



Bachelor Thesis

Highlevel Tracing of Code Modifications in Eclipse

im Studiengang Informatik
des Fachbereichs Mathematik und Informatik
der Philipps-Universität Marburg

vorgelegt von:

Marco Christmann
Matrikelnummer: 2564521

Betreuer: Prof. Dr. Christoph Bockisch

Contents

1	Introduction	1
2	Logging code modifications in Eclipse	3
2.1	Approach to log the code modification changes	5
2.2	Iteration of the steps for a refactoring	8
2.3	Iteration of the steps for an other code modification	15
3	Elaboration of the results	18
3.1	Testing the plug-in Retestoring	18
3.2	Critical discussion of completeness	19
4	Conclusion	21
4.1	Summary	21
4.2	Result	21

1 Introduction

Nowadays big projects in the development of software systems are realized by spreading sub projects all over the world and assembling the results whenever they are completed. The development of software systems is a lengthy process, once-written code will often be reused, maintained, modified and optimized. Because this is done by different people it is necessary that the code is well readable, understandable and clearly structured. By using higher level programming languages like Java, which give programmers the opportunity to write well readable and understandable code, programmers should do so to benefit from it.

Unfortunately, it is often not the rule to write well readable and clearly structured code. For the majority of developers creating „good“ code is a waste of time because making functioning code more readable does not give additional functions or improvements in speed. The process of making code more readable consumes a lot of time. But it is worth it because roughly 60 percent of time of the development is spent on maintenance [4]. While it is clear for the writer what his code does, someone else might not comprehend the code. At this point the development environments help out and provide some features called „refactorings“ [5]. Refactorings are shortcuts to modify written code in order to make it more readable and structured. Or more precisely „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“ [5].

However, there can occur problems using these refactorings. For example, when the software development is test-driven [6], the developers have to create test cases for each method before they develop the code of the method. When the code is ready and functional the time of maintenance, especially refactorings, begins. Though, code changes by refactorings do not consider the existing test cases. For instance parts of existing methods can be extracted and from these parts new methods are created. The programmer is not notified that there are new methods for which no test case exist. Furthermore, the accessibility from methods or variables can be changed from private to public or vice versa. In the case that a private method is changed to public, the programmer has to create a new test case for this method and that is not trivial. Therefore, it is necessary that new test cases

are developed or written test cases are modified to cover the changes [8]. The consistency between refactored code and test cases can not be obtained automatically and this is still an unsolved problem [9].

A goal is to automatically create an advice for a developer on how to update the test cases after code modifications. In the further course of this paper there will be taken a closer look at Eclipse, which is the most common development environment for developing Java code [7]. For Eclipse there is already one approach on the problem mentioned above, the plug-in *Retesting*. This approach tracks changes applied to the source code during refactorings and analyzes which parts of the API are impacted in which way and how the changes in the program structure should be reflected in the structure of the program's test cases [1].

But this plug-in is not complete and does not intercept all possible refactorings or could contain gaps in the covered cases. That is mainly because there is no documentation how Eclipse manages the provided refactorings in the background. This is why a basis is needed in which every provided possibility of code modifications is recorded and the changes are logged. Furthermore, on that basis new and existing plug-ins can be tested and improved to give complete support.

Besides, in order to enable a similar testing of the plug-ins it is necessary to make the changes that a code modification does reproducible. The first step is to put together a complete listing of all possible code modifications especially the refactorings provided by Eclipse. The second step is to record the changes that each modification makes; the starting situation and the situation after a modification is needed to compare them.

2 Logging code modifications in Eclipse

Before going into creating a basis of what changes a code modification does there must be taken a close look at which modifications are available in Eclipse. In order to get every possible modification all menu bar entries and right click options in Eclipse were checked whether they actually change the code. With the help of this scheme 43 relevant modifications were found. These 43 code modification cases can be grouped into the three following categories:

- Refactoring

Refactoring contains 23 modifications and are mainly used to make written code more readable and improve the structure.

- Manually

The category Manually contains 11 modifications. They are mainly used in the code development progress and are mostly applied by hand.

- Generalized

Generalized contains 9 modifications which are also mainly used in the code development progress. They help the developer to implement commonly used standards, like getters and setters, more quickly.

The complete list of all code modifications and their allocation to the three categories can be seen in Figure 1. Martin Fowler listed 91 refactorings in his book „Refactoring: Improving the Design of Existing Code“ [10] which is a common standard for developers. Eclipse only provides 21 refactorings from this, but it provides six other refactorings. It was noticed that there are some cases which correspond to more than one case in the other list. This is because the list of refactorings from Fowler is more detailed and more general. Additionally there are a few cases where it is not exactly clear whether they correspond or not, due to different naming. An Excel sheet where the two lists are faced and the overlap cases are marked will be provided in the electronic attachment.

Refactorings	Manual	Generalized
Refactor	File	Source
Rename	Move	Override/Implement Methods
Move	Rename	Generate Getters and Setters
Change Method Signature		Generate Delegate Methods
Extract Method	Edit	Generate toString()
Extract Local Variable	Undo	Generate hashCode() and equals()
Extract Constant	Redo	Generate Constructor using Fields
Inline	Cut	Generate Constructors from Superclass
Convert Local Variable to Field	Paste	Surround With
Convert Anonymous Class to Nested	Delete	Externalize Strings
Move Type to New File	Find/Replace	
Extract Superclass	Word Completion	
Extract Interface	Quick Fix	
Use Supertype Where Possible		
Push Down	Search	
Pull Up	File -> Replace	
Extract Class		
Introduce Parameter Object		
Introduce Indirection		
Introduce Factory		
Introduce Parameter		
Encapsulate Field		
Generalize Declared Type		
Infer Generic Type Arguments		

Figure 1: Categories of code modifications

For the in Figure 1 listed code modifications a project in Eclipse was created that covers all the cases. In order to log the changes the code modifications do and also to make them reproducible this project was pushed into a repository hosted on github [11]. This repository is public and can be accessed on the following url:

- <https://github.com/marcochristmann/codemodifications>

Additionally, a copy of the repository and all lists, scripts, files and screenshots will be attached to this paper in electronically form.

