Post-Refactoring Recovery of Unit Tests: An Automated Approach

https://doi.org/10.3991/ijim.v17i08.38785

Abdallah Qusef[1], Sharefa Murad[2], Najeh Alsalhi[3,4](✉), Eman Shudayfat[5]
[1] Princess Sumaya University for Technology, Amman, Jordan
[2] Middle East University, Amman, Jordan
[3] Humanities and Social Sciences Research Center, Ajman University, Ajman, UAE
[4] Deanship of Research and Graduate Studies (DRG), Ajman University, Ajman, UAE
[5] Department of Creative media, Luminus Technical University College, Amman, Jordan
n.alsalhi@ajman.ac.ae

Abstract—In application development lifecycle, specifically in test-driven development, refactoring plays a crucial role in sustaining ease. However, inspite of bringing ease, refactoring does not ensure the desired behaviour of code after it is applied. Because refactoring tends to worsen the alignments between source code and its corresponding units. One significant solution to the aforementioned issue is the technique called unit testing. As unit testing enable the developers to confidently apply refactoring while avoiding undesired code behaviour. Unit testing provides effective preventive measures for avoiding bugs by providing immediate feedback, thus assisting to mitigate the fear of change. In this work, we present a tool called GreenRefPlus which efficiently enables the developers to maintain the veracity of code after the process of refactoring is applied. The proposed tool provides automatic recovery for the unit tests after the code is refactored. In this work, we consider Java as our target programming language and we focus on five various types of refactoring, which include Rename Method, Extract Method, Move Method, Parameter Addition and Parameter Removal. Our experiments indicate that the proposed tool GreenRefPlus enables us to consistently refactor the code and apply unit tests. The results presented in our work reveal that the proposed tool assists developers in saving approximately 43% of the total time required to manually recover from broken unit tests.

Keywords—unit tests, GreenRefPlus, code refactoring, automatic recovery, JUnit, eclipse plug-in, Agile eXtreme Programming, test-driven development

1 Introduction

The process of refactoring has a huge impact on software development lifecycle because it provides the necessary techniques for improving the internal structure of source code, while keeping the desirable output. There are various refactoring opportunities in any source code governed by few factors. First, the modifications required for the improvement in quality; and second, modifying or reorganizing the code in such as way that its output is preserved [1]. According to the author in [2], refactoring is defined as:

"changes made in the internal structure of the software, thus making it easier to understand and cheaper to modify without changing its observable behavior." However, it has been observed that refactoring may leads to variety of issues and the introduction of bugs in the source code.

Unit testing is the testing process of individual or groups of related units of any source code. Unit testing allow the developers to ensure that a small chunk of code (or unit) is able to meet the design requirements or behaves in accordance with the predefined design [3].

When deployed in conjunction, refactoring and unit testing become powerful tools to simplify, clarify and improve the structure of code in a safely manner. This is due to the fact that unit testing becomes a safety net for developers against the introduction of bugs during the process of refactoring.

In Test Driven Development (TDD), the unit test's coding is performed before the development of source code [4]. In TDD, refactoring not only enables the developers to remove code duplications and code complications, instead, it also assists in the incremental and step-by-step evolution of the code design [5]. For instance, in Agile eXtreme Programming (XP), the development of the software is done in TDD environment, therefore, the refactoring and testing is accomplished frequently [6]. The code development in XP using TDD environment is focused on three measures, First, writing the code for failed tests; second, development of production code such that the aforementioned tests pass; and lastly, refactoring the code to remove bad smells prevailing in the code [7].

As discussed, there is a high probability that the refactoring may result in the undesired behavior of the test codes [8-9]. For instance, the refactoring applied for method renaming consequently introduces an inconsistency in the source code's structure and its corresponding unit tests. In order to mitigate these worsened alignments, the method renaming has to be applied across all the corresponding unit tests. The aforementioned issue forms the basis of this work, i.e., refactoring leading to undesired behavior of unit tests.

In this work, we propose an efficient tool called *GreenRefPlus.* The proposed tool allows the developers to automatically recover the unit test after failures are introduced in source code due to refactoring. In addition, we also study the impact of refactoring on unit tests which are caused due to slow and sometimes risky manual refactoring methods. We also focus on the effect of undesired behavior of automated tools for refactoring. The work presented in this paper deals with two major questions,

Q1: If there is a need to develop such a tool that assists in the recovery of unit test during the process of refactoring?

Q2: Is there a possibility of developing such a tool that has acceptable performance?

In [10], authors propose a tool called *GreenRef.* This tool is developed to assist the developers for automatic refactoring. The tool proposed in [10] enables the developers to automatically apply 3 types of refactoring in Java programming language, i.e., rename method, parameter addition and parameter removal. In this work, we extend the tool presented in [10] and two other crucial types of refactoring namely extract method and move method. The newly introduced techniques are introduced for Java programming language. A rigorous analysis for the new types of refactoring is done. The results presented in this work reveal that the proposed techniques adequately assist developers in automatic refactoring of source codes.

The rest of the manuscript is organized as follows.

In Section 2, we present the literature review focusing on the major concerns that we used in the development of this work. In Section 3, we present the methodology for the proposed tool *GreenRefPlus*. We also focus on various steps used for the development of this tool. In, Section 4, we present the setup for the experiments we conducted for this research work. In Section 5, we present the results for the experiments conducted for this research work. In Section 6, we present a discussion on the finding of the study conducted in this paper. In addition, we also present the limitations of the proposed methods of refactoring in Section 7. Finally, in Section 8, we conclude our work.

