

## **ABSTRACT**

MEHTA, DIVYA. A Study of Refactoring Projects Implemented in an Object-Oriented Design Class. (Under the direction of Dr. Edward F Gehringer).

Changing an existing software system is a constant process. Refactoring is a technique that makes these changes simple by altering the internal structure of the code without modifying the external behavior. This work provides an extensive study of refactorings in open-source software projects performed by the students in object-oriented design class. The manual inspection of the revisions of an existing code base provides us interesting results. It shows that students catch around 62% of the refactorings correctly and most common among them are Move Method, Rename Variable, Add Method, etc. The most commonly missed refactorings are Introduce Explaining Variable, Hide Variable, and Extract Method.

We present an approach for the students to perform refactorings in a more reliable way. Our approach defines two steps: first to acquire sufficient knowledge about refactorings before its implementation and second to follow the test-driven refactoring approach. Since testing is complementary to refactoring, this approach aims to minimize the missing refactorings and to maximize the test coverage.

A Study of Refactoring Projects Implemented in an Object-Oriented Design Class

by  
Divya Mehta

A thesis submitted to the Graduate Faculty of  
North Carolina State University  
in partial fulfillment of the  
requirements for the degree of  
Master of Science

Computer Science

Raleigh, North Carolina

2012

APPROVED BY:

---

Dr. Edward F Gehringer  
Committee Chair

---

Dr. Emerson Murphy-Hill

---

Dr. Laurie Williams

## **DEDICATION**

To my parents, Poonam and Prashant...

## **BIOGRAPHY**

Divya Mehta was born on September 5, 1984 in Patiala, Punjab, India. She finished her schooling in 2000. She received her degree of Bachelor of Technology in Computer Science from Punjabi University, Punjab, India in 2006. She worked as Software Engineer with Infosys Technology Ltd., Chandigarh, India from September 2006 to July 2008, and then worked as Senior Systems Engineer at Infosys Technology Ltd. itself from July 2008 to April 2010. She worked as Software Engineering Intern at Red Hat, Raleigh, North Carolina from May 2011 to August 2011 and also did a Co-op at NVIDIA, Santa Clara, California as GPU Architecture Intern from September 2011 to December 2011. She joined the graduate program in Department of Computer Science at North Carolina State University in Fall 2011. She has been working with Dr. Edward F Gehringer on her master's thesis since April 2011. This period provided her with lot of challenges and opportunities that helped to shape her future career. With the defense of this thesis, she is receiving the degree of Master of Science in Computer Science from North Carolina State University.

## ACKNOWLEDGMENTS

I would like to extend my thanks to the following people without whom it would not have been possible for me to achieve this work.

I owe my sincere thanks to Dr. Edward Gehringer, my advisor, for his all time support and guidance. I am very grateful to him for his valuable time, continuous advice, and his feedback throughout this thesis that helped me to improve my work. I learned a lot from our weekly meetings and his insights helped me to generate creative ideas in my mind.

I would like to thank my other committee members, Dr. Emerson Murphy-Hill, for helping me to work in the right direction and for all his guidance and discussions throughout my thesis work. Dr. Laurie Williams, for being on my committee and providing her inputs and valuable feedback about the work.

I thank my parents, for always being my source of inspiration and motivation. They always encouraged me to pursue my desires in an extraordinary way. I thank my sister Poonam, and my brother Prashant for their all time support and help.

I thank all my friends for their advice and cooperation at every moment.

Finally, I am thankful to God, for granting me the capability, skills and opportunity that made this possible.

## TABLE OF CONTENTS

LIST OF TABLES .....	vi
LIST OF FIGURES .....	vii
LIST OF ABBREVIATIONS .....	viii
Chapter 1 – Introduction .....	1
Chapter 2 - Related work .....	4
2.1 Refactoring .....	4
2.2 Code metrics for refactoring.....	4
2.3 Survey of software refactoring and tools.....	5
Chapter 3 – Study objective and approach .....	7
3.1 Study objective .....	7
3.2 Study approach .....	7
3.2.1 Identification of bad smells using code metrics .....	7
3.2.2 Formalization of refactoring projects.....	8
3.2.3 Submission of projects .....	8
3.2.4 Manual inspection of submitted projects .....	9
3.2.5 Assumptions.....	10
Chapter 4 – Findings .....	11
4.1 How did students perform refactorings? .....	11
4.2 What did students miss? .....	26
4.2.1 Missing refactorings.....	26
4.2.2 Why did students miss refactorings? .....	28
4.3 Other observations.....	31
4.4 How should students refactor the code? .....	33
4.4.1 Refactoring approach .....	33
4.5 Limitations.....	37
Chapter 5 – Future work and Conclusion .....	38
5.1 Future work .....	38
5.2 Conclusion.....	38
REFERENCES .....	40
APPENDICES .....	42
Appendix A .....	43
Appendix B.....	44
Appendix C.....	45
Appendix D .....	51
Appendix E.....	53
Appendix F.....	54
Appendix G .....	55

## LIST OF TABLES

Table 1: Number of refactorings in each project .....	13
Table 2: Refactorings implemented by students .....	13
Table 3: Number of instances of each refactoring in OSS projects .....	14
Table 4: Category of refactorings .....	18
Table 5: Refactorings for specific code smells .....	18
Table 6: Long method code smell.....	19
Table 7: Unreasonable method name code smell .....	20
Table 8: Unreasonable variable name code smell.....	21
Table 9: Method with two responsibilities and duplicate code.....	22
Table 10: Code metrics for classes .....	23
Table 11: Missing refactorings in each project.....	27
Table 12: Most commonly used and missed refactorings.....	29
Table 13: Test case scenario for projects .....	31
Table 14: Survey questionnaire .....	43
Table 15: Summary of survey results .....	44
Table 16: Survey results.....	45
Table 17: List of OSS projects in Expertiza in fall '12 .....	51
Table 18: Revised branch statistics for each project.....	54

## LIST OF FIGURES

Figure 1: Total refactorings in floss and root-canal refactoring projects.....	15
--------------------------------------------------------------------------------	----

## LIST OF ABBREVIATIONS

1. OSS Open Source Software
2. OOD Object-Oriented Design
3. DRY Don't Repeat Yourself
4. DECOR DEtection and CORrection
5. LOC Lines Of Code
6. TDD Test Driven Development
7. XP Extreme Programming

## Chapter 1 – Introduction

“*Refactoring* is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure” [1]. When software is delivered and deployed, it is important to maintain it for future changes. Changes can be adding a new feature or deleting or modifying an existing feature. Refactoring makes these changes easy by improving the design of the application, hence increasing the developer’s productivity.

