse[1] IDE. Most of the editing scenarios have been collected from our experiments. [17, 14] has also contributed to our list.

3.1.1 UC001 Consistent Tracking

Consistent tracking for the same goal in Section 2.15 through different actions in Section 2.14. List of common actions are explained 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	Re-factoring Tools
8	Code Completion

Stated briefly, irrespective of the way user makes any change, CSeR should treat them the same. For example consider Figure 3.2, here "b" is updated to "c". Two ways of doing this in an editor like Eclipse are given below.

1. Type "c" over the character "b".

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

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

So this use case says, change tracking should be independent of actions in code. For the example specified above, CSeR would show the change as in Figure 3.2, no matter how user makes the change. This use case is applicable to use cases mentioned 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, 3.1.11

3.1.2 UC002 Simple Name

This use case corresponds to changing identifiers inside a class. The common modification involving the 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	fPostion → f. Position

3.1.3 UC003 Statements

This use case corresponds to changing statements 2.9 inside a Block 2.12. The common modification involving the statements are listed in Table 3.3. A case which is not mentioned in the table is updating a statement. Since this is a very general case, We deal that as a separate case. For example updating arguments in a method is also updating a 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. The common modification involving arguments 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. The common modification involving the parameters 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 would be the most general case, as change in expression can have different variations. Our strategy is to support the most common cases. The common modification involving expressions 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 section does not refer that the tool has to track the changes in comment. The change referred is the changes in AST by introducing or removing comments. Most common comment related modifications are give 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

These are the reserved words in java. A complete set of keywords in java can be found from the Table 3.9. Most common modification involving keywords 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

The fields in classes correspond to method declaration and field declaration. In JDT[2]2.17 terminology it is called Bodydeclarations2.10. 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	in k; → int k=0;

3.1.10 UC010 Single Statement with Multiple Changes

These cases include the scenario where UC002-06 occur all together or as a combination of few of them. The tool should be able to show the change consistently and sensibly. Few cases of multiple edits are discussed in Table 3.11 This use case is applicable 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

In the case of all delete operations, the tool should be able the log the code that was removed. This could be an important source of information for the programmer when he reviews it later or when someone else reviews it. Delete operations can be from any of the applicable use cases mentioned above. This use case is applicable to the use cases in Section 3.1.3, 3.1.4, 3.1.5, 3.1.6, 3.1.7, 3.1.8, 3.1.9

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

3.1.12 UC012 Condition In Conditional Or Loop Statements

In this case, just expressions are inserted specifically for "If" statements, even though this can be for any loop statements. This is different from UC003 because here we are not inserting a statement but just an expression and a keyword as listed in Table 3.12 This use case is applicable to the use cases in Section, 3.1.3, 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
11	UC012	Conditional Statements	UC003, UC006

3.2 Implementation

For the requirements explained above we have come up with a design and implementation. Current implementation is done in Eclipse IDE as a Eclipse plugin. The implementation involves an editor and a simple view. The CSeREditor2.18 is a super set of default Eclipse java editor features and CSeR tools. The editor will not be any different from the default java editor until the programmer does a copy or paste, or activates CSeR explicitly. The plugin makes use of JDT API and plugins that ship with Eclipse default installation. The

only other external library used in the API is for implementing the levenshtein distance2.21. It is taken from Apache[3] library. The simple view gives the list of all clones collected via copy-paste. Clones can be opened in a diff view to see the comparison between the clones.

3.2.1 Trap "Copy and Paste"

In order to manage the clones that are created by copy and paste, we have to trap all the copy and paste operations done in IDE. In the current implementation, we have modified the default copy and paste operation in JDT 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 copy and pasted inside editor? When the Code 1 CSeR parse the AST into smaller AST's as below. As mentioned earlier we need to keep correspondence between two AST's, for that we have to compare them and conclude about things which are identical, similar and different. Most of the time user makes changes gradually in the code, which gives us the chance to reduce the problem to a smaller scope2.13. We don't have to compare the whole AST most of the time, we just need to find the AST relevant to the edit and compare that AST with the corresponding one in the original2.1 file. We always prefer to compare small AST's as it would be fASTer. But to come up with small AST we have to identify the most relevant AST for every edit. As shown in 3.2.1, each AST is defined with a position2.6. AST information being heavy compared to the light weight position(2 numbers) we always deal with positions, even while saving

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

changes to a particular AST node in a file, internally we are linking the change with the position of AST node. In short with a copy and paste operation inside an IDE a set of positions are added to the database of CSeR from both the copied file and pasted file. As an example consider the class 1 is copied and pasted. The positions that are saved with "HelloWorldApp" is shown in 3.2.1. Notice that there is another set of positions associated with the pasted file called "TestWorldApp".

The criteria for calculating the positions is explained in detail in sections 3.2.4, 3.2.6, 3.2.5. It may be a little surprising that we have to calculate the position of the files separately even though they are clones, the reason is they are not "clones". When you copy and paste especially a file, editors do some obvious re-factoring, like in the previous example when you copy a java file named "HelloWorldApp. java" to "TestWorldApp. java', the editor will change all the occurrences of class "HelloWorldApp" inside TestWorldApp. java to "TestWorldApp", which will in fact result in change of 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 copied file are added to the database. CSeR is a tool to keep track of the changes. The changes that occurred in the current file when compared to the original file. The positions from current file and original file are stored in bidirectional map data structure. In other words given a position in current file, you can find the position in original file. From the definition of position and using JDT API position can be converted to an AST node if there exists one.

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 details of the correspondence relationship is shown in Figure 3.3. CheckPosition is a data structure with two positions, one from the current file and the other from the original file. CheckPositions is a data structure to store checkpositions. In the figure, only selected positions are shown for better clarity.

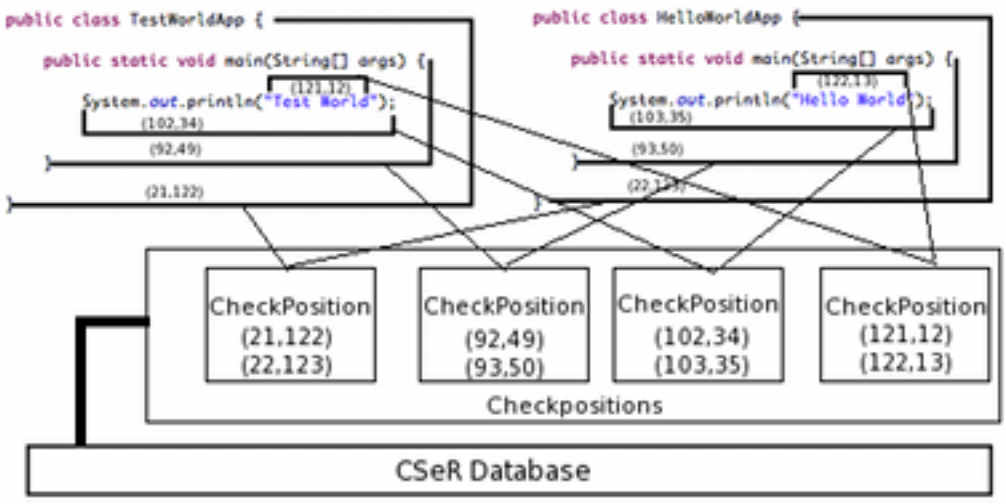

Figure 3.3: Implementation details of Correspondence Relationship

Why are there Position updaters?2.7 Position updaters are listeners which listen to the event of position change. Whenever there is an event which changes positions, CSeR's database has to be updated with the new information. The position updaters do this 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". Here all the operations are assumed to be in the CSeREditor. Finally the class will be as in Code 2. The change of positions due to these operations are explained in Table 3.14.

Operations

1. Select the text"Hello World" inside TestWorldApp file.

2. Type "Hello World" to "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 shows the corresponding positions. Second column gives a short description of the AST node from the original file. The other three columns give positions. Third column specifies to the position of the node in original file, fourth and fifth columns give the position of the same node in current file at different instances of editing. Current position is before any edit, and recent position is after the operation specified in 3.2.2. Few inferences from the table are given below for better understanding.