2 Related work

The research literature presents various methods for manual code refactoring [11-12]. In addition, literature also states that there are various occasions in which developers refactor their code manually, despite being aware of the presence of automated tools for refactoring. This may be due the hindrances and barriers between the refactoring tools and developers [13]. There are many factors that contribute to the formation of these barriers. For instance, discoverability, lack of familiarity and trust, and productivity are the few factors that results in developers refactoring the code manually [14-15]. Authors in [16] discuss that approximately 90% of the developers prefer performing manual refactoring and do so as well. The authors conducted experiments which included almost thirteen thousand developers and four datasets. The authors report that refactoring was frequently applied by the developers, however, the methods of application were mostly manual and without the assistance of any tools for automatic refactoring [16]. Similarly, the experiments conducted in [17] shows that the developers performed approximately 11% more manual refactoring than automated refactoring. According to authors in [18], there is another factor that prevents the developers to perform automated refactoring. The study conducted by the authors show that developers feel restricted and limited while using some of the automated refactoring engines. In addition, the developers also report the presence of several bugs and undesired behaviour in these automatic refactoring tools [18]. These all factors keep developers at bay from using automatic refactoring tools, thus reducing the efficiency.

Authors in [18] conducted a research regarding the advantages and hinderances of the application of refactoring. This research was conducted at Microsoft and comprised of 3 different techniques including a survey, detailed interviews with professional developers and an analysis of software's version history. The authors conclude that there is a significant gap between the theoretical techniques of refactoring and practical methods deployed for refactoring. The study shows that most of the professional developers opined that refactoring is high-risk task and manual refactoring is costly in terms of time resource. However, the quantitative analysis performed by authors undertook Windows 7. The quantitative analysis reveals that the refactoring techniques greatly influenced the Windows 7 development and resulted in a number of successes. Similarly, authors in [19] performed an analysis on various types of refactoring. According to the authors, refactoring may become the cause of various issues in the code as original tests written for unit testing need to be changed. For the purpose of study, authors divided the refactoring types into 5 categories namely A, B, C, D and E. Each class was made by focusing on the effect of refactoring types on unit tests and the recovery procedure for the broken unit tests. This study greatly assists in understanding the paradigm of refactoring and unit testing. The authors also put forward the idea of "test-first refactoring". In this refactoring technique, existing unit tests become the base of finding the suitable refactoring practices. In addition, authors also present the notion of "refactoring-session", which includes the methods to make modifications to the production code as well as the test code.

Authors in [20] performed an analysis of the effect of refactoring on API coverage of unit tests. The authors proposed a new plug-in in Eclipse IDE that has the tendency to track the edits made during the course of refactoring. The analysis of the authors shows the most suitable techniques that can be used by developers for updating the test suite. The proposed method is equally beneficial for all types of refactoring.

The research presented in [17] presents a new technique for the analysis of impact caused by refactoring on the case of regression tests. The authors investigate the reasons of regression test failures due to the influence of regression. The authors conduct a study by making a relation between the type of refactoring and the broken unit tests. The results show that proposed approach has a precision of 80%. Please note that this study was conducted for 5 open-source applications. In a similar fashion, authors in [21] performed an analysis of the modification made during refactoring on the regression tests. This was done using the development version history of open-source project developed in Java programming language. The analysis performed by the authors were refactoring reconstruction analysis and an analysis on modifications. The relationship analysis of refactoring types and refactoring location was performed using REFFINDER tool and the modification analysis was performed using FAULT-TRACER tool. The experiment shows that approximately 38% of the unit tests effected are related to refactoring; half of which include broken tests.

Authors in [12] conduct a study that how manual refactoring is an error-trap. In order to cope with issues of manual refactoring, authors propose a tool called BeneFactor that assists developers in performing automatic refactoring. However, the tool proposed in this work surpasses BeneFactor in terms of saved time.

3 Research methodology

In this section, we present the proposed *GreenRefPlus*. We also present the steps that have been taken for the development of this tool for automatic refactoring in detail.

Please note that the basic theme for the development of this tool is to efficiently facilitate the developers for refactoring and unit testing in a reliable manner. The proposed tool is designed using a process that comprises four major stages discussed below.

3.1. Analysis of techniques for code refactoring

In this stage, we research and study the techniques and methods that are necessary for making refactoring easy and reliable. The research conducted for the development of this assistance tool focuses on Java programming language.

Source code format. The code writing style varies from developer to developer. This is quite true for the two developers working in the same workspace or environment.

In order to read the Java files, all the custom prepared files need to be imported at the beginning of *GreenRefPlus* implementation. Uniform formatting needs to be applied on source code every time modification is made for reliable analysis. Therefore, a custom format XML file is designed in eclipse so that it can be imported at the project configuration stage.

Refactoring history. The automated refactoring tool presented in this work namely *GreenRefPlus* is refactoring-aware code review tool. This tool has the capability to effectively detect the two types of refactoring modifications, i.e., refactoring-type and refactoring-location. *GreenRefPlus* accomplishes this by performing a comparison of two main version of the code. Now, please note that in order to successfully accomplish this, both versions of the program should be readily available. The main target is to keep the files outside the Eclipse projects while being in-sync with Eclipse project files. The availability of the both versions of the source code is ensured by designing *FileSync* eclipse plugin. *FileSync* is a file synchronization tool that is instigated within *GreenRefPlus* development environment.

Post-refactoring automatic build. As discussed in Section 1, the most basic objective of this work is the avoidance of unit test breaking while performing refactoring. Thus, there needs to be a constant inspection of the refactoring process during the code development phase until the end. In other words, as soon as the developer saves the modifications made during refactoring, the effects on the source code must be continuously monitored. This is done by automatically building the modified portions of Java source code by the help of *Apache Ant* builder. The *Apache Ant* builder can be integrated within eclipse with ease and builds edited Java source code. For this process, the *Ant* build file is regarded as the central control unit because of its ability to efficiently handle the following processes.