Refactoring is an essential part of agile methodologies, such as Extreme Programming, which follows an incremental development process. It can be carried out successfully by following certain sequential steps. First, identify the areas in the code where refactoring is required. Second, determine different refactoring types that should be applied to remove that bad code. Third, implement those refactorings, and fourth, run the test cases to guarantee that external behavior is still preserved although internal structure changes. Identification of what is called ‘bad smells’ (i.e., areas in the code that should be reorganized in a way that makes the future changes easy to implement) is not as easy as it is said. It requires good amount of knowledge, effort and time to identify these code smells. Kent Beck and Martin Fowler proposed the way to look for these suitable spots in the code where refactorings should be applied [1]. Fowler also provided comprehensive details of 72 possible refactorings that can be used to eliminate these code smells.

Our work presents the study of refactoring projects done by the students in NCSU’s CSC/ECE 517 (Object Oriented Languages and System) class. We consider the refactoring projects carried out on an open-source *Expertiza* application. *Expertiza* is a peer-review

system where students can submit their work and peer review learning objects, e.g., prose articles, code, exercises, etc. [4]. Open-source software is widely used in software industries and academia. It allows the software developers around the world to contribute by volunteer participation.

The benefits of engaging students in open source software [5,6] are many and some of these are:

- Open-source software provides them good learning opportunities and real life experiences on large-scale projects.
- It allows them to learn software engineering practices more precisely.
- It shortens their learning curve when they work in software companies and improves their professional communication.
- It also serves the open-source community.

Expertiza has been developed since 2005 and various new features keep on being added to the project. The need to improve its code is obvious, to improve the understandability and to increase the developer productivity. As a project in the course, students worked on different refactoring projects in Expertiza. We carry out extensive analysis of these projects in our study. We compare the revised code with its previous version from the repository and identify different refactorings performed by the students. This work specifies the details of how students performed refactorings, what did they do well, which refactorings did they use the most and which ones they missed. Not surprisingly, most refactorings that are examined are low-level or medium-level refactorings. The code could be further improved by higher-level analysis. So, our work suggests a refactoring approach that students should follow to

perform refactorings in a more reliable way. We also surveyed the students about their experiences with refactoring which helps us to analyze what is difficult for the students to do well. The survey questionnaire and summary of survey results can be found in Appendix A and B.

The rest of the thesis is organized as follows. Chapter 2 discusses the related research work of refactoring, code metrics and survey of refactoring. Chapter 3 explains the study approach to evaluate these refactoring projects. Chapter 4 presents the comprehensive results of our study. Chapter 5 specifies some of the limitations for this study and future work related to it. Finally, Chapter 6 concludes the thesis.

## **Chapter 2 - Related work**

We provide an overview of the related work in refactoring, code metrics for refactoring, and survey of software refactoring and tools.

### **2.1 Refactoring**

Programmers often perform refactoring in software projects to carry out perfective and corrective maintenance [3]. Refactoring became popular when Fowler formally defined the term and illustrated 72 different refactorings [1]. Robert and Brant were the first to implement the refactoring engine [15]. Mens and Tourwé explained the terms refactoring and restructuring in the context of reengineering and provided an extensive survey of software refactoring [13]. Murphy-Hill et al. studied eight different sets of data and concluded that programmers frequently perform floss- refactoring where they include any kind of program changes or bug fixes along with refactoring of code. He also mentioned that refactorings are not well supported by existing refactoring tools [11].

### **2.2 Code metrics for refactoring**

Several metrics have been proposed to identify candidates for refactoring. Simon et al. used object-oriented concepts to find out the bad smells within the system and recommended refactorings [8]. Their study used distance-based cohesion metrics to identify possible refactorings within the software code. Moha et al. studied different strategies to detect code smells within the system including manual inspection, metric-based approach, visualization techniques, language, and architectural consistency checkers. He also proposed a method called DECOR that specifies the steps to detect the code and design smells in the system [9].

But some studies realized that along with code metrics, manual inspection is also required to find candidates for refactoring [7]. Pizka studied the medium scale commercial project REX. The team decided to improve its code base by tool-supported refactoring. They have realized that only code metrics were not sufficient to identify the candidates for refactoring, it has to be complemented with significant manual inspection [10]. Stroulia and Kapoor studied metrics of a system before and after performing software refactorings. They quantitatively showed how these metrics support improvements in code and system design and thereby showed high cohesion and low coupling for the classes [7].

### **2.3 Survey of software refactoring and tools**

We found from the survey that students did use refactoring tools available within the development environment to some amount. Pizka in his study found that many important refactorings were not applied because of the absence of tool support. In order to carry out large-scale refactoring, tool support should be present [10]. Murphy-Hill et al. identified that students and experienced programmers prefer to refactor manually rather than use tool [11].

Dig et al. implemented an algorithm, which is used as an Eclipse plug-in to automatically detect the refactorings done in the Java code. It detects the refactorings from two revisions of the code base by using syntactic and semantic analysis. This algorithm works with an accuracy of 85% to detect seven types of refactorings [12]. Refactoring prevents the programmer from spending more time on extending the system and correcting bug fixes. Besides this, M Kim et al. showed how refactoring can make the programmer's life more difficult by increasing bugs in the system [14]. These defects may be due to the incomplete refactorings or partial use of tools.

Hence, researchers have uncovered many areas of software refactoring. They studied code bases revisions, found statistics of code metrics used, explored ways to identify bad smells in code and design, programmers tool usage, refactoring benefits in the long run etc. Unlike all these studies, our work evaluates the code base revisions 100% manually to fetch precise and correct results. Although many studies have also examined open-source project code written by programmers, unlike them we emphasize on refactorings performed in the student-written code in open-source project.

## **Chapter 3 – Study objective and approach**

In this chapter, we provide our study approach for the refactoring projects in the Object-Oriented Design class. We specify what metrics are used to identify bad smells in the code, how we formalize projects, and how we analyze code using code revisions from the repository.

### **3.1 Study objective**

The main objective of this work is to extensively study refactoring projects in class to see what students could have done better and how they can be guided to do better in the future. We also suggest some improvements and the approach they should follow while refactoring code.

### **3.2 Study approach**

We selected the open-source application Expertiza, as the subject of our study. It consists of about 150 classes, and is implemented in Ruby on Rails. In this application, we inspected bad smells in the code using lines of code (LOC) as our code metric and formalized the findings into projects to be worked for. Each student team signed up for a project, refactored it, and submitted it via a github repository. We examined these refactoring project versions. The steps are explained below.