2.1 Approach to log the code modification changes

In order to log the changes made by the category Refactoring a scheme with 8 steps was created. For the categories Manually and Generalized the scheme is slightly different and will be described afterwards.

The steps for every single code modification case for the Refactorings are as follows:

1. Starting situation

The starting situation for every modification case is always the same. Therefore, the changes are reverted after each pass and are marked in the repository with „reset“ commits. For a better overview screenshots are only taken of the relevant sections of the code where the modifications happen.

2. Marking and settings

To reproduce the code modifications screenshots of the code where the marked part is shown will be taken. If the current modification provides a window with settings for this modification it will also be visible in the screenshot. The settings will be left the standard settings which are set because this will cover the most common cases. This will not capture every single possible case, but with this guidance readjusting the missing cases will be feasible.

3. State after modification

After the code modifications are executed screenshots are taken of the relevant sections of the code. This will make it easier to see all changes at a glance by comparing it to the screenshot taken in step 1.

4. History

Eclipse provides a build in function called „History“. It is available by clicking „Refactor“ in the menu bar and then on „History“. In the History all executed refactorings in the workspace are listed. The entries show the date, the time, the name of the refactoring and the marked part of code. For each entry it displays some more details: the type of the marked code, the fully qualified name, the name of the project where the refactoring took place, the original element and the chosen settings.

5. Script

Eclipse provides a build in function called „Create Script“. It is available by clicking „Refactor“ in the menu bar and then on „Create Script“. This function creates a script file for executed refactorings listed in the History (see step 4). There can be chosen one or more entries to create a new script file or even merge the newest changes to an existing script. However, these script files can be executed by the Eclipse function „Apply Script“. Once script file for every separate case will be created. With the script files it is very easy to reproduce every code modification case in the exact same way.

6. Github

Recording all changes made by the code modification by hand would be very complex and error-prone. In order to make this automatically the online platform github, which hosts repositories, was used.

a) Commit

After the code modifications are executed a commit with the name of the code modification is made. The current state of the code is pushed to github to get access to the provided features.

b) Changes

For every commit github displays the changes since the last commit and also the names of the changed files. Changes are highlighted in the following colors, red for deleted code and green for new code. In addition, deleted code lines are marked with a leading „-“ and new code lines with a leading „+“. Unchanged code is not highlighted. Because it makes it very comfortable to determine all changes at a glance screenshots of this will also be taken.

c) Diff-file

Using the git command „git diff“ all the changes are stored in a text file, called „diff-file“. It contains the file name of the changed source code files and lists every change below. Changes are marked with a leading „-“ for

deleted code lines and leading „+“ for new code lines. Unchanged code is not marked.

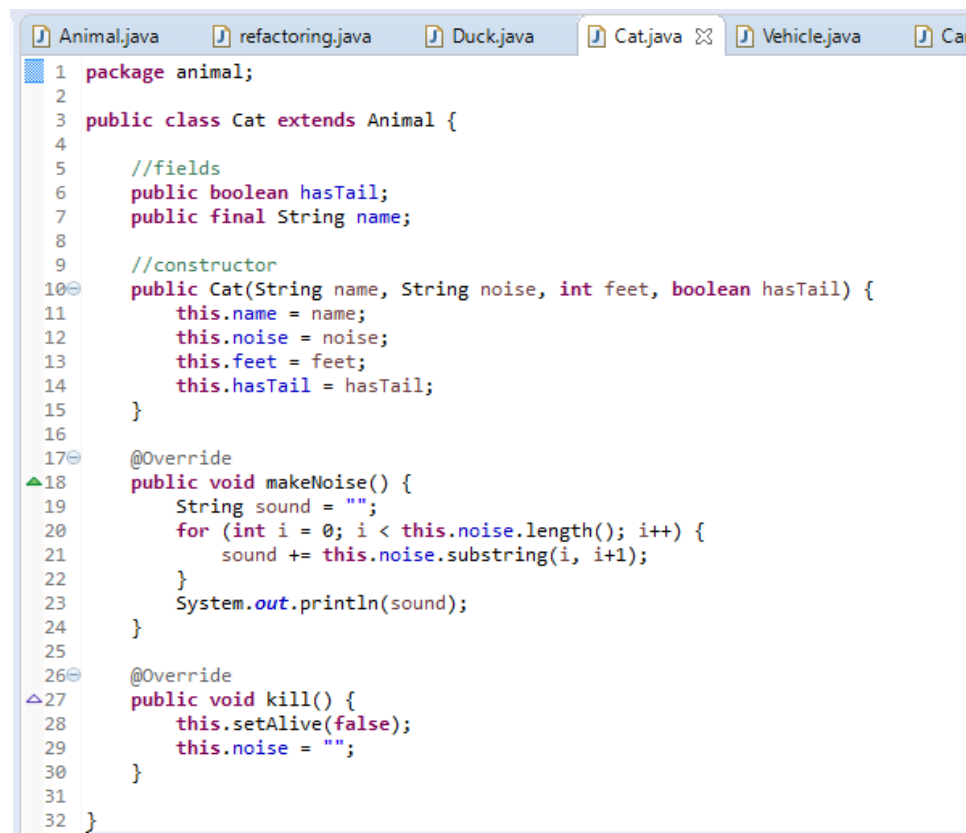
For the two remaining categories Manually and Generalized step 4 (History) and 5 (Script) are not available because they only work with the refactoring menu entries. In order to make them reproducible a text file with precise instructions was created. Each step that was performed is listed in this text file.

By providing the repository, all screenshots, files and scripts, it will be possible for plug-in developers to use this as a basis to test their plug-ins whether they are complete and can detect all changes. By using the starting situation and executing the provided scripts the changes will be easily reproducible and are the same for everyone.

