Automatic Recovery of Unit Tests after Code Refactoring

Alaa Jaradat
Department of Computer Science
Princess Sumaya University for Technology
Amman, Jordan
alaamjaradat@gmail.com

Abdallah Qusef
Department of Software Engineering
Princess Sumaya University for Technology
Amman, Jordan
a.qusef@psut.edu.jo

Abstract-- Unit testing allows developers to refactor their code confidently, these tests act as a safety net against producing bugs and provide immediate feedback during the refactoring process and furthermore help developers overcome the fear of change. When performing refactoring, the design of code is changed or restructured according to a predefined plan, after refactoring is applied, the alignment between source code and its corresponding unit tests could be broken which creates a problem that needs to be solved.

This paper introduces an approach in which code refactoring can maintain the integrity of the previous unit tests; the tool called *GreenRef* demonstrates this work. This tool supports an automatic recovery for the unit tests after performing of three particular refactoring types for Java programming language: Rename Method, Add Parameter and Remove Parameter.

The achieved results indicate that *GreenRef* facilitates consistent use of refactoring and unit tests, and save about 43% of the time required to recover broken unit tests manually.

Keywords— Refactoring, Unit Testing, Test-Driven Development, JUnit, Eclipse plug-in, Agile eXtreme Programming.

1. Introduction

Unit testing and Refactoring is an ideal companion developed to be used in conjunction; as Refactoring improves, clarifies and simplifies the design of existing code, and unit testing makes it possible to refactor safely. For example, in Agile eXtreme Programming (XP) methodology [1], where software is developed via incremental Test-Driven Development (TDD) programming: testing and refactoring must be done regularly at each iteration. XP adapted TDD style [2], in which three activities are tightly interwoven: write failed tests, write production code that makes that tests pass and finally refactor to clean up the smells in code written,

Refactoring changes the software system in such a way that it does not alter the external behavior but improves the internal structure [3].

In Practice, the quality of test suites might be affected in the presence of refactoring [4]. For example, Rename Method refactoring introduces inconsistency between the structure of refactored code and its corresponding unit tests. This inconsistency should be solved by renaming the method in all related unit tests.

This study is motivated by the observation that common refactoring can easily and accidentally affect the running of unit tests [5].

Recent studies show that developers often apply refactoring manually despite their awareness of automated refactoring tools. Liu *et. al.* [6] Draws a conclusion using four datasets spanning more than 13,000 developers, and finds that refactoring are frequently performed, almost 90% of refactoring are performed manually without the help of tools and programmers frequently interleave pure refactoring with behavior-modified edits.

We studied the impact of the refactoring on unit tests especially in cases of incomplete refactoring caused by slow and risky manual edits or by automated engines where recent researchers found several limitations and dozens of bugs in their implementations [8].

This paper introduces *GreenRef* tool and addresses the following research questions:

RQ1: Is there a need for a tool for unit tests recovery while refactoring?
RQ2: Can such a tool be built with acceptable performance?

The remainder of this paper is structured as follows: in section 2 we introduce the main concepts used in our study, in addition to related work. Section 3 explains the proposed methodology, and the steps that have been applied while developing *GreenRef.* Section 4 describes our experimental setup, while section 5 presents the results of the experiment. In section 6 we discuss our findings and present threats to validity. Finally, Section 7 concludes this paper.

2. Related work

Zimmermann and Nagappan [8] presented a field study of refactoring benefits and challenges at Microsoft Company through three complementary study methods: a survey, semi-structured interviews with professional software engineers, and quantitative analysis of version history data. A study found that refactoring definition in practice differs from the pure theoretical refactoring; interviewed developers recognized refactoring as a costly and risky task. On the other hand; the quantitative analysis of Windows 7 version history indicating a visible benefit of refactoring.