#### **3.2.1 Identification of bad smells using code metrics**

Although various tools like Rhodi, reek, etc., for Ruby are available to automatically detect some code smells; but we often need manual investigation to find others. So, in our study we followed manual approach to detect bad smells. Large controller and model classes were

identified as candidate classes for refactoring. Classes of more than 200 LOC were examined. The code smells found in the system are:

- Large classes
- Long methods
- Duplicate code
- Complex code
- Spaghetti code
- Unreasonable names

Using this technique, we found 43 candidate classes that should be refactored to improve system design and reusability.

### **3.2.2 Formalization of refactoring projects**

Before presenting the projects to the students, we combined related classes into projects, occasionally adding to the agenda a bug fix or a small extension. Some projects involve refactoring only a single large file of several hundred LOC, whereas others involve a bundle of related classes. An example of a project description given to students is shown in Appendix E. So, following this approach we came up with the total of 13 refactoring projects. The complete list of projects is given in Appendix D.

### **3.2.3 Submission of projects**

The projects were allocated to the students as part of their OSS project assignment. Students in a team of 2-3 signed up for a project and the timeline was of 3 weeks for submission, followed by a review process. They needed to follow the steps below:

1. Fork the latest Expertiza code to their repository (using github).

2. Create a new branch with any reasonable name.
3. Commit and push the work periodically.
4. After submitting the code, submit a pull request (a *pull request* allows one to push the code to github and let other people review the code and discuss the changes)

Testing is also mandatory for any refactoring project. Refactoring requires rigorous testing to ensure the consistency of external behavior of the system. Students should also add new tests in the project if it involves any additions or changes while refactoring; else they need to run the existing test suite. Therefore, students need to push new tests for each project along with other changes.

### **3.2.4 Manual inspection of submitted projects**

We study the commits for each project, where each commit may include more than one file. It shows the total of 164 commits in all the projects. Appendix F shows the revised branch statistics with total commits, total additions and deletions, etc. of each refactoring project.

To evaluate the projects, we compared the master branch of the Expertiza repository with the commits made by the students in their committed branch. The command ‘git diff’ allows us to look for all the additions, deletions or changes made in a file. In our study, we found 67 (this count contains only distinct files even if they have been changed by more than one project) total files were changed in the repository, and 16 new files were added. We saw that the number of files affected is quite a bit larger than the number of candidate refactoring classes, which were 43.

### **3.2.5 Assumptions**

The assumptions followed in this study include; 1) Students implemented all the refactorings identified by them. 2) We speculated the reasons for missing refactorings: some refactorings are harder to identify, some are harder to implement, and some are due to lack of knowledge (also evident from survey results). 3) Students did not follow a test-driven approach while refactoring (also evident from observation of test cases).

## Chapter 4 – Findings

The rigorous analysis of 13 refactoring projects done by the students in class provides us substantial data about how did they perform refactorings and what did they do well. Section 4.1 shows the details of refactorings performed by students in each project. Section 4.2 mentions the missing refactorings that students could have included in their work. Section 4.4 provides the suggestions that students should follow while performing refactoring activities. We also mention the refactoring approach and recommendations to improve them.

### 4.1 How did students perform refactorings?

To understand how students have performed refactorings and what types of refactorings they have implemented, we first see the total number of refactorings done by students in each project. Table 1 shows the number of refactorings implemented in each project.

From Table 1 we can calculate the *average number of refactorings* performed by students in each project. On calculation, we found that on an average each project performs  $171/13 = 13.15$ , i.e., approximately 13 refactorings. Although the average number of refactorings that students implement seemed to be adequate, considerable amount of time and effort is needed to identify and implement these refactorings. From survey results summary mentioned in Appendix B, we found that the majority of students who participated in the survey (9 out of 12 of them) found it more difficult to identify the refactorings than to perform the refactorings.

Some projects have merely 3 to 8 refactorings, whereas others have more than 20 refactorings. This wide difference in the refactoring count among the projects is due to the reason that each project is of different scope; i.e., some are purely refactoring projects called

as *Root-Canal Refactoring* whereas others include functional changes or bug fixes along with some scope of refactoring called as *Floss Refactoring* [11]. From Table 1, we see that projects with a smaller or average number of refactorings, particularly in the range of 3–10, are mostly floss refactoring projects, viz., E 202, E 204, E 205, E 211, E 212; whereas the projects with a large number of refactorings are generally root-canal refactoring projects, viz., E 201, E 203, E 209, E 210, E 216 and E 219. Besides this, some floss refactoring projects also have a good amount of refactorings, e.g., E 207 and E 208, which have 13 and 23 refactorings respectively.

Table 1 also shows that total number of floss refactorings is 58, which is 34% of total refactorings whereas total root-canal refactoring count is 113, i.e., 66%. So, it is evident that although floss refactoring projects are more 7/13, the number of floss refactorings is quite less as compared to root-canal refactoring. This is shown in Figure 1.

We found that students implemented 14 different refactoring types in their projects, shown in Table 2.

**Table 1: Number of refactorings in each project**

<b>Refactoring Project</b>	<b>Floss / Root-Canal Refactoring Project</b>	<b>Number of Refactorings</b>
E201-Break up assignment.rb	Root Canal	27
E202-Refactor assignment controller	Floss	10
E203-Refactor AssignmentParticipant	Root Canal	13
E204- Remove Individual Assignments	Floss	1
E205 – Users	Floss	7
E207- Questionnaires and advice	Floss	13
E208- Changes to permissions	Floss	20
E209- Refactor SignUp Code	Root Canal	15
E210- Refactor review_mapping_controller	Root Canal	16
E211- E-mailer improvements	Floss	4
E212- WikiSpider	Floss	3
E216- Refactor TeamController and ParticipantChoicesController	Root Canal	11
E219-Miscellaneous Controller CleanUp	Root Canal	31

**Table 2: Refactorings implemented by students**

<b>Refactorings in student's code</b>	<b>Category</b>
Add Class (AC) *	L
Delete Unused Class (DC) *	L
Extract Class (EC)	M
Add Method (AM) *	L
Delete Unused Method (DM) *	L
Extract Method (EM)	M
Move Method (MM)	L
Rename Method (RM)	L
Hide Method (HM)	L
Delete Unused Variable (DV) *	L
Rename Variable (RV)	L
Hide Variable (HV)	L
Add Parameter (AP)	L
Replace Conditional by using Database Field Value (RCFV)*	H