Afterwards, the steps are demonstrated in a simple example of each category. The first one will be the category Refactoring and the second the categories Manual and Others are combined because the steps do not differ.

2.2 Iteration of the steps for a refactoring

As a simple example of the category refactorings the refactoring called „Rename“ was chosen. The rename refactoring can change the name of classes, local variables, fields, methods, constants, packages or projects. Eclipse gives one setting to choose whether to update references or not. The standard setting is to update references. This will change the selected part and all corresponding occurrences.



```
1 package animal;
2
3 public class Cat extends Animal {
4
5     //fields
6     public boolean hasTail;
7     public final String name;
8
9     //constructor
10    public Cat(String name, String noise, int feet, boolean hasTail) {
11        this.name = name;
12        this.noise = noise;
13        this.feet = feet;
14        this.hasTail = hasTail;
15    }
16
17    @Override
18    public void makeNoise() {
19        String sound = "";
20        for (int i = 0; i < this.noise.length(); i++) {
21            sound += this.noise.substring(i, i+1);
22        }
23        System.out.println(sound);
24    }
25
26    @Override
27    public void kill() {
28        this.setAlive(false);
29        this.noise = "";
30    }
31
32 }
```

Figure 2: Starting situation

Figure 2 shows the starting situation. The name of the String *sound* in line 19 will be changed to *shout*.

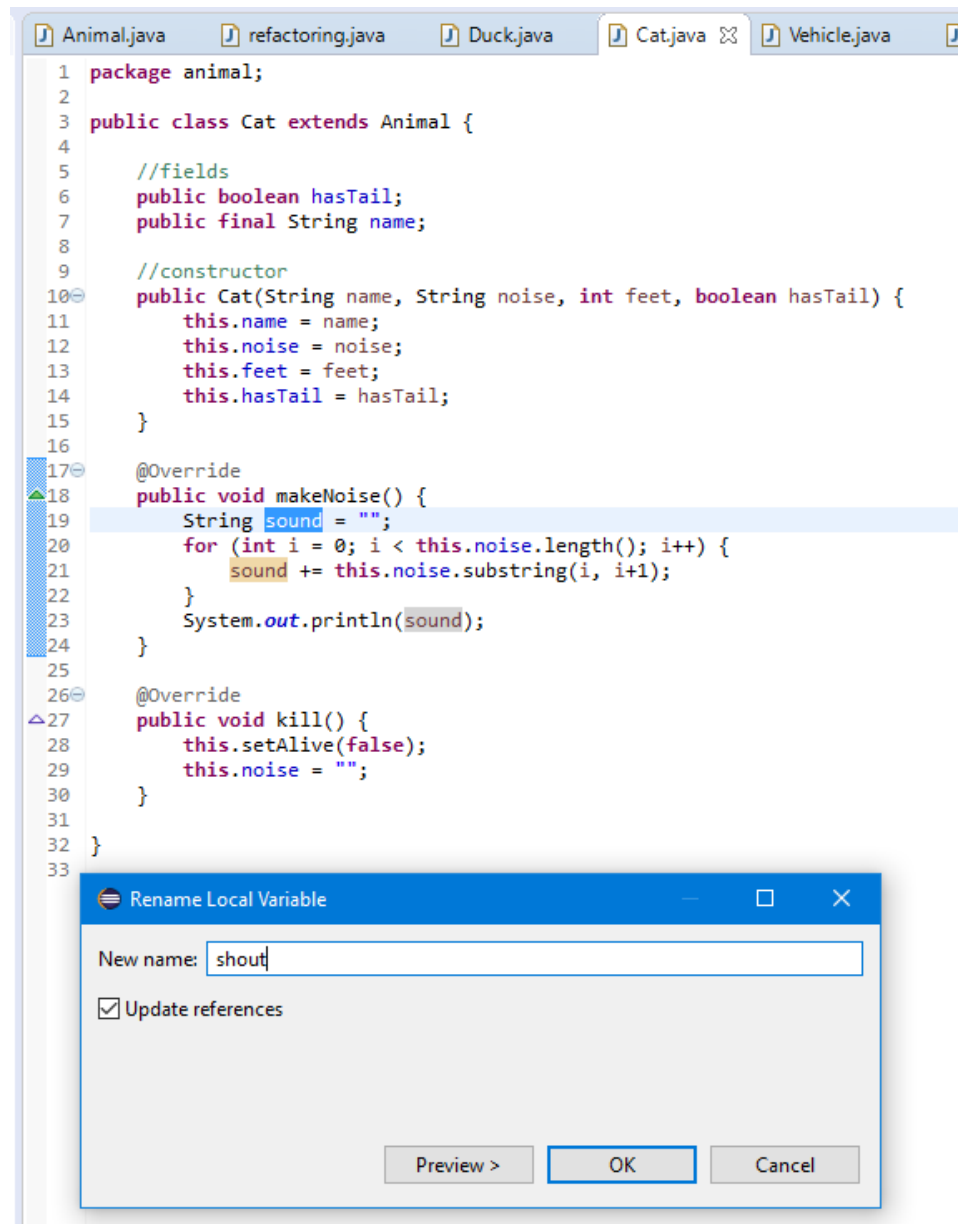
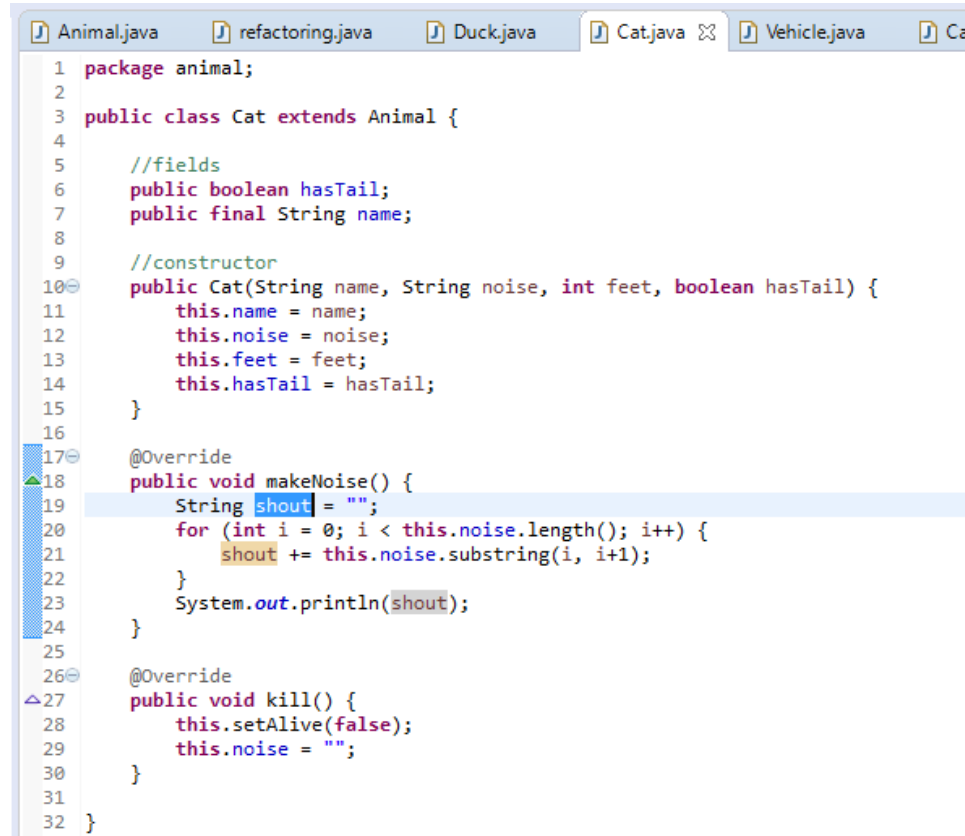