978-1-7281-3010-1/19/$31.00 ©2019 IEEE

Van Deursen and Moonen [9] analyzed refactoring types in case problems might arise because the original tests need to be modified, Types was divided into categories from A to E according to their effect on unit tests and recovery mechanism. They also proposed the notion of *test-first refactoring*, which uses the existing unit tests as the starting point for finding suitable refactoring practice. Moreover, proposed the idea of a *refactoring session,* capturing a coherent series of separate steps involving changes to both the production code and the test code. This study helped in understanding the interaction between refactoring practices and unit tests.

Passier *et. al.* [10] described different ways in which refactoring can impact the API coverage of unit tests, and developed an Eclipse plug-in that tracks all changes to the program code that are made during refactoring, analyzing their influence on the existing test suite and giving advice to developers on how to update the test suite to migrate it. This Approach applies to all refactoring.

Liu *et. al.* [6] proposed an approach to analyzing refactoring' impact on regression test cases; analyzed why regression test cases failed and the influence of refactoring on software interfaces. Based on the analysis they built a mapping between refactoring and test case failure, used for guiding test case automated recovery tools where test cases are made obsolete by refactoring. The approach was evaluated on five open-source applications. And the Evaluation results suggest that the precision of the approach is higher than 80%.

Rachatasumrit and Kim [11] investigated the impact of refactoring edits on regression tests using version history of Java open source projects, by using refactoring reconstruction analysis and a change impact analysis in tandem, they investigated the relationship between the types and locations of refactoring edits identified by REFFINDER and the affecting modifications and affected tests recognized by the FAULTTRACER change impact analysis. The results on three open source projects show that's, 38% of affected tests are relevant to refactoring and refactoring are involved in almost half of the failed test cases.

3. Research Methodology

This section explains the proposed methodology, which defines the steps that have been applied while developing *GreenRef.*

The goal of the methodology used is to develop a tool that facilitates more consistent and reliable use of refactoring, and unit tests for specific types of refactoring. The research methodology consists of four main steps:

3.1. The Study of the technologies needs to analyze the refactoring practice of the code.

In this step, the study focused on the required technologies to facilitate analyzing refactoring changes and how to configure a Java project to be applied to these projects:

3.1.1 Source Code Format

To facilitate the analyzing source code, the format has to be unified. Code writing is the practice that defers from developer to another, reading java file relies on all files that are formatted in a custom format prepared to be imported at the beginning of *GreenRef* implementation.

For accurate analysis, unified formatting will be applied to a source code automatically each time code modified by the developer. Custom format file designed as XML file prepared in eclipse to be imported at project configuration step.

3.1.2 History of Refactoring Edits

GreenRef is a refactoring-aware code review tool which detects type and location of refactoring edits by comparing two versions of a program. Here comes the need to have always two different versions of the same source code.

For this purpose, *FileSync*[1] eclipse plugin is implemented within *GreenRef* development environment. *FileSync* plugin is a file synchronization tool. The main goal is to keep files outside of Eclipse projects in-sync with Eclipse project files.

3.1.3 Automatic build after applying the refactoring

The primary objective of this study is maintaining unit tests during refactoring edits, this means that reviewing the refactoring has to be a continues process till that end; after developer saved refactoring edits on the affected source code, test suites needed to be inspected.

In this step, automated build for the affected source code Java files will take place using apache ant builder who used effectively to build Java applications as it easily integrated with eclipse.

The Ant build file is considered as the control unit for the proposed approach, as it responds to handle the following processes automatically:

- **Compile changed source code.**

In this step only modified files will be compiled by executing javac task, where you can choose compiler to be used by setting compiler attribute, include Ant runtime whether to include the Ant run-time libraries in the classpath.

- **Run program's unit tests.**

This task runs tests from the JUnit testing framework, and generate Test report summary for each test class under the TestReports directory.

- **Run *GreenRef* recovery tool.**

This task will run the external tool jar file.