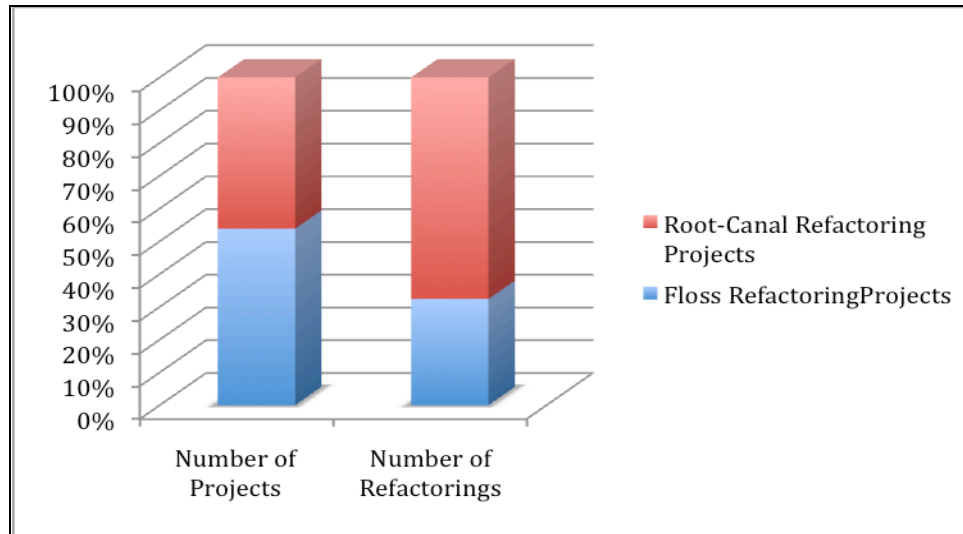
**Table 3: Number of instances of each refactoring in OSS projects**

	A C	D C	E C	A M	D M	M M	E M	R M	H M	D V	R V	H V	A P	RC DF
E 201	3		3			21								
E 202				7		1	2							
E 203			1	1		6		3	2					
E 204														1
E 205	1		1	1	1	1	2							
E 207	5		1	3		3		1						
E 208											20			
E 209			3			12								
E 210				2	4	5	1	1		1	1	1		
E 211				1									3	
E 212							3							
E 216		2		2	3	2		2						
E 219			2	4	3	11		10			1			
<b>Total</b>	<b>9</b>	<b>2</b>	<b>11</b>	<b>21</b>	<b>11</b>	<b>62</b>	<b>8</b>	<b>17</b>	<b>2</b>	<b>1</b>	<b>22</b>	<b>1</b>	<b>3</b>	<b>1</b>

Common Refactorings

Average Refactorings

Rare Refactorings



**Figure 1: Total refactorings in floss and root-canal refactoring projects**

These are the major refactorings carried out by students in their OSS projects. Opdyke also mentioned these refactorings as low-level, such as creating and deleting a class, variable or function; changing variables and functions attributes like visibility scope, arguments; or moving methods or variables between classes [1]. We can see the number of instances of each refactoring type carried out in each project in Table 3.

In Table 2, we marked some refactorings with asterisk, since these are not mentioned in Fowler's 72 refactorings. We realized the need for these refactorings in our code,

1. *Delete unused class, Delete unused method and Delete unused variable* come under the same category. This refactoring is applied when the class, method or variable is no more used in the system. Changes in the internal design make the class or method or variable unusable in the application.

2. *Add Class and Add Method* is used when internal design is changed such that one change leads to addition of other classes or methods. For example, since our application uses model-view-controller architecture, when one controller file is extracted from another controller file, it generates the need for its corresponding model and view files. So its view files are also created to fulfill the purpose of refactorings performed.
3. *Replace Conditional by using Database Field Value* is used to simplify the design that requires a conditional check to be used at large number of places (20 in our case) to implement similar functionality slightly differing in behavior. So, we added one database field and used its value as a conditional check. We let the implementation to remain same without any conditional check. It reduces the number of conditional checks drastically (from 18 to 3 in our case) to be used only where required.

From Table 3, we see that the most common refactorings (numbers highlighted in green) performed in the application are Move Method—the highest with 62 instances, Rename Variable (22), Add Method (21), and Rename Method (17). Students also identified average number of instances (numbers highlighted in yellow) for Extract Class (11), Delete Method (11), Add Class (9) and Extract Method (8) refactoring types. It also describes couple of instances for rare refactorings (numbers highlighted in red) for Add Parameter (3), Hide Method (2), Delete Class (2), Delete Variable (1), Hide Variable (1) and Replace Conditional with Database Field Value (1).

In Table 4, we classify which refactorings are most common (major refactorings), which are average (average refactorings) and which are less common (minor refactorings) in

our projects. We believe that the reasons for most common refactorings are their ease of implementation and comfort of using tool support, e.g., renaming entities. Murphy-Hill et al also show that developers commonly implement low-level refactorings and refactorings for which they are comfortable with the tool support [11].

The major **code smells** found in the application are already mentioned in Section 3.2.1. Kerievsky and Fowler proposed different refactorings to resolve various code smells [1, 2]. In Table 5 we show the list of **refactorings** that students adopted to handle different code smells.

**Table 4: Category of refactorings**

<b>COMMON Refactorings</b> (Highlighted in green)	<b>AVERAGE Refactorings</b> (Highlighted in yellow)	<b>RARE Refactorings</b> (Highlighted in red)
Move Method (62)	Extract Class (11)	Add Parameter (3)
Rename Variable (22)	Delete Method (11)	Delete Class (2)
Add Method (21)	Add Class (9)	Hide Method (2)
Rename Method (17)	Extract Method (8)	Delete Variable (1)
		Hide Variable (1)
		Replace Conditional with Database Field Value (1)

**Table 5: Refactorings for specific code smells**

<b>Code Smells</b>	<b>Refactorings Performed By Students</b>
Large Class (>200 LOC)	Extract Class, Move Method, Delete Methods
Long Method (>50 LOC)	Extract Method for different responsibilities, Extract Method for duplicate code
Duplicate Code	Extract Method for duplicate code
Complex Code	Extract Method to modularize the responsibilities, Replace Conditional by using Database Field Value
Spaghetti Code	Extract methods in object oriented way Add Variables Add Method Add Class
Improper Names	Rename Variable, Rename Method
Others	Hide Variable, Hide Method, Modify Method Signatures

We illustrate the excerpts from the source code for some of the common bad smells and the refactorings carried out to eliminate them.