Figure 3: Marking and settings

Figure 3 shows the selected code part highlighted in the color blue, here it is `sound` in line 19. Eclipse automatically highlights all corresponding occurrences in line 21 and line 23. After clicking „Refactor“ and navigate to „Rename“ in the menu bar of Eclipse the window below line 33 appears. In this window the new name for `sound` namely `shout` is typed. The other

settings are left untouched and confirmed by clicking on OK.



```
1 package animal;
2
3 public class Cat extends Animal {
4
5     //fields
6     public boolean hasTail;
7     public final String name;
8
9     //constructor
10 public Cat(String name, String noise, int feet, boolean hasTail) {
11     this.name = name;
12     this.noise = noise;
13     this.feet = feet;
14     this.hasTail = hasTail;
15 }
16
17 @Override
18 public void makeNoise() {
19     String shout = "";
20     for (int i = 0; i < this.noise.length(); i++) {
21         shout += this.noise.substring(i, i+1);
22     }
23     System.out.println(shout);
24 }
25
26 @Override
27 public void kill() {
28     this.setAlive(false);
29     this.noise = "";
30 }
31
32 }
```

Figure 4: State after modification

Figure 4 shows the code in the state after the modifications from the rename refactoring are executed. The selected *sound* in line 19 has changed to *shout* just like the other two corresponding occurrences in line 21 and line 23. The code highlighting has not changed.

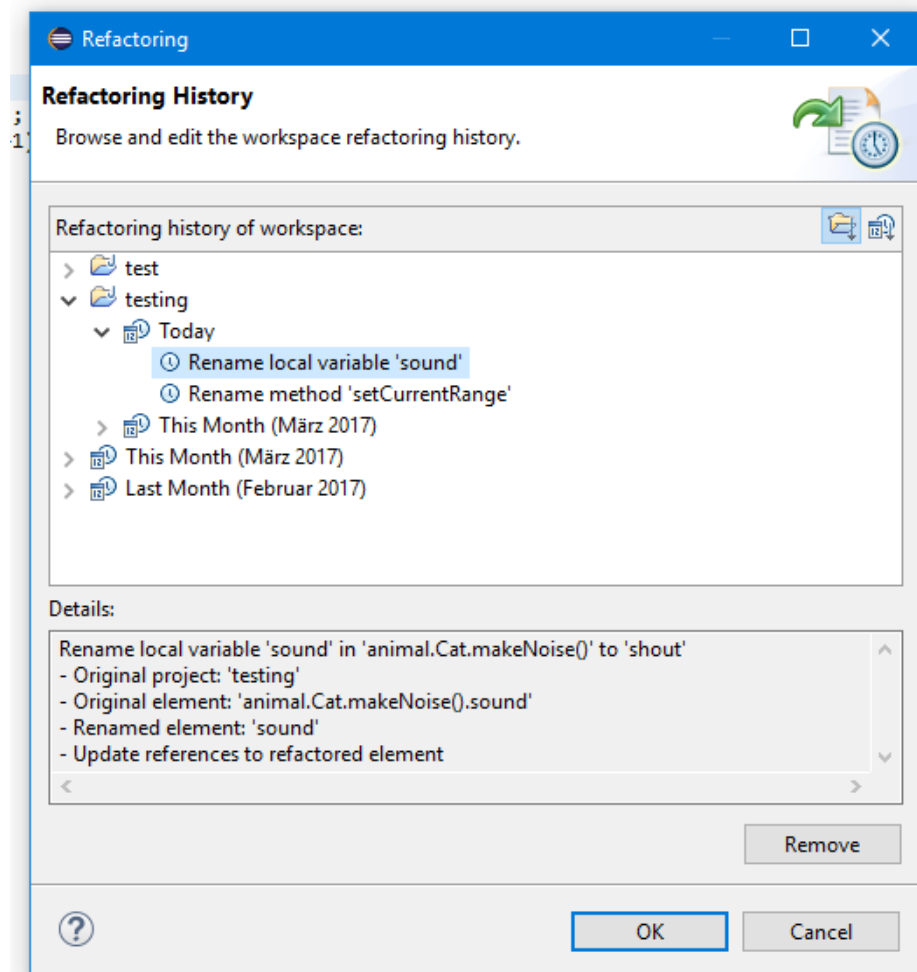


Figure 5: History

Figure 5 shows the entry of the rename refactoring from *sound* to *shout* in the History function of Eclipse. It is listed as the newest entry. In the Details the type of the marked code: local variable, the fully qualified name: `animal.Cat.makeNoise()`, the name of the project: `testing`, the original element: `sound` and the chosen settings: `Update references to refactored element` are displayed.

