ABSTRACT

QUICK JAVA REFACTORING TOOL

By

Shendun Liu

August 2013

Object Oriented Programming(OOP) is a contemporary favorite method of programming. OOP provides better flexibility, source codes are more organized and systematized, and it enables a group of developers to easily work with each other. Nevertheless, a poorly designed system will not only defeat the intention of coding with OOP, but also will make the software extremely difficult to maintain.

Refactoring is a powerful way to improve existing code. It only changes the structure of the source code without changing its functionality. Manually refactoring larger systems not only consumes large amounts of time and money, but it also happens to be incredibly inaccurate. As a result, quick and easy refactoring with partial automation is extensively discussed in the software realm.

This thesis presents algorithms for implementing nine refactorings that work on the fly for JAVA source code. The refactorings algorithms are implemented in a stepwise manner by initially selecting the source code portion for refactoring, and by choosing the correct refactoring method, and then programmatically changing the selected source code to achieve refactoring.

QUICK JAVA REFACTORING TOOL

A THESIS

Presented to the Department of Computer Engineering

and Computer Science

California State University, Long Beach

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

Committee Members:

Michael Hoffman, Ph.D. (Chair)
Shui Lam, Ph.D.
Tracy Maples, Ph.D.

College Designee:

Burkhard Englert, Ph.D.

By Shendun Liu

B.A., 2004, Arizona State University

August 2013

ACKNOWLEDGEMENTS

I would never have been able to finish my dissertation without the guidance of my thesis advisor, help from my friends, and support from my family.

I would like to offer my sincerest gratitude to my advisor, Dr. Michael Hoffman, who has supported me throughout my thesis with his great patience, assistance, and immense knowledge. I could not have imagined having a better advisor and mentor for my MSCS study.

I would also like to thank my committee members for allowing me to utilize their time for my thesis defense.

Lastly, I would like to thank my family. My parents, both of them encouraged me throughout my study. My wife, Amy, who was always there cheering me up and standing by me; and my son, Alex, who was born while I started this thesis. He gives me the courage and hope to finish this thesis.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

FIGURE Page

CHAPTER 1

INTRODUCTION

Software maintenance is statistically proven to be expensive and difficult. There is also a sense that software maintenance is inevitable due to the fact that new technologies, such as new hardware are constantly being introduced; changes in law and regulations happen regularly; business entities are re-organized frequently; and lastly, customers' requirements are constantly changing and evolving.

According to Martin Fowler [1], "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure." Additionally, in an article titled "Refactoring-Way for Software Maintenance", Arora et al. said that through the practice of refactoring, one can accomplish re-modularization of the software, and make the software system easier to read, to understand, to extend, and to maintain [2]. Many of the refactorings described are common sense programming techniques, such as *move field;* some refactorings are common software design practice, such as *encapsulate field;* some refactorings improve readability and maintainability, such as *reverse conditional.*

As a result, refactoring is all about improving software design and structure. A poorly designed or engineered software system will cause problems for others. Source code decays over time; particularly, in the year 2000, the infamous Y2K bug drew worldwide fear and attention for software maintenance.

This thesis describes nine algorithms which implement nine refactorings. A prototype tool named Quick Java Refactoring Tool (QJRT) was created to validate the algorithms.

CHAPTER 2

REFACTORINGS

Refactoring is an important step in software maintenance. The first reason to refactor is that large software usually is developed by many people. Thus, different people usually have different thoughts on how to write the source code even if the project starts with a perfect UML (Unified Modeling Language) class diagram as its blueprint. In addition, software does age, and needs to be taken care of. As Pippin says in an online post, “I get the sense that tells me it is some of the best code I’ve written. It’s better than the code I wrote yesterday, and it’s better than the code I wrote a year ago” [3].

The classic approach to software development is the waterfall model. Let us assume the waterfall model worked great on some projects. However, after a period of time, new technology has been introduced, and the system must keep up in order to use the new technology (e.g., to enhance security). Refactoring comes in handy under such situations. “[It] is the process of changing a software system in such a way that it doesn’t alter the external behavior of the code yet improves its internal structure. [1]”

Refactoring is divided into four procedures: detect a problem, analyze the problem, find a solution, and change the source code accordingly [4]. To detect a problem, Fowler [1] used Bad Smells to refer to source code that can be improved. In order to analyze a problem, a series of questions must be asked, such as: Why is it necessary to change something? What are the benefits? Are there any risks [4]? To find a solution to the problem, the goal is eliminating potential bugs; yet, the outcome stays the same. Lastly, the source code must be modified to archive the goal.

Fowler discussed a total 72 refactorings in his book, and later provided a few more on his website [5]. The 72 refactorings are catalogued into seven areas: (1) composing methods; (2) moving features between objects; (3) organizing data; (4) simplifying conditional expressions; (5) making method calls simpler; (6) dealing with generalization; (7) big refactorings [1].

The scope of this thesis is limited to describing nine algorithms for implementing nine refactorings and developing a tool (QJRT) in the process. Those nine refactorings are categorized under three areas as follows:

1. Simplifying Conditional Expressions: Consolidate Conditional Expression (CCE) refactoring combines a sequence of conditional tests into a single conditional expression and extract it. Consolidate Duplicate Conditional Fragments (CDCF) refactoring moves the same fragment of code outside of the expression. Replace Nested Conditional with Guard Clauses (RNCGC) refactoring uses guard clauses for special cases and make clear the normal path of execution.

2. Dealing with Generalization: Pull Up Method (PUM) refactoring moves the specific methods with identical results from subclasses to the superclass. Pull Up Field (PUF) refactoring moves the specific identical field from subclasses to the superclass. Push Down Method (PDM) refactoring moves a specific method from the superclass to the only subclass in which the method makes sense. Push Down Field (PDF) refactoring moves a specific field from superclass to a subclass, in which the field is only used.

3. Others / Miscellaneous: Reverse Conditional (RC) refactoring makes negative conditional positive. Reverse Conditional Advance (RCA) refactoring changes the conditional to the opposite meaning.

All of them are described by Fowler [1], and later on Bill Murphy [5] and others' works were added to Fowler's refactoring website [1, 5]. Every refactoring is defined, and is followed by a sample of how the refactoring should be done in a manual fashion. QJRT provides quick and easy refactoring with knowledge of programming, which includes finding a bad smell, analyzing the problem, and then choosing the correct refactoring. The QJRT changes the source code for the developer on the spot, and the developer can verify the outcome of the finished code right away.

CHAPTER 3

SIMPLIFYING CONDITIONAL EXPRESSIONS

This section describes the algorithm for applying refactorings that simplify conditional expressions. The refactorings included in this section are Consolidate Conditional Expression (CCE), Consolidate Duplicate Conditional Fragments (CDCF), and Replace Nested Conditional with Guard Clauses (RNCGC).

As Martin Fowler[1] pointed out in his book, the conditional expressions have their own ways of getting logically tricky. The goal is to change the conditional expressions for better readability, while not changing the meaning of the expressions, after all, that is what refactoring is for.

CCE would be used when running multiple different if-statements' tests, no matter what the tests are, the results from the different if-statements would return the same. CDCF is applied to remove any duplicated codes within if-statement's body. RNCGC is used while an if-statement has nested conditionals, usually more than three nested level, where the codes are hard to read. Based on developer's preference, one or more of these three techniques can be applied to make the code more readable and tricky logic more understandable.

Consolidate Conditional Expression

The CCE basically involves combining all of the conditional expressions into one conditional expression. It results in less coding and better understanding. In addition, this step of refactoring makes the code highly cohesive.

Structure

```
double disabilityAmount() {
     if (_seniority < 2) return 0;
     if (_monthsDisabled > 12) return 0;
     if (_isPartTime) return 0;
     // compute the disability amount
```

```
 double disabilityAmount() {
     if ((_seniority<2)||
          (_monthsDisable > 12) ||
          (_isPartTime)) return 0;
     // compute the disability amount
```

FIGURE 1. Consolidate Conditional Expression [1].

Refactoring Steps

1. Developer opens the file from the File menu; the file's entire source code will be loaded in the left panel of the QJRT, and highlights the portion of the code s/he wants to refactor.

2. User clicks Select button, and QJRT extracts the selected code to the bottom panel for individual analysis.

3. Developer chooses the Consolidate Conditional Expression refactoring from the drop-down menu bar under Refactoring menu.

4. Upon choosing the refactoring method, QJRT displays CCE refactoring on a message label to present and confirm which refactoring is being executed. It also passes the parameter to QJRT to prepare CCE refactoring. And lastly, it sets the Refactoring button visible for user to click on.

5. User clicks Refactoring button, and this is where all the wonderful mechanics happen. QJRT takes the selected source code from the bottom panel, uses Abstract Syntax Tree (AST) to parse it. In detail, AST checks if the selected source codes are multiple if-statements. If all of them are valid if-statements, then AST checks individual if-statement, and CCE compares then- and else- Statement. If all thenStatements are same; meanwhile, all elseStatements are same; CCE will retrieve all expressions and constructs a new if-statement.

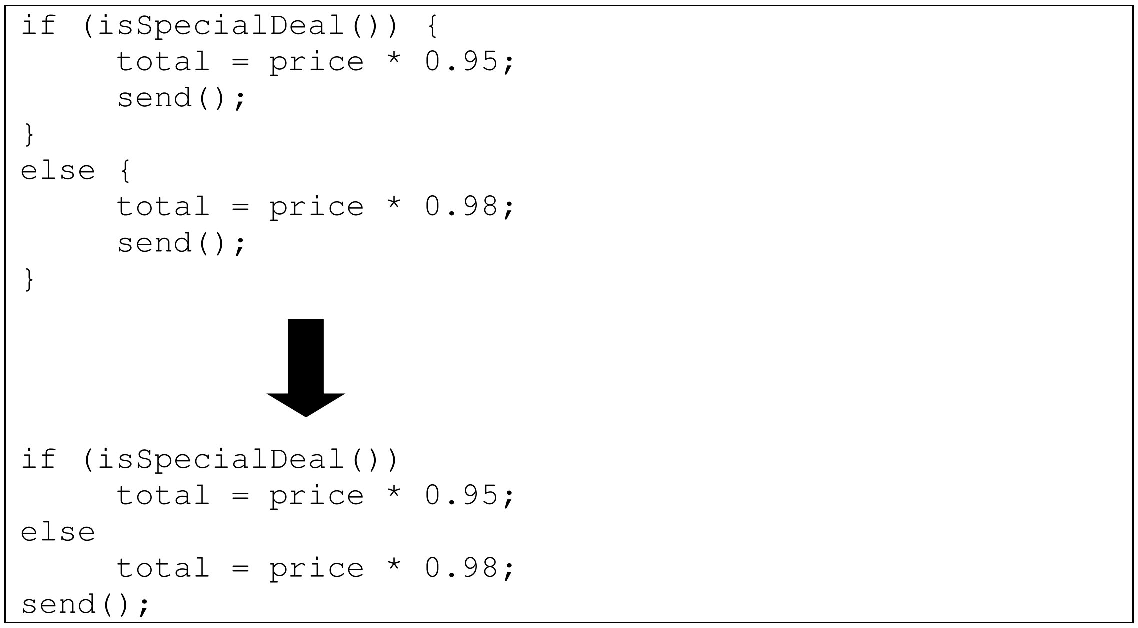
6. After the above step is completed in the background, the new and improved source code shows on the right panel of QJRT. The original source code is replaced with the refactored source code in the color red with highlights in yellow to show the difference. The unchanged source codes are still in black color. The left side panel with the original source codes remains the same.

7. Now, the developer has the option of saving the refactored source code. S/he simply clicks File menu, chooses the option "save as …" to save to a new file or to overwrite the old one. Nonetheless, if there are doubts, the developer can simply cancel or exit the whole process. An example of this refactoring is presented in Chapter 8 under heading "Sample Input and Validating of *Consolidate Conditional Expression*."

Consolidate Duplicate Conditional Fragments

CCE refactoring improves the source code for better readability and makes it more straightforward by combining the conditional expressions. Moreover, according to Folwer, CCE is important for two reasons. “First, it makes the check clearer by showing that you are really making a single check that’s or’ing the other checks together. The sequence has the same effect, but it communicates carrying out a sequence of separate checks that just happen to be done together. The second reason [...] is that it often sets you up for Extract Method” [1]. Similarly, the CDCF refactoring method eliminates the same fragment code showing all over inside the body of the if-statement. In other words, CDCF extracts the same fragments of code out for individual standing. This improves source code for better readability, and clarifies logically what changes and what does not.

Structure



```
if (isSpecialDeal()) {
     total = price * 0.95;
     send();
}
else {
     total = price * 0.98;
     send();
}
```

```
if (isSpecialDeal())
     total = price * 0.95;
else
     total = price * 0.98;
send();
```

FIGURE 2. Consolidate Duplicate Conditional Fragments [1].

Refactoring Steps

1. As usual developer opens the file from the File menu, and highlights the portion of the code s/he wants to refactor from the chosen file.

2. User clicks Select button, and QJRT extracts the selected code to the bottom panel for individual analysis.

3. Developer chooses the Consolidate Duplicate Conditional Fragments refactoring from the drop-down menu bar under Refactoring menu.

4. Upon choosing the refactoring method, QJRT displays CDCF refactoring on a message label to present and confirm which refactoring is being executed. It also passes the parameter to QJRT to prepare CDCF refactoring. And lastly, it sets the Refactoring button visible for user to click on.

5. Developer clicks Refactoring button, and QJRT takes the selected source code from the bottom panel and uses AST to parse it. AST verifies the selected code is valid if-statement, and also verifies there are any identical fragment in the thenStatement and elseStatement. Next step is QJRT determines the identical fragments' position. If the position matches, for example, both of them are at the beginning of the statement block; or at the end of the statement block. Then, CDCF extracts the identical fragment out, and put it either in front or at the end of the if-statement according to the position of the fragment.

6. After the refactorings are completed in the background, the new and improved source code shows on the right panel of QJRT. The original source code is replaced with the refactored source code in the color red with highlights in yellow to show the

difference. The unchanged source codes are still in black color. The left side panel with the original source codes remains the same.

7. Developer now has the option to save the refactored source code to a new file, or to overwrite the old one. S/he simply clicks File menu and from the drop-down menu, choose option "save as …" to save the file or file the old one to replace. Nonetheless, if there are doubts, developer can simply cancel or exit the whole process before the step to overwrite the original file. An example of this refactoring is presented in Chapter 8 under the heading "Sample Input and Validation of *Consolidate Duplicate Conditional Fragments*."

Replace Nested Conditional with Guard Clauses

RNCGC is used to re-arrange one normal condition per if-statement to the readers while eliminating the nested conditional. This action will make the codes look clean, as well as make the code logic easy to understand. Many software developers, including myself, intend to use many nested if-statements when deciding an outcome of a result. The way of using the nest if-statements is not illegal, but it certainly can be treated as insufficient, or in other words, very poorly designed. "If the method is clearer with one exit point, use one exit point; otherwise don't." [1]

Refactoring Steps

1. Developer opens the file from the File menu, and highlights the portion of the code s/he wants to refactor from the chosen file.

2. User clicks Select button, and QJRT extracts the selected code to the bottom panel for individual analysis.

Structure

```
double getPayAmount(){
     double result;
     if (_isDead) result = deadAmount();
     else {
          if (_isSeparated) result = sepratedAmount();
          else {
               if (_isRetired) result = retiredAmount();
               else result = normalPayAmount();
          };
     }
return result;
};
```

```
 double getPayAmount () {
     if (_isDead) return result = deadAmount();
     if (_isSearated) return result = separatedAmount();
     if (_isRetired) return result = retiredAmount();
     return result = normalPayAmount();
 }
```

FIGURE 3. Replace Nested Conditional with Guard Clauses [1].

3. Developer chooses Replace Nested Conditional with Guard Clauses refactoring from the drop-down menu bar under Refactoring menu.

4. Upon choosing the refactoring method, QJRT displays RNCGC refactoring on a message label to present and confirm which refactoring is being executed. It also passes the parameter to QJRT to prepare RNCGC refactoring. And lastly, it sets the Refactoring button visible for user to click on.

5. Developer clicks Refactoring button, QJRT takes the selected source code from the bottom panel, uses AST to parse it. In detail, AST checks if the selected source

code is valid if-statement. Then from the top level if-statement down to inner level if-statement, RNCGC spins off a new if-statement constructed with expression and thenStatement.

6. After the refactorings are completed in the background, the new and improved source code shows on the right panel of QJRT. The original source code is replaced with the refactored source code in the color red with highlights in yellow to show the difference. The other unchanged source codes are still in black color. The left side panel with the original source codes remains the same.

7. Developer now has the option to save the improved source code to a new file, or to overwrite the old one. S/he simply clicks File, and from the drop-down menu, choose option "save as …" to save the file or file the old one to replace. However, if there are doubts, developer can simply cancel or exit the whole process before the step to overwrite the original file. An example of this refactoring is presented in Chapter 8 under heading "Sample Input and Validation of *Replace Nested Conditional with Guard Clause.*"

CHAPTER 4

DEALING WITH GENERALIZATION

This section includes refactorings that work around a hierarchy of inheritance, moving fields or methods up and down between classes.

Pull Up Method

The definition of Pull Up Method is to move the same function source code from two subclasses to the superclass. Similar to Pull Up Field, the motivation of this refactoring method is to eliminate duplicate source code in all subclasses. Furthermore, the benefit of this refactoring method is that developer can reduce the risk of changing only one method while not changing the others. It is easy to miss the duplicate method at subclass level, especially when the subclasses are developed by different people. The refactoring method is generally used when a method needs to be used by multiple implemented subclasses.

Structure

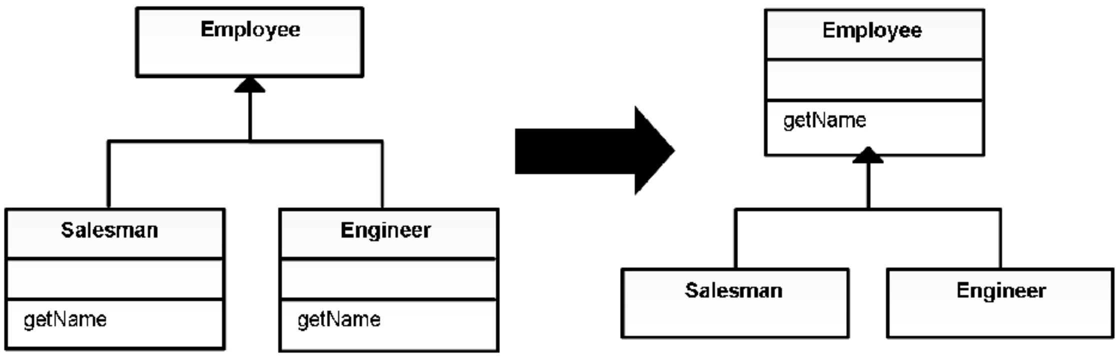


FIGURE 4. Pull Up Method [1].

Refactoring Steps

1. Developer chooses a subclass from the File menu, and highlights the complete method source code s/he wants to pull up to the superclass.

2. User clicks Select button, and QJRT extracts the selected code to the bottom panel for individual analysis.

3. Developer chooses Pull Up Method refactoring from the drop-down menu bar under Refactoring menu.

4. Upon choosing the refactoring method, QJRT displays PUM refactoring on a message label to present and confirm which refactoring is being executed. It also passes the parameter to QJRT to prepare PUM refactoring. And lastly, it sets the Refactoring button visible for user to click on.

5. Developer clicks Refactoring button, and QJRT takes the selected source code from the bottom panel and uses ASTParser to parse it. The first step is to check if the selected methods' class has a superclass, which is done through analysis of the AST. Then, PUM verifies if the selected code is a valid method, which should have basic method signature, includes following key elements: modifiers, return type, method name, parameter list in parenthesis, an exception list, and method body. PUM extracts value of those elements from AST, and temporarily save in memory. Then, QJRT finds the root of the chosen file (based on current file location); following the root path, QJRT discovers all Java files under root, and saves file name and location into an ArrayList. Next, QJRT parses each file by ASTParser to locate current file's superclass, and all related subclasses. The final step is to add the selected method to the superclass, while removing the same methods from all the subclasses.

6. After refactoring is complete, QJRT pops up a question window that gives the developer the option to view the modified files and the original files with the default notepad program.

7. In addition, on the right panel of the QJRT, it reflects the changes that have been made to the current class. The bottom panel shows all the location of the affected super- and subclasses. An example of this refactoring is presented in chapter 8 under heading "Sample Input and Validation of *Pull Up Method*."

Pull Up Field

The definition of PUF is to move a field from any subclasses to the superclass. Mostly, the pull up field refactoring won't be applied when the source code was written by an individual, but as Fowler[1] mentioned, pull up field is used when the subclasses are built separately by different developers, or the subclasses are combined through refactoring. From combining the classes to a package, subclasses mostly do have duplicate features, and having identical fields is just one of the duplicate features.

The benefit of PUF eliminates the duplicate data declaration; moreover, while this action moves the field behavior from subclasses to superclass, similar to pull up method, it reduces the risk of changing only one field while not changing the others.

Refactoring Steps

1. Developer chooses a subclass that contains the field s/he wants to move to the super class from the File menu, and then highlights the individual field.

2. User clicks Select button, and QJRT extracts the selected code to the bottom panel for individual analysis.

Structure

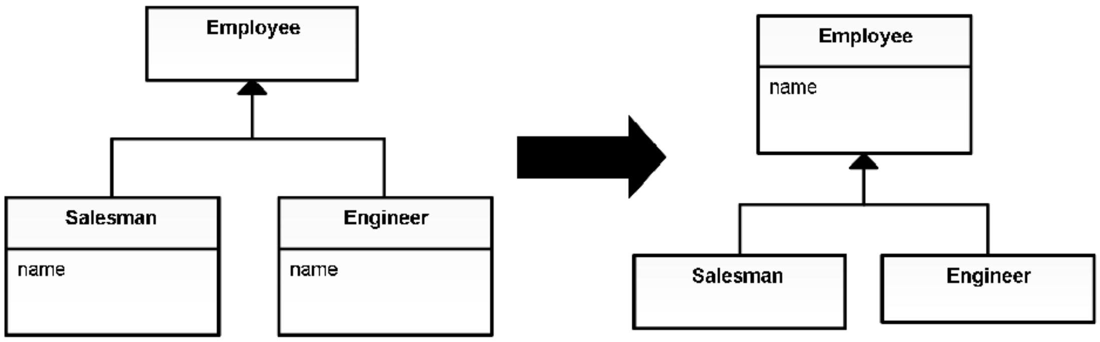


FIGURE 5. Pull Up Field [1].

3. Developer chooses Pull Up Field method from the drop-down menu bar under Refactoring menu.

4. Upon choosing the refactoring method, QJRT displays PUF refactoring on a message label to present and confirm which refactoring is being executed. It also passes the parameter to QJRT to prepare PUF refactoring. And lastly, it sets the Refactoring button visible for user to click on.

5. Developer then clicks Refactoring button, and QJRT takes the selected field from the bottom panel and uses ASTParser to parse it. The first step is to check if the selected field's class has a superclass, which is done through analysis of the AST. Then, PUF verifies if the selected code is a valid field, which should have basic field signature, includes following key elements: modifiers, type, field name, expression. PUF extracts value of those elements from AST, and temporarily save in memory. The following step is to construct a new field with these key elements, one extra step PUF will do is to change "private" to "protected" in modifier: this new field will be saved temporarily and

be extracted to superclass. Then, QJRT finds the root of the chosen file (based on current file location); following the root path, QJRT discovers all Java files under root, and saves file name and location into an ArrayList. Next, QJRT parses each file by ASTParser to locate current file's superclass, and all related subclasses. The final step is to add the new constructed field to the superclass, while removing the same field from all the subclasses.

6. After refactoring is complete, QJRT pops up a question window that gives the developer the option to view the modified and original files with the default notepad program.

7. In addition, on the right panel of the QJRT, it reflects the changes that have been made to the current class. The bottom panel shows the locations of the original and modified files. An example of this refactoring is presented in Chapter 8 under heading "Sample Input and Validation of Pull Up Field."

Push Down Method

PDM is the opposite of PUM. The definition is to move the same function source code from a superclass to a specific subclass. Similar to Push Down Field, the motivation is to eliminate source code in a superclass, since only one subclass implements it; in other words, the method only makes sense in the specific subclass, yet it is declared in the superclass. Furthermore, the Push Down Method makes the subclass highly cohesive, and in general makes the entire software more well designed.

Refactoring Steps

1. Developer chooses a class from the File menu, and highlights the complete method source code s/he wants to push down to the subclass.

Structure

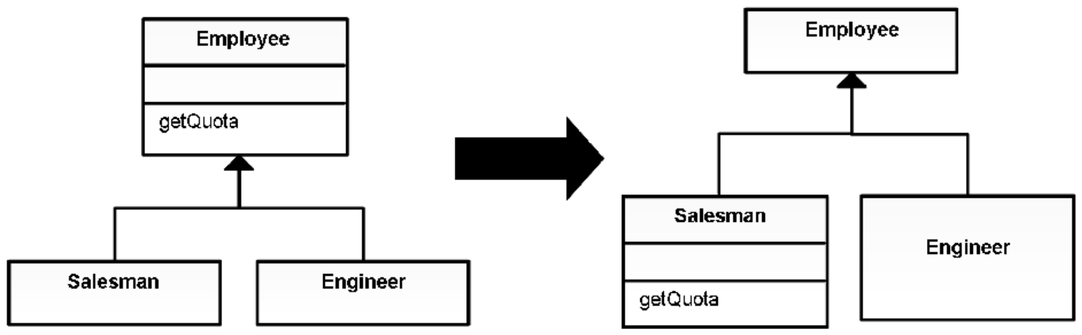


FIGURE 6. Push Down Method [1].

2. User clicks Select button, and QJRT extracts the selected code to the bottom panel for individual analysis.

3. Developer chooses Push Down Method from the drop-down menu bar under Refactoring menu.

4. Upon choosing the refactoring method, QJRT displays PDM refactoring on a message label to present and confirm which refactoring is being executed. It also passes the parameter to QJRT to prepare PDM refactoring. And lastly, it sets the Refactoring button visible for user to click on.

5. Developer clicks Refactoring button, and QJRT takes the selected source code from the bottom panel and uses ASTParser to parse it. The first step is PDM verifies if the selected code is a valid method, which should have basic method signature, includes following key elements: modifiers, return type, method name, parameter list in parenthesis, an exception list, and method body. PDM extracts value of those elements from AST, and temporarily save in memory. Then, QJRT finds the root of the chosen

file (based on current file location); following the root path, QJRT discovers all Java files under root, and saves file name and location into an ArrayList. Next, QJRT parses each file by ASTParser to locate current file's all subclasses. The final step is to add the selected method to the subclasses, while removing the same methods from superclass.

6. After refactoring is complete, QJRT pops up a question window that gives the developer the option to view the modified and the original files with the default notepad program.

7. In addition, on QJRT's right side panel, it presents the changes that have been made to the current class. The bottom panel clears the selected method, and updates the location of affected super- and subclasses. An example of Push Down Refactoring is presented in Chapter 8 under heading "Sample Input and Validation of Push Down Method."

8. PDM requires user intervention: User is required to review and manually delete the unwanted method in subclasses.

Push Down Field

PDF is the opposite of PUF. The definition of PDF is to move a field from a superclass to a subclass. Similar to Push Down Method, the motivation is the field in the superclass does not make sense in some subclasses at all; in other words, the specific field makes sense only in one of the subclasses. Thus, there is no reason to keep the field in the superclass. The benefit is less coding on superclass, and a highly cohesive method of design on subclass.

Structure

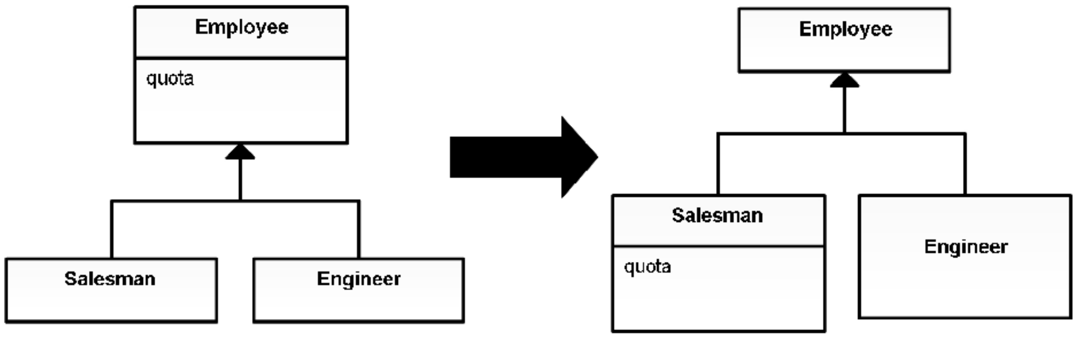


FIGURE 7. Push Down Field [1].

1. User clicks Select button, and QJRT extracts the selected code to the bottom panel for individual analysis.

2. Developer chooses Push Down Field refactoring from the drop-down menu bar under Refactoring menu.

3. Upon choosing the refactoring method, QJRT displays PDF refactoring on a message label to present and confirm which refactoring is being executed. It also passes the parameter to QJRT to prepare PDF refactoring. And lastly, it sets the Refactoring button visible for user to click on.

4. Developer then clicks Refactoring button, and QJRT takes the selected field from the bottom panel and uses ASTParser to parse it. The first step is PDF verifies if the selected code is a valid field, which should have basic field signature, includes following key elements: modifiers, type, field name, expression. PDF extracts value of thoseelements from AST, and temporarily save in memory. The following step is to

construct a new field with these key elements: this new field will be saved temporarily and be extracted to subclasses. Then, QJRT finds the root of the chosen file (based on current file location); following the root path, QJRT discovers all Java files under root, and saves file name and location into an ArrayList. Next, QJRT parses each file by ASTParser to locate current file's all subclasses. The final step is to add the new constructed field to the all subclasses, while removing the same field from the superclass.

5. After refactoring is complete. QJRT pops up a question window giving the developer the option to view the modified and the original files with the default notepad program.

6. In addition, on the right panel of the QJRT, it reflects the changes that have been made to the current selected class. The bottom panel now shows the locations of the original and the modified affected class fields. An example of *Push Down Field* refactoring is presented in Chapter 8 under the heading "Sample Input and Validation of Push Down Field."

7. PDF requires user intervention: user is required to review and manually delete the unwanted field in subclasses.

CHAPTER 5

MISCELLANEOUS

This section is added for items not recorded in the refactoring book by Fowler, but are mentioned on Fowler's website, which currently is still being updated [4]. This section includes the basic Reverse Conditional, and further discussion regarding Reverse Conditional (Reverse Conditional Advance). When a conditional's logic is twisted (e.g., !(a==1)), the purpose of the conditional not only causes confusion to any developers, but also makes debugging difficult in software maintenance.