- Compiling Modified Source Code

During this stage, the modified source code files are compiled. This is done by executing the *javac* task. The compiler to be used for compiling the Java files can be selected by compiler attribute and by including the *Ant* runtime. Please note that the *Ant* runtime libraries need to be included in the class-path.

- Running Program's Unit Tests

This step runs the unit tests available in the JUnit testing framework. The resultant output of these tests is a summary for every test class. The summary is stored in TestReports directory.

- Running *GreenRefPlus* Recovery Tool

At this stage, the *GreenRefPlus* external jar file is executed.

3.2. Analysis of code refactoring practices

In addition to avoiding breaking of unit tests, analysis of code refactoring practices forms another important aspect of the work done in this paper. During this analysis stage, all the modified code files that are synced during stage 1 are analyzed one line at a time. Each line of the modified code is compared with the code's previous version with the help of refactoring-type and refactoring-location before the process is completed. There are few sub-stages of this step presented below.

Reading modified files. When the developer performs the refactoring and save the changes, all the edited Java source files are synced in a folder outside Eclipse.

Converting Modified Java files in XML format. The modified Java files are converted to XML file format for making the comparison process easy and quick. The generalized name of the modified Java file after conversion is *"packageName.className.meta.xml"*.

XML files analysis. Please note that after conversion, the original and un-modified code is also available in XML format. In order to identify the differences between the original and modified versions of the source code, both files are compared, and the differences are stored in the resultant file as described in previous detail.

Comparing original and modified XML files. The original and modified XML files are compared line by line by focusing on the following refactoring types.

- Rename Method: For this refactoring type, the comparison requires the list of method's parameters and the method's body of the original and modified XML files.
- Extract Method: In this type of refactoring, we have two or more code fragments that can be grouped together. Extract method refactoring is applied because if there are more lines in a method then it becomes harder to identify the method's function. Extract method refactoring also allow to mitigate rough edges in the source code. This type of refactoring is done by creating a new method with such a name that makes its function self-evident. Then the code body is moved inside the newly created method. Please note that if the variable used by the code fragment which is being moved are declared prior to the code fragment, then these variables need to be passed to the newly created method in order to make it behave in a desired fashion.
- Move Method: This type of refactoring is applied when a method is used in another class than the class in which it is created. The refactoring of moving such method comprises of moving the method to the class that contains most of the data being used in the method's computation process. This type of refactoring makes the class

more internally coherent. This type of refactoring is done by creating a new method in that class which makes most of the use of this method. The code from the old method is moved to this newly created method. Finally, the code of the original method is transformed into a reference to the new method in other class.

- Parameter Addition: This refactoring type also requires method's parameters and method's body of the original and modified XML files. Please note that this does not check for overloaded function.
- Parameter Removal: Similar to parameter addition and rename method, this type of refactoring also requires the method's name and method's body for performing comparison. Again, this does not include an overload function.

Result file preparation. After the analysis of the modification made during refactoring is complete, all the applied practices are listed in an output file. The output file also has the XML format.

Reading test reports. Initially, all the unit test reports are loaded. The report for each unit test in located in the following hierarchy.

Refactored project base directory\\Refactoring\\Test Reports.

3.3. Unit tests automatic recovery

The unit tests are implemented in the refactored project. When the process of refactoring is completed and saved, and the modified files are built, the unit test available within the refactored project also execute. The resultant output file is generated and saved in *Test Reports* directory located within the refactored project directory.

In case of broken unit tests, following array of techniques is applied for recovery.

Analysis of loaded reports. After the unit test reports are loaded, an analysis is performed on these reports. This information in these reports state the cause of unit test failure and the location of failure. This information is utilized to identify the cause of break by establishing a connection between applied refactoring techniques and unit test failure type. Figure 1 illustrates the classification of test case failure with possible reasons that contributed in causing failure [17].

In this work, we focus on the breaks caused by errors, for instance, we specifically focus on the error type "*No Such Method*". An instance of report for failed unit tests is presented in Figure 2 and Figure 3. The information presented in the report generated by *GreenRefPlus* is crucial for the recovery of broken unit tests.

Moreover, the analysis of data type of method's parameter is also significant. This is due to the fact that the tool does not rely on the conventional naming techniques of Java. For instance, the symbol "I" in the generated report refers to the "int" type. Table 2 presents the data types in the report generated by *GreenRefPlus* and the corresponding data types in Java programming language.

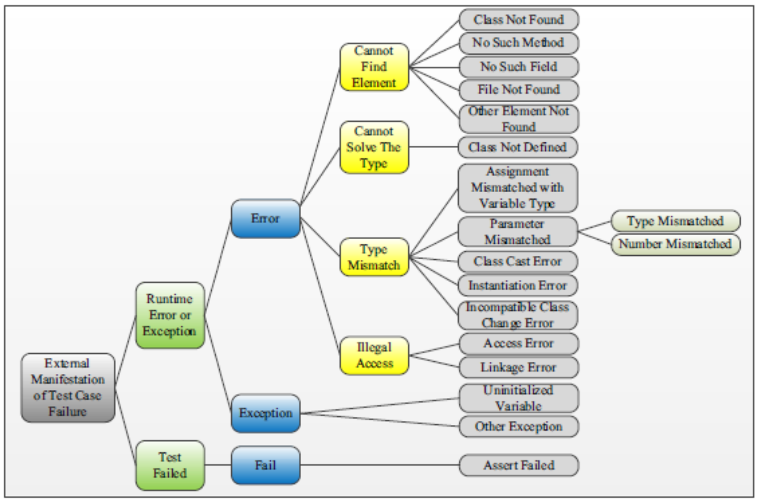
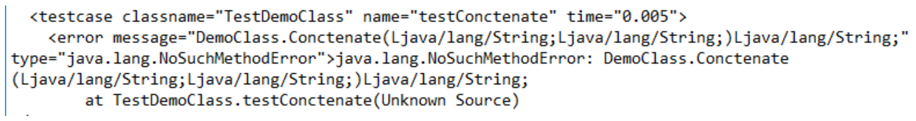
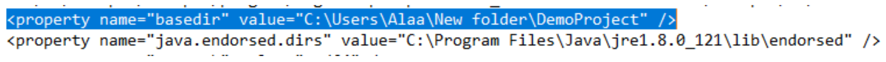