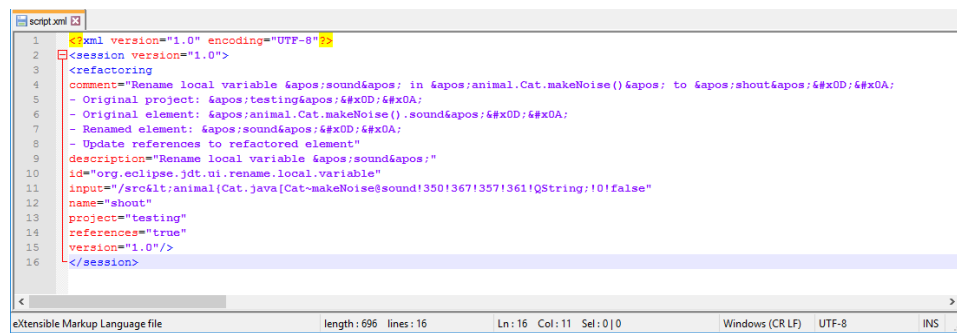


Figure 6: Script

Figure 6 shows the created script in xml format. The first entry „comment“ includes the details of the refactoring History, followed by the entry „description“ which corresponds to the description in the refactoring History (see Figure 5). The remaining entries contain the required information for Eclipse to reconstruct the executed refactoring. By applying the script to the code in starting position Eclipse executes the chosen refactoring automatically. This automates the reproduction of the code modifications always in the same way and can be used to eliminate errors.

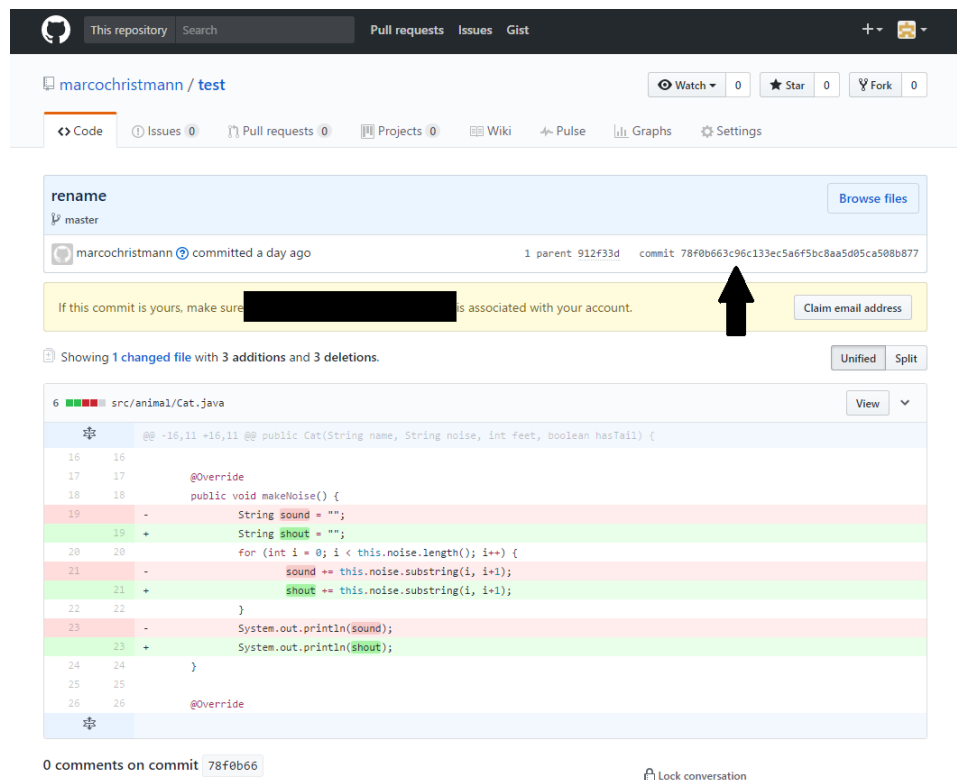


Figure 7: Changes

Figure 7 shows how the changes are displayed on github. It includes the changes since the last commit and the names of the changed files. In the middle can be seen that only one file, `Cat.java`, has changed and that there are 3 changes in this file. The lines where changes happened are highlighted red and green highlighted with leading „-“ and „+“. The actual change in this line is marked in a stronger shade. In the case of the rename refactoring from *sound* to *shout*, the highlighted lines are 19, 21 and 23, the source word *sound* is highlighted in the color red because it was deleted, the word *shout* is highlighted in the color green because it was added.

```

diff --git a/src/animal/Cat.java b/src/animal/Cat.java
index d702d71..de90006 100644
--- a/src/animal/Cat.java
+++ b/src/animal/Cat.java
@@ -16,11 +16,11 @@ public class Cat extends Animal {
     @Override
     public void makeNoise() {
-        String sound = "";
+        String shout = "";
+        for (int i = 0; i < this.noise.length(); i++) {
+            sound += this.noise.substring(i, i+1);
+            shout += this.noise.substring(i, i+1);
+        }
-        System.out.println(sound);
+        System.out.println(shout);
     }
     @Override