Inferences from Table 3.14

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

2. Current position in 2.1 before any edits for Class declaration is different (by length, 1) from the position in original file because of the internal re-factoring that took place inside TestWorldApp during the copy-paste, changing all occurrences of "HelloWorlApp" to "TestWorldApp". Since the names differ by one and there is only one occurrence ("public class HelloWorldApp...") of HelloWorldApp inside TestWorldApp the length of class declaration in the current file before any edit differs from original file by 1.

3. Current position for class declaration after the edit seems to be different from the class declaration before edit, again by 1. This is because the operation done was removing

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

the text "Hello World"(11 characters) and inserting "Test World"(10 characters), is quantitatively equivalent in removing 1 character.

3.2.3 Trap-Edit Approach

Every file opened inside the IDE editor is represented by a document2.22 object. The document object contain positions. Whenever there is any change in the document documentListeners are activated and listeners will have more specific information about the change. This information include positions where the change occurred and the nature of the change (insert, delete or replace etc..). Algorithm 1 describes how to get the smallest AST containing that change, when user is editing inside a clone. In the algorithm "Compare" is not explained, as implementation of it will vary based on the type of the node we compare. For example statements cannot be compared the same way as identifiers are compared. All the functions or 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 of the editing scenarios are explained below to clarify the working of the algorithm, given the information in Figure 3.3. Only selected positions are shown for better clarity.

1. Consider adding a space in front of "Test World" string, the position of edit would "121, 1" the, (lines 1-5) algorithm will add all the four positions to the unFilteredList as all of them contains the position of edit. In the next step (line 6) algorithm calculates the smallest position but it won't be "121, 12" but "121, 13". This is because of the position updaters included in the CSeR editor. Position updaters update all the occurrence of "121, 12" with "121, 13" throughout the CSeR database. The Eclipse design calls position updaters before document listeners, hence all the positions in document listeners are updated positions. Once you have the smallest position you calculate the current Node " Test World", and then using the Table 3.14, we can calculate the original position and finally original node. Once both the nodes are known we check the type, as in this case the type is not block or

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

typedeclaration it will go to the default case(Here the type is "String Literal"). A message is calculated using the current Node, original Node and displayed in editor using annotation markers. More details about this comparison is explained in Section 3.2.6. Figure 3.4, change 1 shows the representation of the change in CSeR editor.

2. Consider pasting 'doA();" after the print statement inside the method, (lines 1-5) algorithm will add two positions to the unFilteredList (Block node and the class declaration node) as they are the only one containing the position of edit, and the smallest one being the block node ("92, 49") is selected, Current block node and original block node are calculated as mentioned above and passed to specific comparator. More details about this comparison is explained in Section 3.2.4 Figure 3.4, change 2 shows the representation of the the change in CSeR editor.

3. Consider pasting a declaration for the "doA" after the "main" method, (lines 1-5) algorithm will add just one positions to the unFilteredList (Class declaration node) as that is the only one containing the position of edit, and the smallest one being itself, Classdeclaration of TestWorldApp and HelloWorldApp passed to specific comparator. More details about this comparison is explained in Section 3.2.5 Figure 3.4, change 3 shows the representation of the the change in CSeR editor. The case where adding a field is also the same.

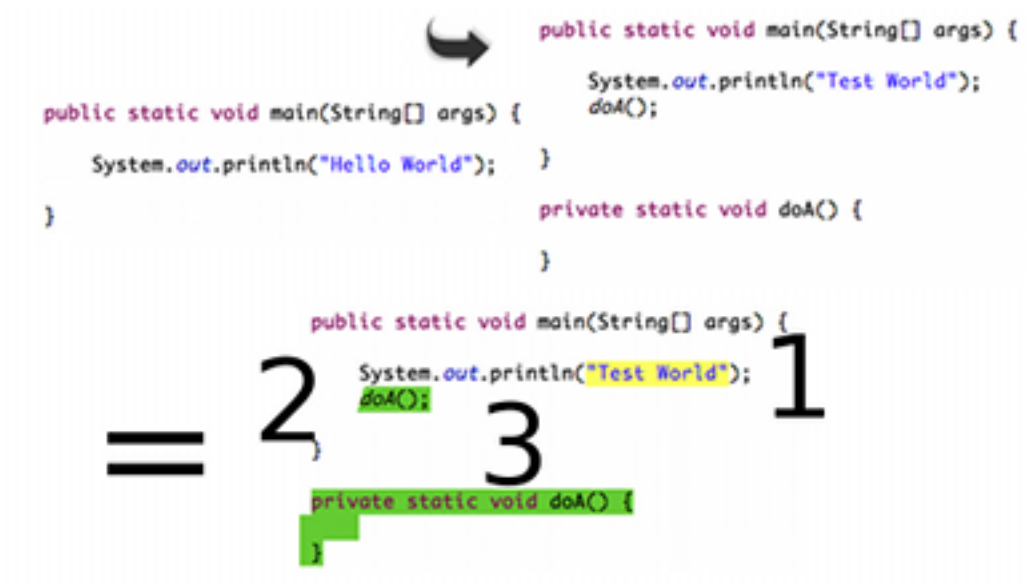

Figure 3.4: Understanding TRAPEDIT Algorithm with TestWorldApp

Using TRAPEDIT algorithm, every edit operation will result in an anchor AST, which can be Type declaration, Block or anything else. When the anchor AST is type declaration, we will be comparing body declarations. In case of block, comparison has to done in statements and all the other cases we have to consider the expression nodes.

The approach for doing comparisons for statements, bodydeclarations and expression is explained in following sections.

3.2.4 Statements

By the existing design when the anchor AST, smallest AST containing the position of edit is Block node, we have to compare the statements. Let us go back to the HelloWorlApp example, consider the situation we are adding a new statement, say “int i=0;” inside the main method as the first line.

Inserting can be either pasting entire line or typing character by character or a combinatation of both. As a design decision CSeR wont be showing any change until the AST is complete, means if you are typing a “i” as in ““i”nt i=0;”, until user types the lAST charac-

ter ";" CSeR wont be doing any change calculation(But the positionupdaters are not AST aware, they will be still keeping the correspondence between the positions for every edit). Once the AST is complete CSeR starts processing, first fetch the block from the current file and find corresponding AST(Listing 11) from the other clone and finally compare, if it doesn't match show it as a change. In the example TestWorldApp the block in the current file will be Code 3. So in this section the question is how to compare two ASTNodes (3, 11) which are of type Block.

Code 3 AnchorAST after inserting a new statement in TestWorldApp.

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

In case of Block comparison, break down each Block into non overlapping statements and expressions, then compare them individually

How to find non overlapping statements and expressions in a Block? If you use the statements as defined by the JLS3 [4] grammar there will be overlapping statements which is not supported by the existing design. CSeR has visitors which visits the Statements and will give a list of non overlapping statements and expressions. The visiting nodes of the Statement visitor are listed in Table 3.17. As an example only the expressions are visited for a "ForStatement" since all the statements inside the "for" loop will be will visited by some of the other visiting nodes. Figure 3.5 shows how a simple for loop is parsed by the Statement visitor.