Reverse Conditional

RC refactoring is used to make source code easier to understand. "Often conditionals can be phrased in way that makes them hard to read. This is particularly the case when they have a not in front of them. If the conditional only has a "then" clause and no "else" clause, this is reasonable. However if the conditional has both clauses, then you might as well reverse the conditional." [5]

Refactoring Steps

1. Developer chooses a file from the File menu, and highlights the complete if-else statement s/he wants to refactor.

2. User clicks Select button, and QJRT extracts the selected code to the bottom panel for individual analysis.

Structure

```
if ( !isSummer( date ) )
     charge = winterCharge( quantity );
else
     charge = summerCharge( quantity );
```

$$\Downarrow$$

```
if ( isSummer( date ) )
     charge = summerCharge( quantity );
else
     charge = winterCharge( quantity );
```

FIGURE 8. Reverse Conditional [1].

3. Developer chooses Reverse Conditional refactoring method from the drop-down menu bar under Refactoring menu.

4. Upon choosing the refactoring method, QJRT displays RC refactoring on a message label to present and confirm which refactoring is being executed. It also passes the parameter to QJRT to prepare RC refactoring. And lastly, it sets the Refactoring button visible for user to click on.

5. Developer clicks Refactoring button and QJRT takes the selected source code from the bottom panel and uses AST to parse it. AST verifies the selected code is valid if-statement, and also verifies if the conditional is negative. RC re-constructs new if-statement by deleting the "NOT" operator, and switch the thenStatement and elseStatement.

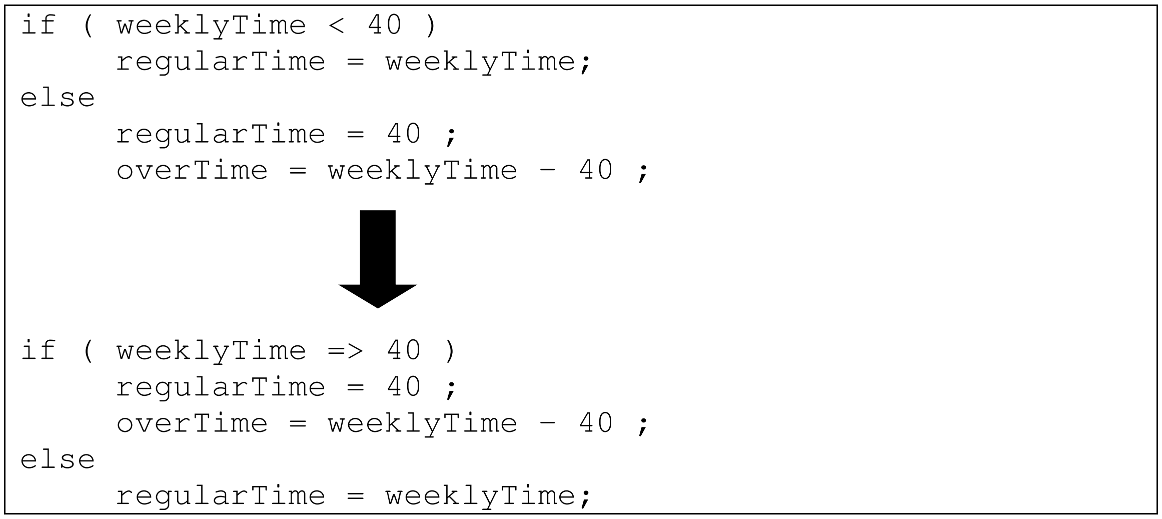
6. Once the refactoring is complete, the user has the option to either save or discard the changes. If discarding the changes, the user simply exits QJRT; if saving the changes, the user clicks the file menu and chooses “save as …” to a new file, or to overwrite the original file.

7. At the same time, on QJRT’s right side panel, it shows the changes in source code in red with yellow highlights. An example of *Reverse Conditional* is presented in Chapter 8 under the heading “Sample Input and Validation of Reverse Conditional.”

Reverse Conditional Advance

RCA refactoring is an expanded version of RC. This advanced version not only deals with the “NOT” operator, but also reverse the general operators such as greater, greater and equal, less, less than, etc. In the end, it supplies more complicated reverse operations on conditionals.

Structure



```
if ( weeklyTime < 40 )
     regularTime = weeklyTime;
else
     regularTime = 40 ;
     overTime = weeklyTime - 40 ;
```

```
if ( weeklyTime => 40 )
     regularTime = 40 ;
     overTime = weeklyTime - 40 ;
else
     regularTime = weeklyTime;
```

FIGURE 9. Reverse Conditional Advance [1].

Refactoring Steps

1. Developer chooses a class from the File menu, and highlights the complete source code s/he wants to refactor.

2. User clicks Select button, and QJRT extracts the selected code to the bottom panel for individual analysis.

3. Developer chooses Reverse Conditional Advance refactoring from the drop-down menu bar under Refactoring menu.

4. Upon choosing the refactoring method, QJRT displays RCA refactoring on a message label to present and confirm which refactoring is being executed. It also passes the parameter to QJRT to prepare RCA refactoring. And lastly, it sets the Refactoring button visible for user to click on.

5. Developer then clicks Refactoring button, and QJRT takes the selected source code from the bottom panel and uses AST to parse it. AST checks if it is an if-statement. Then by determining the expression. RCA re-construct new if-statement by reversing the operation, such as PrefixOperator (ex: ! – RCA will remove !); InfixOperater (ex: < LESS – RCA changes to >= GREATER_EQUALS), and then RCA switches the thenStatement and elseStatement.

6. Once the refactoring is complete, the user has the option to either save or discard the changes. If discarding the changes, user simply exits QJRT; if saving the changes, user clicks the file menu and chooses "save as …" to a new file, or to overwrite the original file.

7. At the same time, on QJRT's right side panel, it shows the changes in source code in red with yellow highlights. An example of Reverse Conditional Advance is

presented in Chapter 8 under the heading “Sample Input and Validation of Reverse Conditional Advance.”

CHAPTER 6

RELATED WORK

The first refactoring work was done by Bill Opdyke. Prior to Opdyke, his advisor, Ralph Johnson was the first person to define refactorings [6]. Many Integrated Development Environments (IDE) consist refactoring functions used by software engineers. Moreover, lots of plugins work with various IDEs, and vice versa. Developers benefit from Fowler's work on refactorings; Consell and Swift studied and analyzed 14 refactorings mentioned by Fowler [7].

Eclipse Software Development Kit (SDK) is a platform to build and assemble programs [8], and series refactorings are included in its platform IDE [8, 9]. However, "Eclipse's refactoring tools aren't intended to be used for refactoring at a heroic scale [10]." For example, after renaming a package, Eclipse debugger doesn't report any errors, or the source code doesn't show any highlight of errors, but the program won't build any more. To fix the problem, I had to create a new project, and copy the source code over with intended package's name, and then everything works. This is just an example of some plugins wouldn't work appropriately with certain IDEs.

The basic idea of how to refactor consists two parts: the first part is for parser to parse the source code, and the second part is to re-construct the destination source code without changing its external behavior. Many parsers can be used for this purpose. I have tested ANTLR [11] and Gold Parser [12] to a different degree. ANTLR implements LL parsing algorithm (top-down). Gold parser implements LALR parsing algorithm (look-ahead). When working with ANTLR, I had to manually write interface from scratch on each abstract syntax tree (AST) node, and provide complicated methods to retrieve and store all the information from AST. It is same as to write an entire ASTParser system (similar to Eclipse ASTParser), which the workload is not only time consuming but also out of scope of this thesis (which is to focus on refactoring tool, not on lexer and parser). Eclipse AST provides complete interface to interact with ASTNode. The AST is the base framework, and maps Java source code in a tree form [8]. QJRT utilizes Eclipse AST for parsing function.

JavaRefactor is another refactoring product, it serves as a plugin for jEdit. It implemented few refactorings such as renaming class, field, method, and package, also includes Push Down and Pull Up of methods and fields [13]. JavaRefactor does refactoring step by step, but it doesn't synchronize code. It only can be viewed as a plugin, and it cannot run without jEdit; while QJRT is a stand-alone tool, doesn't rely on any host software. QJRT also implemented Push Down and Pull Up of methods and fields.

JRefactory, as a plugin for Java refactoring, is used in various IDEs and programs. They are jEdit, JBuilder, Ant, and NetBeans [14]. It applies refactoring by using a generated UML-diagram interface. The core part of JRefactory is Java Parser, that intake the tokenizer for analysis, and then the source code can be viewed as an AST. It could be a stand-alone application for loading view webstart purpose due to the fact it lacks editing function.

RefactoringNG is used for NetBeans as a plugin. The developers can define the refactoring rules by tweaking RefactoringNG [15]. The basics refactoring is achieved by obtain and modify the code's AST. One of benefit is that it can create batches for automatic code upgrade when updating the library. This tool would be useful for advanced developer on refactoring, as well as for system manager perform maintenance on large numbers of software systems at once.

There are other plugins, such as once famous Refactor-J [16] for IntelliJ IDEA [17], and jFactor [18] etc. Many refactoring plugins no longer exist or are not up to date. Many IDEs already consists powerful refactoring tools without external plugins. NetBeans's refactoring tools are available to use right after installation. There aren't any other plugins to configure. Basically, modern IDEs are bundled with refactoring features.

QJRT, developed in this thesis, is a quick refactoring tool. The basics concept of QJRT is using Eclipse AST as a help class to retrieve source code's information. QJRT's refactorings are implemented with minimal user interaction (ex. current file's inheritance hierarchy is automatically detected).

CHAPTER 7

QUICK JAVA REFACTORING TOOL

Quick Java Refactoring Tool (QJRT) is a portable program performed on individual refactoring. Each refactoring section is divided into two parts. First is user interactions, and the other is refactoring.

The user interaction section includes opening the file, selecting the code, and choosing the refactoring method. The refactoring section includes two modules, one is parsing and data collection, and the other is re-constructing and generating new code. The first module is parsing the selected code, collecting related data and saving the information to memory; this module also handles all the errors. Second module is to reconstruct the source code. This section runs automatically at the background once it's been kicked off, and it will save new code to memory or a new file.

User Interaction

QJRT is designed with minimal user input to refactor Java code. The GUI (graphical user interface) includes a menu bar, a tool bar, and three blank text areas: two text areas are on the top side by side and another area on the bottom alone. The tool bar resides between the top and bottom text areas. A message label is also included within the tool bar for showing which refactoring is in selected, and it is only visible while the refactoring is in process.

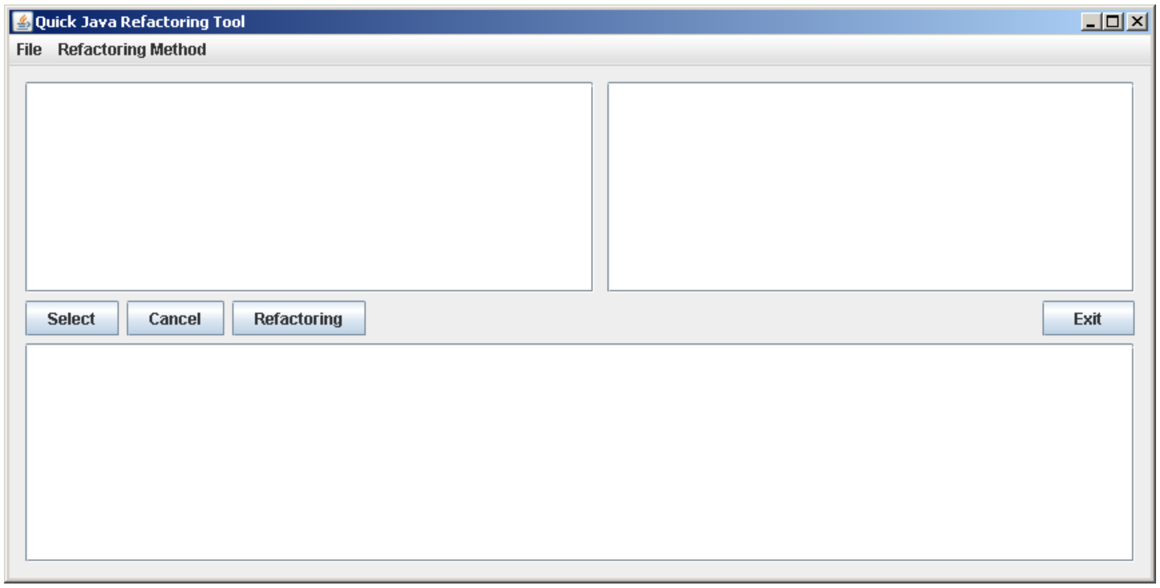


FIGURE 10. Initial screen of QJRT.

Initial screen of QJRT contains of a menu bar and a tool bar. The menu bar contains two menu items, one is *File* and another is *Refactoring Method.* Tool bar has *select*, *cancel*, *refactoring*, and *exit* buttons. The *Refactoring* button is disabled initially, and it will be available upon a refactoring method is chosen by user.

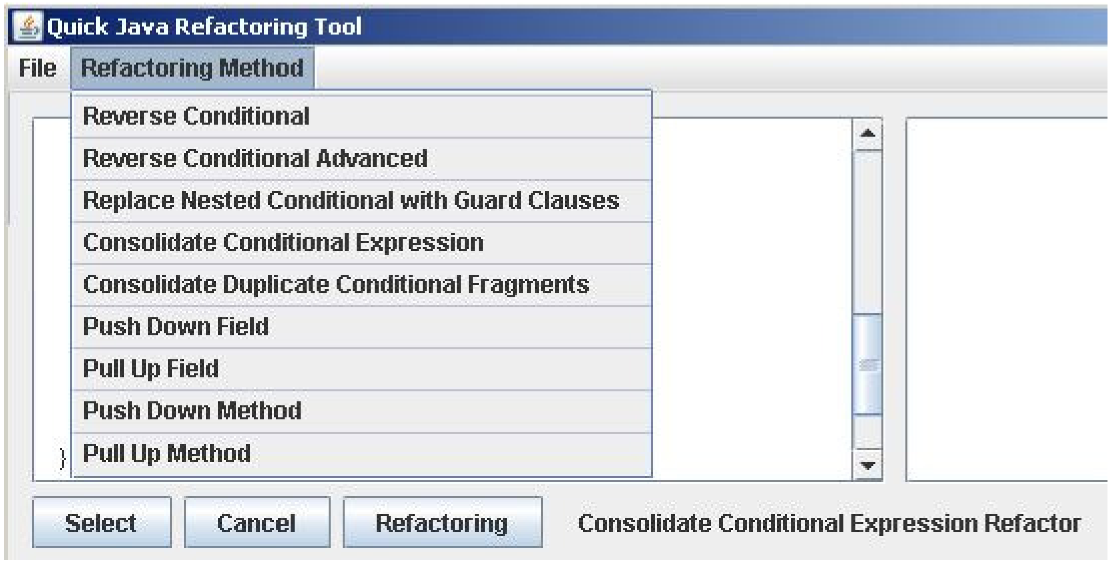
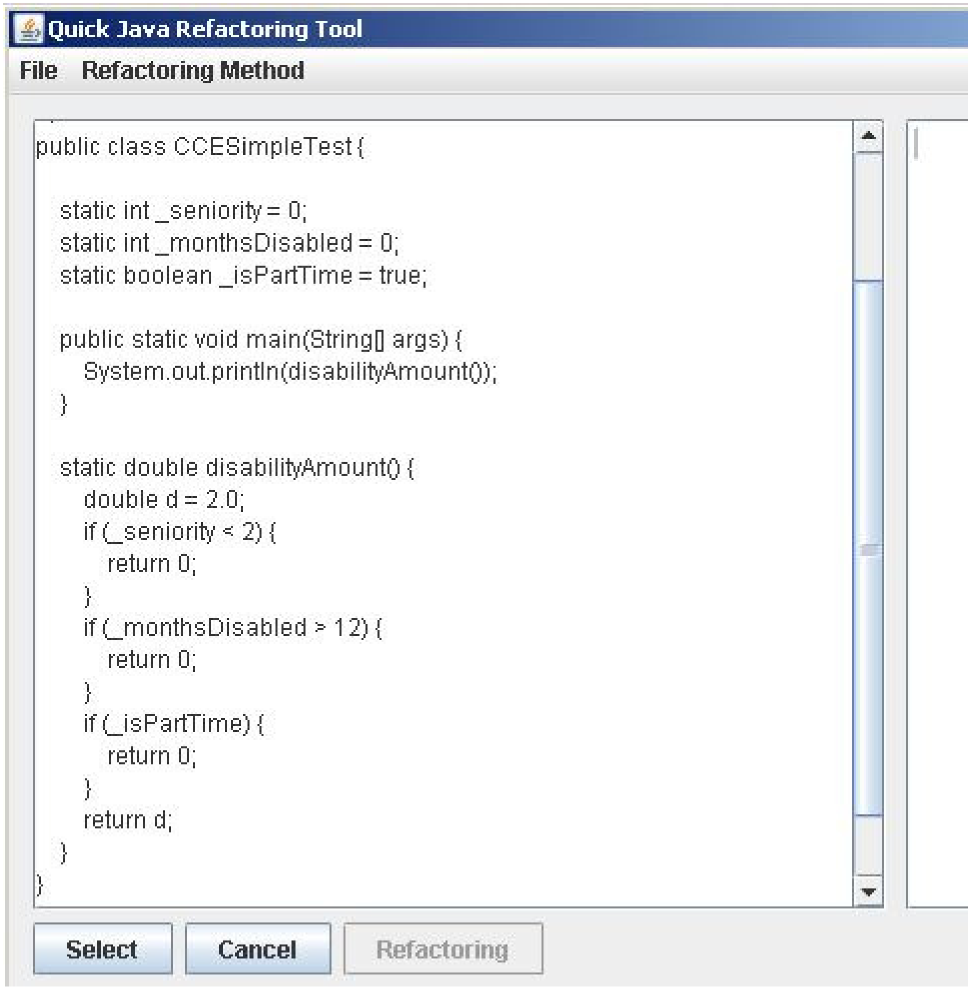

Quick Java Refactoring Tool
File
Refactoring Method
Reverse Conditional
Reverse Conditional Advanced
Replace Nested Conditional with Guard Clauses
Consolidate Conditional Expression
Consolidate Duplicate Conditional Fragments
Push Down Field
Pull Up Field
Push Down Method
Pull Up Method
Select
Cancel
Refactoring
Consolidate Conditional Expression Refactor

FIGURE 11. Refactoring Method menu options in QJRT.

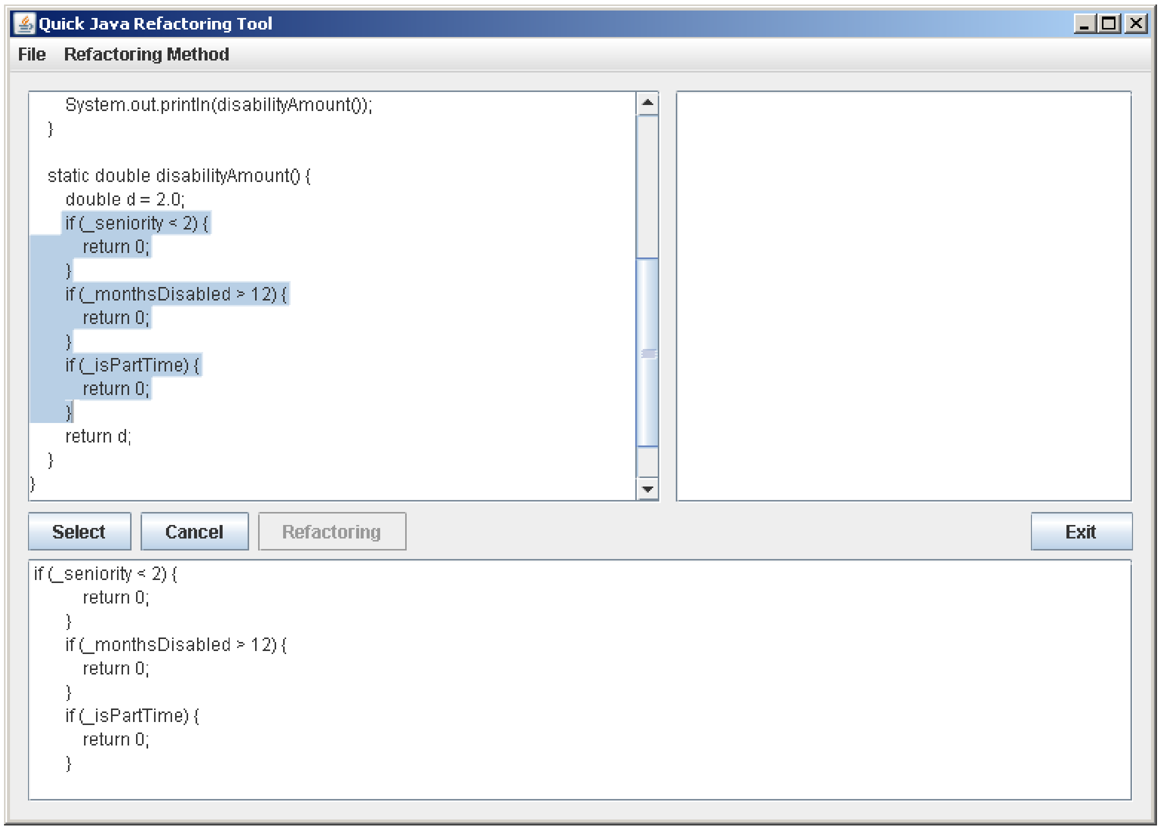
The Figure 11 shows the refactoring methods implemented in QJRT. Upon user selecting a refactoring method from this menu, it passes the parameters to QJRT to prepare the current refactoring.



```
public class CCESimpleTest {

  static int _seniority = 0;
  static int _monthsDisabled = 0;
  static boolean _isPartTime = true;

  public static void main(String[] args) {
    System.out.println(disabilityAmount());
  }

  static double disabilityAmount() {
    double d = 2.0;
    if (_seniority < 2) {
      return 0;
    }
    if (_monthsDisabled > 12) {
      return 0;
    }
    if (_isPartTime) {
      return 0;
    }
    return d;
  }
}
```

FIGURE 12. Initial text area for original code viewer in QJRT.

QJRT uses window's open file dialogue box to let developer choose Java file to open. The upper left text area is used for displaying Java code contained in the file (shown in figure 12). The code is taken in by QJRT as string in local memory. Developer then highlights the targeted code. Upon clicking *Select* button, the highlighted code is copied to the bottom text area (shown in figure 13). After choosing the correct refactoring method, the *refactoring* button is enabled (shown in figure 11) and user then clicks this button to initialize refactoring process. At this point, the user interaction is finished.



```
    System.out.println(disabilityAmount());
  }

  static double disabilityAmount() {
    double d = 2.0;
    if (_seniority < 2) {
      return 0;
    }
    if (_monthsDisabled > 12) {
      return 0;
    }
    if (_isPartTime) {
      return 0;
    }
    return d;
  }
}
```

```
if (_seniority < 2) {
      return 0;
    }
    if (_monthsDisabled > 12) {
      return 0;
    }
    if (_isPartTime) {
      return 0;
    }
```

FIGURE 13. Highlighting and selecting targeted code to bottom text area.

Automatic Refactoring - Data Collection

The first module is for parsing the selected code. QJRT uses eclipse's AST (Abstract Syntax Tree) [5] as a library plugin, and uses AST to parse the code. The ASTParser is used for every refactoring. In detail, if the selected source code is a method or a field, K_COMPILATION_UNIT is used to parse; if the selected source code is if-statement/s, K_STATEMENTS is used to parse. Furthermore, to validate a selected method, K_CLASS_BODY_DECLARATIONS is used. While parsing the source code, QJRT also collects the necessary information on the parsed codes.

Each refactoring uses and requires different algorithm. Each refactoring is an individual file (class) in order to parse the code, collect data based on the code, and store all necessary information to local memory. All of these steps start with ASTParser to obtain the information then pass to QJRT to store it to local variables. Similar classes do have same local variables named with same intension. For example, all fields defined in table 1 is used by both Pull Up and Push Down Field.

TABLE 1. Table for Pull Up Field and Push Down Field Refactoring

Field Name	Description
file	Selected source file for refactoring
str	Selected field's name
line	Selected field's line number within source file
rootPath	Source file's location
superClassName	Super class's name (string)
childclass	childclass list to store all sub class
supclass	Initial null string for super class //used for in pull up field
CompilationUnit	Eclipse's AST on selected file (type of the root of an AST)
TypeDeclariation	Verify super class's existence
FieldDecalration	Validate the selected field
PackageDeclaration	All associate java files under package
m_modifiers	Field's modifiers
m_type	Field's type
m_name	Field's name
m_exp	Field's expression

Table 1 lists the fields used in Pull Up and Push Down field refactoring. Both refactoring methods are separated files due to the fact of loose coupling, the major data gathering is similar, except one field (supclass) is designated to *pull up field* refactoring only. All information such as file name, field name, starting line number, root path, etc. are store individually in memory. File, str, line are from user's input, and all other information are obtained from AST. In addition, the field's signature information (m_modifier, m_type, m_name, and m_exp), supclass and childclass are used when reconstruct the code.

TABLE 2. Table for Pull Up and Push Down Method Refactoring

Field Name	Description
file	Selected source file for refactoring
str	Selected method's name
line	Selected method's line number within source file
rootPath	Source file's location
superClassName	Super class's name
CompilationUnit	Eclipse's AST on selected file
TypeDeclariation	Verify super class's existence
MethodDecalration	Validate the selected method
PackageDeclaration	All associate files under package
childclass	childclass list to store sub class
supclass	Initial null string for super class // for only pull up method
m_name	Method's name
m_retype	Method's return type
m_block	Method's body
m_modifiers	Method's modifiers
m_params	Method's paramethers
m_throw	Method's exceptions

Table 2 lists the fields used in Pull Up and Push Down Method refactoring. Similar to Table 1, both refactoring methods are separated files due to the fact of loose coupling, the major data gathering is similar, except one field (supclass) is designated to pull up method refactoring only. All information such as file name, field name, starting line number, root path, etc. are store individually in local memory. File, str, line are from user's input, and all other information are obtained from AST. In addition, the method's signature information (m_name, m_retype, m_block, m_modifier, m_params, and m_throw), supclass and childclass are used when reconstruct the code.

TABLE 3. Table for Reverse Conditional and Reverse Conditional Advance Refactoring

Field Name	Description
str	Selected block of code as string
stmt	Validate ifStatement
ifstmt	ifStatement
ifexp	ifStatement's expression
st1	thenStatement
st2	elseStatement
PrefixExpression	Verify ++,--,+,-,~ operators before expression
PrefixExpression	Verify! operators before expression // only in reverse cond
PostfixExpression	Verify ++,--,+,-,~,! operators after expression
ParenthesizedExpression	Verify parenthesis operator
InstanceofExpression	Verify instanceof type
InfixExpression	Verify all operator (+,-,*,/,%,<<,>>,>>>,<,>, <=,>=,==,!=,&,|,&&,||)

Table 3 lists the fields used in Reverse Conditional and Reverse Conditional Advance refactoring. Both refactoring methods are separated files due to the fact that Reverse Conditional is introduced solely by Fowler and Murphy [1, 5], and they discussed further on advanced version on this topic. The major data gathering is similar, except when analysis the expression, on *Reverse Conditional Advance,* more conditionals are considered, they are: PostfixExpression, ParenthesizedExpression, InfixExpression, and InstanceofExpression. One common field is used on both refactoring, it is PrefixExpression. The usage for Reverse Conditional Advance, it verifies major operators in front of expression, while for Reverse Conditional, it only verifies the NOT (!) operator in front of expression. Str is from user's input (selected ifStatement), and all other information are obtained from AST. In addition, the ifStatement's signature information (st1 - thenStatement, st2 - elseStatement, ifexp) are used when reconstruct the code.

TABLE 4. Table for Consolidate Duplicate Conditional Fragment Refactoring

Field Name	Description
str	Selected block of code as string
stmt	Validate ifStatement
ifstmt	ifStatement
ifexp	ifStatement's expression
s01	thenStatement
s02	elseStatement
b1	Block to store statement
b2	Block to store statement
ls1	List to store block 1[b1]statement
ls2	List to store block 2[b2]statement
frontlist	statements move to front
endlist	statements move to end

Table 4 lists the fields used in Consolidate Duplicate Conditional Fragment refactoring. All information is store individually in local memory. Str is from user's input (selected ifStatements), and all other information are obtained from AST. In addition, the ifStatement's signature information (ifexp, ls1, ls2), frontlist and endlist are used when reconstruct the code.

TABLE 5. Table for Consolidate Conditional Expression Refactoring

Field Name	Description
str	Selected block of code as string
sts	Validate ifStatement
ifstmt	ifStatement
exp_list	List to store expressions
s01	First thenStatement
s02	First elseStatement
s1	Second thenStatement
s2	Second elseStatement

Table 5 lists the fields used in Consolidate Conditional Expression refactoring. All information is store individually in local memory. Str is from user's input, and all other information are obtained from AST. In addition, the ifStatement's signature information (s01, s02, exp_list, s1, s2) is used when reconstruct the code.