## I. Long method

**Table 6: Long method code smell**

Before Refactoring	After Refactoring
<pre> <b>def</b> create   .....    <b>if</b> params[:days].nil? &amp;&amp; params[:weeks].nil?     @days = 0     @weeks = 0   <b>elsif</b> params[:days].nil?     @days = 0   <b>elsif</b> params[:weeks].nil?     @weeks = 0   <b>else</b>     @days = params[:days].to_i     @weeks = params[:weeks].to_i   <b>end</b>    @assignment.days_between_submissions =   @days + (@weeks*7)   .....   .....  <b>end</b> </pre>	<pre> <b>def</b> create   .....    set_days_between_submissions   .....  <b>end</b>  <b>def</b> set_days_between_submissions    <b>if</b> params[:days].nil? &amp;&amp; params[:weeks].nil?     @days = 0     @weeks = 0   <b>elsif</b> params[:days].nil?     @days = 0   <b>elsif</b> params[:weeks].nil?     @weeks = 0   <b>else</b>     @days = params[:days].to_i     @weeks = params[:weeks].to_i   <b>end</b>    @assignment.days_between_submissions = @days + (@weeks*7) <b>end</b> </pre>

Explanation: The *create* method was a very long method of 111 LOC with lot of functionality that can be extracted out into different methods. Here, we show only one set of statements that is extracted into a new method. LOC for this method reduces to 56 after

refactoring. The methods 'set\_days\_between\_submissions' of 12 LOC and 'set\_due\_dates' of 60 LOC is extracted out.

Refactorings done here: Extract Method.

## II. Unreasonable names

- a) Unreasonable method names

**Table 7: Unreasonable method name code smell**

<b>Before Refactoring</b>	<b>After Refactoring</b>
<pre> class AssignmentParticipant &lt; Participant   require 'wiki_helper'   .....    def <b>submit_hyperlink</b>(hyperlink)     .....   end    def <b>get_course_string</b>     .....   end    def <b>get_scores(questions)</b>     .....   end </pre>	<pre> class AssignmentParticipant &lt;   Participant   require 'wiki_helper'   .....    def <b>submit_hyperlink</b>(hyperlink)     .....   end    def <b>get_course_name</b>     .....   end    def <b>get_scores_given(questions)</b>     .....   end </pre>

b) Unreasonable variable names

**Table 8: Unreasonable variable name code smell**

<b>Before Refactoring</b>	<b>After Refactoring</b>
<ol style="list-style-type: none"><li>1. <code>@Review_of_review_deadline = deadline.id</code></li><li>2. <code>elseif @submission == "review_of_review"</code></li></ol>	<pre>@metareview_deadline = deadline.id elseif @submission == "metareview"</pre>

Explanation: Table 7 shows an example of improper and misspelled method names and Table 8 shows improper variable names that are changed according to new terminology.

Refactorings done here: Rename Method, Rename Variable.

### III. Method handling two responsibilities and duplicate code

Table 9: Method with two responsibilities and duplicate code

Before Refactoring	After Refactoring
<pre> def reset_and_mail_password  self.reset_password!  Mailer.deliver_message(   {:recipients =&gt; self.email,    :subject =&gt;"Your Expertiza password has been reset ",   :body =&gt; {     :user =&gt; self,     :password =&gt; clear_password,     :first_name =&gt; ApplicationHelper::get_user_first_name(self ),     :partial_name =&gt; "send_password"   } } ) end  def email_welcome Mailer.deliver_message(   {:recipients =&gt; self.email,    :subject =&gt;"Your Expertiza password has been creat ed",   :body =&gt; {     :user =&gt; self,     :password =&gt; clear_password, #FIXME     :first_name =&gt; ApplicationHelper::get_user_first_name(self ),     :partial_name =&gt; "user_welcome"   } } ) end </pre>	<pre> def reset_password   randomize_password   save   clear_password end  module MailerHelper def self.send_mail_to_user(user,subject ,partial_name,password)  return Mailer.deliver_message(   {:recipients =&gt; user.email,    :subject =&gt;subject,    :body =&gt; {     :user =&gt; user,     :password =&gt; password,     :first_name =&gt; ApplicationHelper::get_user_first_nam e(user),     :partial_name =&gt; partial_name   } } ) end  def email_welcome  MailerHelper::send_mail_to_user(self, "Your Expertiza password has been cr eated","user_welcome",clear_passwor d)  End </pre>

Explanation: Table 9 shows ‘Mailer.deliver\_message’ method is a duplicate code used twice with different parameters. ‘reser\_and\_mail\_password’ method has two tasks: 1) to reset the password, 2) to send mail to user.

Refactorings done here:

- Extract Method to remove duplicate code following the DRY principle.
- Extract Method to modularize different functionalities.

Students implement these different refactorings to resolve the code smells and thereby improve the software quality. Modularity, low cohesion and high coupling are some of the characteristics of good software design. They allow the application to follow object-oriented principles (e.g., DRY principle, Law of Demeter) more precisely.

In Table 10, we show LOC, responsibilities and collaborators before and after refactoring in some important files of the application. These statistics clearly show that the refactorings have improved the cohesion and coupling measure considerably.

**Table 10: Code metrics for classes**

File Name	LOC		Responsibilities		Collaborators	
	Before	After	Before	After	Before	After
Assignment	677	209	51	16	24	15
SignUpSheetController	626	365	28	14	12	7
QuestionnaireController	242	181	12	10	8	6
AssignmentParticipant	382	311	35	27	14	14
AdminController	120	57	17	10	3	3
AssignmentController	403	532	16	20	12	12

From Table 10, we can see that first five rows (shown in blue) show considerable improvement in *LOC* after refactoring. Assignment and AdminController class's length gets reduced by more than 50%, SignUpSheetController by 42%, whereas QuestionnaireController and AssignmentParticipant show decreases of 25% and 18% respectively.

Table 10 also shows that number of *responsibilities* gets reduced after refactoring when methods are moved to their respective classes. The Assignment class shows the maximum decrease in responsibilities, reducing from 51 to 16 (in green), a 68% decrease. Similarly, the number of *collaborators* has also been reduced by considerable amount, Assignment shows 37.5% decrease, QuestionnaireController 25%. Other files, namely, AssignmentParticipant, AdminController, and AssignmentController, do not show any change in the number of collaborators, which shows that the coupling measure remains the same for these classes whereas their cohesion measure is improved.

One more thing to notice here is that in the AssignmentController (shown in Orange color), *LOC* gets increased after refactoring. This is due to the addition of some more responsibilities for this class, but its cohesion measure gets improved without affecting its coupling measure. Here, we see that the *LOC* measure is still high (>200) for some of these classes, which calls for further possible refactoring in them. This we will check in the next section, to see whether students still miss some of the refactorings, which could have improved these metrics.

This improvement of cohesion and coupling implies the improvement in object-oriented design of the application.