```

Normal text file length: 512 lines: 20 Ln: 1 Col: 1

Figure 8: Diff-file

Figure 8 shows the diff-file. In order to create the diff-file with the git command line the following code line was used:

```
git diff commit_id_1 commit_id_2 >> diff.text
```

The *commit_ids* are the identification of the commit. The first id is the reset commit before the current refactoring, the second one is the commit of the current refactoring. They are accessible on the github repository by clicking the desired commit. Moreover, the position on the example of *commit_id₂* in Figure 7 is marked with arrow. The *commit_id_1* is 912f33da6aed0ec0022d908352f7d1ffd62d6095 so the complete code line is:

```
git diff 912f33da6aed0ec0022d908352f7d1ffd62d6095 78f0b663c96c133ec5a6f5bc8aa5d05ca508b877 >> diff.text
```


2.3 Iteration of the steps for an other code modification

As a simple example of the remaining two categories the case called „Surround With“ was chosen. Surround With offers 11 different possibilities to surround the selected code part. For demonstration purposes the if statement was chosen. This will insert an if statement above the selected code. The selected code part then will only be executed if the statement is fulfilled. Because the same class was chosen to execute the modification the starting situation is the same as for the rename refactoring and is already shown in Figure 2.

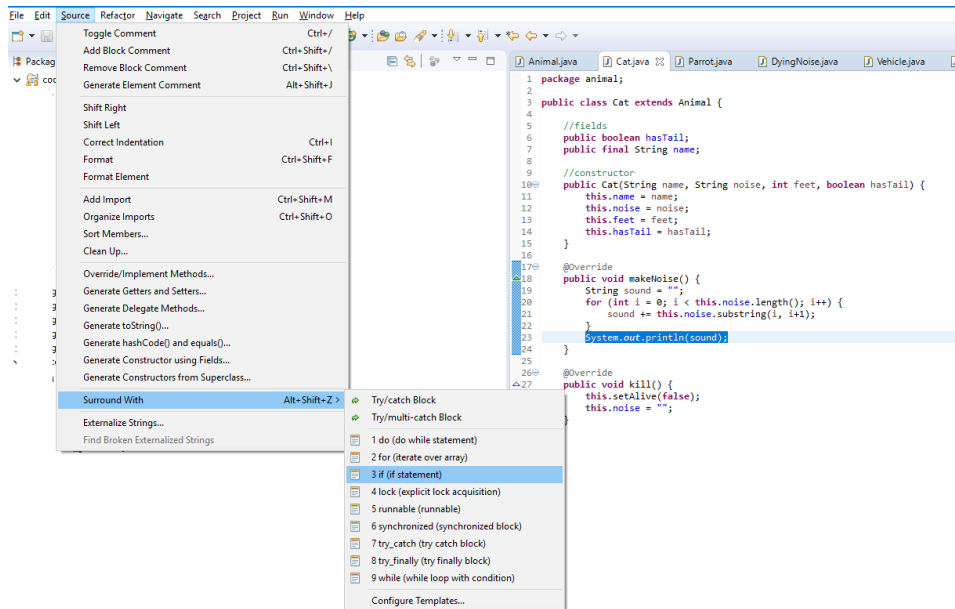
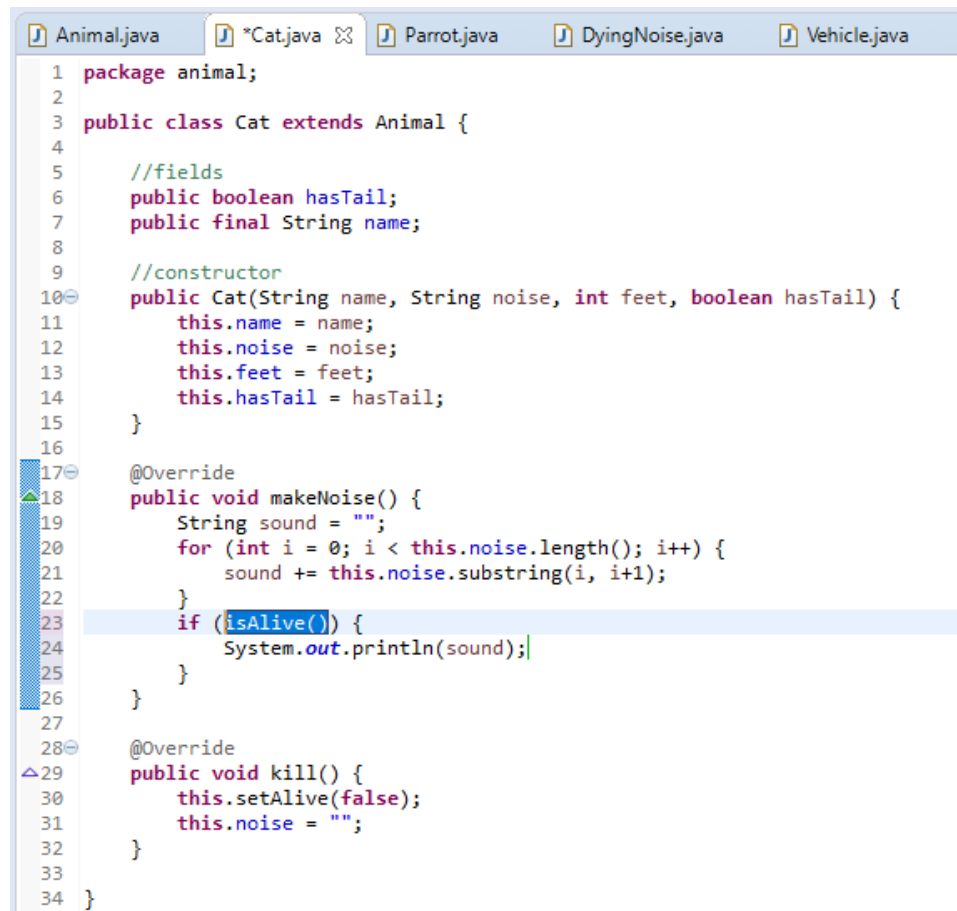