3.2 Study for the process of analyzing code refactoring practice.

Analyzing refactoring practices done by a developer is the core step in paper methodology, in this stage, all modified files that have been synced in step one will be analyzed line by line and compared with the previous version to be finalized with applied refactoring types and location. This step breaks down into multiple steps; each will handle a specific task:

[1] http://andrei.gmxhome.de/filesync/index.html

3.2.1 Read all modified files

After refactoring changes took place and the developer saves their work, all modified java files will be synced into an external folder. For more details of how to configure file synchronization process within eclipse IDE

3.2.2 Convert java files into XML format

For easy and quick comparison between code versions, java files converted into XML format with a unified structure. The new file will have a new name as follow: "packageName.className.meta.xml".

3.2.3 Analyze XML Files

Note that the old version of code also will be located with an XML format in this stage both versions will be compared and find the differences between them, and finally record those changes as refactoring in the result file.

3.2.4 Compare Old and New Versions

In this step XML files contents are compared line by line, tracking for the following refactoring types:

- **Rename Method:** The checkup for this refactoring type required the method's parameters list and method body that is matched between two versions.
- **Add Parameter:** The checkup for this refactoring type required that method's name and method body that is matched between two versions, and this is not an overloaded function.
- **Remove Parameter:** The checkup for this refactoring type required that method's name and method body is matched between two versions, and this is not an overloaded function.

3.2.5 Preparing the result file

In this step, all the applied refactoring practices which will be listed in XML format.

3.3 Unit Tests Automatic recovery

In the first step, after refactoring's practice took place and modified files being built; also unit test implemented within the refactored project will be run and tested reports generated in Test Reports folder located under Refactoring folder.

Recovery for broken unit tests will go through some steps; each step will be discussed briefly.

3.3.1 Read all Test reports:

The first step of recovery is to load all unit tests reports to be analyzed in the next step, where all generated reports will be located under the following directory and been deleted before each run for unit tests:

Refactored project base directory\\Refactoring\\Test Reports

3.3.2 Analyze the testing report:

The information that is returned from the testing report reveals why the test case failed when a failure occurs. In this step the report will be analyzed to find a relation between the refactoring practices and the failure of unit tests.

Figure 1 illustrates the classification of test case failure with conditions that caused each failure [6]:

Research current scope works in the area of failures that caused by errors, specifically error of type "*No Such Method*".

Sample report for broken unit test shown in Figures 2 and 3, Information analyzed from the report will be valuable at recovery step; test class detailed will help in the step of applying recover mechanism (Step four), and code under test information will help in refactoring analyses step (Step three).

Another challenge in test report analysis is to detect the data type method parameters, as it's not using the conventional names as in java source code; for example (I) symbol in testing report refers to (int) data type, also object data type will be started with "Ljava/lang/". In Table II below data types designations used in test reports and what its correspondence type in Java:

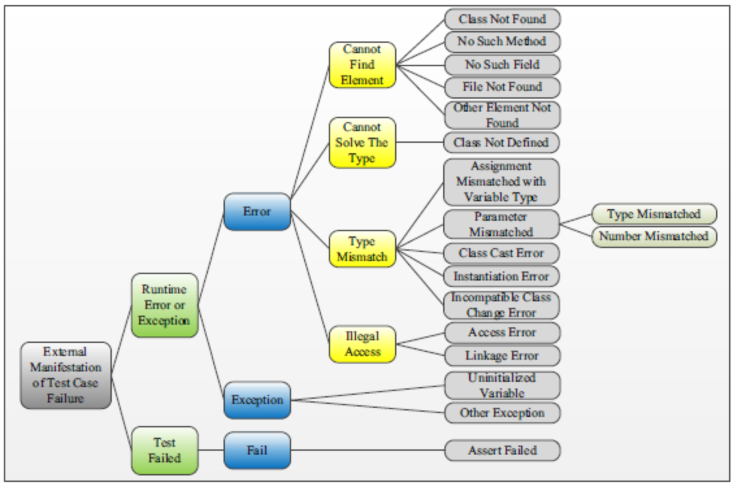
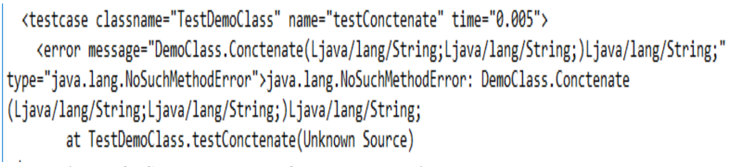
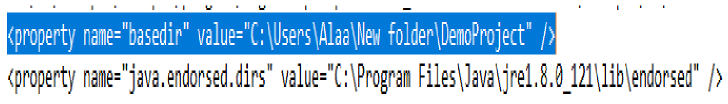