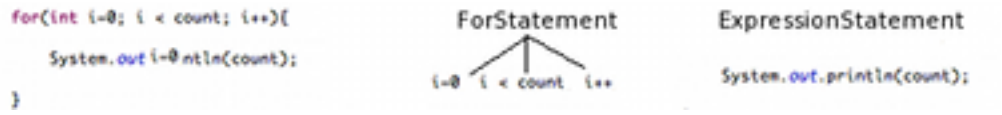

Figure 3.5: Statement visitor visiting a for loop

The output of TestWorldApp and HellWorldApp after the StatementVisitor is shown in

[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");

Table 3.16. Here again we don't convert the positions to AST nodes, we will keep them as positions itself for further processing. In the table it is shown as ASTNodes 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 set of positions such that $p_1{}', p_2{}', \cdots, p_N{}'$ be positions in Current file and $p_1, p_2, \cdots, p_M$ be positions in Original file. Let P_c and P_o be positions of current and original Blocks respectively. $P_c \subset p'$ and $P_o \subset p$. The initial inserted node positions is any position p_i, where $Cor'(p_i')$ (Inverse Corresponding function) is not defined. Similarly initial deleted node position is any position p_j where $Cor(p_j)$ (Corresponding 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 original file. Line 3 will find

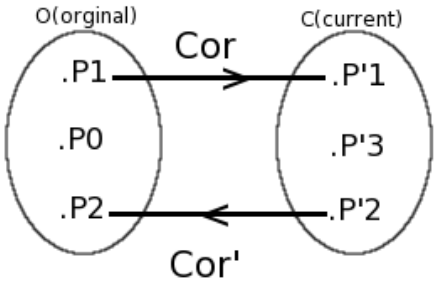

Figure 3.6: Correspondence Relation

the corresponding position of each original positions. Line 4 finds the difference of that positions with the original positions from the original file. Finally "finalPosition" may have two type of positions, ones which exist in the original positions but not in current positions means the deleted positions, others which exist in current positions but not in original positions means newly inserted positions. We mark them 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? The initial inserted nodes or deleted node positions are the positions without correspondence. As seen from the Figure 3.6, $P_3{}'$ and P_0 do not have correspondence position on the other side. Inserting a new node or removing a node is a situation where this can happen but this is not the only situation. As an example, "undo" operation is another situation.

Consider a statement is removed and then typed back as the same sentence again. Another technical detail involved here is whenever an AST is removed from file, the position linked with it is removed. The position updaters linked with it also will be removed.

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

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

Algorithm 2 CMPSTATEMENTS - Comparing statements

Require: currentBlock originalBlock
Ensure: currentBlock, originalBlock not null

```
1: currentPositions ⇐ geStatementPositions(currentBlock)
2: originalPositions ⇐ geStatementPositions(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)
```

Typing/Pasting that AST back will not bring back either the position or the position updaters. So there is a chance that once deleted nodes can be inserted back.

In the case where deleted nodes are inserted back, inserted and deleted nodes have to be processed further to see whether they are final inserted/deleted nodes. Algorithm PROCESSINSDELNODES does that for CSeR.

Algorithm PROCESSINSDELNODES starts by assuming the final deleted and inserted positions are the same. Then it iterates over insertedPositions and for every insertedPositions it iterates over deletedPositions. For every such iteration it will check the inserted node first, and the deleted node match (line 5). This matching algorithm, Algorithm 4 is just the default ASTmatcher that is shipped with JDT. Once it is found matching it can be of two cases, first they can be identical, second they can be similar.

What are "identical" statements? To define identical nodes in CSeR, we introduce two more terms for statement comparison, upper and lower statement. As the name suggests, the statement which appears above the statement is the upper statement and the one which appears below is the the lower statement. For example, "int i=0;" is the upper statement for the statement "System. out. println("Test World");" and lower statement of

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
```

it is null. Similarly, the upper statement of "int i=0;" is null. CSeR considers a statement in current file to be identical to the statement in original file if both the statements are matching and also both the statements have matching neighbors. Matching neighbors means the upper statement of that statement and the upper statement of the original match. Same is the case of lower statement. This is explained in Algorithm 5. In short for statement level comparison we consider the order in which statements occur. A very usual case of identical nodes would be undo operations.

CSeR assumes a statement in current file with no correspondence to be identical to another statement in original file with no correspondence if both the statements are matching and have matching neighbors.

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 don't match"
7: end if
```

Algorithm 5 AREIDENTITCAL - Checking Identical node, Undo Operations

Require: Position1, Position2, PositionList1, PositionList2
Ensure: Position1, PositionList1 ∈ originalFile AND Position2, PositionList2 ∈ currentFile

```
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
```

What are "moved statements"? If a newly inserted statement in current file matches with a deleted statement in original file but are not identical means they differ in the order they occur in current and original file, those statements are called move operations. Algorithm PROCESSINSDELNODES 3 finds out moved statements for CSeR.

What are "updated" statements If a newly inserted statement in current file is of the same type as a deleted statement and levenshtein distance 2.21 is less than the configured value but they don't match, then CSeR defines it as an updated statement. Algorithm PROCESSINSDELNODES 3 finds out 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 appears to be very similar. We used CSeR to modify a method named "canEvaluate" in JavaDocContext to convert to the

[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 "Postion1 is not an update of Position2"
9: end if
```

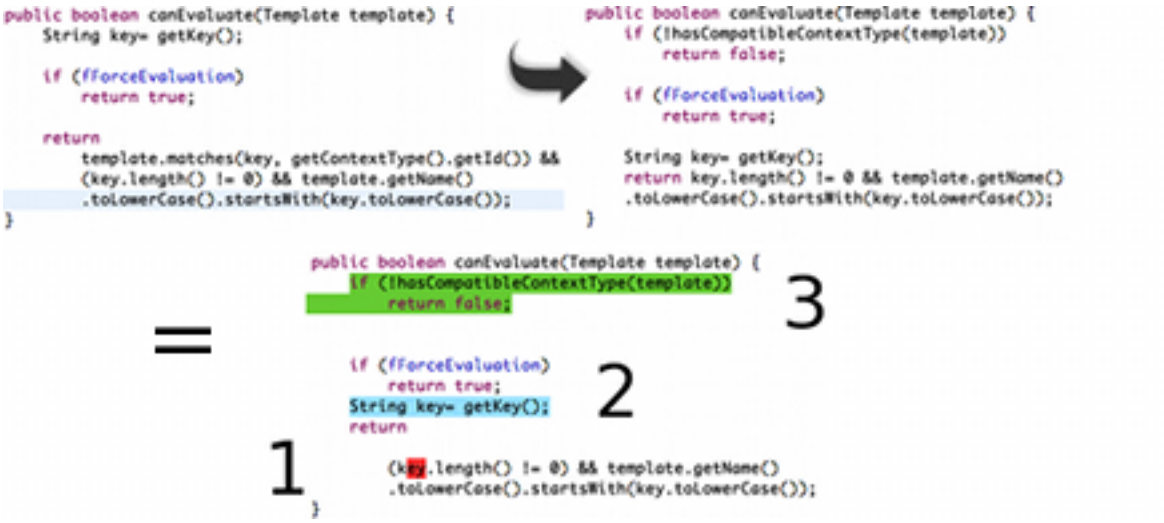

Figure 3.7: Statement changes in CSeR Editor using "canEvaluate" method in JavaContext and JavaDocContext

corresponding method in JavaContext. Changes that appear in CSeR is shown in Figure 3.7. Change that is marked "3" shows the insertion of new "IfStatement", change that is marked "2" shows a move operation and finally "1" shows the delete operation.

3.2.5 Body Declarations

From the TRAPEDIT algorithm, when the anchorAST is TypeDeclaration node we need to compare the BodyDeclarations 2.10. Let us try introducing a change in the TestWorldApp. For the Code 4 anchorAST would be entire class TestWorldApp. In this case also, inserting can be either pasting a code block or typing it. Again CSeR wont be showing any change

until the AST is complete, means if you are typing a “p” as in ““p”rivate int i=0;”, until user types the lAST character “;” CSeR wont do any change calculatation (but PositionUpdaters will be working for any edit irrespective of its occurrence, say TypeDeclaration, Block or any node). Once the AST is complete CSeR starts processing. It first fetches the classdeclaration from the current file and finds the corresponding AST from the original file and finally compares the both. If they do not match, it is shown up as 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.
}
```

Thus, once we have the typedeclaration we have to find the bodydeclarations that can occur inside the class but outside methods and then follow a field level comparison to find the difference. An important difference in this case with the lAST one is the order of the occurrence of fields. Order of fields do not change the meaning of a class in Java. Another difference is the way in which fields are calculated in the two classes. Here also we need to have non overlapping field positions.