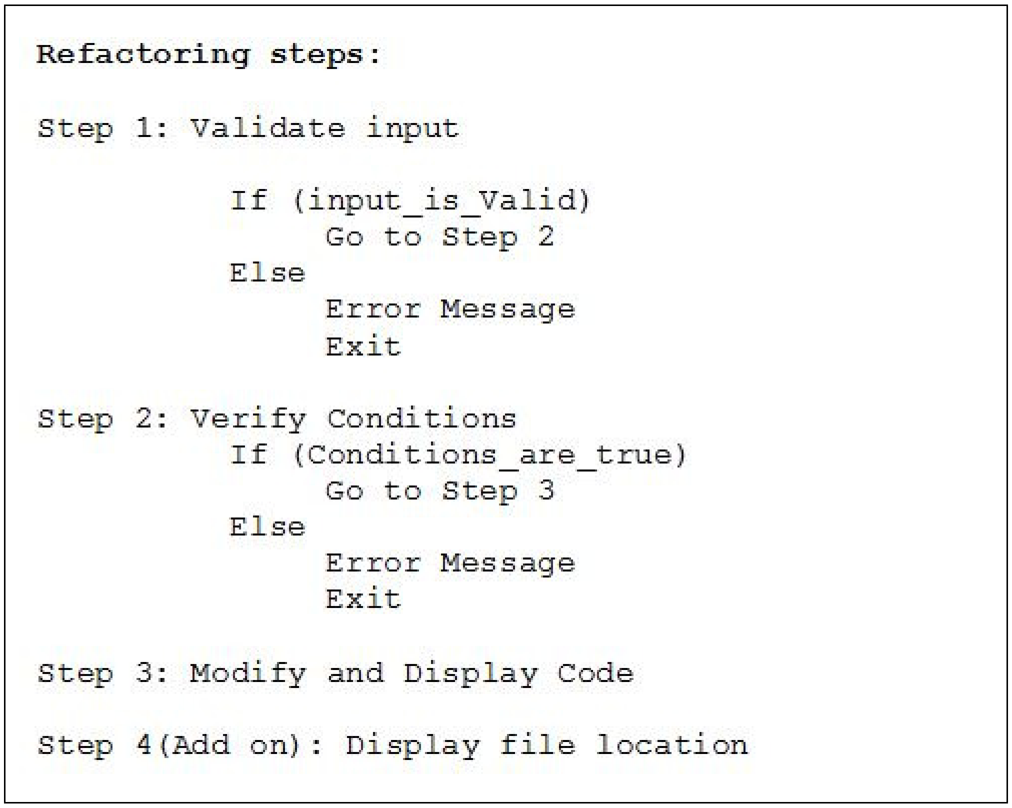
TABLE 6. Table for Replace Nested Conditional with Guard Clause Refactoring

Field Name	Description
str	Selected block of code as string
stmt	Validate ifStatement
ifstmt	ifStatement
ifexp	Initial ifstatement's expression
new_exp	New ifstatement's expression
new_if	New if statement
new_rt	New ReturnStatement
s1	thenStatement
s2	elseStatement

Table 6 lists the fields used in Replace Nested Conditional with Guard Clause refactoring. All information is store individually in local memory. Str is from user's input, and all other information are obtained from AST. In addition, the ifStatement's signature information (s1, s2, ifexp), new_exp, new_if, new_rt are used when reconstruct the code.

Automatic Refactoring - Overview

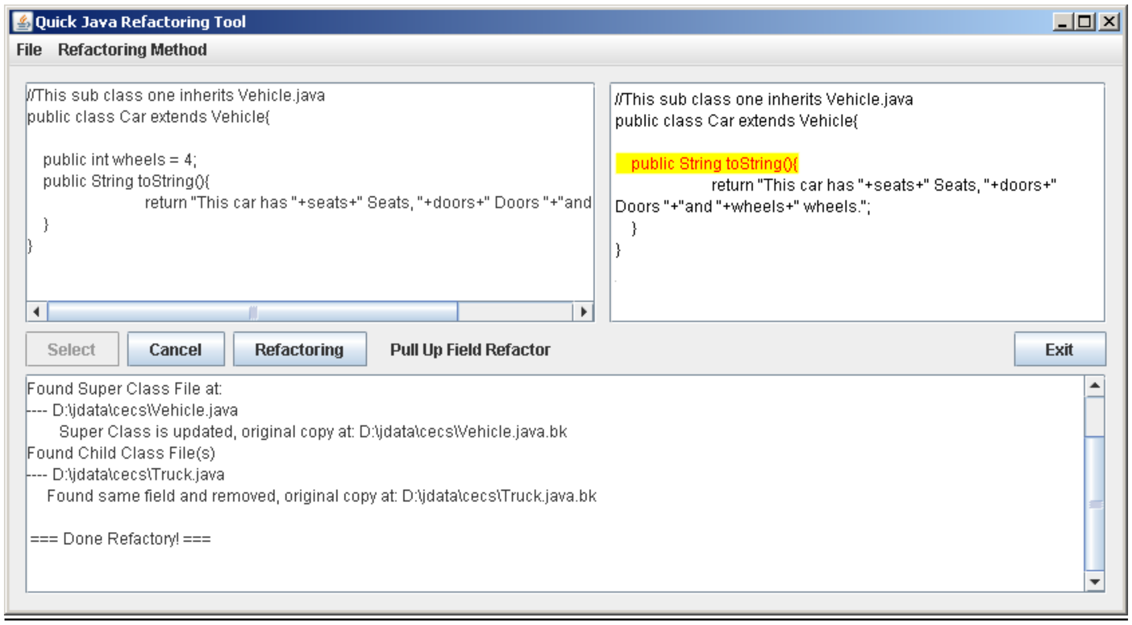
The first step of refactoring is to validate the input from user. Second step is to check the conditions for each refactoring, and the third step is to modify and display the refactored code, finally, if needed, QJRT displays the affected file locations. The stepwise procedure is shown in Figure 14.



```
Refactoring steps:

Step 1: Validate input

          If (input_is_Valid)
               Go to Step 2
          Else
               Error Message
               Exit

Step 2: Verify Conditions
          If (Conditions_are_true)
               Go to Step 3
          Else
               Error Message
               Exit

Step 3: Modify and Display Code

Step 4(Add on): Display file location
```

FIGURE 14. Refactoring steps in QJRT.

In Step 2, the conditions are the requirement based on each individual refactoring. If only the requirements are met, the step 3 is reached. Otherwise, an error message is displayed and the program ends. Step 4 is an additional step for multiple files are affected during refactoring. For example, the pull up field refactoring requires at least three different files including the one user selected (one sub class), a super class (obtained by program in background), and at least one other sub class (obtained by program in background). In step 4, QJRT will display the file location in bottom text area (shown in Figure 15).



```
//This sub class one inherits Vehicle.java
public class Car extends Vehicle{

   public int wheels = 4;
   public String toString(){
                    return "This car has "+seats+" Seats, "+doors+" Doors "+"and
   }
}
```

```
//This sub class one inherits Vehicle.java
public class Car extends Vehicle{

   public String toString(){
                    return "This car has "+seats+" Seats, "+doors+"
Doors "+"and "+wheels+" wheels.";
   }
}
```

```
Found Super Class File at:
---- D:\jdata\cecs\Vehicle.java
     Super Class is updated, original copy at: D:\jdata\cecs\Vehicle.java.bk
Found Child Class File(s)
---- D:\jdata\cecs\Truck.java
   Found same field and removed, original copy at: D:\jdata\cecs\Truck.java.bk

=== Done Refactory! ===
```

FIGURE 15 . Text area to display file location.

CHAPTER 8

VALIDATION OF TOOL: QJRT

This chapter describes how QJRT was validated. This thesis includes three different areas (Simplifying Conditional Expressions; Dealing with Generalization; Miscellaneous) of refactoring described by Fowler[1]. Each area has at least one set of testing code to demonstrate. Some would have multiple sample codes for extensive testing.

The whole validation process is presented with the help of screenshots of QJRT; some steps only include partial screenshots, while some steps include UML diagrams to help clarify the source code's structure. In the end, the screenshots show the changes that have been done throughout QJRT, as well as screenshot shows the output (external behavior) of the program are same before and after performing refactoring.

Sample Input and Validation of Consolidate Conditional Expression

To test the tricky conditional logic of Simplifying Conditional Expressions, first in Consolidate Conditional Expression refactoring, a sample code is used from a single Java class named "CCESimpleTest." This class has a single main method with a print line to print a double primitive type's value. The double's value is determined by a few if-statements under different conditionals to return to.

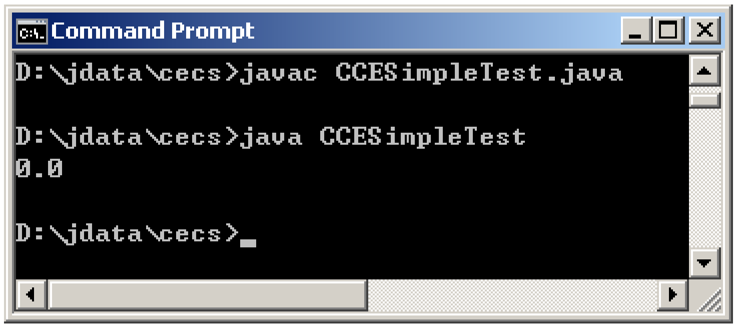
```
CCESimpleTest.java

public class CCESimpleTest {

    static int _seniority = 0;
    static int _monthsDisabled = 0;
    static boolean _isPartTime = true;

    public static void main(String[] args) {
        System.out.println(disabilityAmount());
    }

    static double disabilityAmount() {
        double d = 2.0;
        if (_seniority < 2) {
            return 0;
        }
        if (_monthsDisabled > 12) {
            return 0;
        }
        if (_isPartTime) {
            return 0;
        }
        return d;
    }
}
```

FIGURE 16. The sample code for CCE.

Sample Input

CCESimpleTest.java is selected to open in the left panel of the tool. The if-statement inside the primitive type double will be consolidated since the return value is the same for each if-condition. The new and improved if-condition will replace the old if-condition upon applying refactoring. The output stays the same before and after applying refactoring.

Output Before Applying Refactoring



```
D:\jdata\cecs>javac CCESimpleTest.java

D:\jdata\cecs>java CCESimpleTest
0.0

D:\jdata\cecs>_
```

FIGURE 17. Output before applying CCE refactoring

Refactoring Process

Since this is the first sample of the validation, the process will be as detailed as possible to present the procedure of refactoring. It will go step by step, and each step will be accompanied by a single screenshot.

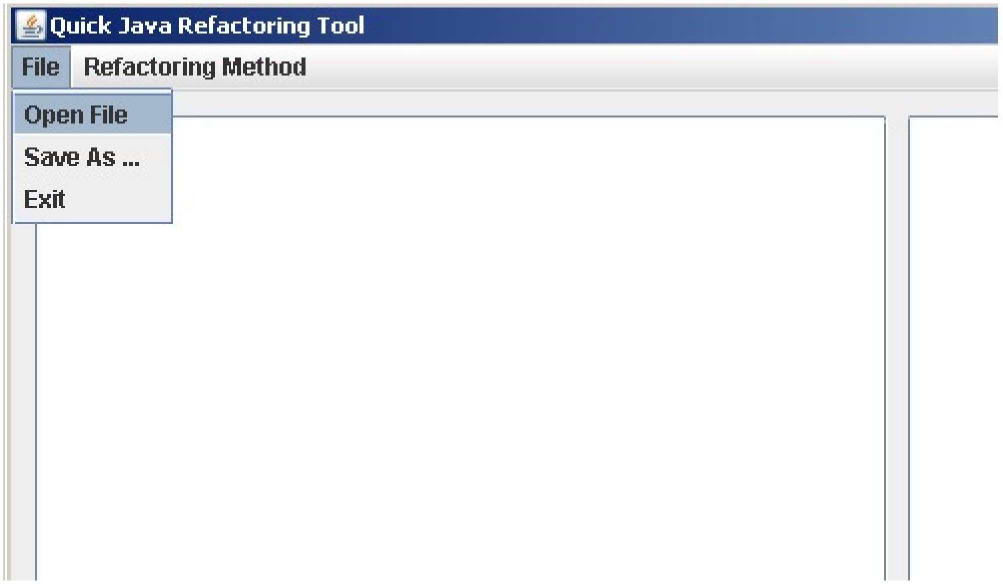
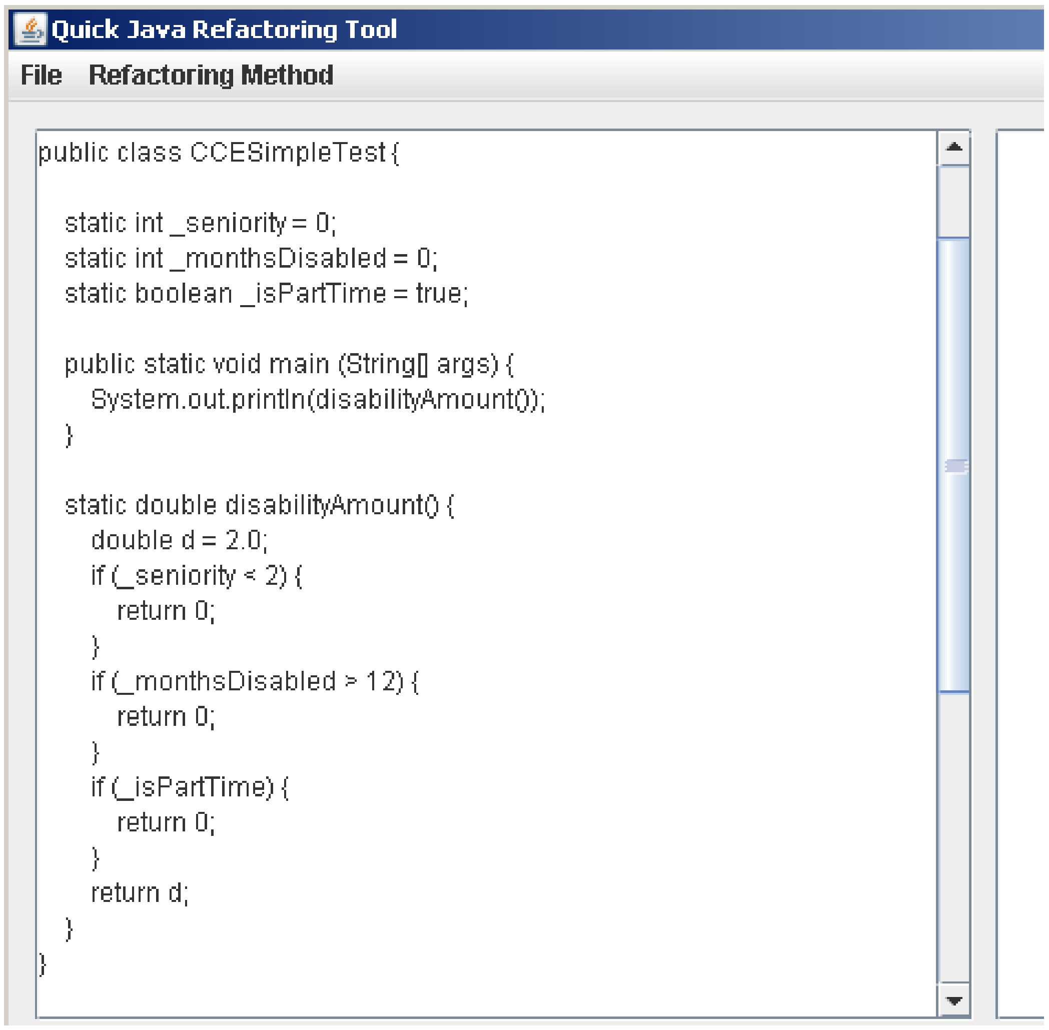
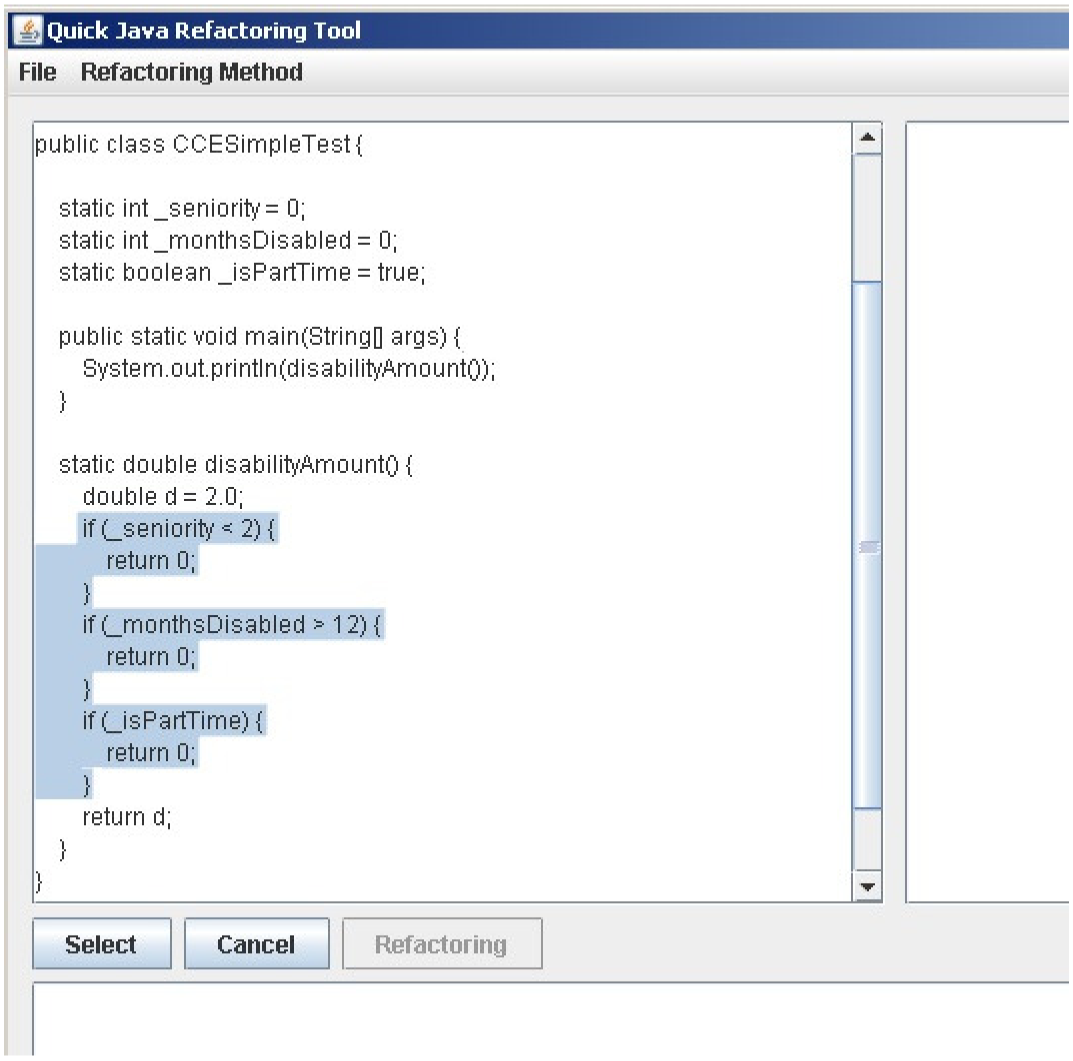


FIGURE 18. Opening a file.



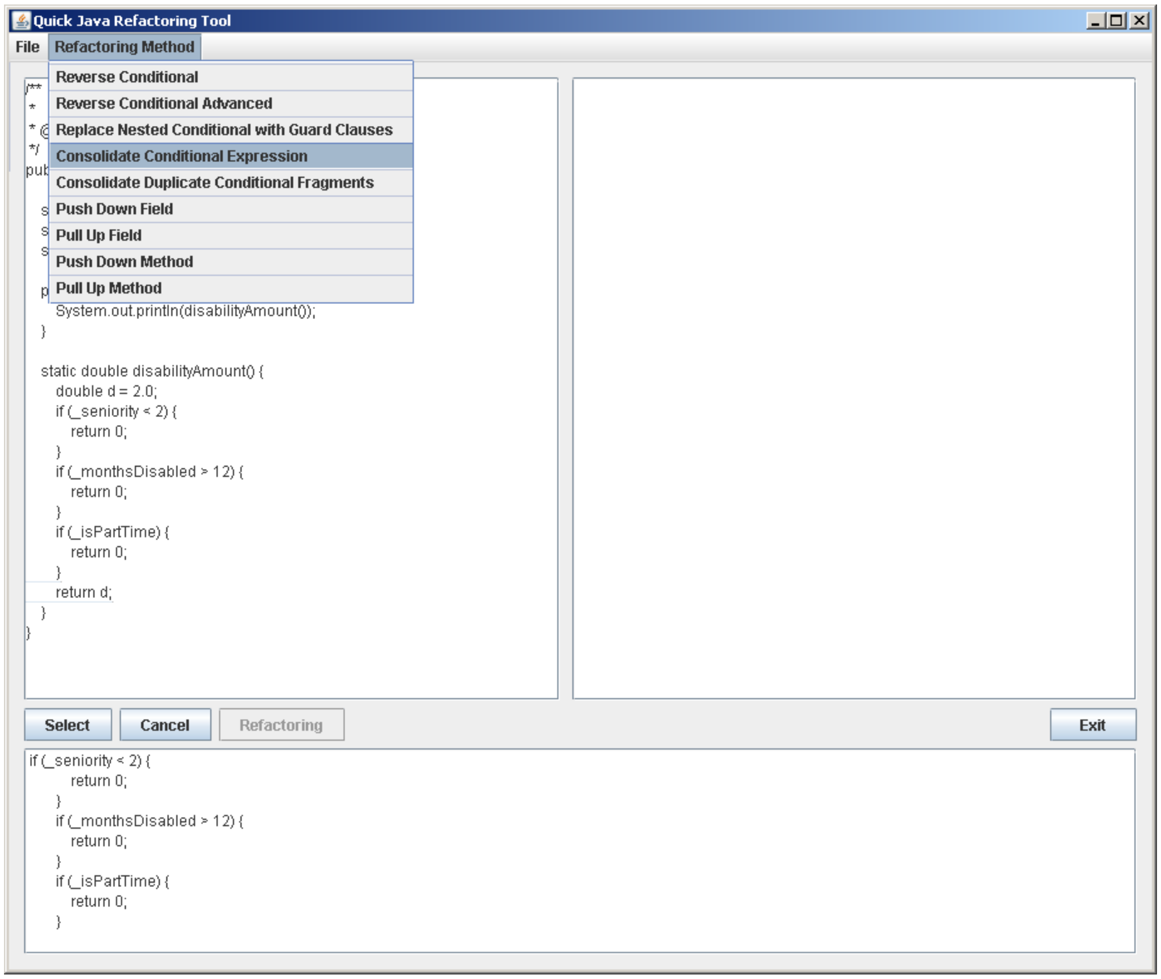
```
public class CCESimpleTest {

  static int _seniority = 0;
  static int _monthsDisabled = 0;
  static boolean _isPartTime = true;

  public static void main (String[] args) {
    System.out.println(disabilityAmount());
  }

  static double disabilityAmount() {
    double d = 2.0;
    if (_seniority < 2) {
      return 0;
    }
    if (_monthsDisabled > 12) {
      return 0;
    }
    if (_isPartTime) {
      return 0;
    }
    return d;
  }
}
```

FIGURE 19. Left panel of the source code.



```
public class CCESimpleTest {

  static int _seniority = 0;
  static int _monthsDisabled = 0;
  static boolean _isPartTime = true;

  public static void main(String[] args) {
    System.out.println(disabilityAmount());
  }

  static double disabilityAmount() {
    double d = 2.0;
    if (_seniority < 2) {
      return 0;
    }
    if (_monthsDisabled > 12) {
      return 0;
    }
    if (_isPartTime) {
      return 0;
    }
    return d;
  }
}
```

FIGURE 20. Selecting the portion of source code.

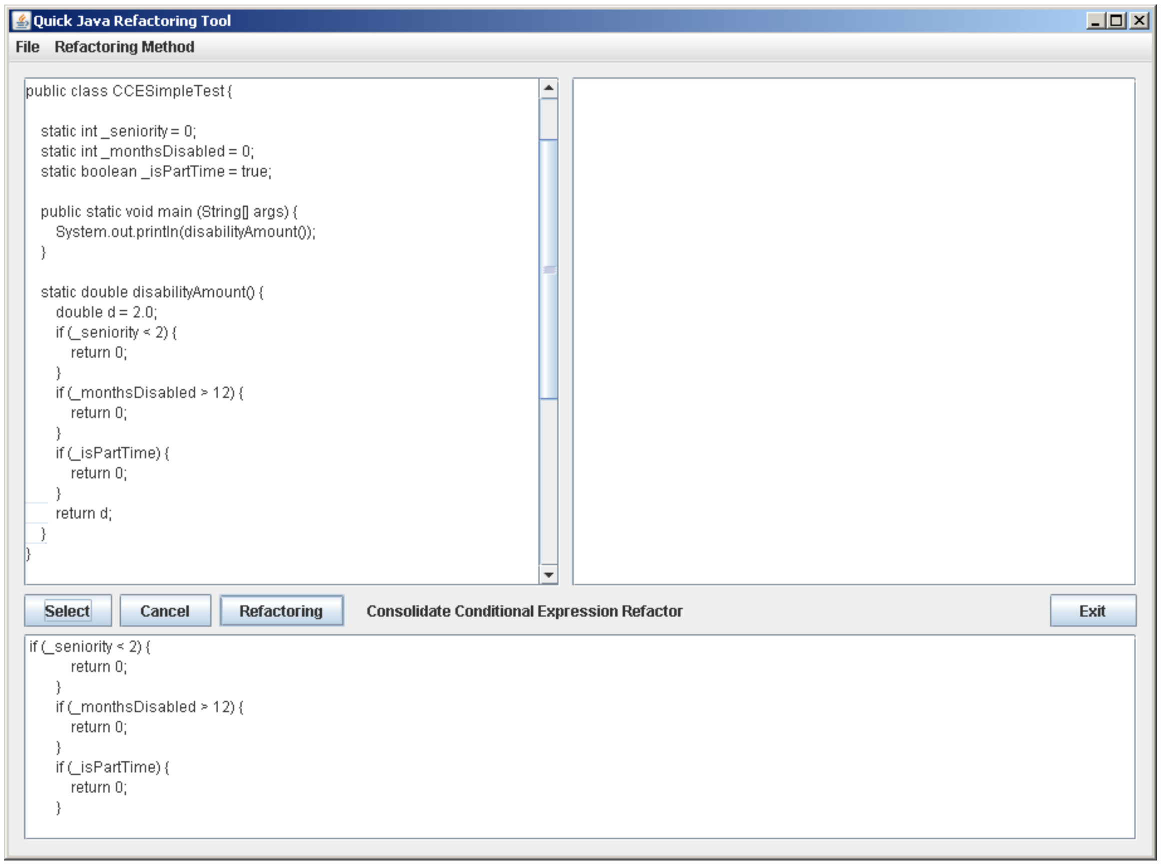


Quick Java Refactoring Tool
File
Refactoring Method
Reverse Conditional
Reverse Conditional Advanced
Replace Nested Conditional with Guard Clauses
Consolidate Conditional Expression
Consolidate Duplicate Conditional Fragments
Push Down Field
Pull Up Field
Push Down Method
Pull Up Method
Select
Cancel
Refactoring
Exit

```
      System.out.println(disabilityAmount());
   }

   static double disabilityAmount() {
      double d = 2.0;
      if (_seniority < 2) {
         return 0;
      }
      if (_monthsDisabled > 12) {
         return 0;
      }
      if (_isPartTime) {
         return 0;
      }
      return d;
   }
}
```

```
if (_seniority < 2) {
         return 0;
      }
      if (_monthsDisabled > 12) {
         return 0;
      }
      if (_isPartTime) {
         return 0;
      }
```

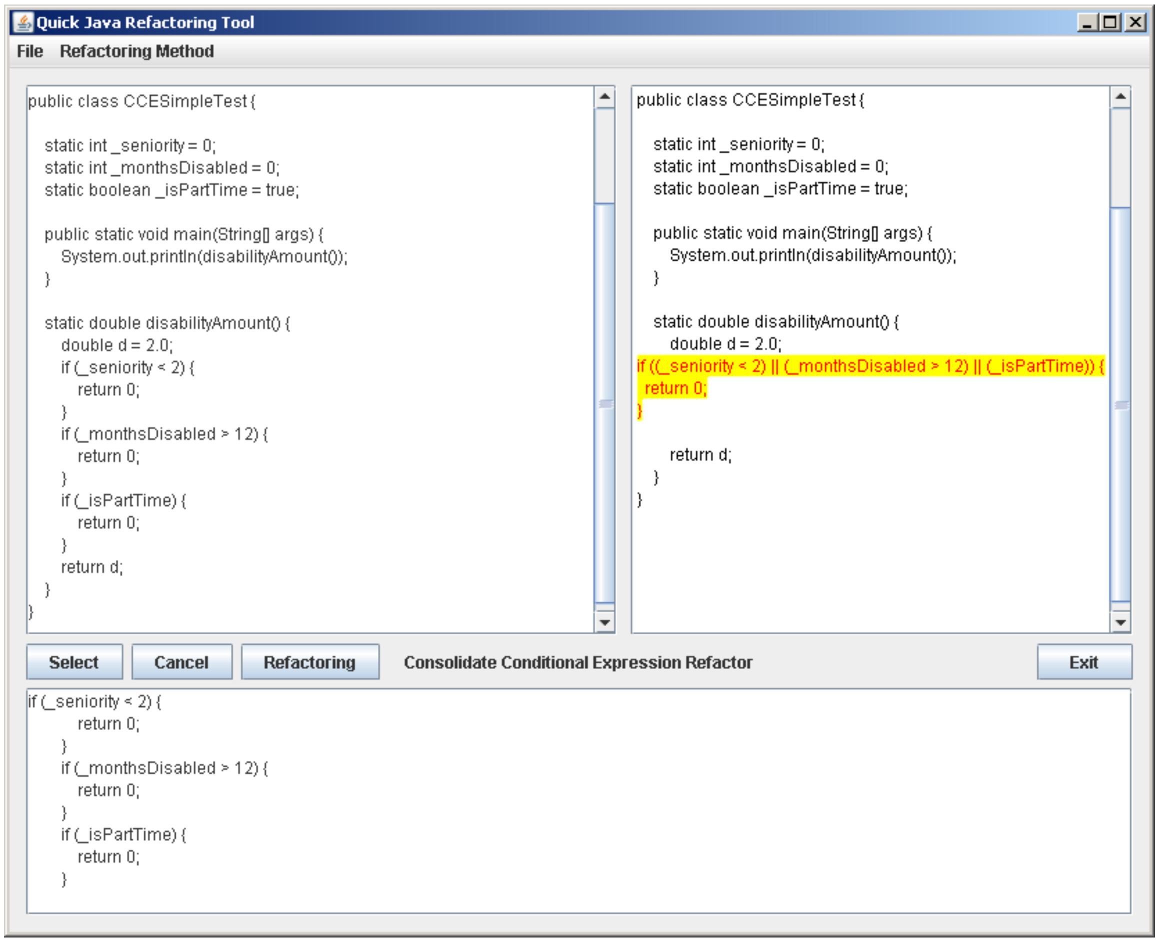
FIGURE 21. Selecting the correspondence refactoring method.


Quick Java Refactoring Tool
File
Refactoring Method
Select
Cancel
Refactoring
Consolidate Conditional Expression Refactor
Exit

```
public class CCESimpleTest {

   static int _seniority = 0;
   static int _monthsDisabled = 0;
   static boolean _isPartTime = true;

   public static void main (String[] args) {
      System.out.println(disabilityAmount());
   }

   static double disabilityAmount() {
      double d = 2.0;
      if (_seniority < 2) {
         return 0;
      }
      if (_monthsDisabled > 12) {
         return 0;
      }
      if (_isPartTime) {
         return 0;
      }
      return d;
   }
}
```

```
if (_seniority < 2) {
         return 0;
      }
      if (_monthsDisabled > 12) {
         return 0;
      }
      if (_isPartTime) {
         return 0;
      }
```

FIGURE 22. After selected refactoring method.