Figure 9: Marking and settings

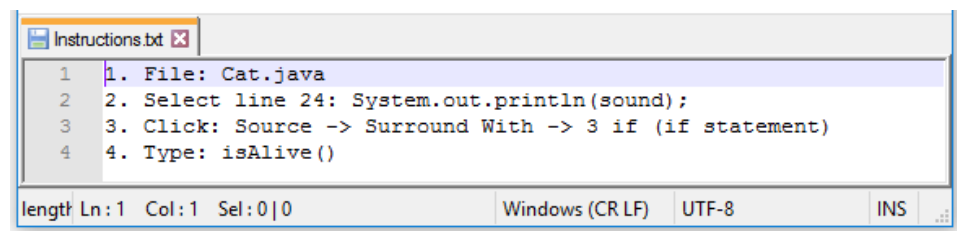
Figure 9 shows the selected code part highlighted in the color blue, here it is `System.out.println(sound);` in line 23. Next steps are clicking „Source“ in the menu bar of Eclipse, navigating to „Surround With“ and choosing „3 if (if statement)“.



```
1 package animal;
2
3 public class Cat extends Animal {
4
5     //fields
6     public boolean hasTail;
7     public final String name;
8
9     //constructor
10 public Cat(String name, String noise, int feet, boolean hasTail) {
11     this.name = name;
12     this.noise = noise;
13     this.feet = feet;
14     this.hasTail = hasTail;
15 }
16
17 @Override
18 public void makeNoise() {
19     String sound = "";
20     for (int i = 0; i < this.noise.length(); i++) {
21         sound += this.noise.substring(i, i+1);
22     }
23     if (isAlive()) {
24         System.out.println(sound);
25     }
26 }
27
28 @Override
29 public void kill() {
30     this.setAlive(false);
31     this.noise = "";
32 }
33
34 }
```

Figure 10: State after modification

Figure 10 shows the code in the state after the modifications from Surround With are executed. The selected code in line 23 has moved to line 24 and is surrounded with an if statement. Furthermore, the if statement is selected and was changed to `isAlive()` by typing.



```
Instructions.txt
1 1. File: Cat.java
2 2. Select line 24: System.out.println(sound);
3 3. Click: Source -> Surround With -> 3 if (if statement)
4 4. Type: isAlive()

length Ln: 1 Col: 1 Sel: 0|0 Windows (CR LF) UTF-8 INS
```

Figure 11: Instructions

Figure 11 shows the instruction file where each performed step is listed in chronological order. These instructions replace the provided scripts for the Refactorings. By performing each step the changes will be reproducible.

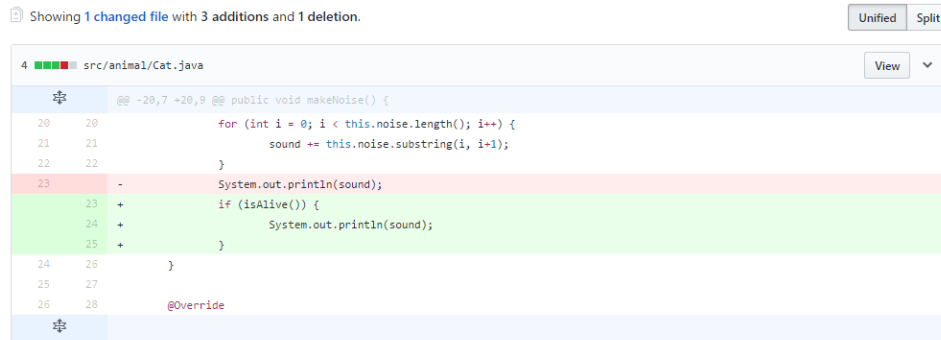


Figure 12: Changes

Figure 12 shows how the changes are displayed on github. On the top is shown that only one file, Cat.java, has changed since the last commit. In this case the code in line 23 was moved to line 24 and was surrounded by the if statement in line 23 and 25.

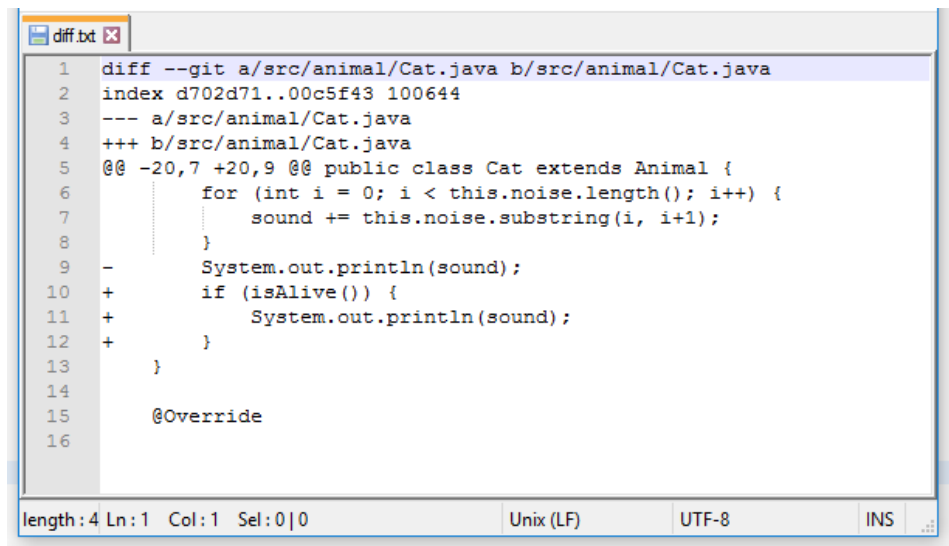


Figure 13: Diff-file

Figure 13 shows the diff-file and includes the changes shown in Figure 12.