Fig. 1. Classification of unit test failures and their corresponding reasons



```
  <testcase classname="TestDemoClass" name="testConctenate" time="0.005">
    <error message="DemoClass.Conctenate(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;"
type="java.lang.NoSuchMethodError">java.lang.NoSuchMethodError: DemoClass.Conctenate
(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;
        at TestDemoClass.testConctenate(Unknown Source)
```

Fig. 2. A sample report for broken unit test (test case tag)



```
<property name="basedir" value="C:\Users\Alaa\New folder\DemoProject" />
<property name="java.endorsed.dirs" value="C:\Program Files\Java\jre1.8.0_121\lib\endorsed" />
```

Fig. 3. A sample report for broken unit test (basedir property)

Therefore, the analysis of the report generated by the tool presents useful information, such as the information regarding the failed unit tests, the type of error and some other additional information regarding the code that is subject to unit testing. We present this information is Table 1.

Table 1. Analysis of the unit test breaking report generated by the proposed tool *GreenRefPlus*

Test Case Tag's Property	**Analyzed Data**	**Value**
Classname	Name of testing class	TestDemoClass
Name	Name of the testing method	testConctenate
Error Tag's Property	**Analyzed Data**	**Value**
Message	Name of the class under test	DemoClass
Message	Name of the method under test	Concatenate
message	Parameters data types list for the method under test	String, String
type	Test error type	"No Such Method Error"
basedir Property	**Analysed Data**	**Value**
value	The path for unit test java file	C:\Users\Alaa\New folder\DemoProject

Table 2. Various data types in report generated using *GreenRefPlus*

Datatype in generated report	Corresponding datatype in Java programming language
I	Int
Z	Bool
C	Char
B	Byte
S	Short
J	Long
F	Float
D	Double
[I or any data type	Array of int
Ljava/lang/….	object data type like string, dictionary

Testing report analysis. The proposed tool *GreenRefPlus* is environment aware and understands that refactoring has been applied to the source code using one of more than one of the refactoring types. In this step, the analysis of the file *result.xml* is performed. Please note that this file is generated during the execution presented in step 2. The content of this report comprises of the modification performed during refactoring, the name of the package, the class name, the name of the method and location of the refactoring in the source code. Finally, the analysis concludes with matching the unit test breaking error. Please note that that the identification of this failure was performed in the previous stage. After the analysis is complete, the *GreenRefPlus* starts the execution for automatic recovery.

Automatic recovery for unit test. In the previous stages, the proposed tool *GreenRefPlus* successfully identified the issues. First, the *GreenRefPlus* determined the unit test which broke during the execution. Second, the tool automatically identifies the refactoring practice that resulted in the breaking of the unit test. In addition, the tool is aware of the error type and the location of modification made during the process of refactoring. *GreenRefPlus* makes use of all this information to execute recovery tests. Figure 4, Figure 5, Figure 6 and Figure 7 present the scheme that is deployed by the proposed tool *GreenRefPlus* for the automatic recovery for the broken unit tests.

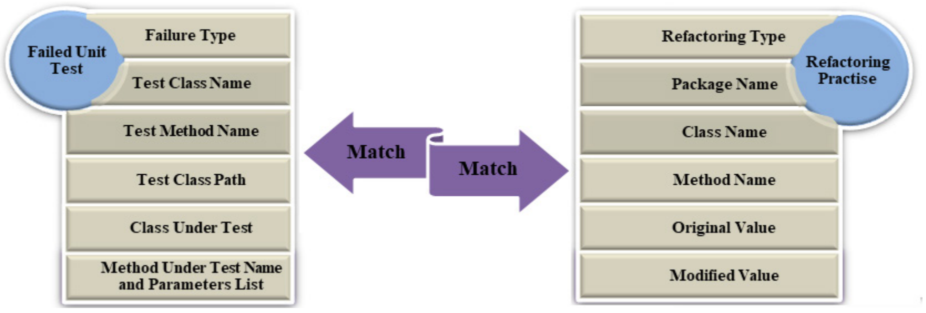


Fig. 4. The automated process of unit test recovery

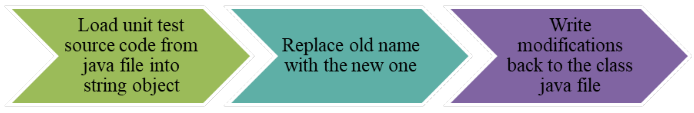

Load unit test source code from java file into string object
Replace old name with the new one
Write modifications back to the class java file

Fig. 5. Unit test recovery – Rename Method refactoring

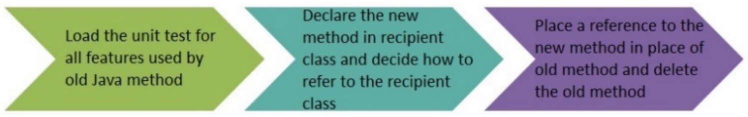

Load the unit test for all features used by old Java method
Declare the new method in recipient class and decide how to refer to the recipient class
Place a reference to the new method in place of old method and delete the old method