Quick Java Refactoring Tool
File
Refactoring Method
Select
Cancel
Refactoring
Consolidate Conditional Expression Refactor
Exit

```
public class CCESimpleTest {

  static int _seniority = 0;
  static int _monthsDisabled = 0;
  static boolean _isPartTime = true;

  public static void main(String[] args) {
    System.out.println(disabilityAmount());
  }

  static double disabilityAmount() {
    double d = 2.0;
    if (_seniority < 2) {
      return 0;
    }
    if (_monthsDisabled > 12) {
      return 0;
    }
    if (_isPartTime) {
      return 0;
    }
    return d;
  }
}
```

```
public class CCESimpleTest {

  static int _seniority = 0;
  static int _monthsDisabled = 0;
  static boolean _isPartTime = true;

  public static void main(String[] args) {
    System.out.println(disabilityAmount());
  }

  static double disabilityAmount() {
    double d = 2.0;
if ((_seniority < 2) || (_monthsDisabled > 12) || (_isPartTime)) {
 return 0;
}

    return d;
  }
}
```

```
if (_seniority < 2) {
      return 0;
    }
    if (_monthsDisabled > 12) {
      return 0;
    }
    if (_isPartTime) {
      return 0;
    }
```

FIGURE 23. After applying refactoring.

The above figure on the right side panel shows refactored source code. Since the file name stays the same as the original, that QJRT didn't implement rename refactoring, so the refactored class name will need to be changed manually to "CCEtestDone.java," and the same manual renaming process needs to be done with the class's file name.

Output After Applying Refactoring

The output of the sample code is the same as the output listed in Figure 17. Both outputs, before applying refactoring and after applying refactoring, are identical.

```
public class CCEtestDone {

    static int _seniority = 0;
    static int _monthsDisabled = 0;
    static boolean _isPartTime = true;

    public static void main (String[] args) {
        System.out.println(disabilityAmount());
    }

    static double disabilityAmount() {
        double d = 2.0;
if ((_seniority < 2) || (_monthsDisabled > 12) ||
(_isPartTime)) {
  return 0;
}
        return d;
    }
}
```

FIGURE 24. After applying refactoring's Source Code.

Summary

Consolidate Conditional Expression is used in order to make code more readable by combining the expressions inside the conditionals. The structure of the source code didn't change except the conditional. Fowler also suggests using Extract Method on the condition after applied CCE refactoring [1]. Therefore, the complicated conditional expression once again can be replaced by a single method to express the purpose of the conditionals more clearly.

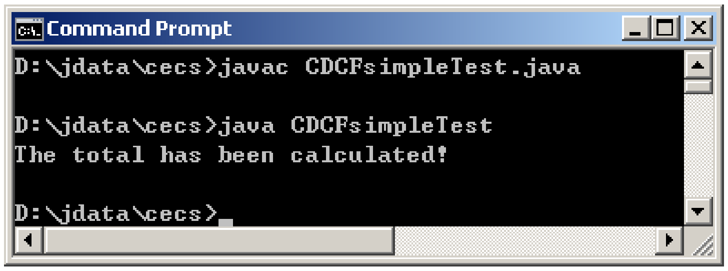
Sample Input and Validation of Consolidate Duplicate Conditional Fragments

Secondly, to test Consolidate Duplicate Conditional Fragments refactoring, a sample code is used from a single Java class named "CDCFsimpleTest." This class has a single main method with two doubles and one boolean. An if-statement is used, the conditional is determined by the declared boolean's value, and inside the if-statement, there exists an identical method, where CDCF will extract this identical method from both the thenStatement and elseStatement. The process of taking out the identical method is the purpose of Consolidate Duplicate Conditional Fragments.

```
public class CDCFsimpleTest {
    public static void main(String[] args) {
        double total = 0;
        double price = 1;
        boolean isSpecialDeal = false;

        if (isSpecialDeal()) {
            total = price * 0.95;
            send();
        } else {
            total = price * 0.98;
            send();
        }
    }
    public static void send() {
        System.out.println ("The total has been
calculated!");
    }
    public static boolean isSpecialDeal() {
        return true;
    }
}
```

FIGURE 25. The sample code for CDCF.

Output Before Applying Refactoring



```
D:\jdata\cecs>javac CDCFsimpleTest.java

D:\jdata\cecs>java CDCFsimpleTest
The total has been calculated!

D:\jdata\cecs>_
```

FIGURE 26. Output before applying CDCF refactoring.

Sample Input

CDCFsimpleTest.java is selected to open in the left panel of the tool. The whole if-else statement is selected and the duplicated fragments will be extracted. The new and improved if-statement body will replace the old if-condition upon applying refactoring. The output stays the same before and after applying refactoring.

Refactoring Process

The very first step is to open the source code in the panel.

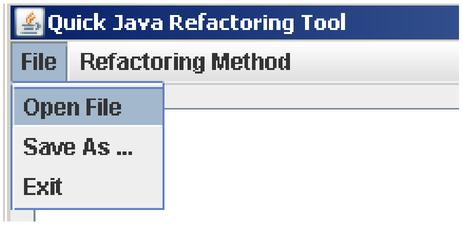
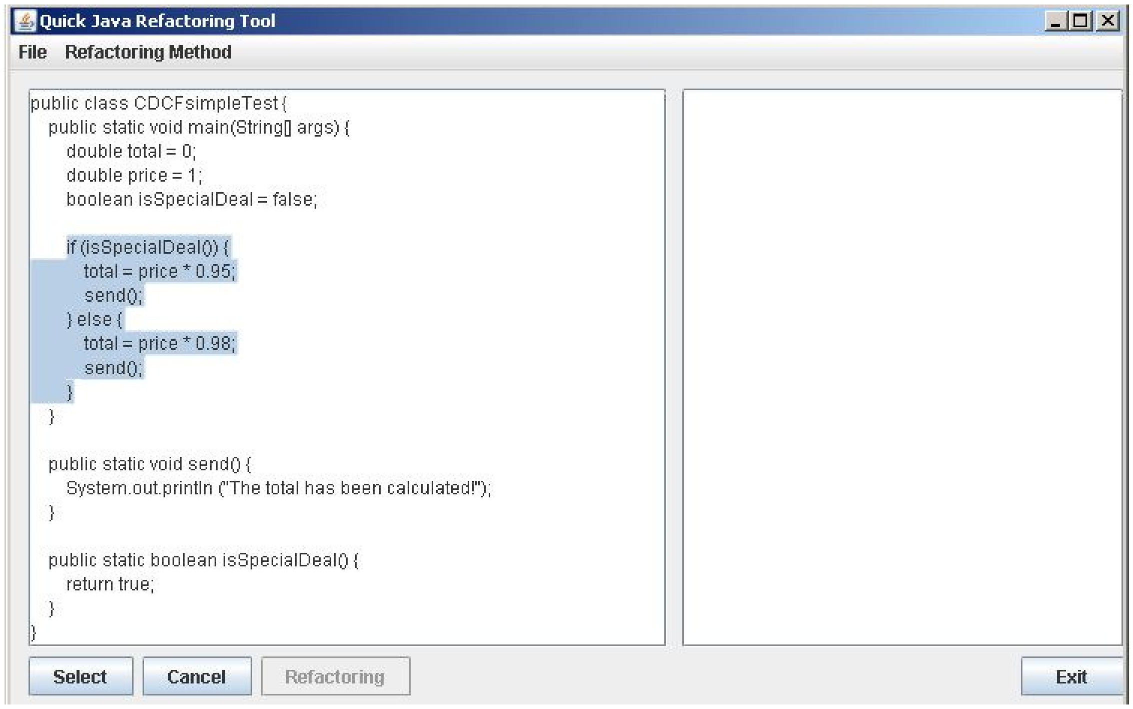
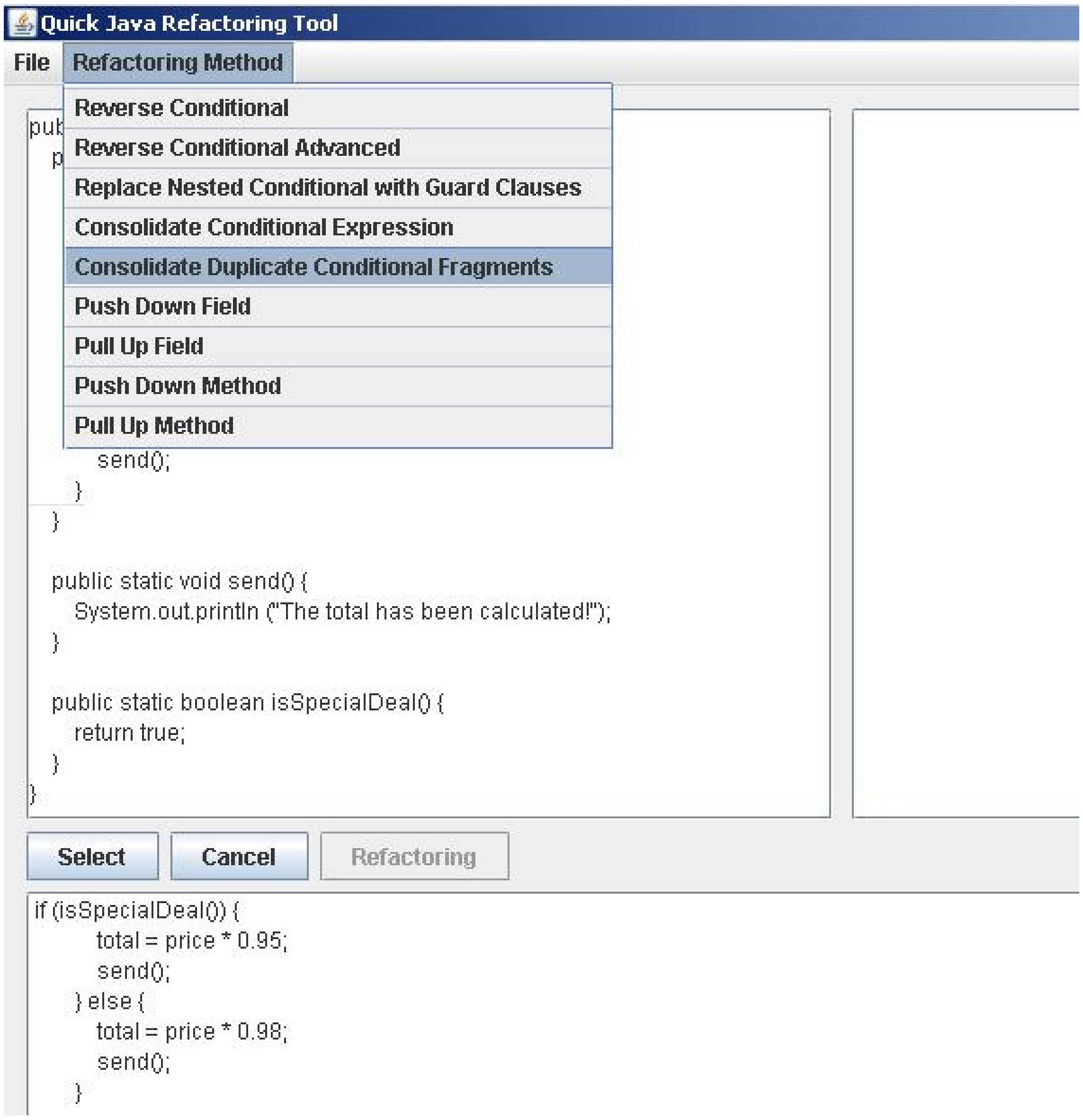


FIGURE 27. Opening the java file for refactoring.



```
public class CDCFsimpleTest {
   public static void main(String[] args) {
      double total = 0;
      double price = 1;
      boolean isSpecialDeal = false;

      if (isSpecialDeal()) {
         total = price * 0.95;
         send();
      } else {
         total = price * 0.98;
         send();
      }
   }

   public static void send() {
      System.out.println ("The total has been calculated!");
   }

   public static boolean isSpecialDeal() {
      return true;
   }
}
```

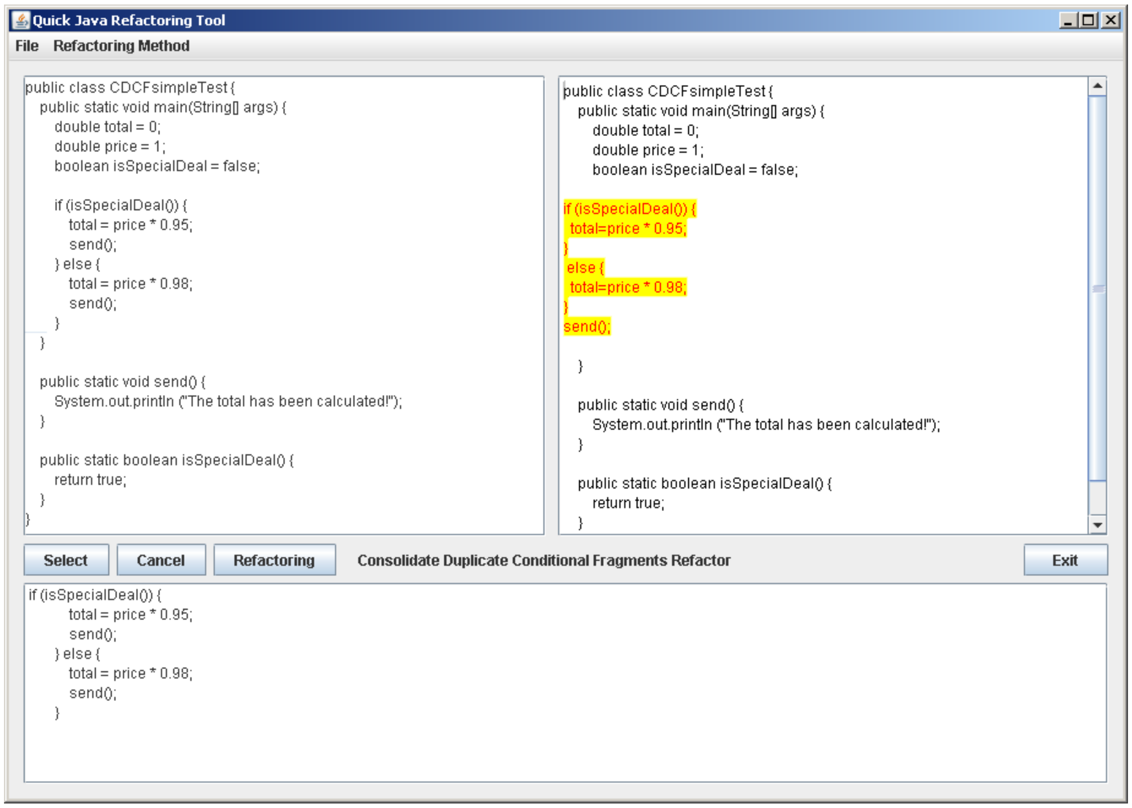
FIGURE 28. Highlighting the source code for refactoring.


Quick Java Refactoring Tool
File
Refactoring Method
Reverse Conditional
Reverse Conditional Advanced
Replace Nested Conditional with Guard Clauses
Consolidate Conditional Expression
Consolidate Duplicate Conditional Fragments
Push Down Field
Pull Up Field
Push Down Method
Pull Up Method
Select
Cancel
Refactoring

```
pub
  p
      send();
    }
  }

  public static void send() {
    System.out.println ("The total has been calculated!");
  }

  public static boolean isSpecialDeal() {
    return true;
  }
}
```

```
if (isSpecialDeal()) {
      total = price * 0.95;
      send();
    } else {
      total = price * 0.98;
      send();
    }
```

FIGURE 29. Selected the highlighted code and choosing CDCF.


Quick Java Refactoring Tool
File
Refactoring Method
Select
Cancel
Refactoring
Consolidate Duplicate Conditional Fragments Refactor
Exit

```
public class CDCFsimpleTest {
   public static void main(String[] args) {
      double total = 0;
      double price = 1;
      boolean isSpecialDeal = false;

      if (isSpecialDeal()) {
         total = price * 0.95;
         send();
      } else {
         total = price * 0.98;
         send();
      }
   }

   public static void send() {
      System.out.println ("The total has been calculated!");
   }

   public static boolean isSpecialDeal() {
      return true;
   }
}
```

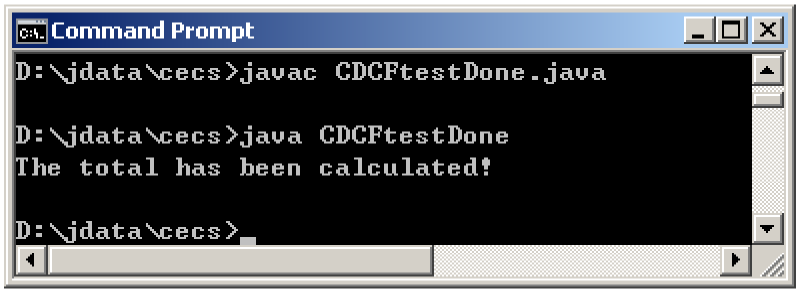
```
public class CDCFsimpleTest {
   public static void main(String[] args) {
      double total = 0;
      double price = 1;
      boolean isSpecialDeal = false;

if (isSpecialDeal()) {
 total=price * 0.95;
}
 else {
 total=price * 0.98;
}
send();

   }

   public static void send() {
      System.out.println ("The total has been calculated!");
   }

   public static boolean isSpecialDeal() {
      return true;
   }
```

```
if (isSpecialDeal()) {
         total = price * 0.95;
         send();
      } else {
         total = price * 0.98;
         send();
      }
```

FIGURE 30. After applying CDCF refactoring.

The above figure on the right side panel shows the refactored source code. It must be saved to the new file name “CDCFtestDone.java”, and accordingly the class name should be changed manually.

Output After Applying Refactoring



```
D:\jdata\cecs>javac CDCFtestDone.java

D:\jdata\cecs>java CDCFtestDone
The total has been calculated!

D:\jdata\cecs>_
```

FIGURE 31. Output after applying CDCF refactoring.

```
public class CDCFtestDone {
    public static void main(String[] args) {
        double total = 0;
        double price = 1;
        boolean isSpecialDeal = false;

if (isSpecialDeal()) {
  total=price * 0.95;
}
 else {
  total=price * 0.98;
}
send();

    }

    public static void send() {
        System.out.println ("The total has been 
calculated!");
    }

    public static boolean isSpecialDeal() {
        return true;
    }
}
```

FIGURE 32. After applying refactorings, the names are changed manually.

Summary

Consolidate Duplicate Conditional Fragments is used to eliminate duplicate code. It results in cleaner source code. The structure of the source code didn’t change except the conditional. Moreover, inside the if-statement’s body, the logical is straightforward with less code presents, which means less potential for making mistakes. In the end, this refactoring makes the conditional clearer.

Sample Input and Validation of Replace Nested Conditional with guard Clauses

Finally, under Simplifying Conditional Expression's Replace Nested Conditional with Guard Clauses, a sample code used is a single Java class named "RNCGCsimpleTest." This class has a single main method to print a double's value, where the double's value is determined by a few nested if-statements, and each of them is determined under a different boolean value. The purpose of RNCGC is to simplify the nested conditional to a list of plain view results.

The sample source code's three boolean variable values will be changed to test the refactoring thoroughly. As a result, four different inputs and outputs of the program will be shown. The first set of inputs occurs when any one of three booleans' values is true, which will return the double's correspondence value. The second set of input occurs when all booleans are false, which will return the default value.

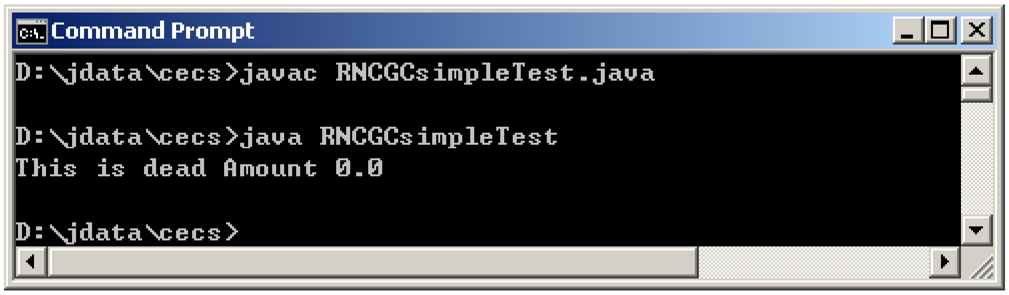
```
Public class RNCGCsimpleTest{

     static boolean _isSeparated, _isRetired, _isDead =
true;

     public static void main(String[] args) {
          System.out.println(getPayAmount());
     }

     static double getPayAmount(){
        double result;
        if (_isDead) {
            result = deadAmount();
        } else {
            if (_isSeparated) {
                result = separatedAmount();
```

FIGURE 33. The sample code for RNCGC.

```
            } else {
                if (_isRetired) {
                    result = retiredAmount();
                } else {
                    result = normalPayAmount();
                }
            };
        }
        return result;
    };
    static double deadAmount() {
          System.out.print("This is dead Amount ");
          return 0;
    };
    static double separatedAmount() {
          System.out.print("This is separated Amount ");
          return 1;
    };
    static double retiredAmount() {
          System.out.print("This is retired Amount ");
          return 2;
    };
    static double normalPayAmount() {
          System.out.print("This is normal Amount ");
          return 3;
    };
}
```

FIGURE 34. The sample code for RNCGC (continued).

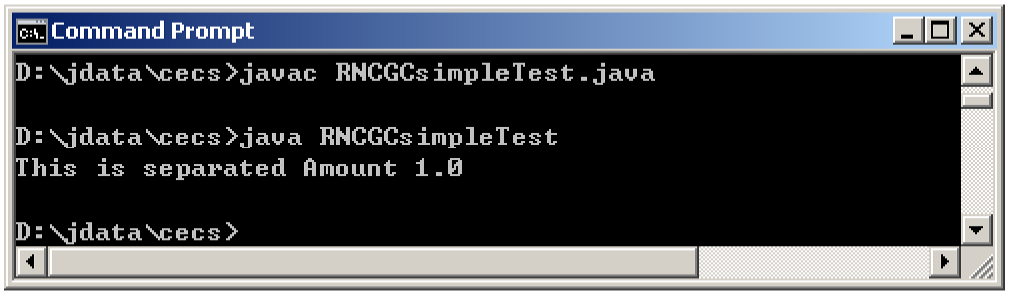
<u>Output 1 Before Applying Refactoring</u>



```
D:\jdata\cecs>javac RNCGCsimpleTest.java

D:\jdata\cecs>java RNCGCsimpleTest
This is dead Amount 0.0

D:\jdata\cecs>
```

FIGURE 35. Output 1 when _isDead = true.

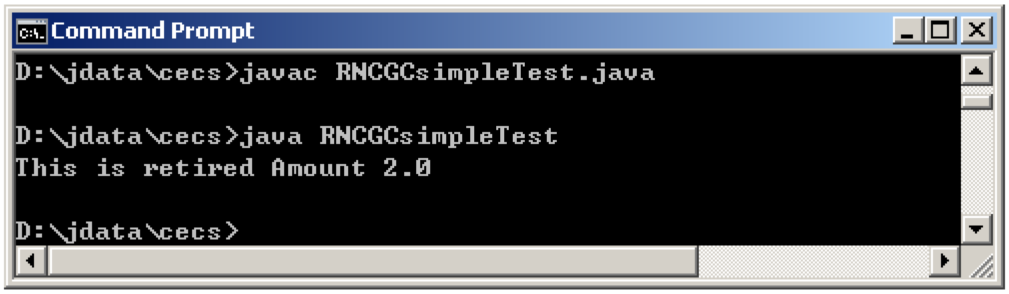
Output 2 Before Applying Refactoring



```
D:\jdata\cecs>javac RNCGCsimpleTest.java

D:\jdata\cecs>java RNCGCsimpleTest
This is separated Amount 1.0

D:\jdata\cecs>
```

FIGURE 36. Output 2 when _isSeparated = true.

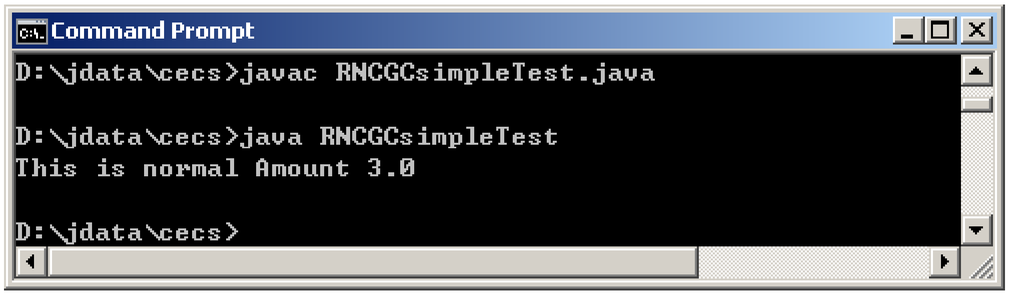
Output 3 Before Applying Refactoring



```
D:\jdata\cecs>javac RNCGCsimpleTest.java

D:\jdata\cecs>java RNCGCsimpleTest
This is retired Amount 2.0

D:\jdata\cecs>
```

FIGURE 37. Output 3 when _isRetired = true.

Output 4 Before Applying Refactoring



```
D:\jdata\cecs>javac RNCGCsimpleTest.java

D:\jdata\cecs>java RNCGCsimpleTest
This is normal Amount 3.0

D:\jdata\cecs>
```

FIGURE 38. Output 4 when all three boolean's values are false.

Sample Input

RNCGCsimpleTest.java is selected to open in the left panel of the tool. The conditional nested if-else statements are selected. Upon applying refactoring, the new and improved if-else statement body will replace the old one. The output stays the same before and after applying refactoring.

Refactoring Process

The very first step is to open the source code in the panel.

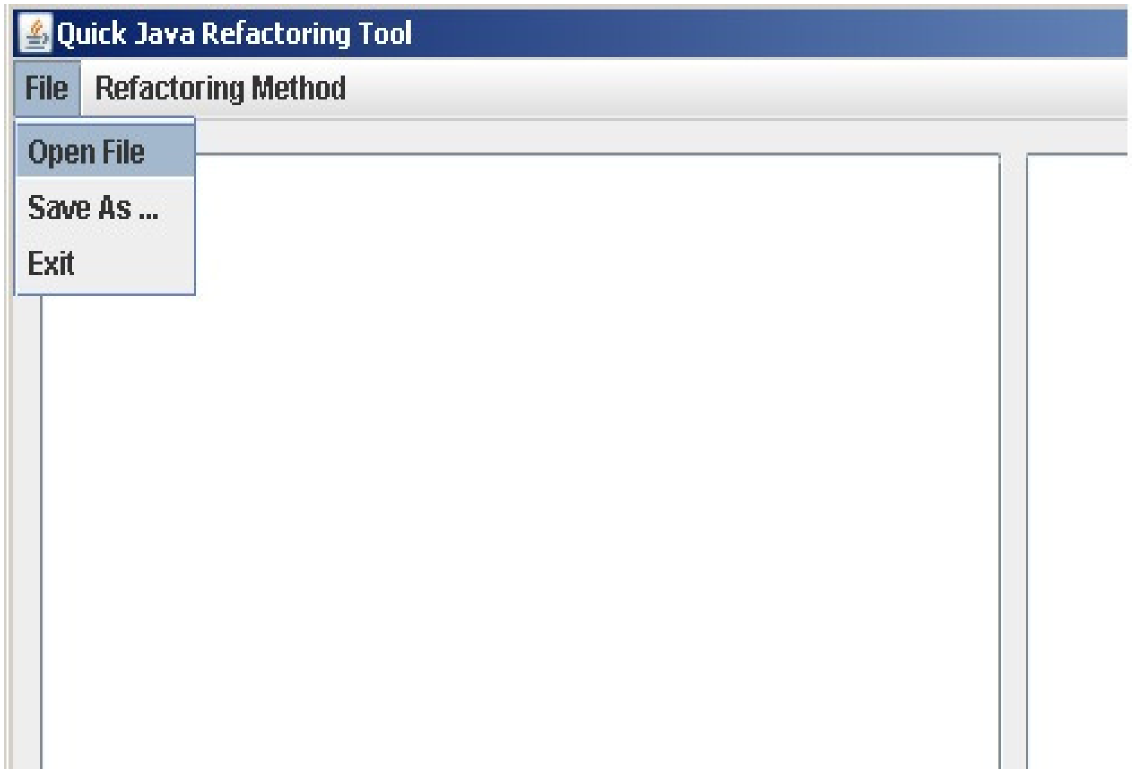
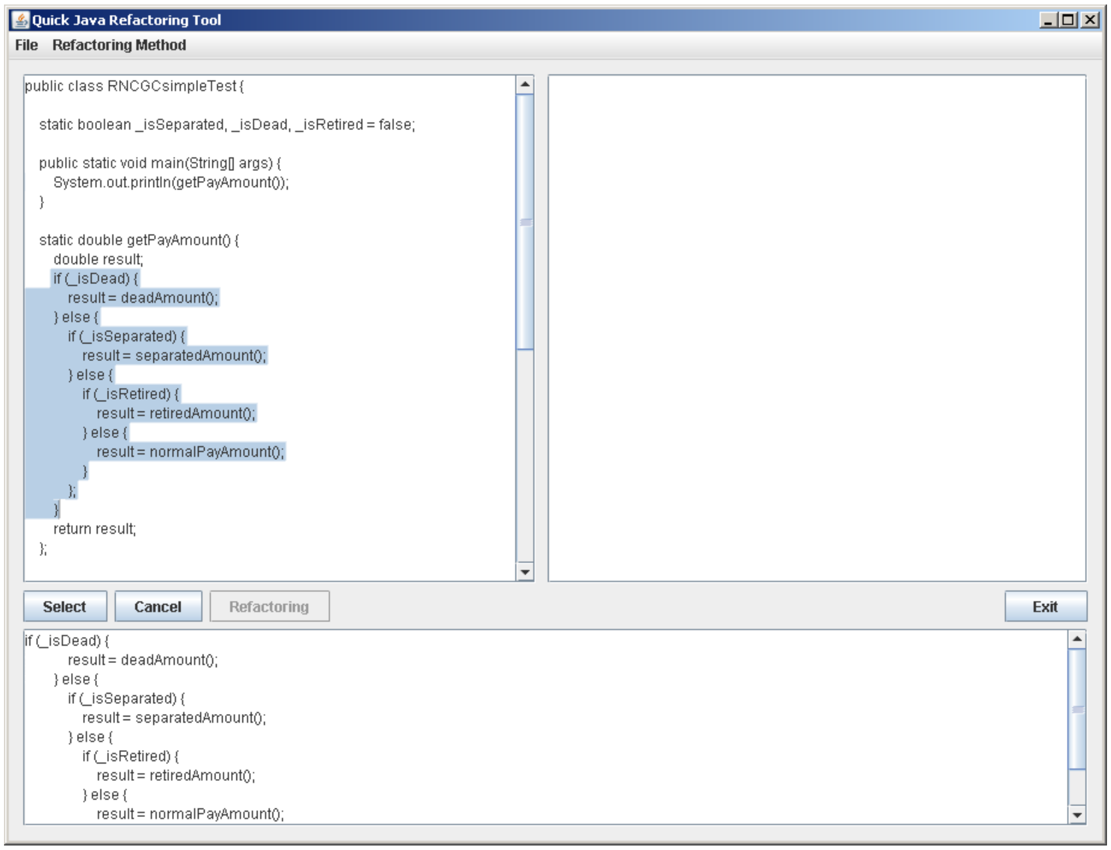