Summarizing the above discussion we can clearly say that students have performed pretty well in their projects:

- They employed low-level and medium-level refactorings.
- They have used these refactorings to accommodate and resolve different code smells.
- The code structure has been greatly improved by refactoring.
- Refactoring reduces the code complexity and also accounts for high cohesion and low coupling.
- It improves the code reusability, understandability, and maintainability.
- It allows the application to follow object-oriented principles more closely like DRY principle and Law of Demeter.

Although students have done quite well, they have still missed some points. In the next section, we provide an analysis of missed refactorings and possible reasons for them.

## **4.2 What did students miss?**

Students performed pretty well in their projects by applying total of 171 refactorings in the code but there is still lot of scope for additional refactorings to be incorporated. We found many missing refactorings.

### **4.2.1 Missing refactorings**

We showed most common refactorings applied by the students in Section 4.1. But there are many other refactorings that students totally missed in their projects and these are:

1. Introduce Explaining Variable
2. Replace Magic Number by Symbolic Constant
3. Decompose Conditional
4. Remove Middle Man
5. Replace Conditional with Polymorphism
6. Rename Class
7. Replace Template by Query

These refactorings could be implemented at many places in the application, but all students missed them. Secondly, students also missed many instances of already used refactorings mentioned in Section 4.1 at various places in the code.

From study of these projects, we collected sufficient data about missing refactorings and determined what students most commonly missed in their projects. In Table 11 we have shown all refactorings missed by the students in each project along with the number of instances missed.

Table 11: Missing refactorings in each project

No.	Refactorings Missed	Refactoring Project	Number of Instances Missed	Total Instances Missed
1	<b>Introduce Explaining Variable</b>	E 205	1	12
		E 211	3	
		E 219	3	
		E 209	4	
		E 204	1	
2	<b>Replace Magic Number by Symbolic Constant</b>	E 205	1	7
		E 204	6	
3	Decompose Conditional	E 219	1	1
4	Remove Middle Man	E 201	2	2
5	Replace Conditional with Polymorphism	E 212	1	1
6	Rename Class	E 204	1	2
		E 209	1	
7	Replace Template by Query	E 209	1	1
8	Hide Method	E 216	1	1
9	<b>Hide Variable</b>	E 202	2	11
		E 210	2	
		E 219	5	
		E 204	2	
10	<b>Extract Method</b>	E 202	2	10
		E 209	2	
		E 212	3	
		E 219	2	
		E 204	1	
11	Move Method	E 203	2	2
12	Extract Class	E 219	2	2
13	Rename Method	E 203	1	4
		E 205	1	
		E 207	1	
		E 208	1	
14	<b>Rename Variable</b>	E 207	1	7
		E 209	3	
		E 210	1	
		E 219	2	
15	Remove Variable	E 219	1	1
16	Remove Unused Method	E 207	1	1
	<b>Total Refactorings Missed</b>			<b>= 65</b>

Table 11 shows that students missed a total of 65 refactorings. Students most commonly missed refactorings like Hide Variable (7), Introduce Explaining Variable (5), Extract Method (5), Rename Method (4), and Rename Variable (4) shown in orange in the table. An example of code excerpt for missed refactoring is shown in Appendix G.

Using Table 1 and Table 11 we figured out that total refactorings missed in floss refactoring projects are 29, whereas in root canal refactorings it is 36. So, the number of missing refactorings in root-canal refactoring projects is 19% more than refactorings missed in floss refactoring projects.

#### **4.2.2 Why did students miss refactorings?**

There could be many possible reasons for students to miss these 16 refactorings mentioned above. Students might have missed them because:

- Some refactorings are *harder to identify* than others. For example, Decompose Conditional, Replace Conditional with Polymorphism, Remove Middle Man and Introduce Explaining Variable are the refactorings that need good amount of effort to identify them.
- Some are *harder to implement* than others, For example, Decompose Conditional, Replace Conditional with Polymorphism, Remove Middleman, Extract Method, Hide Variable and Hide Method are the refactorings that impact lot of files and hence must be performed very carefully. Some them seem to be very straightforward at first glance, but actually it takes a lot of time and effort to figure out the places where it would actually impact the code.

- Students had limited refactoring knowledge about types of refactorings (as shown by the survey results).
- Students had limited time to identify or implement all refactorings.
- Students had limited application knowledge. (from the survey results.)
- Students didn't use tools to identify and implement refactorings. Most of the students only used the refactoring tool available in RubyMine, and not any external tools. Consequently, they couldn't implement a wide variety of refactorings with confidence.

In Table 12 we distinguish most commonly used refactorings and most commonly missed refactorings.

**Table 12: Most commonly used and missed refactorings**

<b>Commonly used refactorings</b>	<b>Commonly missed refactorings</b>
Move Method	Introduce Explaining Variable (12)
Add Method	Hide Variable (11)
Delete Method	Extract method (10)
Rename Method	Replace Magic Number by Symbolic Constant (7)
<b>Rename Variable</b>	<b>Rename Variable (7)</b>
Add Class	

From Table 12, we find some interesting results:

- First, we find that *Hide Variable* is most commonly missed refactoring (9). The reason we can think of is that this refactoring is harder to identify and implement than others. Although it seems simple at first sight, just to change the scope to private, it involves lot of other changes in many files. So, students might find it difficult to identify its impact within the application.

- Second, we find that *Rename Variable* (shown in orange) is the refactoring that students use frequently and also miss frequently. But the question is, Do the same teams who implemented this refactoring miss it? The answer is yes and no. Most commonly the teams who have not implemented it, missed it; others are the same teams who have implemented it too and missed it too.

Since it is easy to implement *Rename Variable* and *Rename Method*, so students most commonly implemented them. But in order to implement these refactorings, students should be well versed in the application terminology and coding conventions being followed. Lack of time can be another reason why they were not able to discover all instances.

### 4.3 Other observations

Students identified and implemented several refactorings, but they also missed them frequently in their work. During investigation of the projects, we determined that students missed many important properties that can make refactoring process more effective. Some properties are crucial for refactoring but others are just good to have. These are:

- **Test cases** – Testing is an integral part of the refactoring process. Regression test suites generally follow refactoring. When refactoring is performed to modify the internal structure, test cases are run to verify that external behavior is not changed. If refactoring is done along with new changes in the system, then user should add new tests to check the required functionality of the system.

From our study, we observe that students did not implement test cases for some projects. Other projects do include test cases, but either they are incomplete or they are not accurate enough to test the required functionality and behavior. Table 13 shows which projects did not include test cases at all and which ones include faulty (incomplete or incorrect) test cases. We find that refactoring projects with missing test cases are all Floss refactoring projects, where students should create unit and functional test cases for new features implemented.