Fig. 6. Unit test recovery – Move Method refactoring

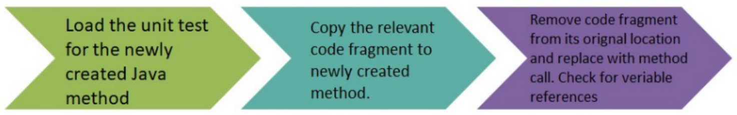

Load the unit test for the newly created Java method
Copy the relevant code fragment to newly created method.
Remove code fragment from its orignal location and replace with method call. Check for veriable references

Fig. 7. Unit test recovery – Extract Method refactoring

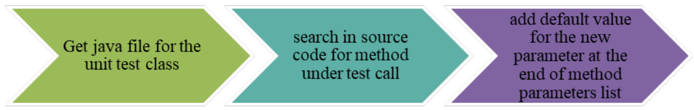

Get java file for the unit test class
search in source code for method under test call
add default value for the new parameter at the end of method parameters list

Fig. 8. Unit test recovery – Parameter Addition refactoring

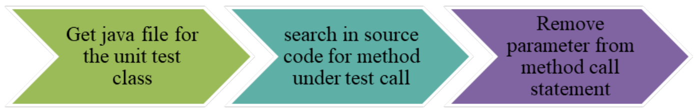

Get java file for the unit test class
search in source code for method under test call
Remove parameter from method call statement

Fig. 9. Unit test recovery – Parameter Removal refactoring

4 Experiment design

In order to perform the analysis of the proposed *GreenRefPlus*, a pretest-posttest experiment was conducted on one group of developers. Please note that in one-group pretest-posttest environment, only one group is tested for the validity of performed research [22-27]. Contrary to the controlled group experimental environment, the same group is subject to an additional test (pretest) before the actual tests (posttest) are commenced. This was done to analyse various factors of *GreenRefPlus*, such as the ability to maintain unit tests validity during the application of code refactoring practices, amount of time saved in comparison with manual refactoring and the usability of refactoring. This test is considered as a baseline for the experimental evaluations. In pretest and posttest stages, the subjects are measured in terms of dependent variables. This type of experiments allows researchers to report on facts of real user-behavior.

In addition, a series of questions was also made available for each individual for investigating the level of selected subjects. The pre-test list of question is presented in

Table 3. As discussed, the major target of these experiments is to evaluate the performance in terms of time saved during the process and the usability of the proposed tool. The evaluation of the proposed tool *GreenRefPlus* is done in terms of five refactoring techniques namely, rename method, extract method, move method, parameter addition and parameter removal. The evaluation process revolves around the time consumed for code refactoring using aforementioned refactoring types and time consumed for the recovery from broken unit tests. The time for manual refactoring and time consumed by refactoring using the proposed tool is detailed.

Pretest Design: For the purpose of pretest, each individual in the experimental group was presented with 5 question. The questions of pretest are presented in Table 3.

Table 3. Pretest list of questions for developer's group

ID	Question	Answer Format
Q1	What is your professional level?	Check the appropriate box.
Q2	What is refactoring according to you?	The subject had to write text for the definition he knows.
Q3	Have you ever applied refactoring on your code?	Yes or no.
Q4	What is your preference; manual refactoring or automated refactoring?	Choose manual or automated.
Q5	Among the five types, please select the refactoring you want to apply.	This question to analyze the popularity of refactoring types used by subjects. In this work we focus on Rename Method, Extract Method, Move Method, Parameter Addition, Parameter Removal.

Posttest Design: After the experiment is conducted and each subject is finished with their work, they are presented with 2 questions. The questions are related to the satisfaction level of the subject for the proposed *GreenRefPlus* and the level of ease for using the proposed tool.

a) Projects:

In order to conduct the experiment, 2 applications were chosen while keeping following characteristics in mind.

- The language for project development should be Java.
- The unit tests should be implemented for the selected applications.
- None of the application should be complex.
- Each selected application should have a small size.

b) Assignments:

The experiment took place at the Princess Sumaya University for technology. The subjects comprised of 21 senior bachelor students of fourth year. Each subject was a student of Software engineering major, with each having a different skill level in programming. Please note the all the students ensured the knowledge of software development and testing. According to the designed experiment, each developer was presented with 2 assignments. First, the development task should be completed using the manual refactoring process. In manual recovery, each developer had to refactor and recover the broken unit tests manually. Second, the same assignment should be developed using the proposed *GreenRefPlus*. In the later assignment, subject had to apply the refactoring and then broken unit tests were automatically recovered.

Each subject was asked to apply the specific refactoring modification chosen from aforementioned refactoring types. In addition, each subject was also asked to perform recovery on broken unit tests, as required. In both the manual and automated refactoring, the time of completion was recorded with care.

Pretests Results

Refactoring Knowledge: Among 21, there were 4 developers that were unable to correctly define the process of refactoring.

Refactoring Skills: 2 subjects declared that they don't have the experience of code refactoring and have never applied refactoring on their codes.

Choosing Manual vs. Automated Refactoring: Among the experimental group, 8 subjects preferred the application of manual refactoring. Contrary, six subjects chose automatic refactoring as the preferred method.

Selection of Refactoring Type: Among all the participants, 8 developers selected rename method, 4 selected parameter addition, 6 selected parameter removal and the remaining 5 selected either extract method or move method.

Posttest Results

After the completion of experiment, each developer was asked about their opinion regarding the proposed tool and if the proposed tool was helpful in refactoring. All the participants agreed that the tool not only made the process easy, but it also helped them to achieve time efficiency. In addition, we also asked the participants if they would feel comfortable and confident while refactoring the unknown code. Almost all the participants opined that they would feel comfortable while refactoring with *GreenRefPlus.*

5 Results and analysis

In this section, we present the results for the experiments performed for measuring the efficiency of the proposed *GreenRefPlus*. The evaluation process is done in terms of performing refactoring manually and with the assistance of the proposed tool.