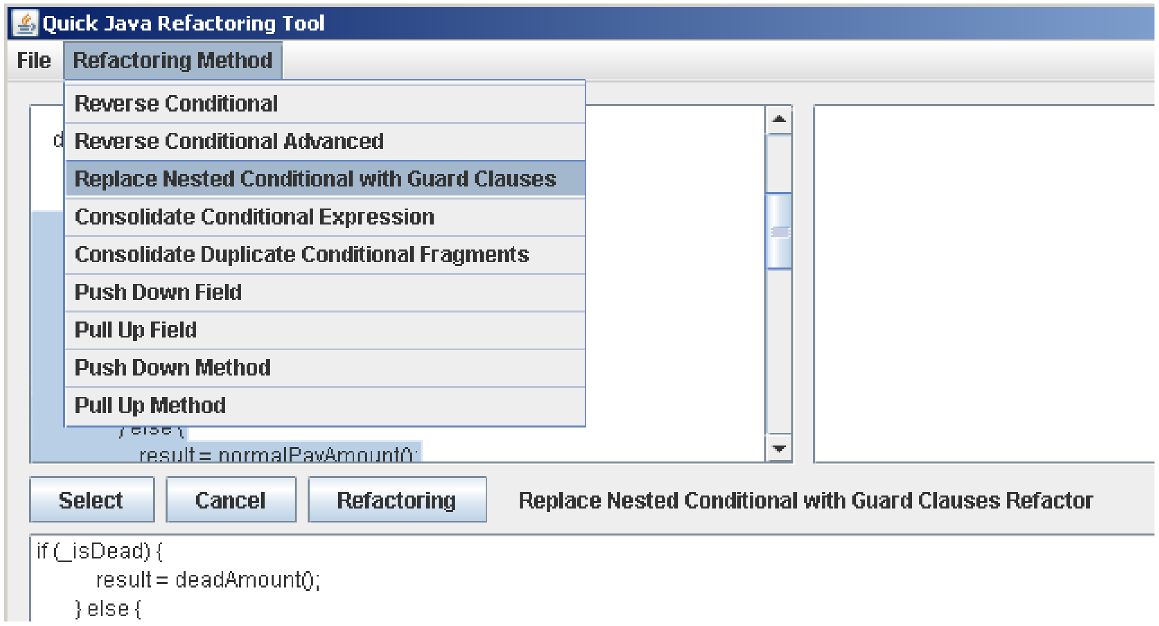
FIGURE 39. Opening the java file for refactoring.



```
public class RNCGCsimpleTest {

  static boolean _isSeparated, _isDead, _isRetired = false;

  public static void main(String[] args) {
    System.out.println(getPayAmount());
  }

  static double getPayAmount() {
    double result;
    if (_isDead) {
      result = deadAmount();
    } else {
      if (_isSeparated) {
        result = separatedAmount();
      } else {
        if (_isRetired) {
          result = retiredAmount();
        } else {
          result = normalPayAmount();
        }
      };
    }
    return result;
  };
```

```
if (_isDead) {
      result = deadAmount();
    } else {
      if (_isSeparated) {
        result = separatedAmount();
      } else {
        if (_isRetired) {
          result = retiredAmount();
        } else {
          result = normalPayAmount();
```

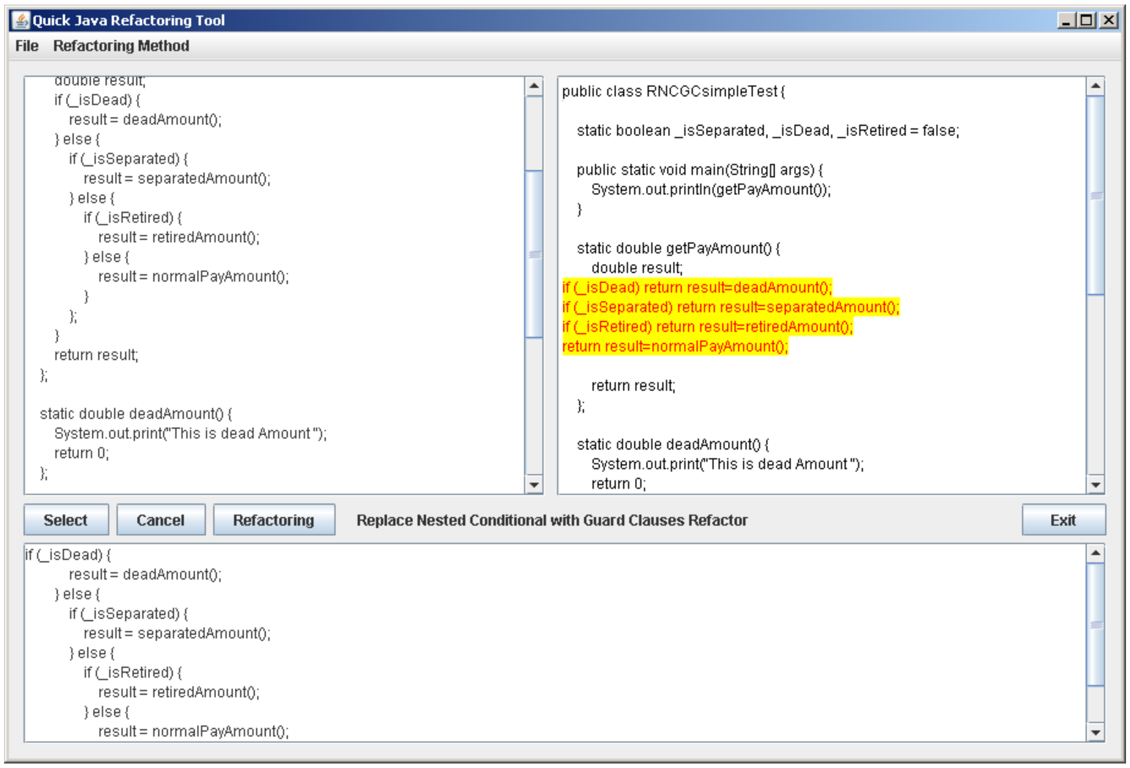
FIGURE 40. Highlighting and selecting the source code for refactoring.


Quick Java Refactoring Tool
File
Refactoring Method
Reverse Conditional
Reverse Conditional Advanced
Replace Nested Conditional with Guard Clauses
Consolidate Conditional Expression
Consolidate Duplicate Conditional Fragments
Push Down Field
Pull Up Field
Push Down Method
Pull Up Method
Select
Cancel
Refactoring

Replace Nested Conditional with Guard Clauses Refactor

```
if (_isDead) {
      result = deadAmount();
    } else {
```

FIGURE 41. Selecting the correspondences refactoring method.


Quick Java Refactoring Tool
File
Refactoring Method
Select
Cancel
Refactoring
Replace Nested Conditional with Guard Clauses Refactor
Exit

```
    double result;
    if (_isDead) {
      result = deadAmount();
    } else {
      if (_isSeparated) {
        result = separatedAmount();
      } else {
        if (_isRetired) {
          result = retiredAmount();
        } else {
          result = normalPayAmount();
        }
      };
    }
    return result;
  };

  static double deadAmount() {
    System.out.print("This is dead Amount ");
    return 0;
  };
```

```
public class RNCGCsimpleTest {

  static boolean _isSeparated, _isDead, _isRetired = false;

  public static void main(String[] args) {
    System.out.println(getPayAmount());
  }

  static double getPayAmount() {
    double result;
if (_isDead) return result=deadAmount();
if (_isSeparated) return result=separatedAmount();
if (_isRetired) return result=retiredAmount();
return result=normalPayAmount();

    return result;
  };

  static double deadAmount() {
    System.out.print("This is dead Amount ");
    return 0;
```

```
if (_isDead) {
      result = deadAmount();
    } else {
      if (_isSeparated) {
        result = separatedAmount();
      } else {
        if (_isRetired) {
          result = retiredAmount();
        } else {
          result = normalPayAmount();
```

FIGURE 42. After applying RNCGC refactoring.

The above figure on the right side panel shows the refactored source code. It must be saved to a new file named "RNCGCtestDone.java," and accordingly, the class name needs to be changed manually.

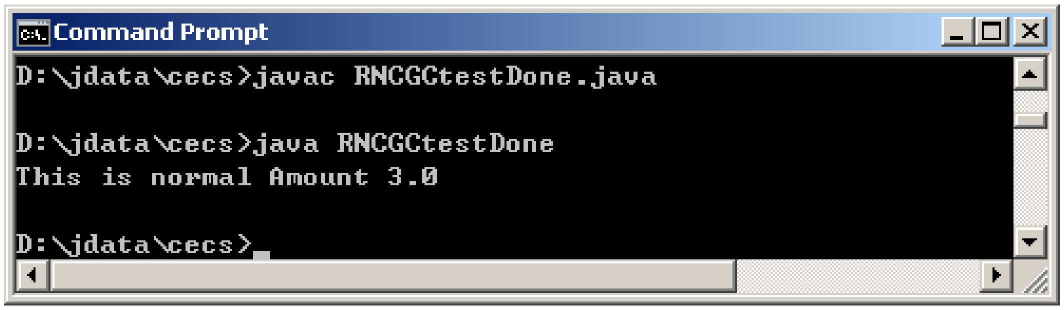
```
public class RNCGCtestDone{

    static boolean _isSeparated, _isDead, _isRetired =
false;

    public static void main(String[] args) {
        System.out.println(getPayAmount());
    }

    static double getPayAmount() {
```

FIGURE 43. After applying refactoring's Source Code with manually changed names.

```
        double result;
if (_isDead) return result=deadAmount();
if (_isSeparated) return result=separatedAmount();
if (_isRetired) return result=retiredAmount();
return result=normalPayAmount();
//         return result;
    };
    static double deadAmount() {
        System.out.print("This is dead Amount ");
        return 0;
    };
    static double separatedAmount() {
        System.out.print("This is separated Amount ");
        return 1;
    };

    static double retiredAmount() {
        System.out.print("This is retired Amount ");
        return 2;
    };

    static double normalPayAmount() {
        System.out.print("This is normal Amount ");
        return 3;
    };
}
```

FIGURE 44. After applying refactoring's Source Code with manually changed names (continuted).

Output After Applying Refactoring



```
D:\jdata\cecs>javac RNCGCtestDone.java

D:\jdata\cecs>java RNCGCtestDone
This is normal Amount 3.0

D:\jdata\cecs>
```

FIGURE 45. Output after applying RNCGC refactoring with default boolean = false.

Summary

Replace Nested Conditional with Guard Clauses (RNCGC) is used to sort the conditional logic in a straightforward manner. The modern languages rule of one entry with one exit point is not useful. Clarity is more important than anything [1]. When the conditions are all equal, there is no reason to weight one more than others. In addition, the unbalanced weighting of the condition might confuse reader. After applying RNCGC refactoring, the entire source code's structure will stay the same, but conditional structures have changed to weight different conditions equally. As a result, the logic of the if-statement becomes clearer. In the end, RNCGC presents the source code's conditional equivalently.

Sample Input and Validation of Pull Up Field

To test Pull Up Field refactoring under "dealing with generalization," there are at least three different source codes used; at least one class is a superclass and two are subclasses. *Car.java, Vehicle.java, Truck.java, and VehicleTest.java* are used to test Pull Up Field refactoring. *Vehicle.java* is the superclass, and *VehicleTest.java* has the main method to test the inherited classes *Vehicle.java.* Inside the *Car.java and Truck.java* classes, there is one identical instance variable named *wheels,* and the value is set at integer of 4. This is the field to be pulled up. Inside the superclass *Vehicle,* there are two fields; one field is named *seats*, with a public value set at integer of 2; another field is named *doors,* with a value set at integer of 3 through class *Vehicle*'s constructor respectively.

```
//This is the super class
public class Vehicle{

    public int seats = 2;
     public int doors;
    Vehicle(){
    doors=3;
    }
}
```

```
//This sub class one inherits Vehicle.java
public class Car extends Vehicle{

    public int wheels = 4;
    public String toString(){
     return "This car has "+seats+" Seats, "+doors+" Doors
"+"and "+wheels+" wheels.";
    }
}
```

```
//This sub class two inherits Vehicle.java
public class Truck extends Vehicle{

     public int wheels =4;
     public String toString(){
     return "This truck has "+seats+" Seats, "+doors+"
Doors "+"and "+wheels+" wheels.";
    }
}
```

```
//This is the driver class
public class VehicleTest{
     public static void main(String args[]){

     Car car = new Car();
     System.out.println(car);
     Truck truck= new Truck();
     System.out.println(truck);
    }
}
```

FIGURE 46. The sample code for Pull Up Field and Push Down Field.

Class Diagram Before Applying Refactoring

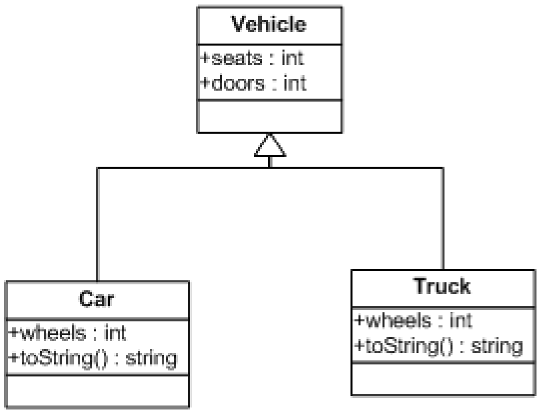
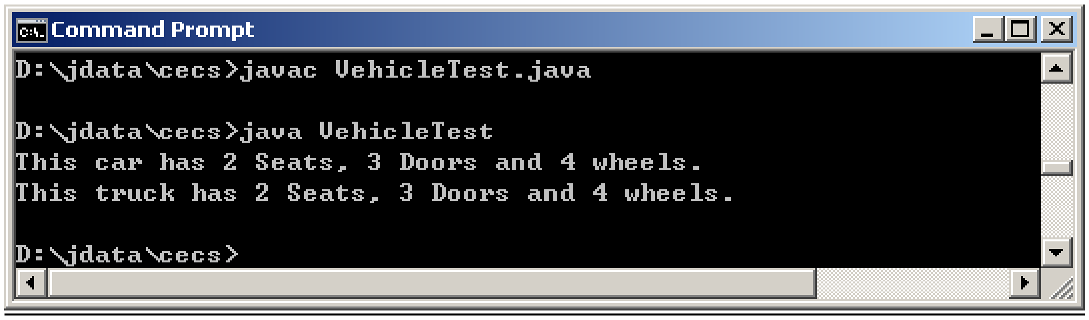


FIGURE 47. Class diagram before applying Pull Up Field refactoring.

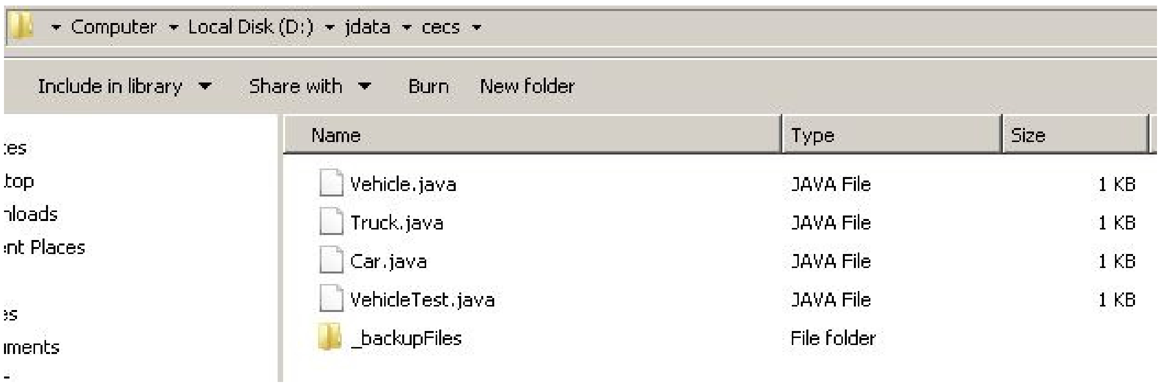
Output Before Applying Refactoring



```
D:\jdata\cecs>javac VehicleTest.java

D:\jdata\cecs>java VehicleTest
This car has 2 Seats, 3 Doors and 4 wheels.
This truck has 2 Seats, 3 Doors and 4 wheels.

D:\jdata\cecs>
```

FIGURE 48. Output before applying Pull Up Field refactoring.

File Location Before Applying Refactoring



Name	Type	Size
Vehicle.java	JAVA File	1 KB
Truck.java	JAVA File	1 KB
Car.java	JAVA File	1 KB
VehicleTest.java	JAVA File	1 KB
_backupFiles	File folder	

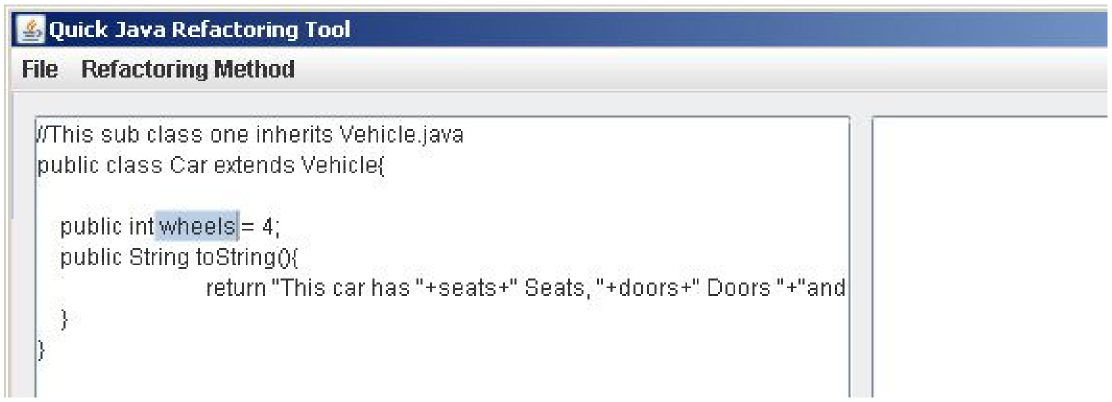
FIGURE 49. All files in the hard disk before applying Pull Up Field refactoring.

<u>Sample Input</u>

Car.java is selected to open in the left panel of the tool. The field *wheels* is selected to refactor.

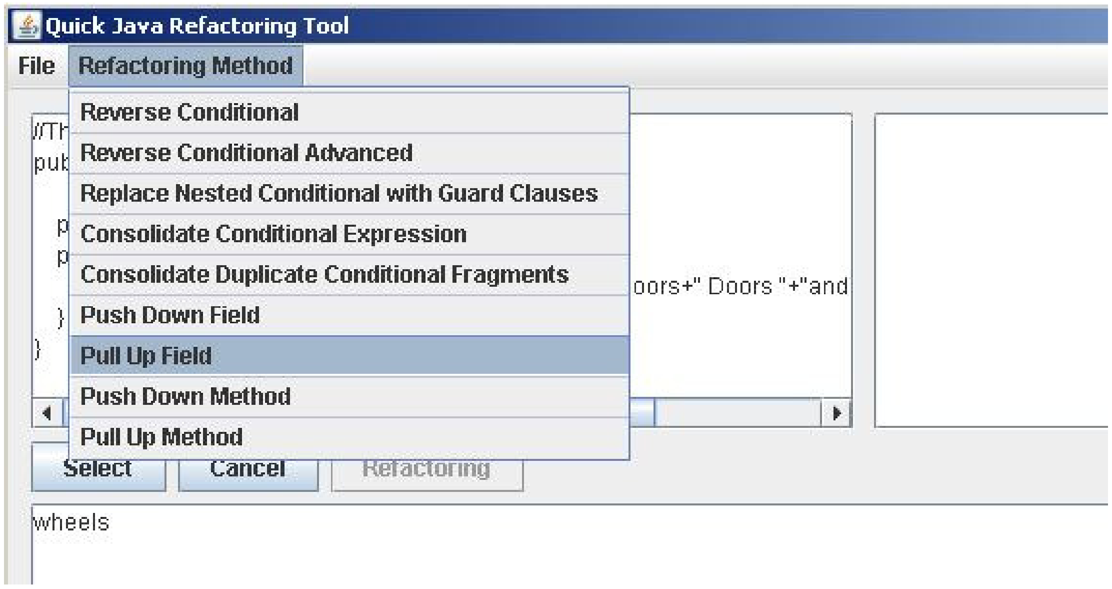
<u>Refactoring Process</u>

The first step is to open the source code in the panel, and highlight the field.



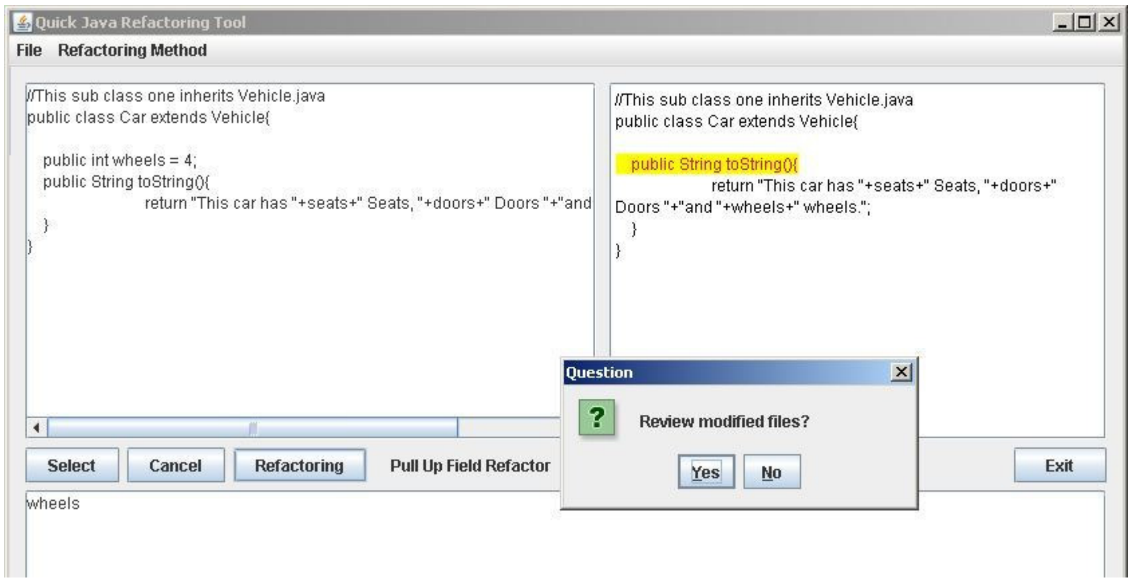
```
//This sub class one inherits Vehicle.java
public class Car extends Vehicle{

  public int wheels = 4;
  public String toString(){
             return "This car has "+seats+" Seats, "+doors+" Doors "+"and
  }
}
```

FIGURE 50. Highlighting the field for pull up field.


Quick Java Refactoring Tool
File Refactoring Method
Reverse Conditional
Reverse Conditional Advanced
Replace Nested Conditional with Guard Clauses
Consolidate Conditional Expression
Consolidate Duplicate Conditional Fragments
Push Down Field
Pull Up Field
Push Down Method
Pull Up Method
Select Cancel Refactoring

```
wheels
```

FIGURE 51. Selecting PUF refactoring.



```
//This sub class one inherits Vehicle.java
public class Car extends Vehicle{

   public int wheels = 4;
   public String toString(){
                    return "This car has "+seats+" Seats, "+doors+" Doors "+"and
   }
}
```

```
//This sub class one inherits Vehicle.java
public class Car extends Vehicle{

   public String toString(){
                    return "This car has "+seats+" Seats, "+doors+"
Doors "+"and "+wheels+" wheels.";
   }
}
```

Review modified files?

wheels

FIGURE 52. After applying PUF refactoring and before viewing modified files

The above figure is the screenshot of applying the PUF. The right side panel, as usual, shows the currently modified class. Moreover, there is a pop up window that asks developer whether to review the modified files, with the default program to open the modified files being *notepad++*, and the default location being "C:\\Program Files\\Notepad++ \\notepad++.exe". If there is no *notepad++* installed, the developer will need to open the files on his or her own. This refactoring method changes the related files in the background. All changed files have a backup file in the same directory.

```
//This is the super class
public class Vehicle{

     public int wheels = 4;
     public int seats = 2;
     public int doors;
```

FIGURE 53. Source codes after applying PUF refactoring.

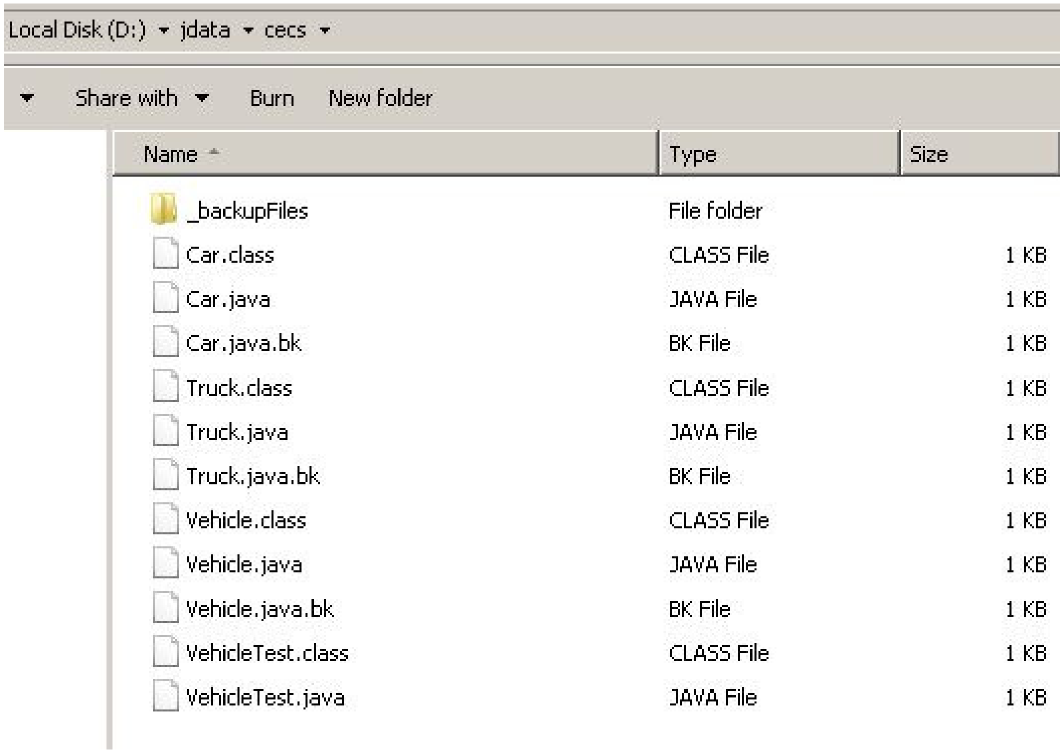
```
    Vehicle(){
    doors=3;
    }
}
```

```
//This sub class one inherits Vehicle.java
public class Car extends Vehicle{

    public String toString(){
     return "This car has "+seats+" Seats, "+doors+" Doors
"+"and "+wheels+" wheels.";
    }
}
```

```
//This sub class two inherits Vehicle.java
public class Truck extends Vehicle{

     public String toString(){
     return "This truck has "+seats+" Seats, "+doors+"
Doors "+"and "+wheels+" wheels.";
    }
}
```

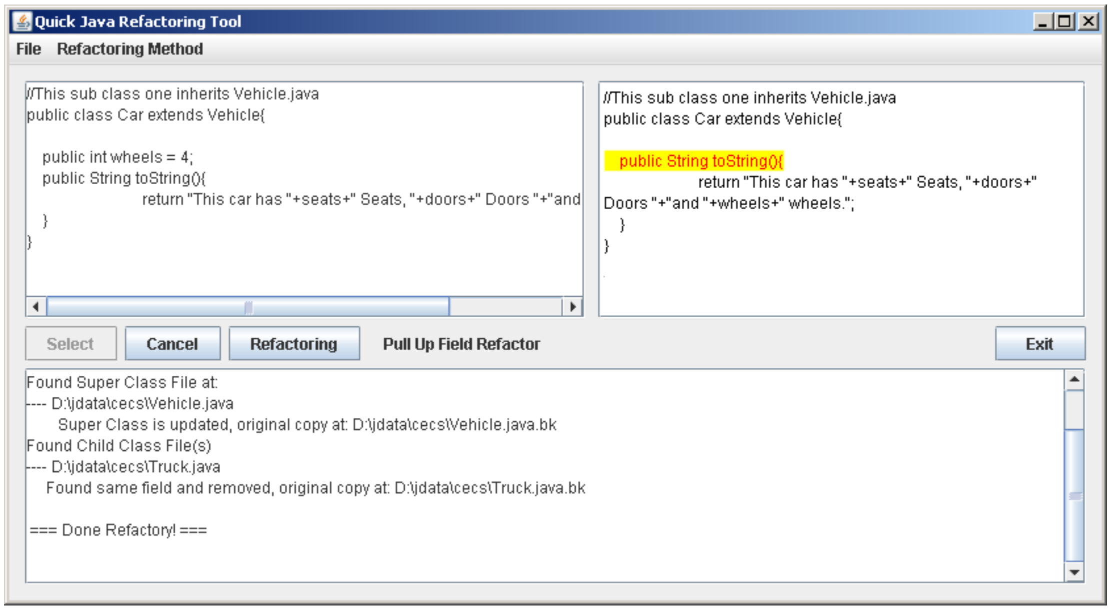
FIGURE 54. Source codes after applying PUF refactoring (continued).