**Table 13: Test case scenario for projects**

<b>Category</b>	<b>Projects</b>
Missing Test Cases	E 207, E 211, 212
Faulty (incomplete or incorrect) Test Cases	E 202

Table 13 shows that 4 out of 13 projects, i.e., 31% of projects, have insufficient testing support. This may not serve our purpose of refactoring as a whole, since only testing can ensure that refactoring objective is met. This is one of the major missing points of these refactoring projects by the students. Other programming practices that should be considered while refactoring are:

- Remove dead code,
- Remove commented-out code,
- Provide sufficient comments for changes made, and
- Remove any print statements from the source code.

Although these practices are not part of refactoring process, they will make the code much cleaner, readable and understandable. It makes sense to consider them during refactoring because programmer has more chance to lookup and clean his code by minimal effort and time.

In the next section, we suggest the refactoring approach and its significance in terms of refactoring coverage and test case coverage necessary for code refactoring. Students should follow this approach while incorporating different types of refactorings to make refactoring projects more effective.

#### **4.4 How should students refactor the code?**

The results in Section 4.1, 4.2 and 4.3 show that students applied substantial refactorings to improve system design, but also missed significant number of them, which is 38% ( $65/171 \times 100$ ) of the applied refactorings, and also the test cases, which are missing in 31% of the projects. So, it is our major concern that students should follow some productive refactoring approach that can minimize the missing refactorings and maximize test coverage to meet the refactoring goal. In this section, we propose test-driven development (TDD) as a refactoring approach that students should follow to achieve a disciplined refactoring process.

##### **4.4.1 Refactoring approach**

To proceed with the refactoring process, first and foremost point is that students should have an adequate knowledge of the application to be refactored and types of refactoring available. So, we divide our approach into two phases:

Phase 1: Acquire adequate knowledge

Phase 2: Follow TDD refactoring approach

##### **Phase 1 – Acquire adequate knowledge**

Although many students learn about refactorings and bad smells in their course, they are not very well versed in techniques for dealing with them. Lack of knowledge about certain refactorings is a major reason why students miss significant refactorings. So, to find as many refactorings as possible, our approach states that students should acquire sufficient knowledge about:

1. possible **bad smells** that may exist in the code,
2. existing **refactoring types** already proposed in studies [1], and

3. the **application** to be refactored.

These three are complementary to each other for an effective refactoring process. Students who are aware of bad smells like large class, long method, long parameter list, improper names as mentioned in Section 3.2.1 can look for those code smells in the application. Knowledge of possible refactorings, at least low-level and medium-level refactorings, allow the students to relate them to particular code smells. Besides this, students need knowledge of the application, including its features, its terminology, coding conventions, and implementation details.

The instructor can help students to gain more knowledge about refactorings by providing them with web links or other valuable material about refactorings and bad smells.

*This approach in Phase 1 ensures that students should not miss any refactorings and its instances due to their ignorance or half knowledge.*

## **Phase 2: Follow TDD refactoring approach**

Test-driven development is an effective software development approach where developers write the tests before they implement any code. The steps in TDD are Red-Green-Refactor [17]. After the developers understand the requirements, they follows three steps: 1) RED - Write test and make them fail, 2) GREEN - Implement code to make those tests pass, and 3) REFACTOR - Refactor the code to improve the internal design. Then run the test cases again to guarantee that changes did not break the code. TDD is one of the practices of agile methodologies and Extreme Programming [17].

A research study [16] shows that a test-first approach improves software quality and helps students to complete their assignments early. Kent Beck also suggested test-driven approach

for implementing additional features in already existing code using agile methodology. Test suites help to proceed with the implementation in steps and perform testing to verify if code meets its requirement [17]. Kerievsky also talked about the test-driven refactoring approach as re-implement and replace technique [2].

When new features are to be added in an application, the developer first determines if the existing design needs to be refactored. So, the developers first write the test cases for the new feature, then write code incrementally to pass the test cases, and then refactor the code if required. If the test fails, it shows the problem in the code at an early stage. So, running the test cases make sure functionality is implemented correctly. To ensure that internal changes did not break the code, these new test cases can be rerun after refactoring is carried out.

When only internal structure needs to be modified, then there are no functional changes in the code. Hence, the developer only need to write unit tests, whereas he can use existing functional tests, i.e., regression test suite for that behavior can be rerun to ensure that external behavior is consistent.

Therefore, in Phase 2, starting with the test-first development approach we define the following steps to perform effective refactorings:

1. Write unit or functional test cases - If pure refactorings are performed in any project, then functional tests will remain same but unit tests may differ. So students need to write unit tests if required. Incase new features are being added along with refactorings in any project, then students should write both functional and unit tests.

*This step ensures the students to have sufficient test coverage for new features and refactorings, which we observed in 4.3.*

2. List out all bad smells in the code - After writing test cases, proceed with the refactoring process and list out all the possible bad smells in the code. Some are already mentioned in Section 3.2.1.
3. Identify refactorings – Determine all possible refactorings to eliminate the abovementioned code smells. Students should relate these refactorings to the listed code smells.
4. Identify all instances where refactorings should be applied - Before starting with the implementation, students should identify all instances in the code for each refactoring where they need to carry out changes, rather than repeating identify-implement cycle.  
*This step greatly reduces the chance for the students to miss any instances of identified refactorings, which we observed in 4.2*
5. Implement refactorings incrementally – For each refactoring type, first implement it and then run its relevant test cases. Repeat this step till all refactorings have been incorporated in the code. This guarantees that external behavior is preserved by these refactorings.
6. Run the test cases to check if the refactorings have preserved the external behavior.

*Phase 2 that follows TDD refactoring approach allows for the maximal test coverage and instance coverage of each refactoring type.*

From the observations in Section 4.2 and 4.3, we concluded that students significantly missed: 1) important refactorings, 2) instances of identified refactorings, and 3) test cases.

Although our approach cannot ensure the complete coverage, it addresses all these three problems by allowing the students to include maximum refactorings, maximum instances of each refactoring and maximum test cases in their projects.

Summarizing, we say that TDD refactoring technique minimizes the missing refactorings and maximizes the test coverage.

#### **4.5 Limitations**

The results of this study clearly show which refactorings students most commonly implemented and which refactorings students most commonly missed. These results are taken from only one open-source application used in the class. Some more open-source applications can be taken for more substantial data. Secondly, the study is based on manual inspection of code, so it is possible that some other refactorings can also be unwrapped by further higher-level analysis of code. Third, the number of additions and deletions mentioned in Appendix F for each project may not be accurate, since these values also contain the measure for whitespaces, newline and indentation changes. So, it did not account for actual measure of changes being done in the application.