Figure 1: Classification of Unit Test Failures and Conditions That Cause Failure[2]



```
 <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)
```

Figure 2: 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" />
```

Figure 3: Sample Report for Broken Unit Test: basedir Property

Analyzing step will give information about the broken unit test, error type, and other information related to the code under test. See Table I.

Table I: Unit Test Report: Analysed Data from Report

TestCase Tag's Property	Analysed Data	Value
Classname	Name of testing class	TestDemoClass
Name	Name of the testing method	testConctenate
Error Tag's Property	**Analysed Data**	**Value**

[2] Gao et al., 2015

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 II: Unit Test Report: Data Types in testing report

Data Type in Testing Report	Data Type in java
I	int
Z	boolean
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,

3.3.3 Analyze the testing report:

After analyzing the test report; errors of type "No Such Method" means that the testing method is trying to call a method that does not exist. That is, the class under test does not have that method definition. From refactoring perspective *GreenRef* suppose that the class under test was refactored recently, specifically rename a method or add a parameter or remove parameter refactoring was applied to the missing method.

In this stage refactoring, result.xml file (generated at step two) will be analyzed to find the refactoring practice that caused an error in this test method.

Details of refactoring practice located in the result.xml file, where refactoring type demonstrate the form of edits applied to the source code, Package Name, Class Name, and Method Name demonstrate the location of refactoring. Finally Original Value, Modified Value used as a history for edit.

The last step in refactoring analysis step is to match unit test error figured in the previous step with its related refactoring practice. Once this step has been done, automatic recovery will be ready to be performed in the next and last step.

3.3.4 Automatic Recovery for unit test:

After successfully determining unit test failure error and the specific refactoring practice caused this error, recovery action has to be performed as a final step in this approach. The first step is to match between failed unit test details and refactoring practice details, see Figure 4.

Recovery mechanism differs for each refactoring type, below more description of how *GreenRef* tool recovers unit after each refactoring type, as Figures 5-7 below:

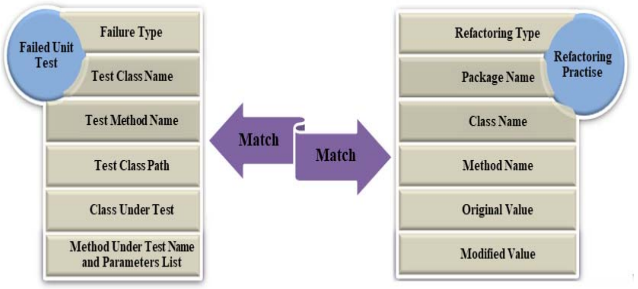


Figure 4: Unit Test Recovery Process

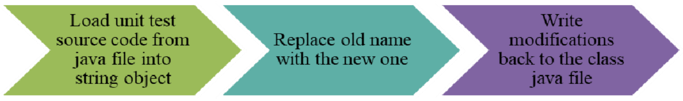

Load unit test source code from java file into string object
Replace old name with the new one
Write modifications back to the class java file

Figure 5: Unit Test Recovery: Rename Method Refactoring

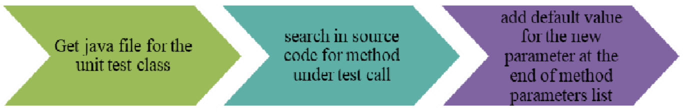

Get java file for the unit test class
search in source code for method under test call
add default value for the new parameter at the end of method parameters list

Figure 6: Unit Test Recovery: Add Parameter Refactoring

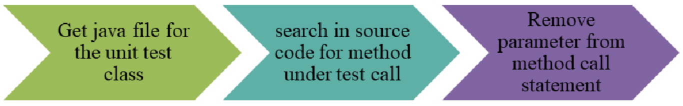

Get java file for the unit test class
search in source code for method under test call
Remove parameter from method call statement

Figure 7: Unit Test Recovery: Remove Parameter Refactoring

4. EXPERIMENT

The one-group pretest-posttest experiment was conducted to verify whether *GreenRef* can maintain unit tests validity during code refactoring practices, and measure time-saving, usability of the refactoring.

In a one-group pretest-posttest pre-experimental [12], only one group is tested. Instead of having a controlled group as in the typically controlled experiment, the experimental group is subjected to an extra test before the experiment. This test considered as a baseline for the experimental measurements. In pretest and posttest steps, the subjects are measured in terms of dependent variables. This type of experiments allows researchers to report on facts of real user-behavior.

For research questionnaires, a list of questions was employed to investigate the level of selected subjects.

Experiment objective is to measure performance (Time Saving and Usability) enhancement gained by applying *GreenRef* tool during code refactoring process for three specific refactoring types: Rename method, add parameters and remove parameters. This target could be accomplished by measure time required to refactor code and then trying to recover broken unit tests manually vs. using proposed recovery tool.