File Location After Applying Refactoring



Name	Type	Size
_backupFiles	File folder	
Car.class	CLASS File	1 KB
Car.java	JAVA File	1 KB
Car.java.bk	BK File	1 KB
Truck.class	CLASS File	1 KB
Truck.java	JAVA File	1 KB
Truck.java.bk	BK File	1 KB
Vehicle.class	CLASS File	1 KB
Vehicle.java	JAVA File	1 KB
Vehicle.java.bk	BK File	1 KB
VehicleTest.class	CLASS File	1 KB
VehicleTest.java	JAVA File	1 KB

FIGURE 55. Source code's file location after applying PUF refactoring.



```
//This sub class one inherits Vehicle.java
public class Car extends Vehicle{

   public int wheels = 4;
   public String toString(){
                  return "This car has "+seats+" Seats, "+doors+" Doors "+"and
   }
}
```

```
//This sub class one inherits Vehicle.java
public class Car extends Vehicle{

   public String toString(){
                  return "This car has "+seats+" Seats, "+doors+"
Doors "+"and "+wheels+" wheels.";
   }
}
```

```
Found Super Class File at:
---- D:\jdata\cecs\Vehicle.java
     Super Class is updated, original copy at: D:\jdata\cecs\Vehicle.java.bk
Found Child Class File(s)
---- D:\jdata\cecs\Truck.java
   Found same field and removed, original copy at: D:\jdata\cecs\Truck.java.bk

=== Done Refactory! ===
```

FIGURE 56. Screen shot after eliminating the pop up window.

The above figure is the screenshot after performing refactoring, and after deciding whether to review the modified files. The bottom panel shows detailed information of super- and subclasses. Furthermore, the files' location is revealed, with detail of all the original files and all the updated files, as well as backups of the original files.

Output After Applying Refactoring

The output of the sample code is the same as the output listed in Figure 48. Both outputs, before applying refactoring and after applying refactoring, are identical.

Class Diagram After Applying Refactoring

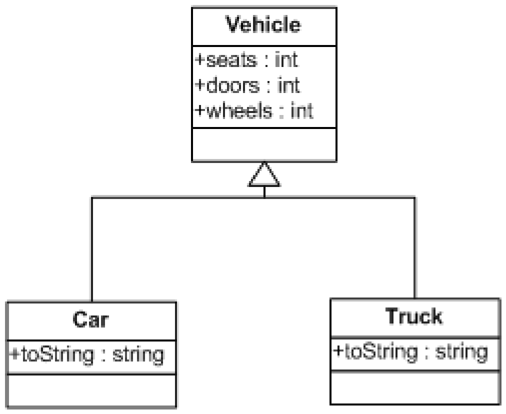


FIGURE 57. Class diagram after applying PUF refactoring.

Summary

Pull Up Field is used to reduce the duplicate fields developed in different subclasses. The structure of the source code is changed, but the output stays the same. The source code is more cohesive in general. More fields can be reused in the superclass. That is one of the goals in refactoring – reusability. In addition, if the fields were private, after Pull Up Field refactoring, the modifier of the field will be changed to *protected* due to inheritance in the superclass.   In the end, Pull Up Field refactor made the superclass more generalized, and easier to use.

Sample Input and Validation of Push Down Field

To test Push Down Field refactoring, previously supplied source codes are used to test. The superclass's field will be pushed down to two subclasses. In detail, the *seats* in *Vehicle.java* will be pushed down to *Car.java and Truck.java* classes.

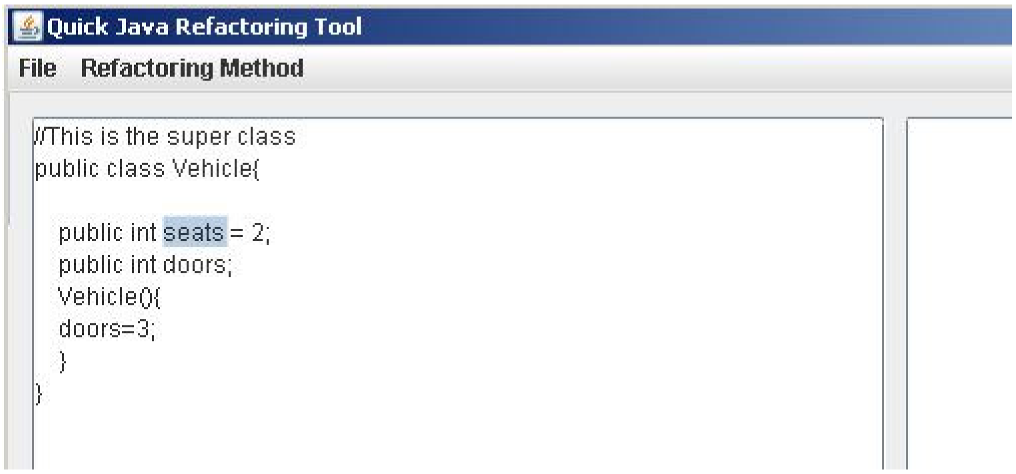
Before applying refactoring, the class diagram, the output, and the file location are all the same since this involves recycling previous source code. The file location is reset before Push Down Field refactoring.

Sample Input

Vehicle.java is selected to open in the left panel of the tool. The field *seats* is selected to refactor.

Refactoring Process

The first step is to open the source code in the panel, and highlight the field.



```
//This is the super class
public class Vehicle{

  public int seats = 2;
  public int doors;
  Vehicle(){
  doors=3;
  }
}
```

FIGURE 58. Highlighting the field for push down field refactoring.

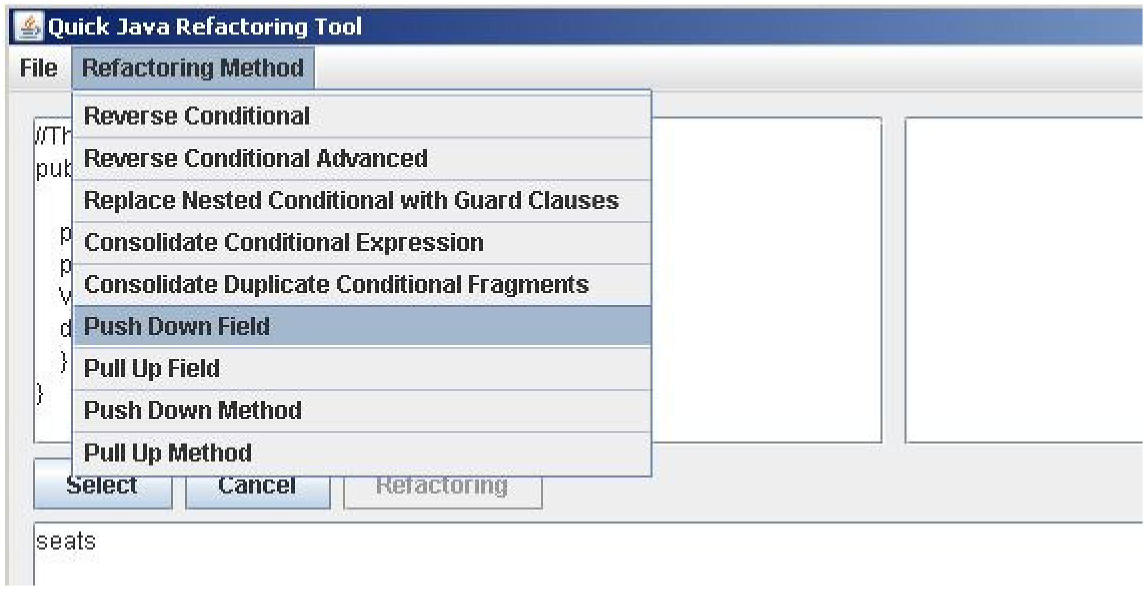
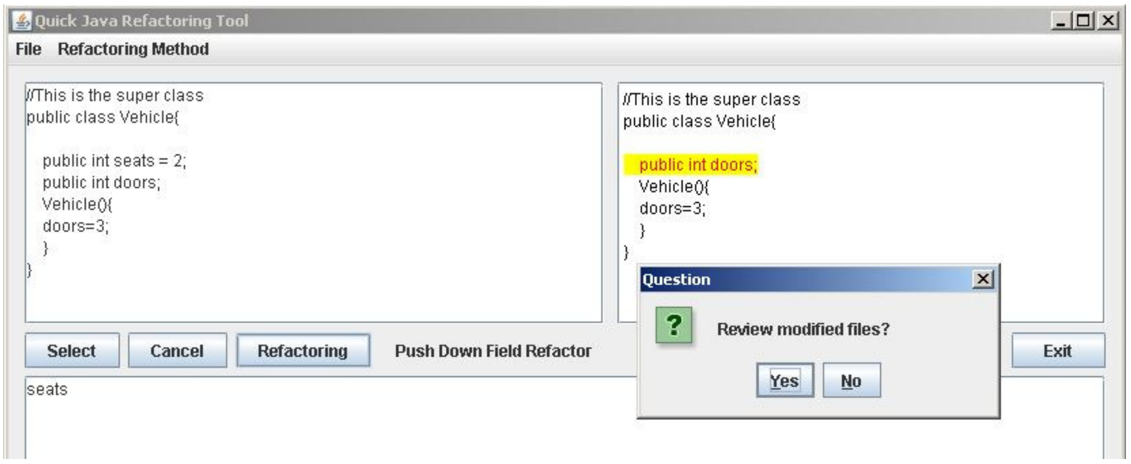

Quick Java Refactoring Tool
File
Refactoring Method
Reverse Conditional
Reverse Conditional Advanced
Replace Nested Conditional with Guard Clauses
Consolidate Conditional Expression
Consolidate Duplicate Conditional Fragments
Push Down Field
Pull Up Field
Push Down Method
Pull Up Method
Select
Cancel
Refactoring
seats

FIGURE 59. Selecting PDF refactoring.



```
//This is the super class
public class Vehicle{

   public int seats = 2;
   public int doors;
   Vehicle(){
   doors=3;
   }
}
```

```
//This is the super class
public class Vehicle{

   public int doors;
   Vehicle(){
   doors=3;
   }
}
```

FIGURE 60. After applying PDF refactoring and before viewing modified files.

The above figure is the screenshot of applying the PDF. The right side panel, as usual, shows the currently modified class. Moreover, the same pop up window asks the developer to review the modified files.

```
//This is the super class

public class Vehicle{

    public int doors;
    Vehicle(){
    doors=3;
    }
}
```

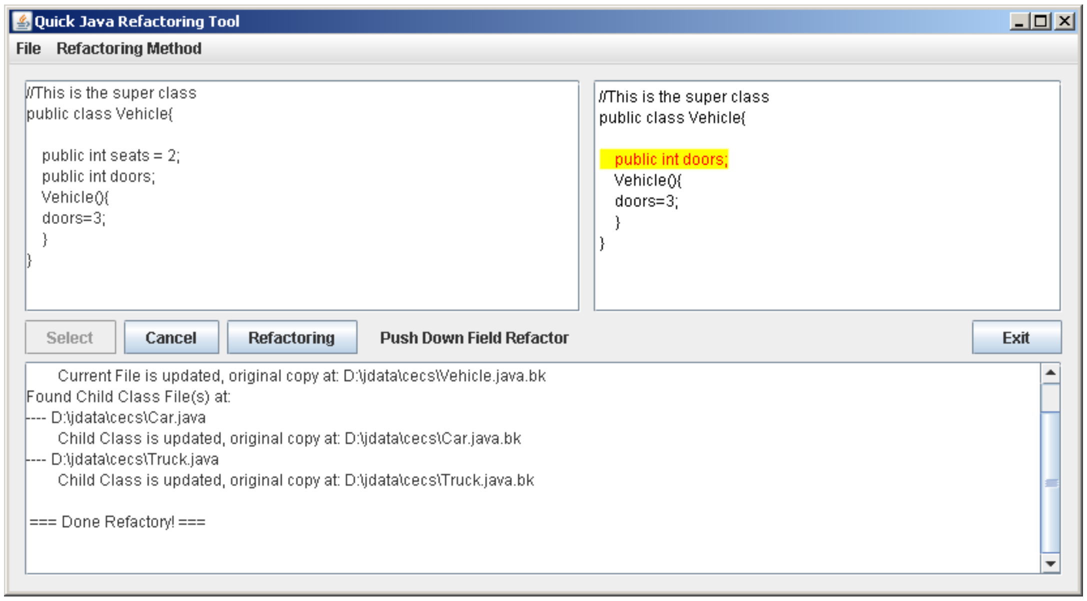
```
//This sub class one inherits Vehicle.java

public class Car extends Vehicle{

     public int seats = 2;
     public int wheels = 4;
    public String toString(){
     return "This car has "+seats+" Seats, "+doors+" Doors
"+"and "+wheels+" wheels.";
    }
}
```

```
//This sub class two inherits Vehicle.java

public class Truck extends Vehicle{

     public int seats = 2;
     public int wheels =4;
     public String toString(){
     return "This truck has "+seats+" Seats, "+doors+"
Doors "+"and "+wheels+" wheels.";
    }
}
```

FIGURE 61. Source codes after applying PDF refactorings.

File Location After Applying Refactoring

The file locations are the same as Figure 55 since the same three classes are affected during the refactoring.



```
//This is the super class
public class Vehicle{

   public int seats = 2;
   public int doors;
   Vehicle(){
   doors=3;
   }
}
```

```
//This is the super class
public class Vehicle{

   public int doors;
   Vehicle(){
   doors=3;
   }
}
```

```
      Current File is updated, original copy at: D:\jdata\cecs\Vehicle.java.bk
Found Child Class File(s) at:
---- D:\jdata\cecs\Car.java
      Child Class is updated, original copy at: D:\jdata\cecs\Car.java.bk
---- D:\jdata\cecs\Truck.java
      Child Class is updated, original copy at: D:\jdata\cecs\Truck.java.bk

=== Done Refactory! ===
```

FIGURE 62. Screenshot after applying PDF and after closing pop up window.

The above figure is the screenshot after performing refactoring. The bottom panel shows detailed information of original and modified files.

Output After Applying Refactoring

The output of the sample code is the same as the output listed in Figure 48. Both outputs, before and after applying refactoring, are identical.

Class Diagram After Applying Refactoring

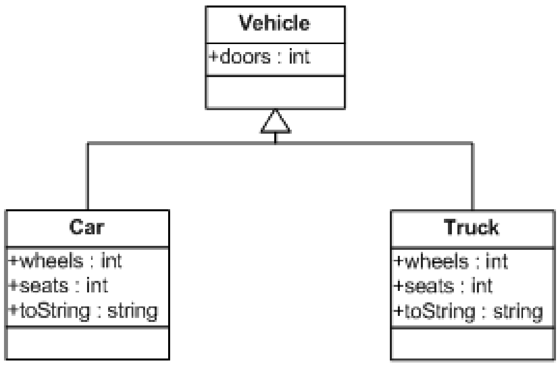


FIGURE 63. Class diagram after applying PDF refactoring.

Summary

Push Down Field is used to reduce the fields in superclass. The structure of the source code is changed, but the output doesn't change in our test. The source code is loosely coupled. The superclass has fewer fields to worry about. To remove the pushed down field in either the *Car.java or Truck.java,* while modifying the source code accordingly, the program will execute with no error. In the end, PDF makes the inheritance classes loosely coupled, which is one feature that shows the high quality of Object Oriented Program.

Sample Input and Validation of Pull Up Method

To test Pull Up Method refactoring, the condition of test files is similar to Pull Up Field refactoring. There are at least three different source codes used. One class is a superclass and two classes are subclasses. *Automobile.java, FamilyCar.java, SportsCar.java,* and *AutoTest.java* are used. *Automobile.java* is the superclass, and *AutoTest.java* has the main method to test the classes inherited with *Automobile.java.* Both *SportsCar.java* and *FamilyCar.java* extend *Automobile.java.* In *FamilyCar.java* and *SportsCar.java* classes, there are four methods, which are identical. The idea method needed to be pulled up is showEngine(), and two other methods are involved in order for showEngine() to work properly because of method call. Those two methods are getType() and getSpeed(). In the superclass, there is another method that is presented, but it is used for Push Down Method testing purposes, which has nothing to do with Pull Up Method right now.

```
//This is the super class
public class Automobile {
    public int seat;
    public String type;
        public void onSalePrice(){
            System.out.println("This car is ON SALE");
        }
}
```

```
//This sub class one inherits Automobile.java
public class FamilyCar extends Automobile {
    FamilyCar() {
        seat = 7;
    }
    public void setType(String s) {
        type = s;
    }
    public String getType() {
        return type;
    }
    public int getSpeed() {
        String t = getType();
        if (t == "e") {
            return 70;
        } else if (t == "g") {
            return 180;
        } else {
            return 0;
        }
    }
    public void showEngine() {
        String s = getType();
        if (s == "g") {
            System.out.println("This is a sports car !
uses " + s + " and max speed is " + getSpeed());
        } else if (s == "e") {
            System.out.println("This is a family car !
uses " + s + " and max speed is " + getSpeed());
        } else {
            System.out.println("NOTHING IS HERE, WRONG");
        }
    }
}
```

FIGURE 64. The sample code for PUM and PDM.

```
//This sub class two inherits Automobile.java
public class SportsCar extends Automobile {
    SportsCar() {
        seat = 2;
    }
    public void setType(String s) {
        type = s;
    }
    public String getType() {
        return type;
    }
    public int getSpeed() {
        String t = getType();
        if (t == "e") {
            return 70;
        } else if (t == "g") {
            return 180;
        } else {
            return 0;
        }
    }
    public void showEngine() {
        String s = getType();
        if (s == "g") {
            System.out.println("This is a sports car !
uses " + s + " and max speed is " + getSpeed());
        } else if (s == "e") {
            System.out.println("This is a family car !
uses " + s + " and max speed is " + getSpeed());
        } else {
            System.out.println("NOTHING IS HERE, WRONG");
        }
    }       }
```

```
//This is the test class
public class AutoTest {
    public static void main(String[] args) {
        FamilyCar fc = new FamilyCar();
        fc.setType("e");
        fc.showEngine();
        fc.onSalePrice();
        SportsCar sc = new SportsCar();
        sc.setType("g");
        sc.showEngine();
        sc.onSalePrice();    }  }
```

FIGURE 65. The sample code for PUM and PDM (continued).

Class Diagram Before Applying Refactoring

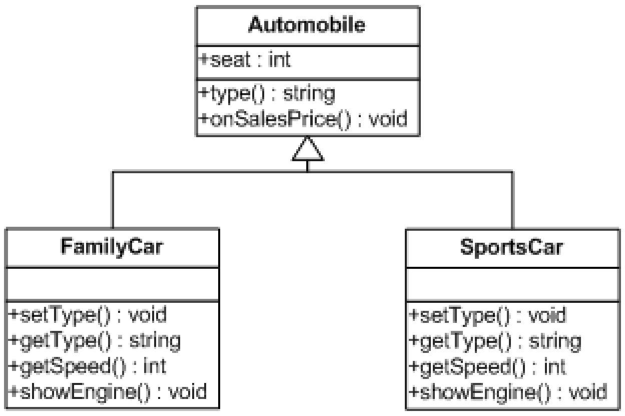
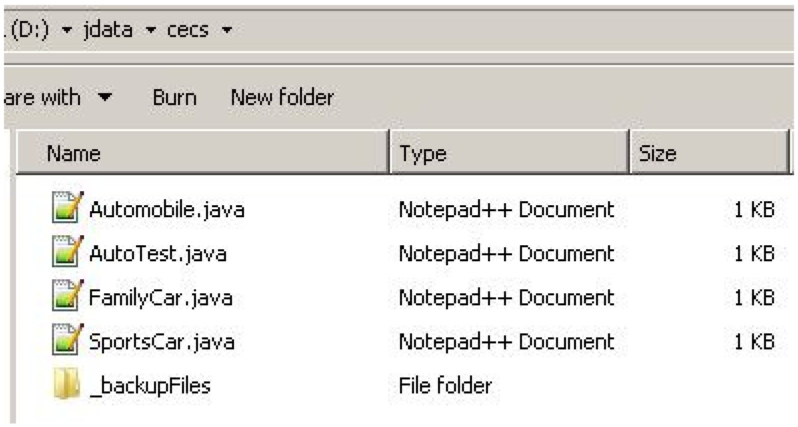


FIGURE 66. Class diagram before applying PUM refactoring.

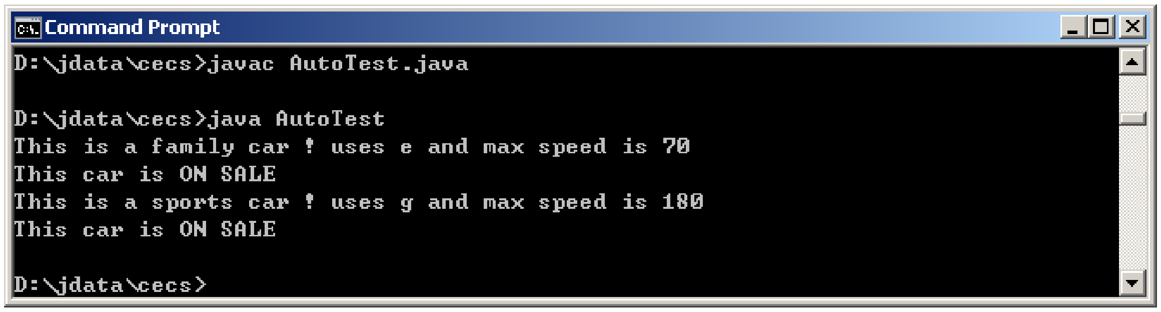
File Location Before Applying Refactoring



Name	Type	Size
Automobile.java	Notepad++ Document	1 KB
AutoTest.java	Notepad++ Document	1 KB
FamilyCar.java	Notepad++ Document	1 KB
SportsCar.java	Notepad++ Document	1 KB
_backupFiles	File folder	

FIGURE 67. All files in the hard disk before applying PUM.

Output Before Applying Refactoring



```
D:\jdata\cecs>javac AutoTest.java

D:\jdata\cecs>java AutoTest
This is a family car ! uses e and max speed is 70
This car is ON SALE
This is a sports car ! uses g and max speed is 180
This car is ON SALE

D:\jdata\cecs>
```

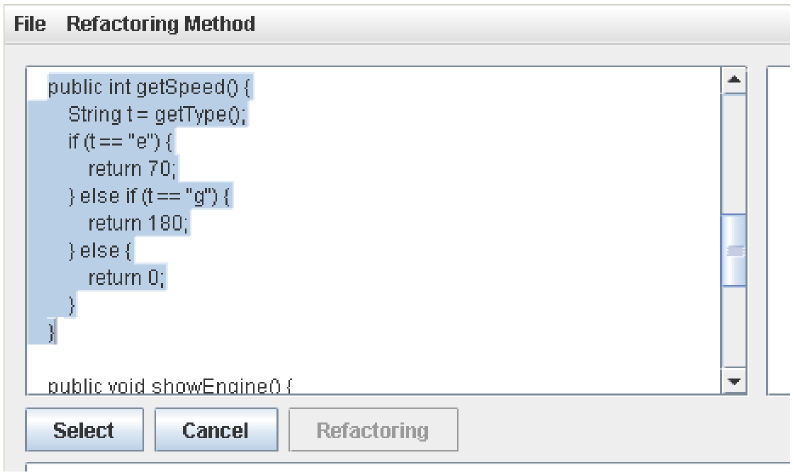
FIGURE 68. Output before applying PUM.

Sample Input

FamilyCar.java is selected to open in the left panel of the tool.

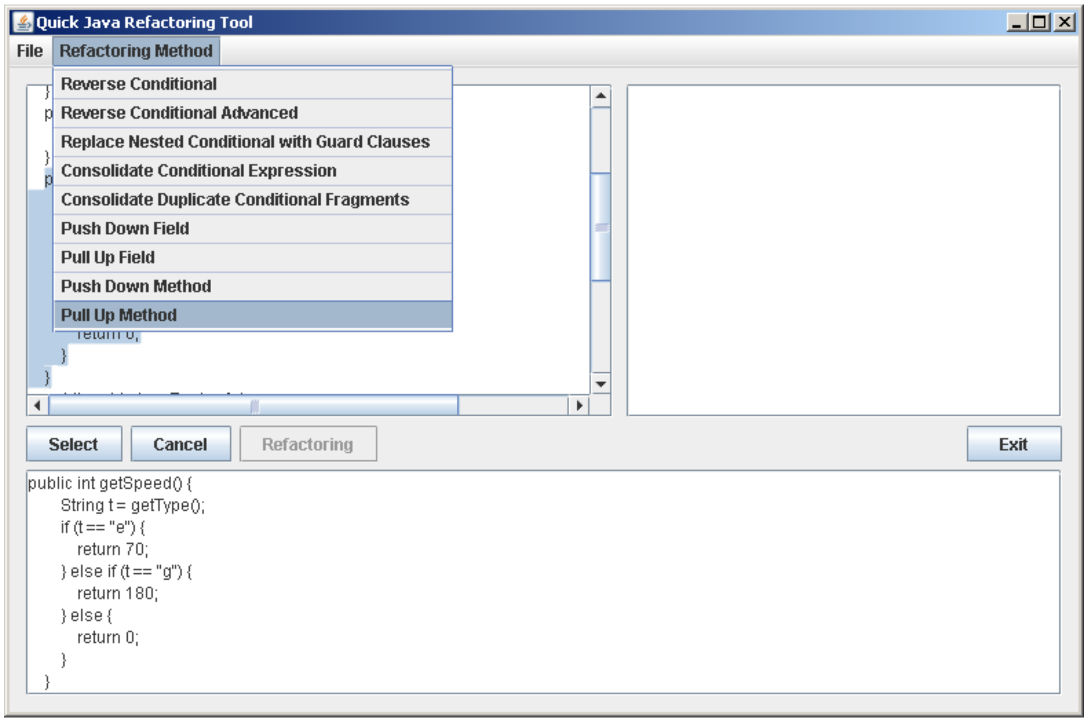
Refactoring Process

The first step is to open the source code in the panel, and analyze the method that needs to be pulled up. The showEngine() calls for two methods; these are getType() and getSpeed(). In order to work after applying refactoring, those two methods need to be in the same class as showEngine() with no reference object showing. Thus, the Pull Up Method must be performed on getType(), and getSpeed().



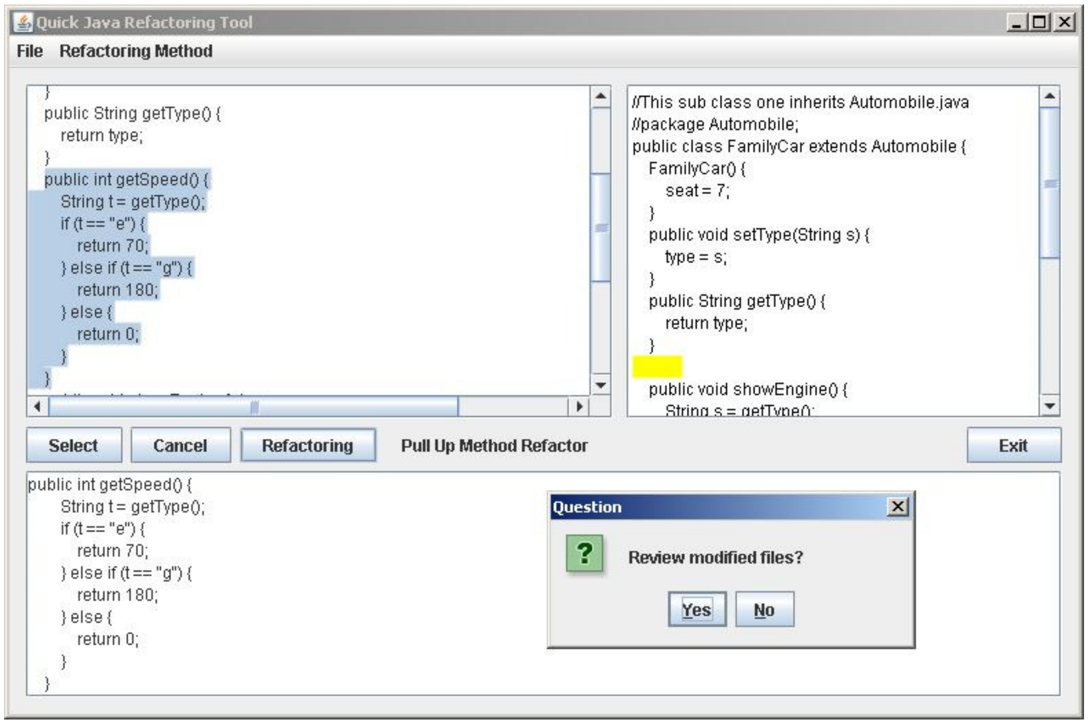
```
public int getSpeed() {
  String t = getType();
  if (t == "e") {
    return 70;
  } else if (t == "g") {
    return 180;
  } else {
    return 0;
  }
}

public void showEngine() {
```

FIGURE 69. Highlighting the getSpeed() method to be pulled up.


Quick Java Refactoring Tool
File
Refactoring Method
Reverse Conditional
Reverse Conditional Advanced
Replace Nested Conditional with Guard Clauses
Consolidate Conditional Expression
Consolidate Duplicate Conditional Fragments
Push Down Field
Pull Up Field
Push Down Method
Pull Up Method
Select
Cancel
Refactoring
Exit

```
public int getSpeed() {
    String t = getType();
    if (t == "e") {
        return 70;
    } else if (t == "g") {
        return 180;
    } else {
        return 0;
    }
}
```

FIGURE 70. Selecting PUM refactoring.



```
}
public String getType() {
    return type;
}
public int getSpeed() {
    String t = getType();
    if (t == "e") {
        return 70;
    } else if (t == "g") {
        return 180;
    } else {
        return 0;
    }
}
```

```
//This sub class one inherits Automobile.java
//package Automobile;
public class FamilyCar extends Automobile {
    FamilyCar() {
        seat = 7;
    }
    public void setType(String s) {
        type = s;
    }
    public String getType() {
        return type;
    }

    public void showEngine() {
        String s = getType();
```

```
public int getSpeed() {
    String t = getType();
    if (t == "e") {
        return 70;
    } else if (t == "g") {
        return 180;
    } else {
        return 0;
    }
}
```

FIGURE 71. After applying PUM on getSpeed() and before viewing modified files.