The time consumed for each task by manual refactoring and automatic refactoring using proposed *GreenRefPlus* is presented in Table 4, 5, 6, 7 and 8. Each table presents the time consumed by the developer for the completion of one task by manual recover method and automatic recovery method. Please note that we have performed experiments for 5 types of refactoring, namely, rename method, extract method, move method, parameter addition and parameter deletion.

Table 4. Experiment results for Rename Method Refactoring (Task 1)

Subject ID	**Assignment # 1 (Manual refactoring)**	**Assignment # 2 (Automatic refactoring using *GreenRefPlus*)**	**Time Saved (s)**	**Time Saved (%)**
1	34	16	18	52.9%
2	38	10	28	73.7%
3	36	25	11	30.6%
4	22	11	11	50.0%
5	26	13	13	50.0%
6	54	13.5	40.5	75.0%
7	27	9	18	66.7%
8	41	10.4	30.6	74.6%
9	57	11	46	80.7%
10	36	15	21	58.3%
11	16	11	5	31.3%
12	17	14	3	17.6%
13	43	17	26	60.5%
14	36	13.4	22.6	62.8%
15	21	12.5	8.5	40.5%
16	30	23	7	23.3%
17	21	14	7	33.3%
18	19	9	10	52.6%
19	38	12	26	68.4%
20	25	12.5	12.5	50.0%
21	46	10	36	78.3%
Average	32.5	13.4	19.1	53.9%

Table 5. Experiment results for Extract Method refactoring (Task 2)

Subject ID	**Assignment # 1 (Manual refactoring)**	**Assignment # 2 (Automatic refactoring using *GreenRefPlus*)**	**Time Saved (s)**	**Time Saved (%)**
1	65	59	6	9%
2	71	63	8	11%
3	58	52	6	10%
4	80	78	2	2%
5	45	44	1	2%
6	54	49	5	9%
7	56	55	1	1%
8	49	41	5	10%
9	79	61	18	23%
10	90	79	11	12%
11	82	67	15	18%
12	72	55	17	24%
13	66	53	13	20%

Subject ID	Assignment # 1 (Manual refactoring)	Assignment # 2 (Automatic refactoring using *GreenRefPlus*)	Time Saved (s)	Time Saved (%)
14	58	48	10	17%
15	57	55	2	4%
16	50	44	6	12%
17	91	78	13	14%
18	77	64	13	17%
19	62	51	11	18%
20	81	77	4	5%
21	69	52	17	25%
Average	67	60	8	13%

Table 6. Experiment results for Move Method refactoring (Task 3)

Subject ID	Assignment # 1 (Manual refactoring)	Assignment # 2 (Automatic refactoring using *GreenRefPlus*)	Time Saved (s)	Time Saved (%)
1	27	15	12	44%
2	35	17	18	51%
3	24	12	12	50%
4	43	23	20	46%
5	39	18	21	53%
6	44	26	18	40%
7	23	11	12	52%
8	30	14	16	53%
9	49	27	22	45%
10	27	13	14	52%
11	28	12	16	57%
12	35	17	18	51%
13	29	21	8	28%
14	41	17	24	59%
15	38	30	8	21%
16	25	15	10	40%
17	37	12	25	67%
18	42	21	21	50%
19	34	20	14	41%
20	21	10	11	52%
21	23	12	11	48%
Average	33	17	15	47%

Table 7. Experiment results for Parameter Addition refactoring (Task 4)

Subject ID	Assignment # 1 (Manual refactoring)	Assignment # 2 (Automatic refactoring using *GreenRefPlus*)	Time Saved (s)	Time Saved (%)
1	22	13	9	40.9%
2	13	12	1	7.7%
3	24	15	9	37.5%
4	15.5	10.5	5	32.3%
5	22	12	10	45.5%
6	25.5	15	10.5	41.2%
7	15	10	5	33.3%
8	20	10	10	50.0%
9	16	13	3	18.8%
10	38.5	13.5	25	64.9%
11	11.5	9	2.5	21.7%
12	13	10.7	2.3	17.7%
13	20	11.8	8.2	41.0%
14	17	12.8	4.2	24.7%
15	18.5	14	4.5	24.3%
16	19	16	3	15.8%
17	15	12	3	20.0%
18	24	11	13	54.2%
19	16.5	11	5.5	33.3%
20	17	10	7	41.2%
21	23	12	11	47.8%
Average	19.3	12.1	7.2	34.0%

Table 8. Experiment results for Parameter Removal refactoring (Task 5)

Subject ID	Assignment # 1 (Manual refactoring)	Assignment # 2 (Automatic refactoring using *GreenRefPlus*)	Time Saved (s)	Time Saved (%)
1	21	13.5	7.5	35.7%
2	23	14	9	39.1%
3	35	13	22	62.9%
4	15	13	2	13.3%
5	26	13.5	12.5	48.1%
6	35.3	14.5	20.8	58.9%
7	19	12	7	36.8%
8	29.5	12	17.5	59.3%
9	23	12.5	10.5	45.7%
10	37	13.5	23.5	63.5%
11	19.5	14	5.5	28.2%
12	17.6	13	4.6	26.1%
13	15	11	4	26.7%
14	23	15	8	34.8%

Subject ID	Assignment # 1 (Manual refactoring)	Assignment # 2 (Automatic refactoring using *GreenRefPlus*)	Time Saved (s)	Time Saved (%)
15	18	11	7	38.9%
16	22	8	14	63.6%
17	21	13	8	38.1%
18	16	8	8	50.0%
19	25.5	14	11.5	45.1%
20	29	12	17	58.6%
21	34	14	20	58.8%
Average	24	12.6	11.4	44.4%

It is evident from the experimental results that each developer took less time for refactoring using *GreenRefPlus* as compared to manual recovery methods. Based on the above presented experimental results, we arrive at following results.