How to find Bodydeclarations? According to JLS3 the statements in JDT can be any of the following listed in Table 3.19. CSeR has visitors which visits the nodes listed in Table 3.19. So we make the node from the current file as well as from the clone to pass through the visitor which will give us the list of bodydeclarations. The output of TestWorldApp is shown in Code 5

Table 3.19: **Bodydeclarations specified by JLS3 grammar**

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

Code 5 Bodydeclarations given from the visitor for TestWorldApp

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

2. private int i=0;

How are bodydeclarations compared? As mentioned earlier Bodydeclarations2.10 are different from statements and they don't necassarily have to be neighbours to be identical. So the same CMPSTATEMENTS algorithm can be used to find the inital inserted and deleted fields, but once they are calculated there is a slight change in the PROCESSINSDEL algorithm, we have to remove the neighbour checking part. Other than this Bodydeclarations follow the same processing as those of statements.

The Algorithm 3 (PROCESSINSDELNODES) is slightly modified to manage the bodydeclarations 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 appears to be very similar. We used CSeR to modify the Bodydeclarations inside NewClassAnnotationWizardPage to convert to NewAnnotationWizardPage. Changes that appear in CSeR

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

Algorithm 7 INSDELBDDLS - Comparing Bodydeclarations Ctnd...

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
```

are shown in Figure 3.8. Change that is marked "2" shows the insertion of new field, change that is marked "1" shows a delete operation, the nodes deleted shown on mouse hovering.

3.2.6 Nodes

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

But a careful analysis would reveal that in this case, anchorAST would be one among the values in Table 3.20. For the Code 6 it would be StringLiteral node. In this case also, inserting can be either pasting a code block or typing it. Here also CSeR makes the assumption that the developer keeps the AST correct.

Once the AST is complete, CSeR starts processing. It first fetchs the node from the current file, here it is "Test World" and finds corresponding AST from the other clone, here it is "Hello World" and finally compares the both. Since they are not same, it will be shown as a change.

What if user is editing in the margin of an ASTNode? In this case TRAPEDIT algorithm would return the parent of the editing node. In case of the example above, it would return the node "System.out.println("Test World")" and then we split this node into small units, the same way we divided the Block into statements, and process unit by unit.

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

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

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, we have visitors with visiting nodes specified in Table 3.20. Making the nodes passing through the visitors will give the smaller non overlapping units which have to 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 with a single difference in the implementation of $diff$ function. Consider in the above example if the user is making a modification in the boundary of 'System.out.println("Test World"), Here there is one, change already at the "TestWorld", if we have the same implementation for $diff$ it wont go in the list of initial deleted or inserted nodes as it has correspondence. Hence the only important change in Node Comparison is the change in

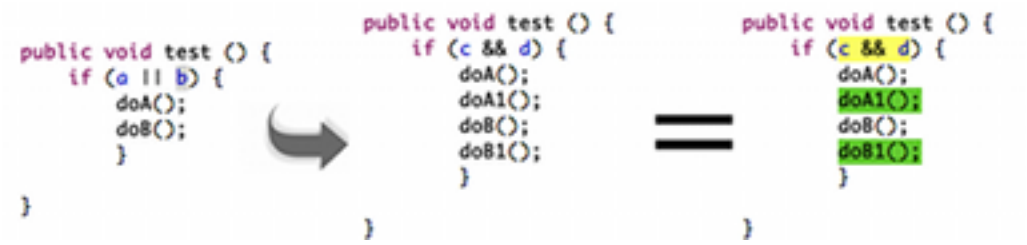

Figure 3.10: Conditional Statement Case 1 : Update Expression

$diff$ implementation. 8 shows how this is done. In the algorithm the function *undid* is supposed to return all the nodes which have correspondence on both sides.

Algorithm 8 NODEDIFF - diff Implementation for nodes

Require: currentPositions originalPosWithCorr originalPositions

1: initialdeleteInsertPositions $\Leftarrow$ $diff$(currentPositions,orgPoswithCorr)
2: positionWithCorrespondence $\Leftarrow$ *undid*(currentPositions,orgPosWithCorr)
3: **for all** position in positionWithCorrespondence **do**
4: **if** $\not$*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] project. The classes FieldTag and FileNameTag appears to be very similar. We used CSeR to modify the only different node inside FieldTag to convert to FileNameTag. Changes that appear in CSeR are shown in Figure 3.9. Change that is marked "1" shows the update of the node, shows the old node on mouse hovering.

3.3 Sample Scenarios

3.3.1 Conditional Statements

Five different cases of Conditional statements are explained below.

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

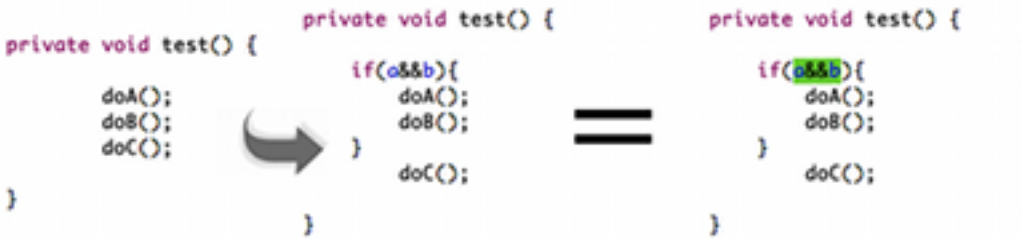

Figure 3.11: Conditional Statement Case 2 : Insert condition

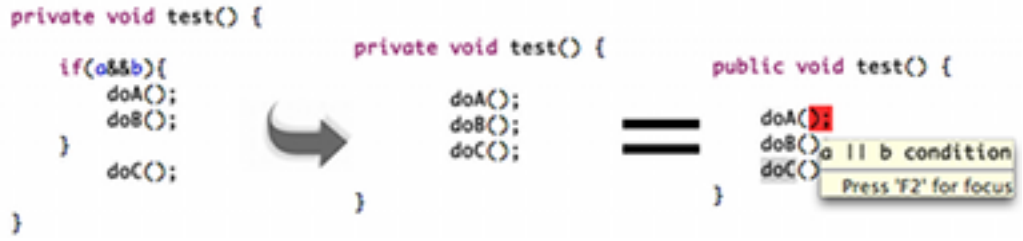

Figure 3.12: Conditional Statement Case 3 : Remove Condition

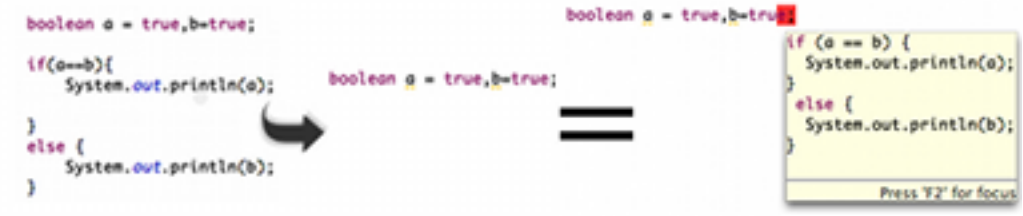

Figure 3.13: Condition Statement Case 4 : Delete An If Statement

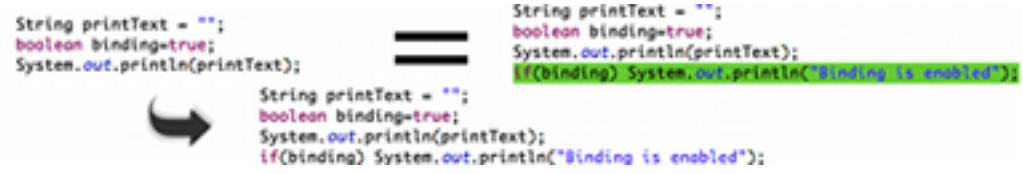

Figure 3.14: Conditional Statement Case 5 : Inserting A Complete If Statement

Figure 3.15: Inserting new arguments

Figure 3.16: Flattening arguments

3.3.2 Arguments

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

3.3.3 Array Initializer

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

3.3.4 Comments

An example of commenting code is shown in Figure 3.18.

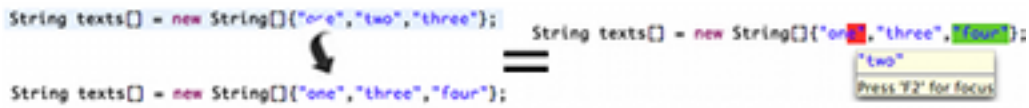

Figure 3.17: Modifying an array initializer

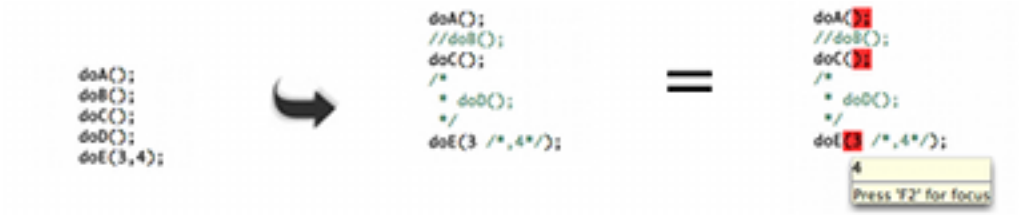

Figure 3.18: Commenting source code

Chapter 4

Validation

4.1 Robustness

In this section we analyze the robustness of the tool. Since CSeR is adding some more features to an existing editor we need to make sure that none of the existing features of the current editor are affected by the new features. Eclipse being an editor with lot of features there is no other way to do this, other than to use the tool itself with other tools in Eclipse. We did almost 50 test cases to ensure that existing features are not affected. This experiment would assure us that our tool works for normal scenarios but a tool is called robust when it can handle the extreme cases. To handle the extreme cases we need to analyze some of the user edits. This is explained in next section.

4.1.1 User Editing

As explained while explaining the requirements there are actions and goals while editing a source file. Same goal can be achieved with different actions. A tool can be robust if it depends only on the goal and not on the actions. Since CSeR is a tool which makes comparisons on every user edits, if necessary the actions play an important role. So here in this section we consider different types of user edits. [17] has heavily contributed to this section.

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		×	

Table 4.1 shows different kinds of useredits and its support for CSeR. Name is referred as anything which is not a keyword in Java. Names can be method names, class names, variable names . Lists corresponds to structures which appear between list delimiters such as {}'s surrounding lists of statements and the () 's surrounding lists of parameters. List elements are delimited by single characters such as ;'s between statements and ,'s between parameters. Actions 2.14 and Goals 2.15 are explained in Definitions chapter.

4.1.2 Special Scenarios