This step is similar to the previous field operation, in the right side panel shows the currently modified class. The pop up window asks the developer whether to review the modified files right away. This refactoring method changes the related files in the background. All modified files have a backup file in the same directory. Before going any further, the developer should execute the program to make sure it still works properly by using the same procedure and refactor for method getType() and showEngine() one by one. The final output is the same as before applying refactoring.

```
//This is the super class
public class Automobile {
    public int seat;
    public String type;
        public void showEngine() {
          String s = getType();
          if (s == "g") {
               System.out.println("This is a sports car !
uses " + s
                         + " and max speed is " +
getSpeed());
          } else if (s == "e") {
               System.out.println("This is a family car !
uses " + s
                         + " and max speed is " +
getSpeed());
          } else {
               System.out.println("NOTHING IS HERE,
WRONG");
          }
     }
          public String getType() {
          return type;
     }
```

FIGURE 72. Source codes after applying PUM refactoring.

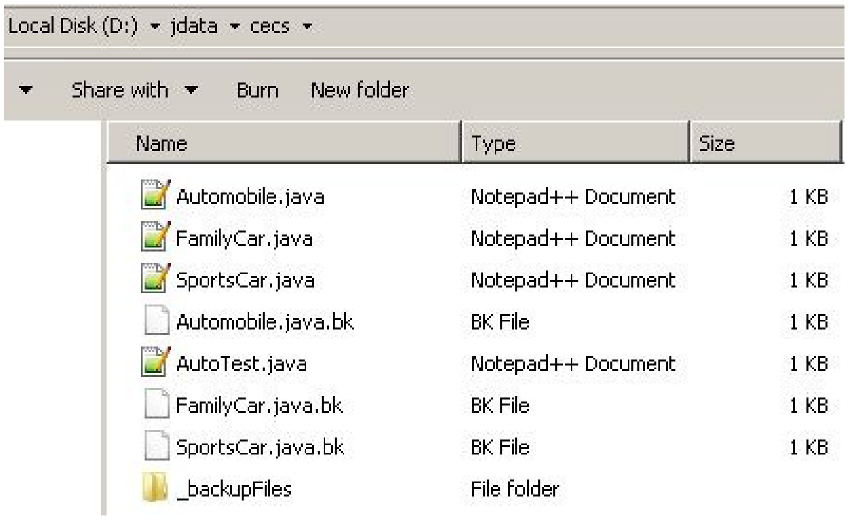
```
          public int getSpeed() {
          String t = getType();
          if (t == "e") {
               return 70;
          } else if (t == "g") {
               return 180;
          } else {
               return 0;
          }
     }
          public void onSalePrice(){
            System.out.println("This car is ON SALE");
        }
}
```

```
//This sub class one inherits Automobile.java
public class FamilyCar extends Automobile {
    FamilyCar() {
        seat = 7;
    }
    public void setType(String s) {
        type = s;
    }
}
```

```
//This sub class one inherits Automobile.java
public class SportsCar extends Automobile {
    SportsCar() {
        seat = 2;
    }
    public void setType(String s) {
        type = s;
    }
}
```

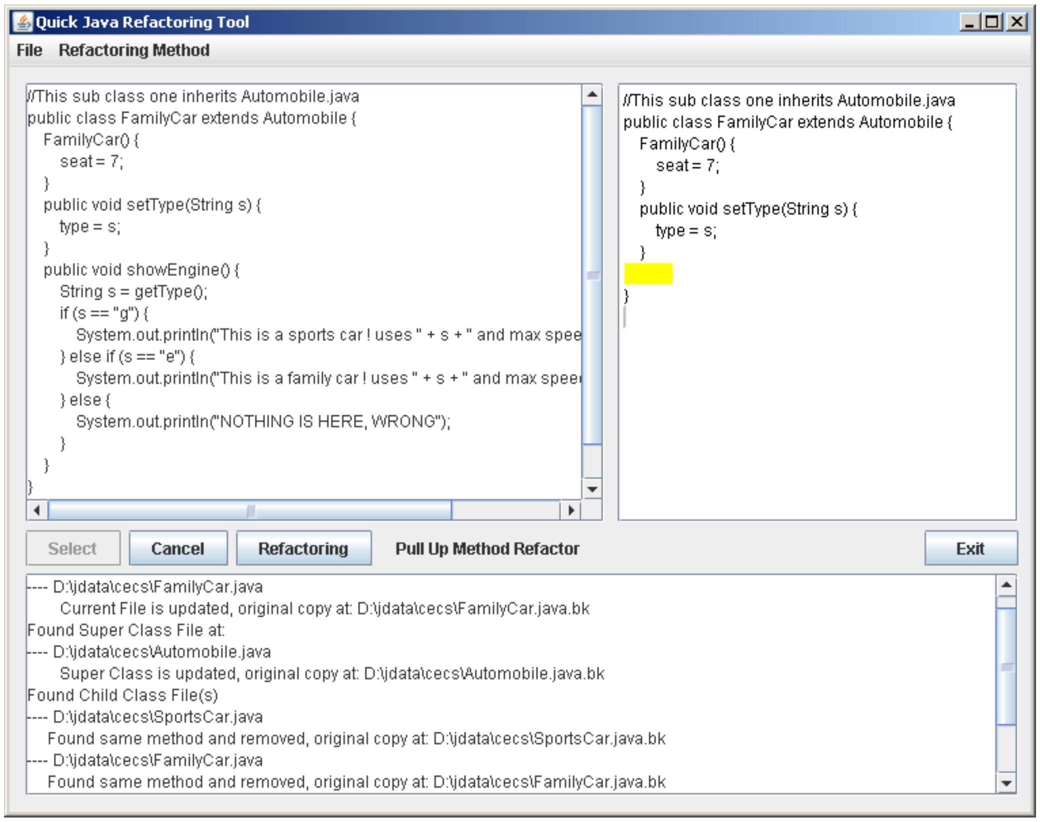
FIGURE 73. Source codes after applying PUM refactoring (continued).

File Location After Applying Refactoring



Name	Type	Size
Automobile.java	Notepad++ Document	1 KB
FamilyCar.java	Notepad++ Document	1 KB
SportsCar.java	Notepad++ Document	1 KB
Automobile.java.bk	BK File	1 KB
AutoTest.java	Notepad++ Document	1 KB
FamilyCar.java.bk	BK File	1 KB
SportsCar.java.bk	BK File	1 KB
_backupFiles	File folder	

FIGURE 74. Source code's file location after applying PUM refactoring.



```
//This sub class one inherits Automobile.java
public class FamilyCar extends Automobile {
   FamilyCar() {
      seat = 7;
   }
   public void setType(String s) {
      type = s;
   }
   public void showEngine() {
      String s = getType();
      if (s == "g") {
         System.out.println("This is a sports car ! uses " + s + " and max spee
      } else if (s == "e") {
         System.out.println("This is a family car ! uses " + s + " and max spee
      } else {
         System.out.println("NOTHING IS HERE, WRONG");
      }
   }
}
```

```
//This sub class one inherits Automobile.java
public class FamilyCar extends Automobile {
   FamilyCar() {
      seat = 7;
   }
   public void setType(String s) {
      type = s;
   }

}
```

```
---- D:\jdata\cecs\FamilyCar.java
      Current File is updated, original copy at: D:\jdata\cecs\FamilyCar.java.bk
Found Super Class File at:
---- D:\jdata\cecs\Automobile.java
      Super Class is updated, original copy at: D:\jdata\cecs\Automobile.java.bk
Found Child Class File(s)
---- D:\jdata\cecs\SportsCar.java
   Found same method and removed, original copy at: D:\jdata\cecs\SportsCar.java.bk
---- D:\jdata\cecs\FamilyCar.java
   Found same method and removed, original copy at: D:\jdata\cecs\FamilyCar.java.bk
```

FIGURE 75. Screenshot after applying PUM refactoring.

The above figure is the screenshot after refactoring, and after deciding whether to review the modified files. The bottom panel shows the detailed information on super- and subclasses. Furthermore, the files' location is presented, with detail of all the original files and all the updated files, as well as backup of the original files. The backup files in this case are the files before performing the Pull Up Method on showEngine().

Output After Applying Refactoring

The output of the sample code is the same as the output listed in Figure 68. Both outputs, before and after applying refactoring, are identical.

Class Diagram After Applying Refactoring

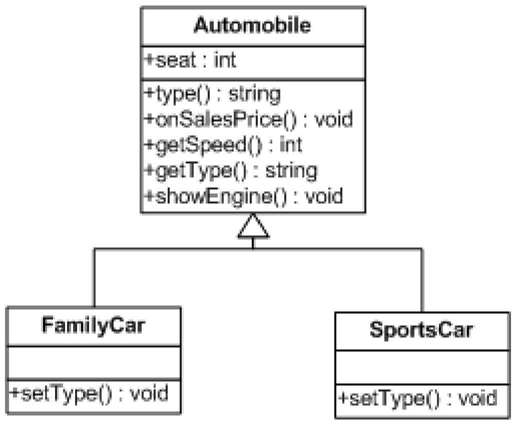


FIGURE 76. Class diagram after applying PUM.

Summary

Pull Up Method is similar to Pull Up Field. The purpose is to reduce the duplicate methods developed in different subclasses. The structure of the source code is changed, but the output stays the same. The superclass is more cohesive, it has more methods that can be reused or inherited. That is one good feature of OOP – reusability. In the end, The PUM cleans up subclasses.

Sample Input and Validation of Push Down Method

To test Push Down Method refactoring, previous source codes for Pull Up Method are re-used. The onSalePrice() method is the superclass since it was the holiday, and every single car is on sale. However, the scenario is that sports cars are on sale with discount prices since it is back-to-school time, but the family car is back to its regular price. Thus, the superclass specifically for sports car method will make sense with sports cars only. There is no reason to keep it in the superclass. Pushing this method to sports car is the best option.

Before applying refactoring, the class diagram, the output, and the file location are all the same since previous source code is being recycled. The file location is cleared out before Push Down Method refactoring.

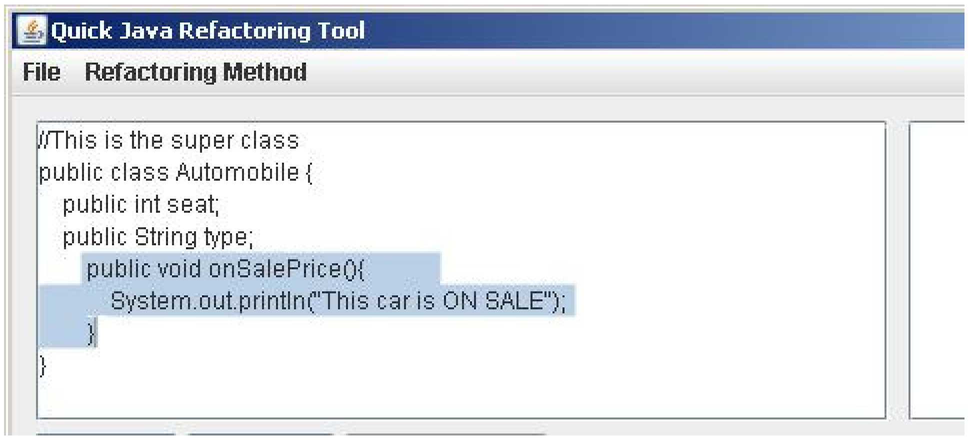
Sample Input

Automobile.java is selected to open in the left panel of the tool. The method *onSalePrice* is selected to refactor.

Refactoring Process

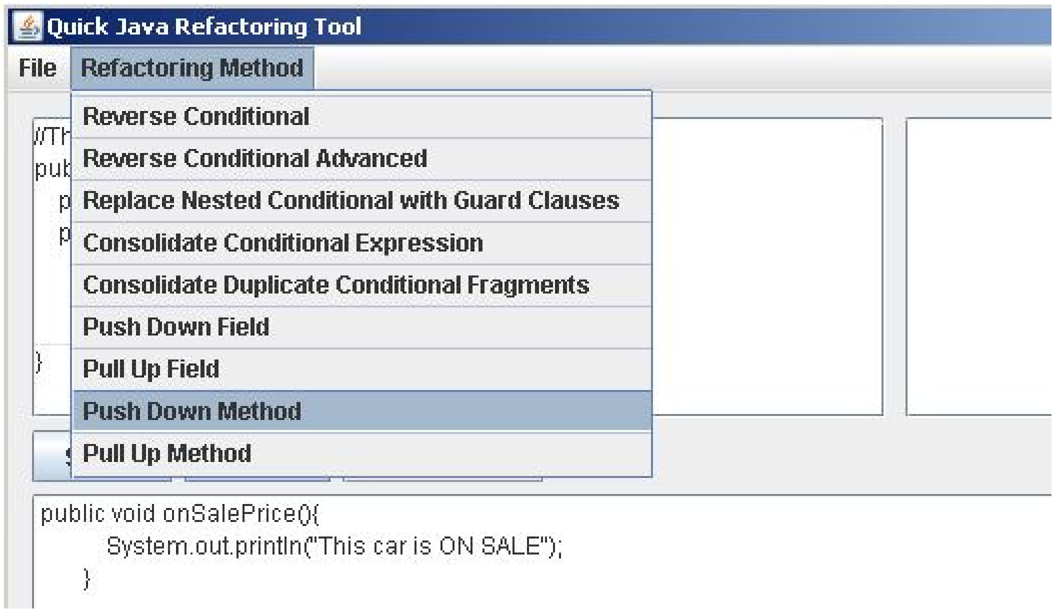
The first step is to open the source code in the panel, and highlight the entire method.

The Figure 79 is the screenshot of applying the PDM. The right side panel, as usual, shows the currently modified class. The same pop up window asks the developer to review the modified files.



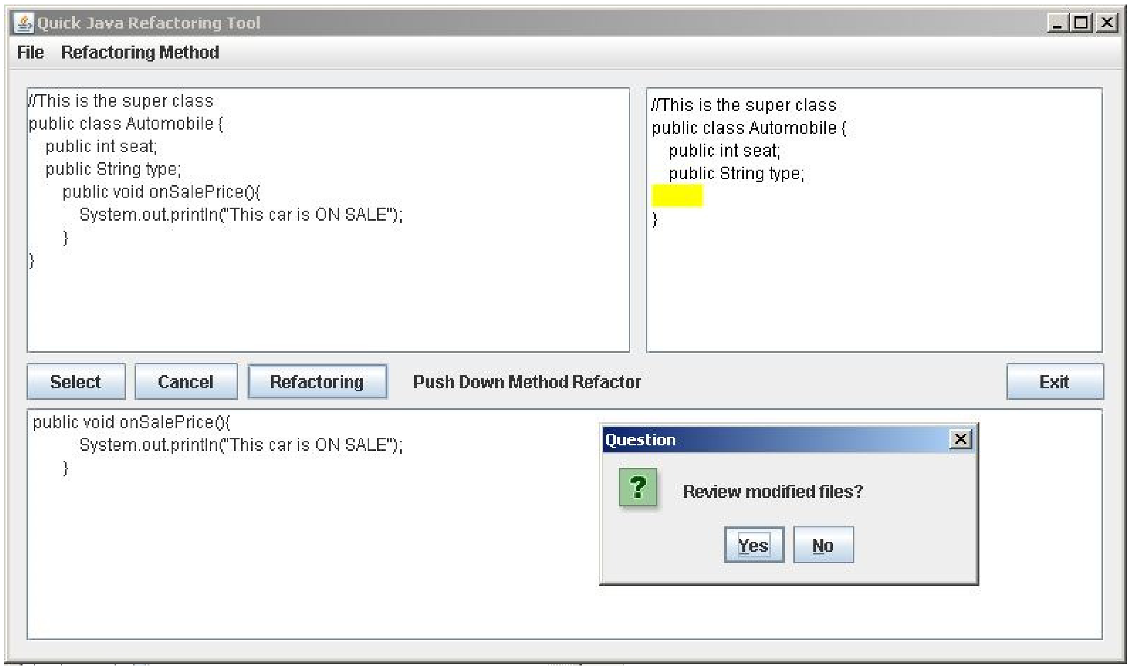
```
//This is the super class
public class Automobile {
   public int seat;
   public String type;
      public void onSalePrice(){
         System.out.println("This car is ON SALE");
      }
}
```

FIGURE 77. Highlighting the method to be pushed down.


Quick Java Refactoring Tool
File
Refactoring Method
Reverse Conditional
Reverse Conditional Advanced
Replace Nested Conditional with Guard Clauses
Consolidate Conditional Expression
Consolidate Duplicate Conditional Fragments
Push Down Field
Pull Up Field
Push Down Method
Pull Up Method

```
public void onSalePrice(){
        System.out.println("This car is ON SALE");
    }
```

FIGURE 78. Selecting PDM refactoring.



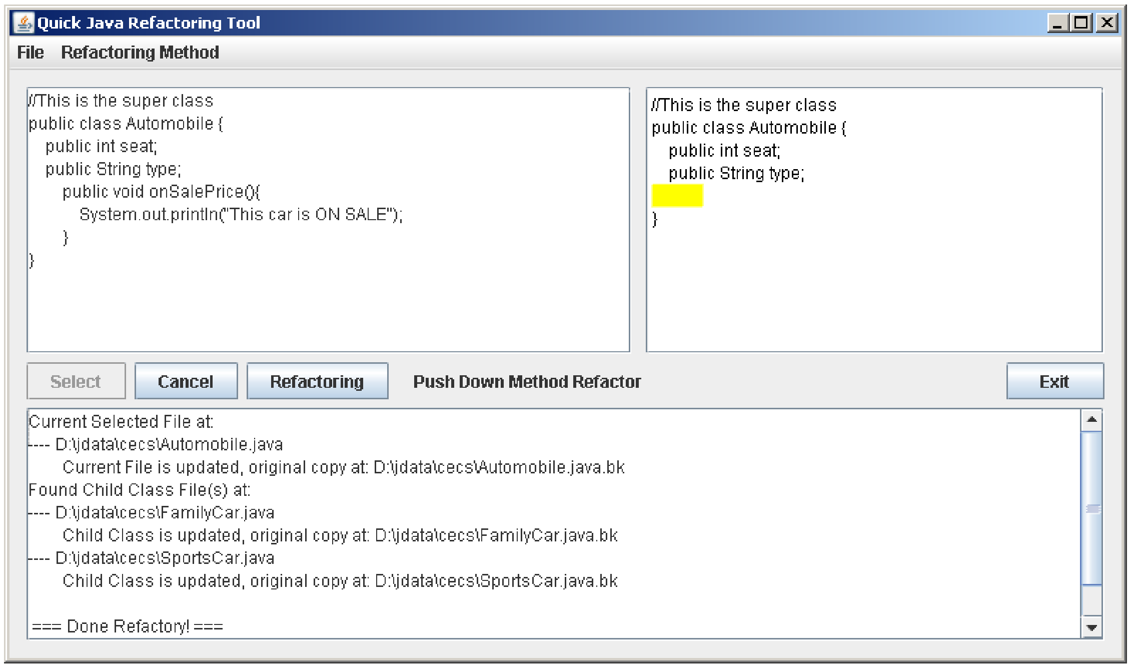
```
//This is the super class
public class Automobile {
   public int seat;
   public String type;
      public void onSalePrice(){
         System.out.println("This car is ON SALE");
      }
}
```

```
//This is the super class
public class Automobile {
   public int seat;
   public String type;

}
```

```
public void onSalePrice(){
         System.out.println("This car is ON SALE");
      }
```

Review modified files?

FIGURE 79. After applying PDM refactoring but before viewing modified files.



```
//This is the super class
public class Automobile {
   public int seat;
   public String type;
      public void onSalePrice(){
         System.out.println("This car is ON SALE");
      }
}
```

```
//This is the super class
public class Automobile {
   public int seat;
   public String type;

}
```

```
Current Selected File at:
---- D:\jdata\cecs\Automobile.java
      Current File is updated, original copy at: D:\jdata\cecs\Automobile.java.bk
Found Child Class File(s) at:
---- D:\jdata\cecs\FamilyCar.java
      Child Class is updated, original copy at: D:\jdata\cecs\FamilyCar.java.bk
---- D:\jdata\cecs\SportsCar.java
      Child Class is updated, original copy at: D:\jdata\cecs\SportsCar.java.bk

=== Done Refactory! ===
```

FIGURE 80. Screenshot after applying PDM refactoring.

The Figure 80 is the screenshot after performing refactoring and closing the pop up window. The bottom panel shows detailed information of original and modified files.

Class Diagram After Applying Refactoring

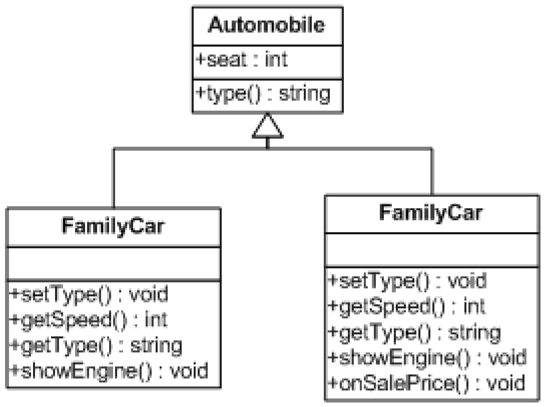


FIGURE 81. Class diagram after applying Push Down Method refactoring.

The developer needs to analyze the program before executing the modified source code. The intention is to push the onSalePrice method to SportsCar class only, but the result is that the method shows up in both subclasses. Therefore, the method in *FamilyCar.java* needs to be deleted manually. The driver class will need to reflect the changes as well. Fowler suggests compile and test [1]. That is what we will do, as well. The output of the sample code will be different from the original to reflect the difference, which is expected.

```
//This is the super class
public class Automobile {
    public int seat;
    public String type;
}
```

FIGURE 82. Source codes after applying PDM refactoring.

```
//This sub class one inherits Automobile.java
public class FamilyCar extends Automobile {
     FamilyCar() {
        seat = 7;
    }
    public void setType(String s) {
        type = s;
    }
    public String getType() {
        return type;
    }
    public int getSpeed() {
        String t = getType();
        if (t == "e") {
            return 70;
        } else if (t == "g") {
            return 180;
        } else {
            return 0;
        }
    }
    public void showEngine() {
        String s = getType();
        if (s == "g") {
            System.out.println("This is a sports car ! uses
" + s + " and max speed is " + getSpeed());
        } else if (s == "e") {
            System.out.println("This is a family car ! uses
" + s + " and max speed is " + getSpeed());
        } else {
            System.out.println("NOTHING IS HERE, WRONG");
        }
    }
}
```

```
//This sub class two inherits Automobile.java
public class SportsCar extends Automobile {
    public void onSalePrice() {
          System.out.println("This car is ON SALE");   }
     SportsCar() {
        seat = 2;
    }
```

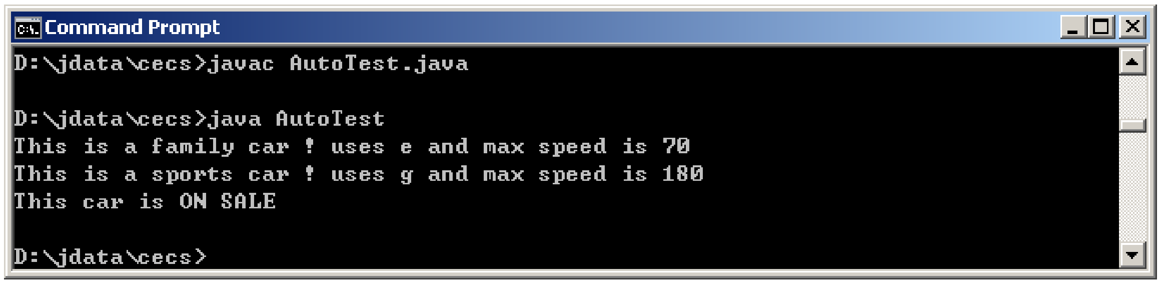
FIGURE 83. Source codes after applying PDM refactoring (continued).

```
    public void setType(String s) {
        type = s;
    }
    public String getType() {
        return type;
    }
    public int getSpeed() {
        String t = getType();
        if (t == "e") {
            return 70;
        } else if (t == "g") {
            return 180;
        } else {
            return 0;
        }
    }
    public void showEngine() {
        String s = getType();
        if (s == "g") {
            System.out.println("This is a sports car !
uses " + s + " and max speed is " + getSpeed());
        } else if (s == "e") {
            System.out.println("This is a family car !
uses " + s + " and max speed is " + getSpeed());
        } else {
            System.out.println("NOTHING IS HERE, WRONG");
        }
    }
}
```

```
//This is the test class
public class AutoTest {
    public static void main(String[] args) {
        FamilyCar fc = new FamilyCar();
        fc.setType("e");
        fc.showEngine();
//        fc.onSalePrice();

        SportsCar sc = new SportsCar();
        sc.setType("g");
        sc.showEngine();
        sc.onSalePrice();
    }
}
```

FIGURE 84. Source codes after applying PDM refactoring (continued).

Output After Applying Refactoring



```
D:\jdata\cecs>javac AutoTest.java

D:\jdata\cecs>java AutoTest
This is a family car ! uses e and max speed is 70
This is a sports car ! uses g and max speed is 180
This car is ON SALE

D:\jdata\cecs>
```

FIGURE 85. Output after applying PDM.

File Location After Applying Refactoring

The file locations are the same as Figure 74 since the same three classes are affected during the refactoring.

Summary

Push Down Method is used to move behavior from a superclass to a specific subclass. The structure of the source code is changed, and the output changes to what was intended to show in our test. The superclass has one less method to care about. By simply deleting the extra method in the subclass, the program executed just fine. In the end, the PDM makes the specific subclass highly cohesive, which is one of the features of the high quality of Object Oriented Programming.

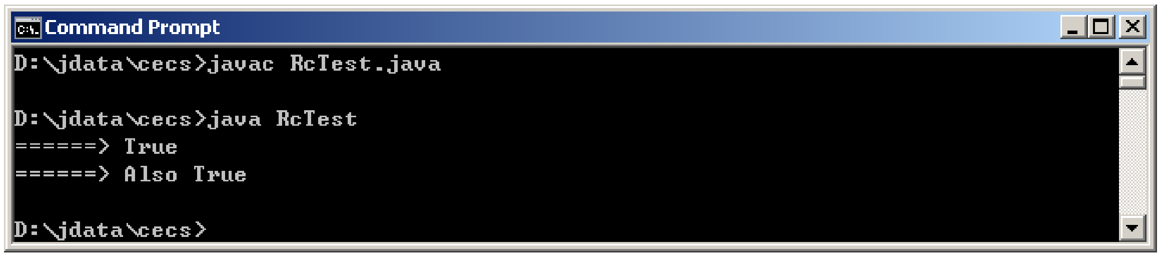
Sample Input and Validation of Reverse Conditional

Lastly, to test Reverse Conditional refactoring, a sample code used is a single Java class named “RcTest.java.” This class has a single main method with two primitive types. It includes two if-statements. The if-statement’s condition is determined by comparing each primitive’s value with a constant integer. The process is changing the conditional to the opposite, then switch thenStatement and elseStatement.

```
public class RcTest {
    public static void main(String[] args) {
        // TODO code application logic here
        int i = 1;
        int j = 2;
        if (!(i != 1)) {
            System.out.println("======> True");
        } else {
            System.out.println("You Shouldn't see me!");
        }
        if (i + j == 3) {
            System.out.println("======> Also True");
        } else {
            System.out.println("You Shouldn't see me!");
        }
    }
}
```

FIGURE 86. The sample code for RC.

<u>Output Before Applying Refactoring</u>



```
D:\jdata\cecs>javac RcTest.java

D:\jdata\cecs>java RcTest
======> True
======> Also True

D:\jdata\cecs>
```

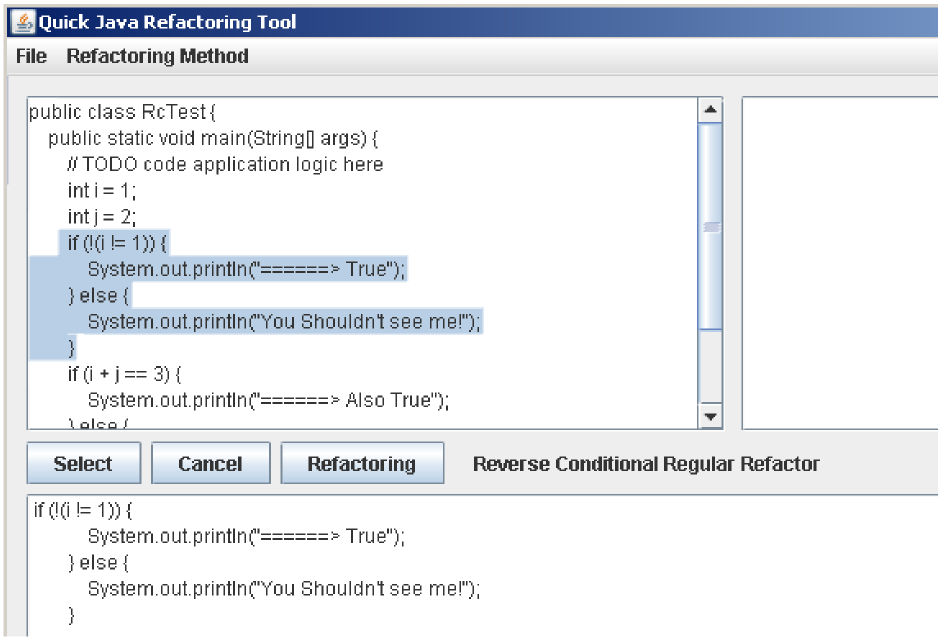
FIGURE 87. Output before applying RC refactoring.

<u>Sample Input Test 1</u>

RcTest.java is selected to open in the left panel of the tool. The whole if-else statement is selected. The new and improved if-else statement body will replace the old if-else conditional upon applying refactoring. The output will be the same before and after applying refactoring.