Pretest Design: for the pretest, a total number of five questions were chosen and can be seen in Table III.

Table III: Pre-test Questionnaire

ID	Question	Answer Format
Q1	Coding professionalism level	Check the box for his level
Q2	How do you define refactoring process	The subject had to write text for the definition he knows
Q3	Have you ever apply to refactor on your code	Yes or no
Q4	Do you prefer to apply manual or automated refactoring	Choose manual or automated
Q5	Select types of refactoring you apply frequently	This question to analyze the popularity of refactoring types used by subjects, we focused on our related types: Rename method, Add Parameters and Remove Parameters

Posttest Design: after the subjects have completed their assignments, they have to answer two questions as a posttest; to measure subject satisfactions regards the proposed tool usefulness and ease of use.

A. Projects:

To perform this experiment, two Java applications with simple functionality were selected according to the following requirements:

- The project should be written in the Java language.
- It should have unit tests implemented.
- It should not be a complicated application in size and functionality.

The selected applications were simple and easy to learn java application, suitable for the development skills level of the subjects.

B. Assignments:

To challenge our subjects with *GreenRef*; two programming assignments were created. The first assignment performed using manual refactoring and manual recovery techniques, while the second one was performed using the proposed tool. The experiment was conducted at the Princess Sumaya University for technology on Bachelor students in fourth year majoring in Software Engineering with different programming levels. All the students had knowledge of both software development and testing.

Subjects were asked to apply specific refactoring edits on two different Java programs and try to recover broken unit tests manually when needed.

Refactoring will be applied manually two times, first try will be applied on a system with no recovery tool where the developer has to recover affected unit test manually, where in the second, unit tests will be automatically recovered.

Each assignment contains three tasks applied to the two applications mentioned earlier.

The main technique represented in this experiment has two levels:

- *Manual recovery:* subjects recovered the broken unit tests affected by refactoring changes manually by running system unit tests and fix the broken ones.
- *GreenRef Auto recovery:* subjects have nothing to do to recover broken tests after refactoring changes have done affected unit tests will be recovered automatically.

Each refactoring type will be performed twice, first time with manual recovery mechanism like searching for broken unit test and try to reflect refactoring changes on it, and the second trial subject will do refactoring with auto recovery mechanism applied using *GreenRef* tool. For performance measurements, the time required to accomplishing task was recorded.

C. *Subject's profile*: experiment done by twenty-one (21) subjects with different levels in java programming. To get accurate results from our experiment, subject's Java programming background in Code Refactoring was analyzed, before doing the experiment subjects had to a pretest questionnaire described in Table III above.

Pretests Results

Knowledge about Refactoring: Unfortunately; four subjects out of twenty-one were correctly define Refactoring technique. But that was not a surprise; since according to [7]; programmers frequently interleave pure refactoring with behavior-modified edits.

Refactoring Skills: Next question was whether subjects do refactoring practices before, only two of them never practice it at all.

Manual vs. Automated Refactoring: It is important to know how subjects (who was familiar with refactoring practices) apply refactoring practices; manually or using refactoring tools supported by IDEs. eight subjects applied manual refactoring, while six subjects applied automated refactoring.

The popularity of refactoring types applied: subjects answered for the refactoring types they applied frequently, eight subjects answered rename method, four answered add parameter, and six answered remove parameter while five answered other types.

Posttest Results