Type Change Edit As a design detail, two important operations that take place during every key stroke are AST comparison and position updation. The positins updation is meant to update the database of CSeR such that for every tracked position there should be an AST. The complexity of AST comparison is determined by the size of AST while complexity of position updation is number of tracked positions. Since the number of tracked positions can be as big as few hundreds we have to keep the operation for a single tracked position as light-weight as possible. Hence CSeR doesn't do any AST parsing in position updation

and this makes the position updation AST unaware. In short the position updation is AST unaware while AST comparison is AST aware.

In the Figure 4.1, while the user types method "doB();" with in the bounds of the "doA();", AST unaware position updation assumes that as an update of the method and include the position of "doB();" also with "doA();". So now the CSeR database would be $(x, 6) \leftrightarrow (x, 12)$, where x is any offset value. As mentioned earlier CSeR always expects all the tracked positions to give an AST for comparison, but in this case second position won't give an AST as this containes two AST nodes instead of one. This is called Type Change Edit as this edit changes a method to two methods or the type of the AST is changed.

This issue can be solved if we can remove the corrupted position from the CSeR database, We have two places where this can be done. First inside the position updation. So we have to make position updation AST aware without sacrificing performance. To achieve that while doing position updation, we have to find the position updaters which involves the editing position. We have to include a check inside those position updaters, whether they give an AST if not remove the position from CSeR database. Second in inside the AST Comparison, if a tracked position doesn't give an AST remove it from the database.

Let us see what happens for the example specified above. Here user finished editing and we removed the tracked position corresponding to "doA();", Once CSeR calculate the deleted and inserted nodes, there will be two inserted nodes and one deleted node Since "doA();" is included in both places and both have identical neighbors, statements would be considered same and no markers are added, while an insert marker is added for "doB();"

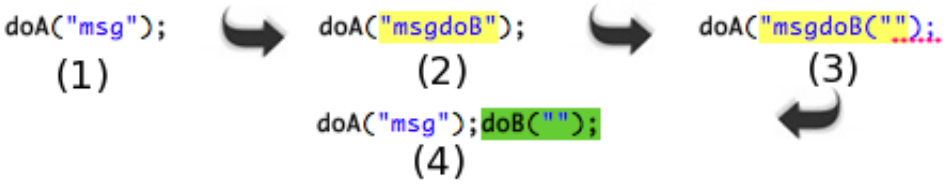

Figure 4.1: Type Change Edit

Unexpected Consistent State CSeR does not compare until there is a valid AST, but there can be situations where a valid state is reached which is AST complete but user has not reached the state he wanted. Consider the Figure 4.2 user want to insert " int j=0" between the two given lines. As shown in Figure a consistent state is reached on step 4, with two statements "int i=0;" and "int j=i++" and on comparison CSeR finds that second statement is new and one statement is deleted "i++". So it put a red mark for that and mark the new statement as green. Once he finishes typing, CSeR recalculates the changes and shows as in step 5.

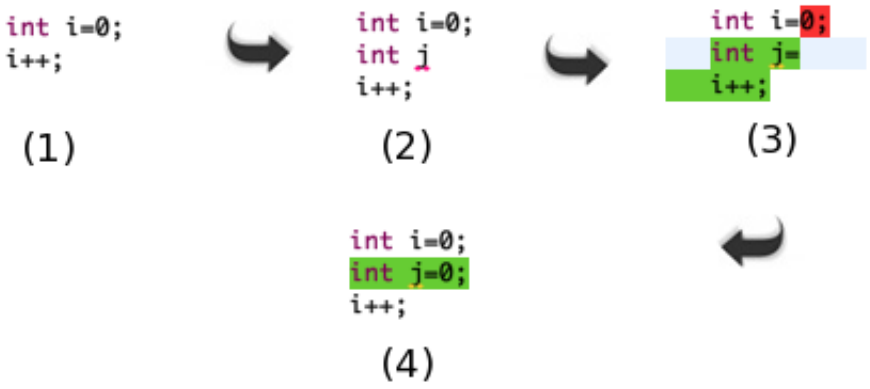

Figure 4.2: Unexpected Consistent State

4.2 Comparison with Existing Tools

In this section we analyze some of the existing tools. We show how some of the existing tools which appear to replace CSeR cannot completely or accurately address the problem we have. Most of the tools analyzed here are source differencing tools. As a general distinction from all the tools from CSeR is, CSeR is not another tool to find difference between two source files at a point, it calculates the differences of two files which were identical at some point and it calculates the differences as an incremental process.

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

There are many text based tools available, most of them make use of the diff algorithmn and the algorithm is based on solving the LCS(Longest Common Subsequence) problem. As this is developed for text files it has obvious disadvantages when used for Java source code. The algorithm won't be able to distingush between source code and comments. Order of fields and methods doesn't change the meaning of class in Java, while for any tool based on diff, this is not the case.

Another disadvantage of this, if the position of two methods are interchanged and say first method is five lines long while the second is ten lines, diff will sacrfice the first method and show the correspondence between the second methods in both the classes. So in short we lose the correspondence between smaller methods or fields in case of move operations.

Compare Editor by default is using an objectified version of diff[1]. So even it is able to identify the java token and show differences in terms of tokens, it will still have some issues since the base algorithm is written for text files. Hence the moved methods and fields will not be identified and even can give wrong information, means they may show correspondence between two methods even though there is a better correspondence.

As an example consider the case taken from JavaContext and JavaDocContext. An analysis of both the classes would reveal that the correspondences are wrong, because the method "evaluate" in the right has a corresponding method "evaluate" on the left side but Compare-Editor consider it as a deleted method. Meanwhile the method "getCharacterBeforeStart" is removed from the JavaContext and it has no correspondence on other side. Figure 4.3 shows the wrong correspondence representation of Compare-Editor. This is because the "evaluate" method is moved in position in the second class.

Version-Editor[1] is a tool which provides tight integration of the revision history and the editor. It makes the version data directly available at the right time in the right context. With in the editor it shows, changes made in the file so far, lines newly added, creation time, status of the change along with some other details. All the changes are calculated

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

with verion in repository. But the comparison is line based and hence all the issues of text based tools over AST based tools is applicable in this case also.

Figure 4.3: Wrong Correspondence in Eclipse Compare Editor

4.2.2 AST Based Tools- Breakaway and ChangeDistiller

Breakaway[5] is for Generalization tasks, it identifies the detailed structural correspondence between class and prints out the correspondence calculated in console. ChangeDistiller[8] is intended to find changes between multiple versions of code. It compares the trees by using the change detection algorithm for hierarchically structured information, the outcome is an edit script which can convert the original version of the tree to the modified one. Both work on by comparing the AST tree of two source files. As explained here both of them don't exactly address the problem we consider. Changedistiller is supposed to work for different versions of a single source file, hence ChangeDistiller expects a higher level similarity, having same class name and will not be useful for the cases to compare two different classes . Breakaway on the other hand is used for generalization process and expects classes is to be similar. Both the tools are different from other source comparison tools since the comparison is AST based.

"Level mismatch"[5] is a common error for tree comparison algorithms, it refers to the situation where a statement node is not found to correspond with, even though they exist at a different level of AST. For example one of the statement is in method declaration and the other is in an if condition in a method declaration. Figure 3.11 3.12 shows CSeR deal with situations like that. Here CSeR is still able to keep the correspondence between the statements even the level of statements has changed.