1. The rename method refactoring consumed about 19 seconds on average using *GreenRefPlus* and saved about 54% of time.
2. The extract method refactoring performed using the proposed tool *GreenRefPlus* accounted for 60 seconds and save approximately 13% of the total time.
3. The automatic refactoring for move method applied using *GreenRefPlus* saved approximately 47% of the total time. The average time for this task is about 17 seconds.
4. The parameter addition refactoring consumed about 7.2 second using *GreenRefPlus* and saved about 34% of the time as compared to manual refactoring
5. The automatic refactoring using *GreenRefPlus* saved 45% of the total time when applying parameter removal. The average time for this task is 11.4 seconds.

6 Discussion

In this section, we build a correspondence between the aforementioned results and the research questions of this work.

Q1: If there is a need to develop such a tool that assists in the recovery of unit test during the process of refactoring?

Authors in [8, 23] discuss that the process of refactoring is governed by behavior. Thus, small changes in the source code have a significant effect on the unit tests. Authors discuss that the recovery of unit tests after refactoring is applied cannot be mitigated completely due to 3 factors. First, a variety of tools for automatic recovery of unit test is unavailable. Second, there is a lack of techniques that can assist in forming a relation between source code and unit tests. Lastly, there is usually little time reserved for code maintenance, both in academia and industry.

As these issues cannot be coped with completely, there is a grave necessity for such tools that can efficiently assist the developers in performing automatic refactoring. Therefore, the proposed tool *GreenRefPlus* is essential and crucial for making the recovery process of broken unit tests automatic. The proposed tool has the capability to maintain the test validity while the process of refactoring is applied on the source code.

Thus, the answer to our question is yes, there is a need for such tool that can effectively assist in automatic recovery after refactoring.

Q2: Is there a possibility of developing such a tool that has acceptable performance?

The proposed tool *GreenRefPlus* is designed as a plug-in in Eclipse IDE. The tools used for the development of *GreenRefPlus* are standard and well-know, such as *Apache Ant*, unit test execution, *FileSync* and other Java Plugins. During the development phase, *GreenRefPlus* runs as a background service and does not account for too much processing. Due to this, many subject developers are inclined to use the proposed tool. Therefore, the answer to this question is yes. This is reflected by the time saved during the experiments.

7 Research limitations

This work has following limitations.

- Keeping in view the internal validity, the tasks chosen for the experiments performed for this work are simple, straightforward and an established and clear functionality. Therefore, it is necessary to test this tool in such development environment where the application is scaled to a huge size with a large number of unit tests and LOCs. This will allow to perform better analysis of the proposed tool.
- Keeping in view the external validity, the skill level of the subject developers was known prior to the experiments. The subjects had different level of Java programming skills and there was a great variation in knowledge of refactoring. Moreover, there were few developers who had no familiarity with the process of refactoring. This issue can be mitigated by conducting a session on refactoring to ensure the related knowledge before experiments are conducted.

8 Conclusion

Refactoring plays a crucial role for the removal of bad smells in the application developing environment. However, refactoring sometimes results in the undesired behavior of the code. Similarly, unit testing is a powerful technique for ensuring that each unit of the code is working in a desired fashion. When used in conjunction, refactoring and unit testing are valuable techniques for maintain the code quality in a reliable manner, as unit tests acts as a safety net for developers. In this work, we propose an automated tool for recovery of the unit tests after refactoring is applied. The proposed tool *GreenRefPlus* provides five different types of refactoring, which include rename method, extract method, move method, parameter addition and parameter removal. For ensuring the validity of the proposed tool, we conduct pretest-posttest experiment on 1 group comprising of 21 subject developers. Each developer has to accomplish 5 tasks namely rename method, extract method, move method, parameter addition and parameter removal. The subject developers have to accomplish 5 tasks by manual refactoring and automatic refactoring using the proposed tool *GreenRefPlus.* The results presented in this work make it evident that the proposed tool assists developers in saving significant time. In addition, the proposed tool *GreenRefPlus* makes the process significantly less cumbersome and also ensures that the final behavior of the code is according to the pre-defined plan.

9 References

[1] Du Bois B., Demeyer S., & Verelst J., "Refactoring-improving coupling and cohesion of existing code," Proc. 11th working conference on reverse engineering, Delft, Netherlands, pp. 144-151, 2004.

[2] Fowler M., Beck K., Brant J., Opdyke W., & Roberts D., Refactoring: improving the design of existing code, Addison-Wesley Professional, Massachusetts, 1999.

[3] Koomen T., & Pol M., Test process improvement: a practical step-by-step guide to structured testing, Addison-Wesley, Massachusetts, 1999.

[4] Beck K., Test-driven development: by example, Addison-Wesley Professional, Massachusetts, 2003.

[5] Guerra E. M., & Fernandes C. T., "Refactoring test code safely," Proc. International Conference on Software Engineering Advances (ICSEA 2007), Cap Esterel, France, pp. 44-44, 2007. https://doi.org/10.1109/ICSEA.2007.57

[6] Wake W. C., Extreme Programming Explored, Addison-Wesley Professional, Massachusetts, 2000.