After the experiment was done, participants were asked about their opinion whether the proposed tool helped them in refactor code safely without invalidating the unit test. Participants gave clear indication that *GreenRef* was indeed helpful during the development tasks. We also asked the participant whether the proposed tool helped them to refactor unknown code confidently and all answers were in the range between four and five. That is another indication that *GreenRef* helps during code maintenance. The final question asked to subject was whether they found the assignment too difficult, answers were between four and five.

5. RESULTS AND EVALUATION

The results from the experiment will be analyzed to discover how *GreenRef* will control a success refactoring process with the existence of unit tests.

The results achieved by the subjects for each task within each assignment were reported in Tables (IV, V, VI) below, tables also report a calculated difference between the result achieved by the subject performing a task using *the GreenRef* tool and the result achieved by the same subject performing the same task manually.

Table IV: Experiment Results for Task One: Rename Method Refactoring

Subject ID	**Assignment#1 (Refactoring with Manual Recovery)**	**Assignment#2 (Refactoring with *GreenRef* Auto Recovery)**	**Saved Time (in seconds)**	**Saved Time %**
Task No.1: Rename Method Refactoring (Time required in seconds)				
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 V: Experiment Results for Task Two: Add Parameter Refactoring

Subject ID	**Assignment#1 (Refactoring with Manual Recovery)**	**Assignment#2 (Refactoring with *GreenRef* Auto Recovery)**	**Saved Time (in seconds)**	**Saved Time %**
Task No.2: Add Parameter Refactoring (Time required in seconds)				
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 VI: Experiment Results for Task Three: Remove Parameter Refactoring

Subject ID	**Assignment#1 (Refactoring with Manual Recovery)**	**Assignment#2 (Refactoring with *GreenRef* Auto Recovery)**	**Saved Time (in seconds)**	**Saved Time %**
Task No.3 : Remove Parameter Refactoring (Time required in seconds)				
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%
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%**

According to readings in the above tables, the following results have been observed:

All subjects took less time to apply to refactor with automated recovery to unit tests than doing it manually.

1. Applying rename method refactoring using *GreenRef* saved time with an average of 19 seconds which less than the time required to do the task manually with a percent of 53.4%.
2. Applying to add parameter refactoring using *GreenRef* saved time with an average of 7.2 seconds which less than the time required to do the task manually with a percent of 34%.
3. Applying to remove parameter refactoring using *GreenRef* saved time with an average of 11.4 seconds which less than the time required to do the task manually with a percent of 44.4%.

6. Discussion

This section will relate the results of the experiment to the research questions that we have presented in the introduction. Then, a number of threats to validity will be discussed.

RQ1: Is there a need for a tool for unit tests recovery while refactoring?

According to [4, 13] studies have shown that even refactoring practices are behavior preserving, they potentially invalidate unit tests, and even small changes on source code can significantly affect unit tests. Unit tests recovery after refactoring is an unsolved problem due to many reasons:

1) Lack of automatic recovery tools.
2) Lack of traceability between source code and unit tests; which may cause by bad naming convention.
3) Lack of time for code maintenance.

Since those problems can't be solved at once; tool-support that automated recovery mechanism can solve these issues, it is in this context that *GreenRef,* eclipse plug-in for Java applications, has been developed. This tool maintains unit tests validity while refactoring, thus the answer to this question is yes there is a need for unit test recovery tool while refactoring.

RQ2: Can such a tool be built with acceptable performance?

GreenRef has been designed as an eclipse plug-in. Many standard tools and libraries used like *Apache ant* for Java application building, unit test running. *FileSync*, Java plug-in, which used for files synchronization. During development, *GreenRef* runs in the background and does not cause a noticeable performance impact for the developer. Usability wise, it comes as no surprise then that majority subject said it was easy to use.