3 Elaboration of the results

After the preparation is done, it is ready for an application case. There already exists an approach, the plug-in Retestoring [1], of catching the code changes and giving the programmer advice to adjust the test cases.

3.1 Testing the plug-in Retestoring

The reproducible code modification cases were run through and the output of the plug-in Retestoring was captured.

Property	Change	Original Value	New Value	Placeholder Data
body	removed	int a = currentRange;		
body	replaced (placeholder)	a	MISSING currentRange	

Figure 14: Retestoring output

Figure 14 shows the output of the plug-in. Under „Changed Node“ the changed method is shown, here it is *remainingRange()*. There are two changes captured, the first one is that $int_a = currentRange()$ was removed and the second one is that *a* was replaced with *currentRange*.

Afterwards, the output was compared to the screenshot of code changes receive from step 6b) (see section 2.1).

Showing 1 changed file with 1 addition and 2 deletions.

Line	Change	Code
15		@@ -15,9 +15,8 @@ public Tires(int maxRange, String manufacturer) {
16		}
17		public int remainingRange() {
18	-	int a = currentRange;
19	-	int b = maxRange;
20	-	int result = a - b;
21	+	int result = currentRange - b;
22		return result;
23		}

Figure 15: Changes

Figure 15 shows the changes captured by github. By comparing the two figures 14 and 15 concludes that this case was captured completely.

In order to get an overview of which cases are captured completely, in which

cases something is missing and in which cases nothing is captured at all the list of code modifications (see Figure 1 was taken to help.

Refactorings	Manual	Generalized
Refactor	File	Source
Rename	Move	Override/Implement Mehtods
Move	Rename	Generate Getters and Setters
Change Method Signature		Generate Delegate Methods
Extract Method	Edit	Generate toString()
Extract Local Variable	Undo	Generate hashCode() and equals()
Extract Constant	Redo	Generate Constructor using Fields
Inline	Cut	Generate Constructors from Superclass
Convert Local Variable to Field	Paste	Surround With
Convert Anonymous Class to Nested	Delete	Externalize Strings
Move Type to New File	Find/Replace	
Extract Superclass	Word Completion	
Extract Interface	Quick Fix	
Use Supertype Where Possible		
Push Down	Search	
Pull Up	File -> Replace	
Extract Class		
Introduce Parameter Object		
Introduce Indirection		
Introduce Factory		
Introduce Parameter		
Encapsulate Field		
Generalize Declared Type		
Infer Generic Type Arguments		

Figure 16: Overview

Figure 16 shows the highlighted list of code modifications in Eclipse. If the plug-in got all the code changes the corresponding case in the list was highlighted in the color green, if the case was not captured complete it was highlighted in the color orange and if no changes were captured it was highlighted in the color red.

3.2 Critical discussion of completeness

Overall, the plug-in got 24 cases completely, in 4 cases something was missing and in 15 cases there was nothing captured. It is noticeable that in the categories Refactorings and Generalized most cases are captured completely, however in the category Manual there is only one captured case. That could be due to the issue, that the code modifications in the category Manual are mostly done by hand and in the progress of code development.

Another critical aspect is that these results could not refer to completeness.

The cases which are chosen in this paper are just one little part of what is possible. Additional to that, only the standard settings were considered. For example, the case „Extract Method“ was not correctly captured in this scenario. If the case is changed to that only one line of code is extracted the plug-in will capture it correctly. In order to make the statement more meaningful it must be tested with several more cases for each existing case. With the help of this paper a scheme was provided to make it possible for developers to create their own test cases.

4 Conclusion

4.1 Summary

This paper shows the problem that code changes do not consider the existing test cases and the programmer is not notified about the changes. It exists one approach to solve this problem, but it does not cover all cases. The necessity of a complete basis of test cases shows up. First a project for Eclipse was created there all possible code modifications are executable. These code modifications are made reproducible and accessible by plug-in developers. Then a scheme was invented to capture all changes the code modification cases make. Afterwards, the created basis was tested for the existing plug-in Retesting.

4.2 Result

With the help of this paper the gap that there is no documentation how Eclipse manages the provided code modifications was closed. A basis in which every provided possibility of code modifications is recorded and the changes are logged was created. The basis contains a complete listing of all possible code modifications provided by Eclipse, the recording of the changes that each code modification makes and the starting situation to reproduce the code modifications. On this basis new and existing plug-ins can be tested and improved to give complete support because the cases are made reproducible.

References

- [1] Passier, Harrie and Bijlsma, Lex and Bockisch, Christoph *Maintaining Unit Tests During Refactoring* 2016
- [2] Buse, Raymond P.L. and Weimer, Westley R. *A metric for software readability* 2008
- [3] Thies, Andreas and Roth, Christian *RSSE: Recommending Rename Refactorings* 2010
- [4] Glass, Robert L and Wesley, Addison *Facts and Fallacies of Software Engineering* ISBN: 0-321-11742-5, 2002
- [5] Fowler, Martin and Beck, Kent and Brant, John and Opdyke, William and Roberts, Don *Refactoring: Improving the Design of Existing Code* 2002
- [6] Kent Beck *Test Driven Development: By Example.* 2002
- [7] *The Market Share of Java IDEs* <http://www.baeldung.com/java-ides-2016>
- [8] Bartosz Walter and Blazej Pietrzak *Automated generation of unit tests for refactoring* Extreme Programming and Agile Processes in Software Engineering, pages 211-214. Springer, 2002
- [9] Tom Mens and Tom Tourwé *A survey of software refactoring* IEEE Transactions on Software Engineering (Volume: 30, Issue: 2), pages 126-39, 2004
- [10] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts *Refactoring: Improving the Design of Existing Code* ISBN-13: 978-0201485677, 1999
- [11] *github* www.github.com

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtliche und sinngemäße Zitate habe ich als solche kenntlich gemacht.

Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum, Unterschrift