Comparing to both the tools CSeR has an upper hand because we start tracking changes

from the time the classes are identical and do the tracking on a continuos process. The changes are tracked based on the user-edit and also with respect to the AST, user edited. As an example, if the parameters in a method declaration are reordered or modified such that the signatures won't correspond Changedistiller fails to find the correspondence as they don't consider the statements inside the method for finding correspondence between two methods. This can result in bad results. In case of Breakaway the fail to consider the neighbors. As an example, While comparing two classes have two fields each of the same type, Breakaway may show wrong correspondence as they don't consider the order or neighbors of each fields. Breakaway also fail to consider statements inside a method while comparing two method declarations. CSeR on the other hand compares do take care of neighbors while comparing statements. In CSeR statement with corresponding function undefined in current file is considered same with another statement in original file only if the statements match and also they have matching or neighbors.

Breakaway applies a greedy algorithm and takes the first corresponding element that is greater then its threshold value. Conceptual disconnect errors referred to in Section 5.1.2 [5] occurs when you have the common elements String, =, and "" that is enough similarity to be greater then the threshold value. Since they don't look for best match they result in "Unordered mismatch and Ordered mismatch" [5]. As seen from the design those errors are irrelevant to CSeR.

ChangeDistiller is an extension of the algorithm for extracting changes in hierarchically structured data by Chawathe et al [3]. It is limited in finding the appropriate number of move operations. In particular, the performances of parameter ordering changes and statement ordering changes. As mentioned in limitation section of [8] changes in parameters can affect finding overall correspondence of the class. Figure 3.15 shows how CSeR handles this changes and this is completely independent of finding the changes inside the method, because changes in parameter are calculated when user edits inside the parameters and changes in statements are calculated when he edits inside a method.

4.2.3 Clone Detectors

A small experiment was conducted to make a study of the clones from clone detectors. The clone detector used in the study is CCFinder [2] . CCFinder was used with the default configuration. All the files used in the examples are relevant as those files are later used for experiment with CSeR as explained in section 4.3.1. The results of the experiment is shown in Table 4.2. Here we consider 5 clone groups. LOC refers to line of code, Missed LOC refers to lines which should have correspondence but was missed by CCFinder, #Clones is the number of clones detected by CCFinder. Missed LOC doesn't include the lines which are only in one file, nor the spaces or comments, It is the number of lines which are corresponding but missed by CCFinder.

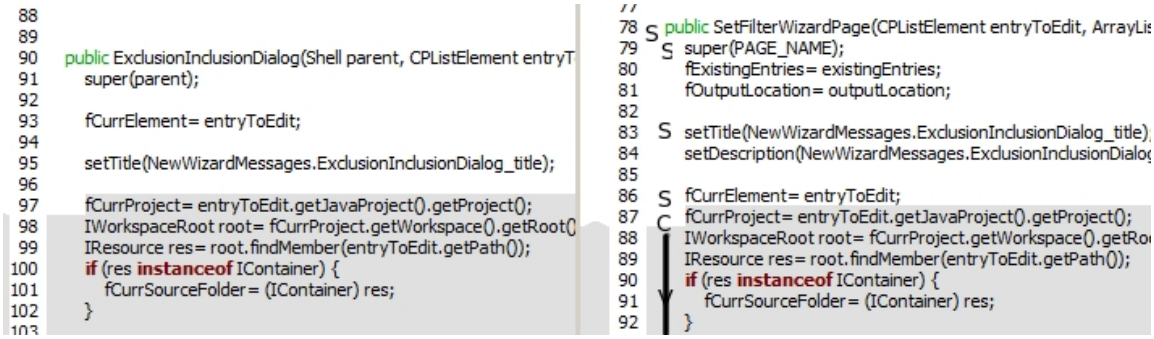

Figure 4.4: Calculation of Missed LOC

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. The lines marked "S" shows those lines are which are treated as skipped lines, those marked "C" are those corresponding lines calculated by CCFinder. The other lines as you can see from the figure exist only in one of the files. In the figure comparison is done with respect the first file on every row.

An ideal case for the experiment would, the clone detector finding a single clone without missed lines. We didn't have single case for the experiment which involved 5 clone groups of 14 files. Most of the cases CCFinder were finding clones for a part of the class as it is

[2] www.ccfinder.net

clear from the Table 4.2(value of #clones). Even if there is only one clone (Clone group 5), there are skipped lines which shows the clone doesn't include the entire class.

In short clone detectors fail to give a high level picture. Even the correspondence given by the code blocks are not complete and accurate. There can be correspondences which are left out by the clone detectors. Our small experiment has revealed that the correspondence between classes can become worse when the location of method declarations and fields are changed and this 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 any tool is the difference it make while using it. This can only be proved by using and showing the difference with the tool. In the context of CSeR, we claim it as a good source of source code documentation and also a guide for the similar activities in future. In the last sections we have the seen the problems with the existing tools which wont correctly or completely address the issues. In this section we prove how effectively CSeR address the issue, How efficiently CSeR manages the clones, How the information is shown and How can programmer benefit from it. We identified some test cases and applied

CSeR on it. The test cases are also a proof for the robustness of the tool, which shows that the tool is made for real world scenario.

4.3.1 Experiment Setup

First of all we need to identify some classes or code which might be created by copy and paste Then we repeat the scenario with CSeR. Identifying proper candidates for our experiment is not an easy job. Our approach is to use a clone detector, CCFinder [3] [13] as a starting point and manually go through the classes in the same package or in related packages. If we find two classes are similar, we screen them for further analysis to see the chances that those are created by copy and paste. To do that, we compare the positions and spacing of statements, comments first at a class level then to a method level and even to a statements level. We even have a case from the JLCP [4] project, where a class file there are two authors, the guy who copied the file from some other file with another author forgot to remove the comments. So once you identify some files which are similar we have to pair them. It is sometimes a tougher decision to identify what is copied from what, or what is the original file and what is the current file. But since CSeR supports two-way correspondence it wont affect our experiments. Since CSeR with the current implementation supports only pair implementation we need to identify pairs, CSeR always shows changes with respect another file. So if you find 3 classes which are similar, we have to split them into two pairs. Say you have two Java files Class A, Class B which are similar we copy Class A and try to make Class B from the copied version. The changes that CSeR shows are tracked and an analysis of these change is done 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. First from Eclipse project, more specifically from JDT UI, JDT Core and SWT projects. Eclipse, the famous open source project is known to be written by efficient programmers is a good source for efficiently

[3] http://www.ccfinder.net
[4] http://sourceforge.net/projects/gotjava

written Java code. Secondly Java Lobby Community Platform (JLCP), aims to write a number of components to produce a a free Java Portal site based on an internal forum system. JLCP, a web based business project which act as another good test case. Finally the literature clones, usually they deal with extreme cases, and also they are diverse domains. Usually they show the most common cases or more generalized cases, so a tool can pass those test cases will certainly help the programmer to a good extend.