Refactoring Process

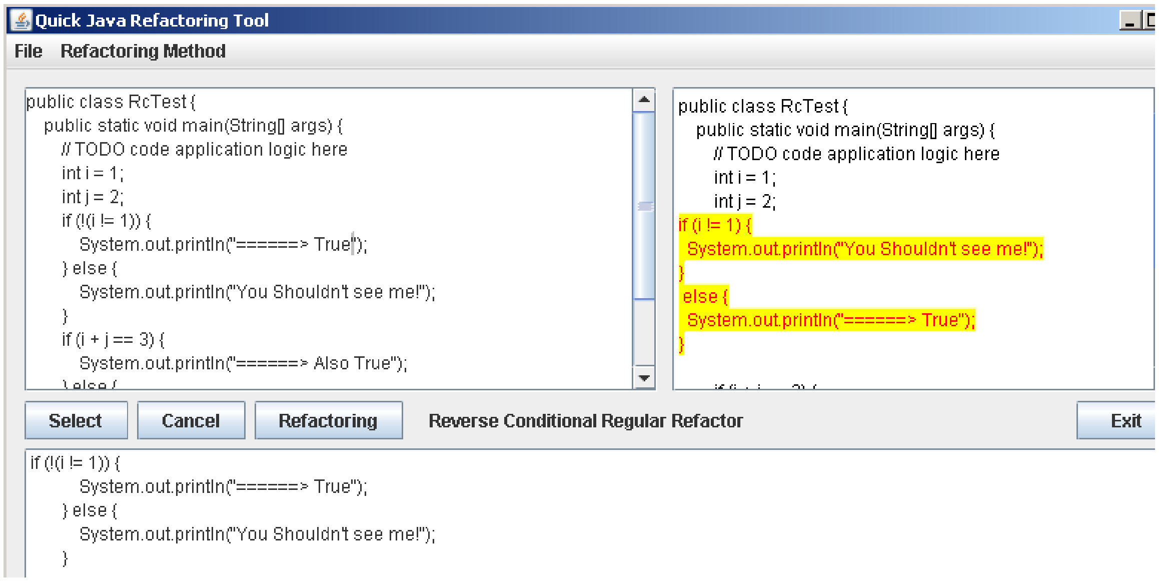
The first step is to select RcTest.java in the panel, and select the entire negative if-else conditional statement for Reverse Conditional.


Quick Java Refactoring Tool
File
Refactoring Method
Select
Cancel
Refactoring
Reverse Conditional Regular Refactor

```
public class RcTest {
  public static void main(String[] args) {
    // TODO code application logic here
    int i = 1;
    int j = 2;
    if (!(i != 1)) {
      System.out.println("======> True");
    } else {
      System.out.println("You Shouldn't see me!");
    }
    if (i + j == 3) {
      System.out.println("======> Also True");
    } else {
```

```
if (!(i != 1)) {
      System.out.println("======> True");
    } else {
      System.out.println("You Shouldn't see me!");
    }
```

FIGURE 88. Highlighting the entire if-else for refactoring.


Quick Java Refactoring Tool
File
Refactoring Method
Select
Cancel
Refactoring
Reverse Conditional Regular Refactor
Exit

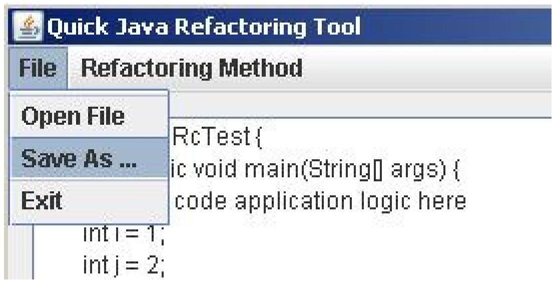
```
public class RcTest {
  public static void main(String[] args) {
    // TODO code application logic here
    int i = 1;
    int j = 2;
    if (!(i != 1)) {
      System.out.println("======> True");
    } else {
      System.out.println("You Shouldn't see me!");
    }
    if (i + j == 3) {
      System.out.println("======> Also True");
    } else {
```

```
public class RcTest {
  public static void main(String[] args) {
    // TODO code application logic here
    int i = 1;
    int j = 2;
if (i != 1) {
 System.out.println("You Shouldn't see me!");
}
 else {
 System.out.println("======> True");
}
```

```
if (!(i != 1)) {
      System.out.println("======> True");
    } else {
      System.out.println("You Shouldn't see me!");
    }
```

FIGURE 89. After applying RC refactoring.

The Figure 89 on the right side panel shows the refactored source code. QJRT has the option to save the modified file, if desired. That implies the original file is not saved automatically like in previous refactorings. Let's save the modified file as "RcTestDone.java," and edit the class name accordingly.



```
RcTest {
ic void main(String[] args) {
code application logic here
    int i = 1;
    int j = 2;
```

FIGURE 90. Saving option in QJRT.

Output After Applying Refactoring

The output of the sample code is the same as the output listed in Figure 87. Both outputs, before and after applying refactoring, are identical.

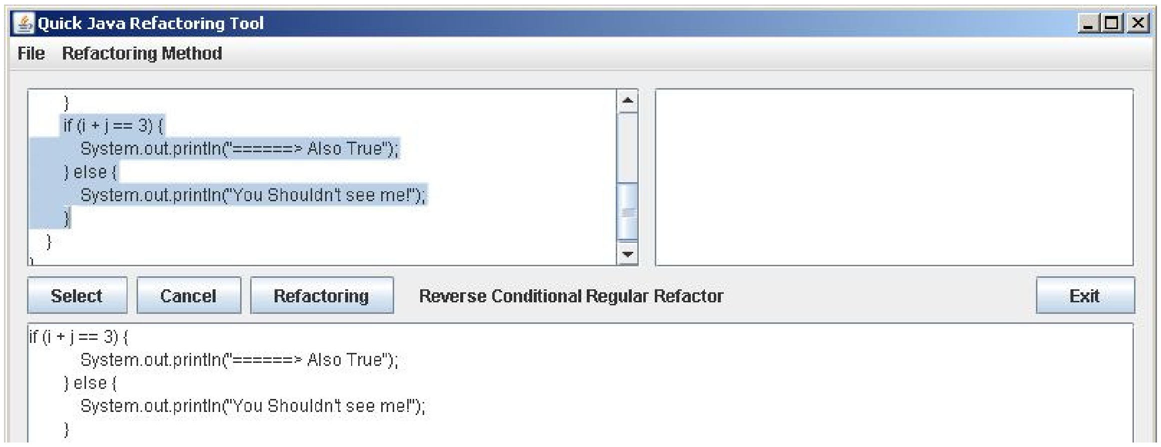
Sample Input Test 2

This input will present if a non-negative if-condition is selected, and QJRT will report error. The same source file is used, but the positive if-else conditional statement is selected to test.

Refactoring Process – with Reporting Error

The first step is to open the source code in the panel, and select the entire positive if-else statement for Reverse Conditional.

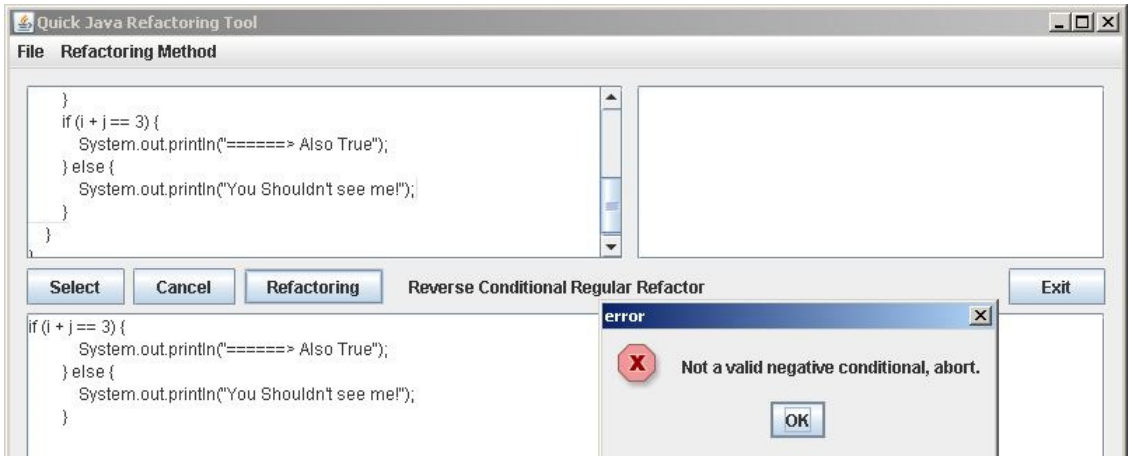
The Figure 92 shows the error message when the selected if-statement's condition is not initialized negative.


Quick Java Refactoring Tool
File
Refactoring Method
Select
Cancel
Refactoring
Reverse Conditional Regular Refactor
Exit

```
    }
    if (i + j == 3) {
      System.out.println("======> Also True");
    } else {
      System.out.println("You Shouldn't see me!");
    }
  }
```

```
if (i + j == 3) {
      System.out.println("======> Also True");
    } else {
      System.out.println("You Shouldn't see me!");
    }
```

FIGURE 91. Highlighting the entire if-else for refactoring.


Quick Java Refactoring Tool
File
Refactoring Method
Select
Cancel
Refactoring
Reverse Conditional Regular Refactor
Exit
error
OK

```
    }
    if (i + j == 3) {
      System.out.println("======> Also True");
    } else {
      System.out.println("You Shouldn't see me!");
    }
  }
```

```
if (i + j == 3) {
      System.out.println("======> Also True");
    } else {
      System.out.println("You Shouldn't see me!");
    }
```

Not a valid negative conditional, abort.

FIGURE 92. After clicking RC refactoring.

Summary

Reverse Conditional is used to make the conditional meaningful. A developer intends to arrange the conditional negatively, but actually the result of RC may be better in most cases. Therefore, this refactoring makes source code easier to understand and better readability. Moreover, this refactoring restricts changing from positive to negative conditional.

Sample Input and Validation of Reverse Conditional Advance

To test Reverse Conditional Advance refactoring, a source code with varied if-else statements is used. The conditional expressions are: parenthesized, Instanceof, and infix expressions. In addition, the infix expression includes less, greater, less and equals, greater and equals, equals, not equals, and others. Each if-else is selected to refactor individually.

Sample Input

RcaTest.java is selected to open in the left panel of the tool.

```
public class RcaTest{
    public static void main(String[] args) {
        // TODO code application logic here
        int i = 1;
        boolean a = true;
        boolean b = false;
        String s = "";
//Parenthesized Expression is following:
        if ((a)) {
            System.out.println("======> True");
        } else {
            System.out.println("You shouldn't see me.");
        }

//Instanceof Expression is following:
        if (s instanceof String) {
            System.out.println("======> True");
        } else {
            System.out.println("You shouldn't see me.");
        }
//infixExpression (other) is following:
        if ((a && a)) {
            System.out.println("======> True");
        } else {
            System.out.println("You shouldn't see me.");
        }
```

FIGURE 93. Source code to be tested on RCA.

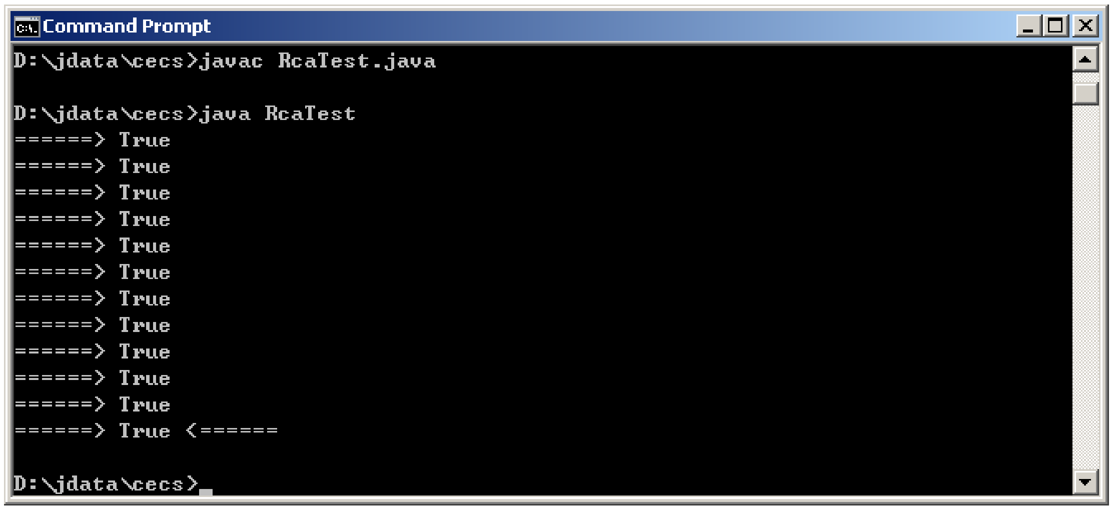
```
        if ((a & b) | (a & !b)) {
            System.out.println("======> True");
        } else {
            System.out.println("You shouldn't see me.");
        }
        if ((!a & b) || (a & !b)) {
            System.out.println("======> True");
        } else {
            System.out.println("You shouldn't see me.");
        }
        if (b && !b || a | b || b) {
            System.out.println("======> True");
        } else {
            System.out.println("You shouldn't see me.");
        }

//InfixExpression ( Less Expression ) is following:
        if (i < 2) {
            System.out.println("======> True");
        } else {
            System.out.println("You shouldn't see me.");
        }

//InfixExpression ( Greater Expression )is following:
        if (i > 0) {
            System.out.println("======> True");
        } else {
            System.out.println("You shouldn't see me.");
        }

//InfixExpression (Less & Equals Expression) is following:
        if (i <= 1) {
            System.out.println("======> True");
        } else {
            System.out.println("You shouldn't see me.");
        }
//InfixExpression(Greater & Equals Expression)is following:
        if (i >= 1) {
            System.out.println("======> True");
        } else {
            System.out.println("You shouldn't see me.");
        }
```

FIGURE 94. Source code to be tested on RCA (continued).

```
//InfixExpression ( Equals Expression) is following:
        if (i == 1) {
            System.out.println("======> True");
        } else {
            System.out.println("You shouldn't see me.");
        }

//InfixExpression ( NotEquals Expression ) is following:
        if (i != 0) {
            System.out.println("======> True <======");
        } else {
            System.out.println("You shouldn't see me.");
        }
    }
}
```

FIGURE 95. Source code to be tested on RCA (continued).

Output Before Applying Refactoring

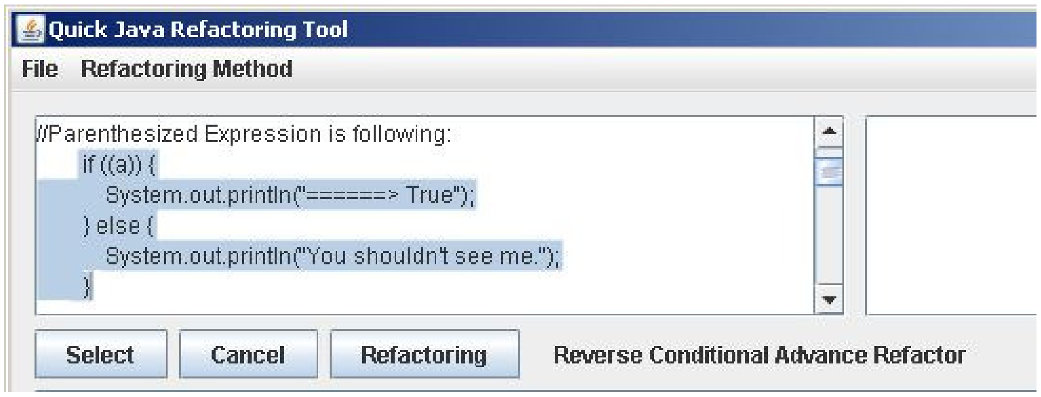


```
D:\jdata\cecs>javac RcaTest.java

D:\jdata\cecs>java RcaTest
======> True
======> True
======> True
======> True
======> True
======> True
======> True
======> True
======> True
======> True
======> True
======> True <======

D:\jdata\cecs>
```

FIGURE 96. Output before applying refactoring.

Refactoring Process and Result

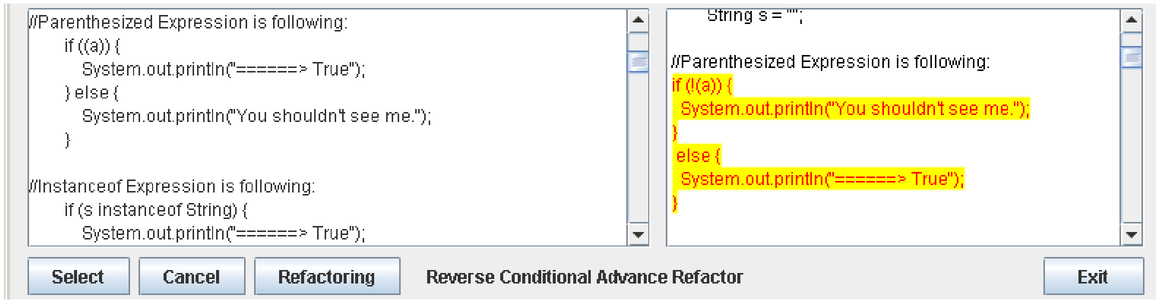
Open the source code in the panel, and then highlight the entire first if-statement to test the parenthesized expression.


Quick Java Refactoring Tool
File
Refactoring Method
Select
Cancel
Refactoring
Reverse Conditional Advance Refactor

```
//Parenthesized Expression is following:
    if ((a)) {
        System.out.println("======> True");
    } else {
        System.out.println("You shouldn't see me.");
    }
```

FIGURE 97. Selecting source code and selecting the refactoring method.

<u>Result After Applying Refactoring</u>

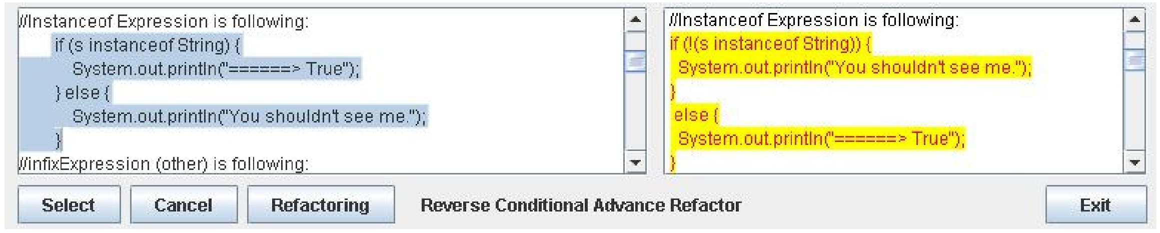


```
//Parenthesized Expression is following:
    if ((a)) {
        System.out.println("======> True");
    } else {
        System.out.println("You shouldn't see me.");
    }

//Instanceof Expression is following:
    if (s instanceof String) {
        System.out.println("======> True");
```

```
    String s = "";

//Parenthesized Expression is following:
if (!(a)) {
 System.out.println("You shouldn't see me.");
}
else {
 System.out.println("======> True");
}
```

FIGURE 98. Screenshot after applying RCA.

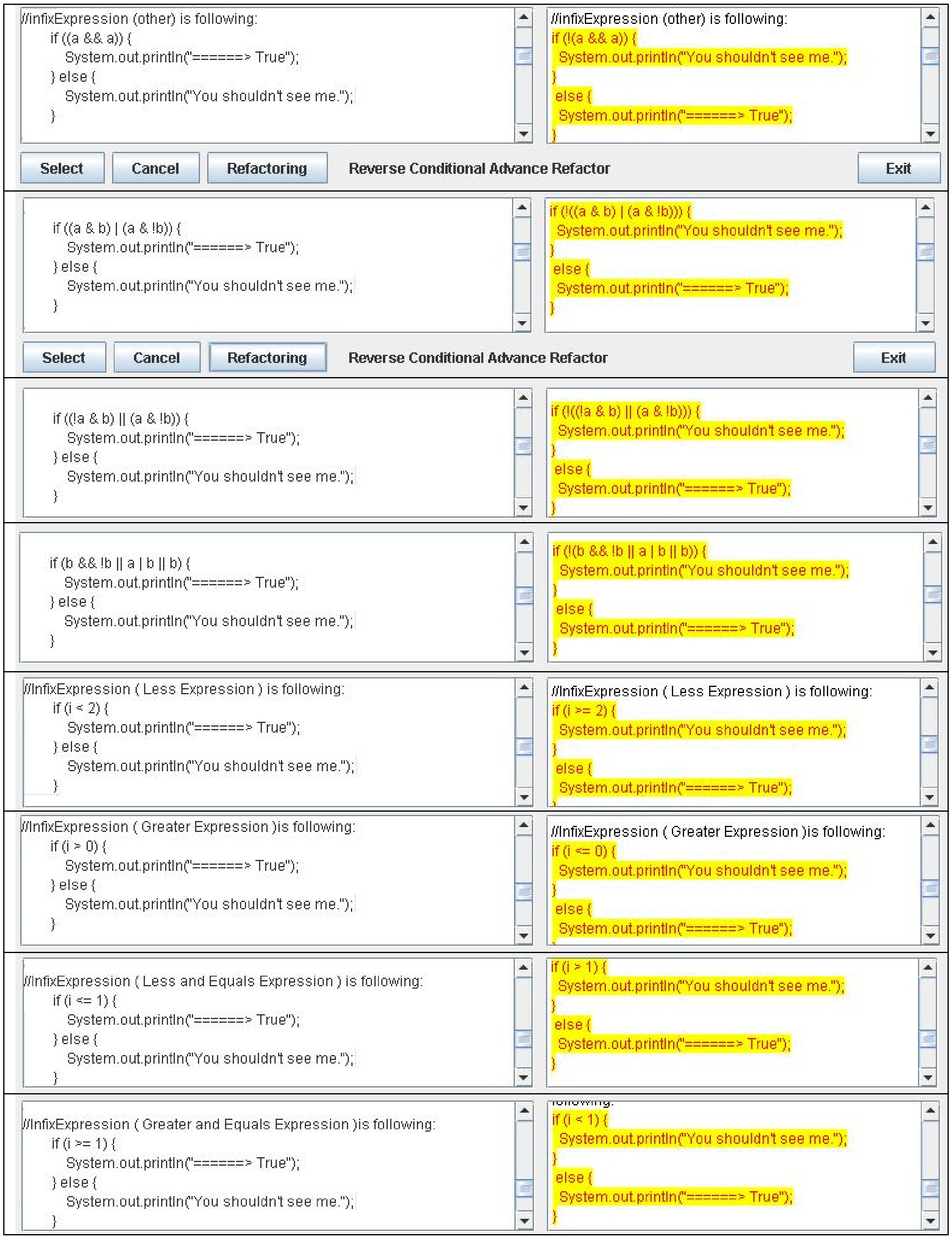
The same procedure should be done for other expressions with the same source code. The results are following:



```
//Instanceof Expression is following:
    if (s instanceof String) {
        System.out.println("======> True");
    } else {
        System.out.println("You shouldn't see me.");
    }
//infixExpression (other) is following:
```

```
//Instanceof Expression is following:
if (!(s instanceof String)) {
 System.out.println("You shouldn't see me.");
}
else {
 System.out.println("======> True");
}
```

FIGURE 99. Screenshots after applying RCA for all conditional.



```
//infixExpression (other) is following:
     if ((a && a)) {
        System.out.println("======> True");
     } else {
        System.out.println("You shouldn't see me.");
     }
```

```
//infixExpression (other) is following:
if (!(a && a)) {
 System.out.println("You shouldn't see me.");
}
 else {
 System.out.println("======> True");
}
```

```
     if ((a & b) | (a & !b)) {
        System.out.println("======> True");
     } else {
        System.out.println("You shouldn't see me.");
     }
```

```
if (!((a & b) | (a & !b))) {
 System.out.println("You shouldn't see me.");
}
 else {
 System.out.println("======> True");
}
```

```
     if ((!a & b) || (a & !b)) {
        System.out.println("======> True");
     } else {
        System.out.println("You shouldn't see me.");
     }
```

```
if (!((!a & b) || (a & !b))) {
 System.out.println("You shouldn't see me.");
}
 else {
 System.out.println("======> True");
}
```

```
     if (b && !b || a | b || b) {
        System.out.println("======> True");
     } else {
        System.out.println("You shouldn't see me.");
     }
```

```
if (!(b && !b || a | b || b)) {
 System.out.println("You shouldn't see me.");
}
 else {
 System.out.println("======> True");
}
```

```
//InfixExpression ( Less Expression ) is following:
     if (i < 2) {
        System.out.println("======> True");
     } else {
        System.out.println("You shouldn't see me.");
     }
```

```
//InfixExpression ( Less Expression ) is following:
if (i >= 2) {
 System.out.println("You shouldn't see me.");
}
 else {
 System.out.println("======> True");
```

```
//InfixExpression ( Greater Expression )is following:
     if (i > 0) {
        System.out.println("======> True");
     } else {
        System.out.println("You shouldn't see me.");
     }
```

```
//InfixExpression ( Greater Expression )is following:
if (i <= 0) {
 System.out.println("You shouldn't see me.");
}
 else {
 System.out.println("======> True");
```

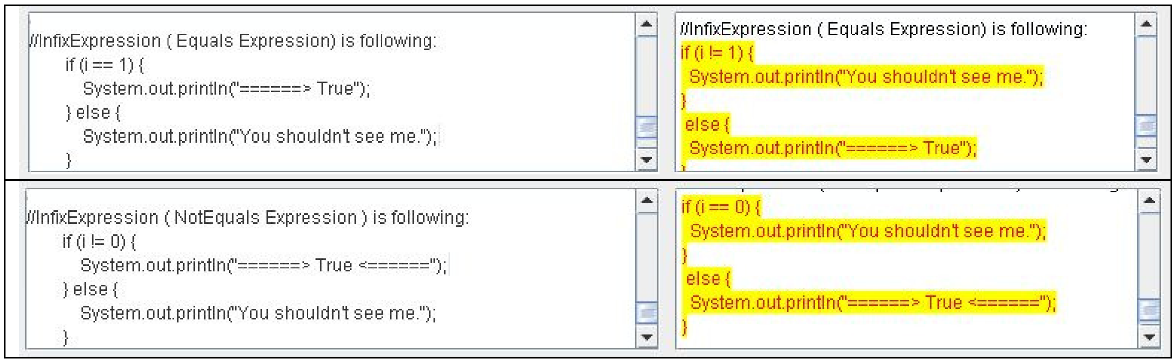
```
//InfixExpression ( Less and Equals Expression ) is following:
     if (i <= 1) {
        System.out.println("======> True");
     } else {
        System.out.println("You shouldn't see me.");
     }
```

```
if (i > 1) {
 System.out.println("You shouldn't see me.");
}
 else {
 System.out.println("======> True");
}
```

```
//InfixExpression ( Greater and Equals Expression )is following:
     if (i >= 1) {
        System.out.println("======> True");
     } else {
        System.out.println("You shouldn't see me.");
     }
```

```
if (i < 1) {
 System.out.println("You shouldn't see me.");
}
 else {
 System.out.println("======> True");
}
```

FIGURE 100. Screenshots after applying RCA for all conditional (continued).



```
//InfixExpression ( Equals Expression) is following:
     if (i == 1) {
        System.out.println("======> True");
     } else {
        System.out.println("You shouldn't see me.");
     }
```

```
//InfixExpression ( Equals Expression) is following:
if (i != 1) {
 System.out.println("You shouldn't see me.");
}
 else {
 System.out.println("======> True");
```

```
//InfixExpression ( NotEquals Expression ) is following:
     if (i != 0) {
        System.out.println("======> True <======");
     } else {
        System.out.println("You shouldn't see me.");
     }
```

```
if (i == 0) {
 System.out.println("You shouldn't see me.");
}
 else {
 System.out.println("======> True <======");
}
```

FIGURE 101. Screenshots after applying RCA for all conditional (continued).

<u>Output After Applying Refactoring</u>

The output of the sample code is the same as the output listed in Figure 96. Both outputs, before and after applying refactoring, are identical.

<u>Summary</u>

Reverse Conditional Advance is an expanded version of Reverse Conditional refactoring. The structure of the source code and the output after applying refactoring stays the same.

CHAPTER 9

CONCLUSION

A total of nine refactorings have been implemented within this thesis. Seven of these refactoring have been published in Fowler's book [1] and the other two have been mentioned on Fowler's website [4]. Eclipse AST is used in QJRT for parsing the source code. Minimal user interaction is required when using QJRT.

The stepwise refactoring implementation gives users a straightforward view. The original and refactored source codes are side by side in most cases, and if default notepad++ is not installed on the system, or users are not using windows based system, QJRT gives users the source files' location in absolute path.

Eliminating inputs from keyboards helps reducing human error; at last, refactoring is part of software maintenance, not to introduce another human error is essential.

CHAPTER 10

FUTURE WORKS

QJRT requires minimum user interface in order to refactor. The User Interface (UI) is very plain sighted; developer doesn't need to spend much time to be able to use it fluently. QJRT's portability allows it to be downloaded and any code to be refactored on the fly. In addition, the program does clean up the source code for better readability, and improves the source code's structure. More importantly, the output is not impacted.

In this thesis, I have simplified one refactoring: *replace nested conditional with guard clause*, it only deals in if-else conditional statement level, but truly it can be expanded to method call level.

QJRT is implemented only one refactoring at a time; however, some complicated refactorings requires multiple other refactorings work together to achieve the goal. QJRT is not implemented with automatic detection of bad smells, nor design patterns. This thesis can be extended to do so. When a similar refactoring may be use, or a bad smell is detected, QJRT could suggest the appropriate refactorings.

REFERENCES

REFERENCES

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[2] M. Arora, S. S.S., V. Rathore, J. Deegwal, S. Arora, “Refactoring, Way for Software Maintenance.” *International Journal of Computer Science Issues,* vol. 8, Mar. 2011, pp. 565-569.

[3] P. Williamson, “Refactor Your Code Over Time,” Oct. 6, 2012; http://pippins plugins.com/refactor-your-code-over-time/

[4] O. Whiler, “Refactoring,” *Methods & Tools,* Spring 2003, pp.22-25.

[5] M. Fowler, “Refactoring Home,” www.refactoring.com

[6] Cunningham & Cunningham, Inc. "History Of Refactoring." Nov. 2008; http://c2.com/cgi/wiki?HistoryOfRefactoring

[7] S. Counsell and S. Swift, "Refactoring steps, Java Refactorings and Empirical Evidence," *Computer Software and Applications,* COMPSAC 08, IEEE Int'l 32nd Ann., 2008, pp. 176-179.

[8] Eclipse.org, “Home Page,” http://www.eclipse.org

[9] B. Marchal, "Working XML: Take advantage of lessons learned by refactoring XM," 2004; http://www.ibm.com/developerworks/library/x-wxxm28/

[10] D. Gallardo, "Refactoring for everyone," 2004; http://www.ibm.com/ developerworks/library/os-ecref/

[11] ANTLR Parser Generator, “ANTLR;” http://www.antlr.org

[12] GOLD Parser, “GOLD Parsing System;” http://goldparser.org/

[13] D. Dig, JavaRefactor. 2002; http://plugins.jedit.org/plugins/?JavaRefactor

[14] “JRefactory;” http://jrefactory.sourceforge.net/

[15] Z. Tronicek, RefactoringNG. 2013; http://kenai.com/projects/refactoringng/

[16] Refactor-J, “Sixth & Red River: Refactor-J;” http://www.sixthandredriver.com /refactor-j.html

[17] IntelliJ IDEA, “the most intelligent Java IDE;” http://www.jetbrains.com/idea/

[18] JFactor, “jFactor: Java Glossary,” 2013; http://mindprod.com/jgloss/jfactor.html