[7] Janzen D., & Saiedian H., "Test-driven development concepts, taxonomy, and future direction," Computer, vol. 38, no. 9, pp. 43-50, 2005. https://doi.org/10.1109/MC.2005.314

[8] Mens T., & Tourwé T., "A survey of software refactoring," IEEE Transactions on software engineering, vol. 30 no. 2, pp. 126-139, 2004. https://doi.org/10.1109/TSE.2004.1265817

[9] Moonen L., van Deursen A., Zaidman A., & Bruntink M. On the Interplay Between Software Testing and Evolution and its Effect on Program Comprehension, In: Software Evolution, Springer, Berlin, Heidelberg, 2008. https://doi.org/10.1007/978-3-540-76440-3_8

[10] Jaradat A., & Qusef A., "Automatic Recovery of Unit Tests after Code Refactoring," Proc. 2019 International Arab Conference on Information Technology (ACIT), Al Ain, UAE, pp. 202-208, 2019. https://doi.org/10.1109/ACIT47987.2019.8990974

[11] Ge X., & Murphy-Hill E., "Manual refactoring changes with automated refactoring validation," Proc. 36th International Conference on Software Engineering, Hyderabad, India, pp. 1095-1105, 2014. https://doi.org/10.1145/2568225.2568280

[12] Ge X., DuBose Q. L., & Murphy-Hill E., "Reconciling manual and automatic refactoring," Proc. 2012 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, pp. 211-221, 2012. https://doi.org/10.1109/ICSE.2012.6227192

[13] Murphy-Hill E., & Black A. P., "Why don't people use refactoring tools?," Proc. Fifth Workshop on Refactoring Tools, Rapperswil, Switzerlande, pp. 60-61, 2007.

[14] Weißgerber P., Biegel B., & Diehl S., Making "Programmers Aware Of Refactorings," Proc. 1st Workshop on Refactoring Tools, Berlin, Germany, pp. 58-59, 2007.

[15] Campbell D., & Miller M., "Designing refactoring tools for developers," Proc. 2nd Workshop on Refactoring Tools, Nashville, Tennessee, pp. 1-2, 2008. https://doi.org/10.1145/1636642.1636651

[16] Gao Y., Liu H., Fan X., Niu Z., & Nyirongo B., "Analyzing Refactorings' Impact on Regression Test Cases," Proc. 2015 IEEE 39th Computer Software and Applications Conference (COMPSAC), Taichung, Taiwan, pp. 222-231, 2015. https://doi.org/10.1109/COMPSAC.2015.16

[17] Negara S., Chen N., Vakilian M., Johnson R. E., & Dig D., "A comparative study of manual and automated refactorings," Proc. European Conference on Object-Oriented Programming, Berlin, Heidelberg, pp. 552-576, 2013. https://doi.org/10.1007/978-3-642-39038-8_23
[18] Kim M., Zimmermann T., & Nagappan N., "A field study of refactoring challenges and benefits," Proc. the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, Cary, North Carolina, p. 50, 2012. https://doi.org/10.1145/2393596.2393655
[19] Van Deursen A., & Moonen L., "The video store revisited - Thoughts on refactoring and testing," Proc. 3rd Int'l Conf. eXtreme Programming and Agile Processes in Software Engineering, Sardinia, Italy, pp. 71-76, 2002.
[20] Passier H., Bijlsma L., & Bockisch C., "Maintaining Unit Tests During Refactoring," Proc. 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Lugano, Switzerland, p. 18, 2016. https://doi.org/10.1145/2972206.2972223
[21] Rachatasumrit N., & Kim M., "An empirical investigation into the impact of refactoring on regression testing," Proc. 2012 28th IEEE International Conference on software maintenance (ICSM), pp. 357-366, 2012. https://doi.org/10.1109/ICSM.2012.6405293
[22] Campbell D. T., & Stanley J. C., Experimental and quasi-experimental designs for research, Houghton Mifflin Company, Boston, USA, 2015.
[23] I. A. Aljazaery, and A. H. M. Alaidi, "Encryption of Color Image Based on DNA Strand and Exponential Factor," International Journal of Online & Biomedical Engineering, vol. 18, no. 3, 2022. https://doi.org/10.3991/ijoe.v18i03.28021
[24] J. Kh-Madhloom, "Dynamic Cryptography Integrated Secured Decentralized Applications with Blockchain Programming," Wasit Journal of Computer and Mathematics Sciences, vol. 1, no. 2, pp. 21-33, 2022.
[25] H. T. H. Hazim, "Enhanced Data Security of Communication System using Combined Encryption and Steganography," International Journal of Interactive Mobile Technologies, vol. 15, no. 16, pp. 144-157, 2021. https://doi.org/10.3991/ijim.v15i16.24557
[26] H. A. Hassan, "Review Vehicular Ad hoc Networks Security Challenges and Future Technology," Wasit Journal of Computer and Mathematics Science, vol. 1, no. 3, 2022.
[27] Elbaum S., Gable D., & Rothermel G., "The impact of software evolution on code coverage information," Proc. IEEE International Conference on Software Maintenance (ICSM'01), Florence, Italy, p. 170, 2001.

10 Authors

Abdallah Qusef received the Ph.D. degree in computer science/software engineering from the University of Salerno, Italy in 2012. Currently, he is an associate professor in Software Engineering at Princess Sumaya University for Technology in Jordan. His research interests include software maintenance and evolution, empirical software engineering, agile software development, software testing, and e-Business topics.

Sharefa Murad received the Ph.D. degree in computer science/Software Systems and Technologies from the University of Salerno, Italy in 2013. Currently, she is an assistant professor in Computer Science department at Middle East University in Jordan. Her research interests include Software Engineering, Software Visualization, and human-machines interfaces.

Najah Al-Salihi is an associate researcher at the Deanship of Research and Graduate Studies (DRG) and the Department of Education at the College of Humanities at Ajman University. Also, at the Humanities and Social Sciences Research Center (HSSRC), Ajman University, Ajman, UAE. Al Salhi Published Most of his Papers at Professional International Conferences and Scientific Journals.

Eman Shudayfat received the Ph.D. degree in virtual reality and augmented reality from Politehnica University of Bucharest. Currently, she is an assistant professor in department of creative media at Luminus Technical University College (LTUC). Her research interests include the applications of virtual reality in various fields.

Article submitted 2023-02-01. Resubmitted 2023-03-07. Final acceptance 2023-03-09. Final version published as submitted by the authors.