According to results measured in the previous section GreenRef decrease development time with average percent equals to 43%. And this is an indication that the answer for the second question is yes, a proposed tool built with acceptable performance in time-saving and usability manners.

7. Threats to Validity

This research has its limitations and outcomes related to the experiment, more details to follow:

- According to Internal Validity, while designing the experiment, the author tried to choose systems with simple architecture and functionality with straightforward and clear tasks and fully described steps. Performing experiment on a real and large application with a large number of LOC and many unit tests classes for sure will give more accurate and noticeable results but again subjects programming skill was considered.
- According to the External Validity, before starting the experiment, subjects had to complete survey questioner to explore their programming skills level. Author noticed that the selected subjects had different levels of Java programming skills and personal abilities. In addition to the different background in software engineering between subjects in the same class; for example, definition of code refactoring was not clear for all of the selected subjects according to questioner only 19% of subjects know the right definition of code refactoring, and around 24% are never heard about refactoring technique before. To avoid this threat session presenting refactoring was arranged.

8. Conclusion

Refactoring and unit testing are valuable techniques for code quality and safety improvement, both are considered as an ideal companion; developer must not start refactoring before ensure that all unit tests are passed and refactoring process not considered successful unless it handles all changes that may cause break for related unit tests. *GreenRef* is an automated approach that facilitates the more consistent use of refactoring and unit tests and subjected to keep unit tests passes while refactoring.

GreenRef is a plug-in for Eclipse IDE to guarantee ease of use, no user interface required all recovery actions will be done in the background without involvement from the developer. *GreenRef* is under development, and for now, it's only implemented recovery for three types of refactoring: Rename Method, Add Parameters, and Remove Parameters.

References

[1] Wake, W. C. (2000). Extreme Programming Explored. *William C. Wake*.

[2] Janzen, D., & Saiedian, H. (2005). Test-driven development concepts, taxonomy, and future direction. *Computer*, *38*(9), 43-50.

[3] Fowler, M., & Beck, K. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[4] Mens, T., & Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on software engineering*, *30*(2), 126-139.

[5] Moonen, L., van Deursen, A., Zaidman, A., & Bruntink, M. (2008). On the Interplay Between Software Testing and Evolution and its Effect on Program Comprehension.

[6] Gao, Y., Liu, H., Fan, X., Niu, Z., & Nyirongo, B. (2015, July). Analyzing Refactoring' Impact on Regression Test Cases. In *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual* (Vol. 2, pp. 222-231). IEEE.

[7] Murphy-Hill, E., Parnin, C., & Black, A. P. (2012). How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, *38*(1), 5-18.

[8] Kim, M., Zimmermann, T., & Nagappan, N. (2012, November). A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (p. 50). ACM.

[9] Van Deursen, A., & Moonen, L. (2002, May). The video store revisited–thoughts on refactoring and testing. In *Proc. 3rd Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering* (pp. 71-76). Citeseer.

[10] Passier, H., Bijlsma, L., & Bockisch, C. (2016, August). Maintaining Unit Tests During Refactoring. In Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (p. 18). ACM.

[11] Rachatasumrit, N., & Kim, M. (2012, September). An empirical investigation into the impact of refactoring on regression testing. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on* (pp. 357-366). IEEE.

[12] Campbell, D. T., & Stanley, J. C. (2015). *Experimental and quasi-experimental designs for research*. Ravenio Books.

[13] Elbaum, S., Gable, D., & Rothermel, G. (2001, November). The impact of software evolution on code coverage information. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)* (p. 170). IEEE Computer Society.