To have a more clear picture about the experiments conducted, we will explain two case studies in detail.

4.3.2 Case Study 1 SetFilterWizardPage And ExclusionInclusionDialog

ExclusionInclusionDialog and SetFilterWizardPage is an interesting case, as they belong to different type hierarchy(parent class of ExclusionDialog is StatusDialog while SetFilterWizardPage is NewElementWizardPage) and yet they are very similar. Details regarding the changes are shown in Table 4.3, 4.4 and a part of code is shown in Figure 4.5. As a rule of thumb red marker shows deletion, green shows insert and yellow for update operations.

From the Figure 4.5, the differences start from the parent, SetFilterWizardPage is a child of NewElementWizardPage while ExclusionInclusionDialog is a child of StatusDialog, this change is marked "1" and is categorized as others in the update change type in Table 4.4. The change marked "2" and "3" shows deletion of 3 fields from the class. The deleted fields can be seen on hovering the mouse near the marker, it is shown in the figure for change "2". Similarly "7" corresponds to the deletion of two statements. As mentioned earlier when there is a delete CSeR always look for a place to fix the marker for the delete operation. Since in this case two statements after "super(parent)" is deleted, CSeR put a marker after that statement ("7"). Change "8" corresponds to a move operation, "fCurrElement=entryToEdit" statement is moved a statement before. The changes "4", "5", "6" corresponds to parameter changes, "5" indicates deletion of two parameters "ArrayList existingEntries" and "IPath outputLocation" while "4", "6" are marked in green showing that they are newly inserted parameters. The inner class in the figure is not shown any

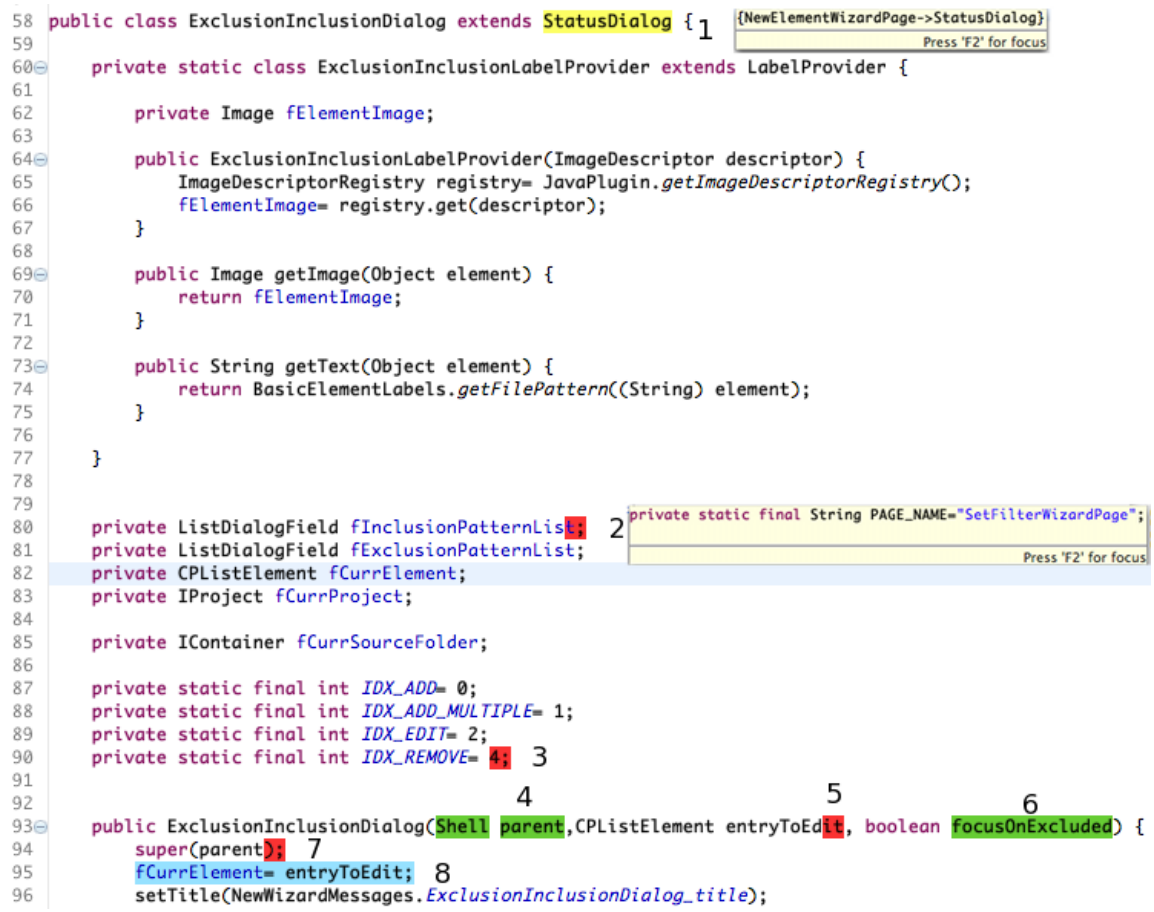

Figure 4.5: CSeR Showing SetFilterWizardPage from ExclusionInclusionDialog

changes as it has no change from the original file other than the position and position on inner class declaration doesn't change the meaning of the class.

4.3.3 Case Study 2 NewClassCreationWizard And Clones

From a quick comparison of the wizards to create Class, Interface, Enum and Annotation, it is obvious that they share a good part of the code. This case study includes 5 classes NewClassCreationWizard, NewEnumCreationWizard, NewAnnotationCreationWizard and NewInterfaceCreationWizard. The changes for a part of code with in the editor are shown in Figure 4.6. Analysis of changes are done in Table 4.4

The figure is with respect to the NewAnnotationCreationWizard. Here the changes are very less, there are only 5 changes. The changes "1", "2" and "5" are variable type changes

Figure 4.6: CSeR Showing NewAnnotationCreationWizard from NewClassCreationWizard

while "3" and "4" are variable name changes. Every update change carry a message showing the original node and it is shown along with the change "5".

Exactly same change is done for both NewInterfaceCreationWizard and NewEnumCreationWizard, In place of NewAnnotationWizardPage it is NewEnumWizardPage and NewInterfaceWizardPage respectively. Similarly the variable names are also changed in very similar fashion. This is in perfect agreement to the argument that programmers copy and paste templates not the code itself.

4.3.4 Result Analysis

The results of the experiments is summarized in Table 4.3, Table 4.4, Table 4.5, Table4.6. A change which was avoided for analysis in all cases is the name of class file, means when you copy NewClassCreationWizard and paste as NewAnnotationCreationWizard, the second class will go through eclipse re-factoring and will change all occurrence of NewClassCreationWizard to NewAnnotationCreationWizard as this is a very obvious change we are not

highlighting that change in the code or included in the data below. After analyzing the changes in different classes we come up with categories for each type of change and the categories for each type of change are explained below.

Division of Changes The division of changes is just a study of changes with CSeR it has no direct relationship with the design of CSeR. As a general point we wont consider overlapping changes. If we have a new method we consider only one change, insert change of method declaration, even though the statements inside the method are also new, we won't consider that, same with the statements, if we have new statements we won't consider the expressions inside that statement. The division based on the experience we gained after analysing the changes. As an example if we had a change called expression update, it would include variable name change, literal change , method invocation name change but since we have ample cases of variable name change and others we decided to treat each of them seperate. Again while considering the literal change we can have string,number and character literal but since our test cases motivated to consider them together than separate.

Update Changes As can be seen from the Table 4.6 major part of the changes was update and in update major portion is covered by variable name and type updates(V,T). An update change is considered as this when the variable name or the type is changed completely or a portion of the old one to new one. An example is, In statement "Expression statement = null" when changed to "Expression expression=null" it can be considered as (V), If it was changed to "Body statement =null" then it is considered as (T) change. Changing the return type of a method is also considered as a type change. L stands for Literal means any string literal, number literal or character literal change is included with this. Method changes can be two ways either in the definition or in the method invocation. There are cases (Case Study 1) where method has to be renamed. In that case the places where that method is called with in the class also has to be changed, both the declaration and invocation changes are considered as method changes.

Special cases in V include "return null → return variableName", " variableName →

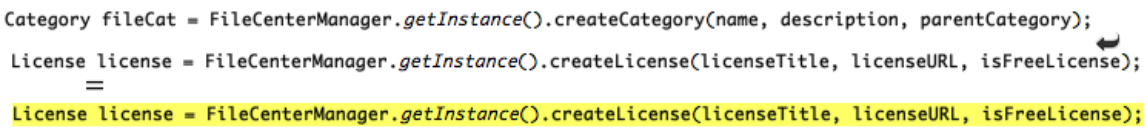

Figure 4.7: Statement Update change

methodName" also. Consider a case as shown in Figure 4.7, it is better to consider it as a statement update rather than few other simple name updates. More details about statement updates are explained a little later. Other updates also include the case where only and "else block" or "else if" are deleted and rest of the statement is maintained. These two are included in the other(O) category.

Statement Updates From the Figure 4.7 it is not possible to consider the statement as an update. But after a careful consideration of the context we get sufficient reasons to argue that the statement is update. In the example give above if we consider individual changes then there would be more than 6 updates, which involve change in name,type method name. Instead of considering this we treat these changes as a single update, that is the statement update.

Insert & Delete Changes S (Statement), M (Method Declaration), F (Field Declaration) are obvious changes. The statement can be an if statement, method invocation in case of anonymous inner classes it is included with C (Class Declaration), C also includes inner classes. P (Parameters in the method declaration). E (Expression) has a bigger domain, it includes the adding new members into the array initializer, adding just a "if" condition, adding new arguments for method invocation, adding new expression to an existing expression. Delete is applicable for all the cases mentioned.

Move Changes S(Statement), M (Method Declaration) and C (Class Declaration) includes the same way as mentioned above but here the operation is 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 (callheirarchy) (185)								
16	HistoryListAction(typeheirarchy) (207)		11		3		1	1	
	HistoryDropDownAction (callheirarchy) (109)								
17	HistoryDropDownAction (typeheirarchy) (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 is based 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 constitute a major source of defects. This implies that cloning can be a substantial problem during developement and maintenance unless special care is taken to find and track existing clones and their evolution. In short, we conclude that changes in clones are frequent, and that changes in clones induce defects.

In the study explained in [10], we proposed the main design elements for proactive clone management that are aimed to provide better support for clone evolution. We described our initial experience with prototyping several features that partially implemented the design outlined for proactive support, including a consistent renaming utility CReN, and CSeR. Our design for proactive support,CnP was partially inspired by related work but differed from them in important ways. Clone case studies were used to motivate individual features and to identify or further refine design requirements. We are planning lab-based user studies to better understand these features and their effectiveness in helping programmers deal with clones.

CnP differs by tracking copied-and-pasted clones directly rather than relying on clone detection tools. As a result, CnP may support those clones that clone detection tools fail to

capture. Moreover, clone detection tools tend to have low precision and recall [2], which can be expensive for the programmer to sort through in batch processing. Proactive support can potentially ease this problem by distributing the effort over time.

Clonescape [4] and CPC [21] are recent projects aimed to develop proactive clone support. CnP differs from Clonescape and CPC in providing features like the accidental capture of external identifiers, consistent renaming (CReN), and clone diff view (CSeR).

CloneTracker [7] proposes a novel representation for clone locations that is independent of physical properties like character offsets or line ranges, and also supports a form of simultaneous editing by using Levenshtein Distances to locate similar lines between clones. But CloneTracker relies on clone detection. CnP may also perform more accurately in cases when Levenshtein Distances fail.

Context Sensitive Cut Copy and Paste (CSCC&P) [14] is a tool implemented in LAPIS[1] pattern matcher platform. The tools aims in detecting violations of a variety of common contextual relationships including semantic relationships. The tools get activated with copy or cut actions but it differs from CSeR in working.

Codelink [20] supports both clone diff views and simultaneous editing. It uses colors to indicate the commonalities between linked clones in blue and differences in yellow and elision to hide the 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 one implemented by the Unix "diff" utility) to determine the commonalities and differences of clones within a clone group. Toomim et al. report two main shortcomings of the LCS algorithm: its potentially long running time and lack of intuitive results. CnP's approach in differencing clones can potentially resolve these problems.

CReN and Rename Refactoring mainly differ in that CReN works in any user-specified region while Rename works only in pre-defined scopes like blocks and classes.

The Breakaway and Jigsaw tools automatically determine the detailed structural corre-

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

spondences between two classes and two methods, respectively [5, 6]. The input to Breakaway and Jigsaw are classes or methods that may contain different code. The input to CSeR, on the other hand, is always identical. CSeR *incrementally* tracks changes as they are made to the related clones.

Chapter 6

Conclusion & Future Work

Developers have to manage the clones in the project efficiently. Significant work has been in the field of clone detection, but very little work has been done in tracking "Copy and Paste"(CnP) operations, even though those operations play an immense role in the formation of clones. We have demonstrated the advantages of tracking CnP operations.

We have presented an approach for clone management by tracking CnP operations. By making use of this approach detailed differences between clones is available with in the editor. The changes are visualized in different colors for easy understanding. The comparison used is context-sensitive and AST based which make it unique. Finally we discussed a proof-of-concept implementation, CSeR(Code Segment Reuse) and its design.

We conducted a small experiment involving 9 pairs of classes to verify CSeR's advantage over clone detectors. The test cases were selected from an industry project. We have demonstrated CSeR's usefulness and robustness by considering 37 test cases selected from industrial projects as well as research publications. The 533 changes identified by CSeR from experiments were carefully analyzed and divided into 20 different changes.

Clone groups CSeR consider clones in pairs, Clone groups refer to cases where there are more than two clones, Say for an example when a class is pasted two times there are three clones and hence it is a clone group, CSeR as the current implementation consider it as two pairs. First clone pair includes first and second and second clone includes first and

third. So even though the second and third could be connected current implementation fails to identify the connection. So a better design would connect all the clones together as a group and hence should be able to view the difference of a file with respect to any other clones. As an example when NewAnnotationCreationWizard, NewEnumCreationWizard is created from NewClassCreationWizard developer would 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 big file and all of a sudden something is not working and he is not sure about what all changes he made. CSeR could be extended to track changes of code with in two time frames of editing. CSeR can even provide a capability to go back before an edit. Basically CSeR calculates the editing regions in clones. The edits are calculated based on AST tree and hence we can integrate with the eclipse template feature. As a rough idea, developer while pasting few lines of code will have an option to activate CSeR template, if he does CSeR calculates the editing regions and pass the information to the template model and they create the template with a key and it can be later used as templates.

Version Control Integration and Side by Side View While calculating changes, CSeR saves all the information related to a class file in a text file with the same name, and this file is read every time someone opens the class file. It could be extended to Version Control, If it can be done then changes could be shared between the users. Comparison could be done with earlier versions with the local machine or in the version control. CSeR doesn't have a side by side view, It would be nice if we could use the Eclipse compare editor with data read from CSeR. The comparison would be more accurate.

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 Jrg 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.