## **Chapter 5 – Future work and Conclusion**

### **5.1 Future work**

This approach will be experimented with in the CSC/ECE 517 Fall 2012 class. The basic rules for implementing the project will remain the same except following our TDD-based refactoring approach. This way we can make sure that students perform sufficient refactorings and minimize the missing instances of it covering all test cases for the project. Moreover, we also propose to use static code analysis tools to automatically detect the bad smells in the code, such as the static code analysis tools Roodi for Ruby. Reek is similar to Roodi, and Saikuro can compute the cyclomatic complexity. A Ruby gem called “excellent” is available for static code analysis.

### **5.2 Conclusion**

This study analyzes refactoring projects that try to improve the internal design and quality of a relatively small application. It shows how refactoring projects can be formalized by determining candidate classes using LOC as code metric. The code is analyzed by manual inspection of revisions submitted by the students. We classified our findings into three main categories: how did students perform refactorings, what did they miss during refactoring and how should they actually refactor code. In first two categories, we indicated the most commonly used and most commonly missed refactorings respectively. We also specified the possible reasons for these commonly missed and used refactorings. We also observed some significant issues with testing in refactoring projects.

The results of the study indicate that students applied substantial refactorings within the code. But, they also missed at least 38% of the total refactorings. We discussed the reasons for missing these refactorings. Finally we proposed a test-driven refactoring approach for students to follow. The approach is to perform refactorings in two phases; first by acquiring sufficient knowledge of refactorings, bad smells and application details and second, by implementing refactoring using test-driven approach. Although being a manual approach, it cannot guarantee the complete coverage of refactorings and its instances, it ensures:

- more coverage of refactoring types,
- more coverage of instances of each refactoring type, and
- more coverage of test cases.

Therefore, it would be very beneficial to follow this framework in class for carrying out refactoring activities effectively in any small sized software application.

## REFERENCES

- [1] Fowler, Martin, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wisley Longman Inc., 1999.
- [2] Kerievsky, Joshua. *Refactoring to Patterns*. Boston, MA: Addison-Wesley Longman Inc., 2005.
- [3] Pressman, Roger S. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 2010.
- [4] Gehringer, Edward F., Luke Ehresman, Susan G. Conger, and Prasad Wagle. "Reusable Learning Objects Through Peer Review: the Expertiza Approach." *Frontiers in Education Conference, 36th Annual* (2007)
- [5] Ellis, Heidi J. C., Gregory W. Hislop, Mel Chua, Clif Kussmaul, and Mathew M. Burke. "Panel — Teaching Students to Participate in Open Source Software Projects." *Frontiers in Education Conference (FIE), 2010 IEEE* (2010): F2B-1-F2B-2.
- [6] Sowe, Sulayman K., and Stamelos Ioannis. "Involving Software Engineering Students in Open Source Software Projects: Experiences from a Pilot Study." *Journal of Information Systems Education* v.18 (2007): 425-436.
- [7] Stroulia, Eleni, and Rohit Kapoor. "Metrics of Refactoring-based Development: an Experience Report." *7th International Conference on Object-Oriented Information Systems* (2001): 113-122.
- [8] Simon, Frank, Frank Steinbrückner, and Claus Lewerentz. "Metrics Based Refactoring." *Fifth European Conference on Software Maintenance and Reengineering* (2001): 30-38.
- [9] Moha, Naouel, Yann- G. Guéhéneuc, Laurence Duchien, and Anne- F. Françoise Le Meur. "DECOR: a Method for the Specification and Detection of Code and Design Smells." *IEEE Transaction on Software Engineering* (2010): 20-26.
- [10] Pizka, Markus. "Straightening Spaghetti-Code with Refactoring?" *Software Engineering Research and Practice* (2004)
- [11] Murphy-Hill, Emerson, Chris Parnin, and Andrew P. Black. "How We Refactor, and How We Know It." *31st International Conference on Software Engineering* (2009)
- [12] Dig, Danny, Can Comertoglu, Darko Marinov, and Ralph Johnson. "Automated Detection of Refactorings in Evolving Components." (2005): Web. <http://hdl.handle.net/2142/11134>.

- [13] Mens, Tom, and Tom Tourwé. "A Survey of Software Refactoring." *IEEE Transactions on Software Engineering* 30.2 (2004).
- [14] Kim, Miryung, Dongxiang Cai, and Sunghun Kim. "An Empirical Investigation into the Role of API-level Refactorings During Software Evolution." *33rd International Conference on Software Engineering* (2011): 151-160.
- [15] Johnson, Ralph, and Donald B. Roberts. "Practical Analysis for Refactoring." *Doctoral Dissertation, University of Illinois at Urbana-Champaign Champaign, IL, USA* (1999).
- [16] Spacco, Jaimo, and William Pugh. "Helping Students Appreciate Test-driven Development." *ObjectOriented Programming Systems Languages and Applications OOPSLA* (2006): 907-913.
- [17] Beck, Kent. *Test Driven Development: By Example*. Addison-Wesley, 2002.

## **APPENDICES**

## Appendix A

**Table 14: Survey questionnaire**

No.	Question
1	In the source files that you refactored, how many lines of code were there when you started refactoring?
2	How many lines of code were there after refactoring?
3	List the main refactorings that you performed & estimate the number of instances of each.
4	Did you use any tools (e.g., Ruby Mine) to carry out any of the refactorings? Which?
5	Did you apply any design patterns while refactoring the code? Which ones?
6	Which object-oriented design principles (e.g., DRY principle, Law of Demeter) have you followed while refactoring?
7	What is more challenging and time consuming—identification of where to apply refactoring, or carrying out the refactoring? Why?
8	Did you find violations of specific OOD principles in the code that you refactored?
9	How many test cases did you write for the code you refactored?
10	How many pre-existing test cases were there for the code you refactored?
11	What kind tests have you written (unit, functional, integration (Cucumber, etc.)?)
12	Do you believe that your refactorings improved performance of the application, or only improved the readability and maintainability of code?
13	In order to finish your refactorings, what did you need to know that was hard to find out?
14	What could we have done that would have made refactoring easier for you?
15	Do you have any other comments?

## Appendix B

**Table 15: Summary of survey results**

<b>List the main refactorings that you performed &amp; estimate the number of instances of each.</b>	<b>Which object-oriented design principles (e.g., DRY principle, Law of Demeter) have you followed while refactoring?</b>	<b>What is more challenging and time consuming - identification of where to apply refactoring, or carrying out the refactoring? Why?</b>	<b>Do you believe that your refactorings improved performance of the application, or only improved the readability and maintainability of code?</b>	<b>In order to finish your refactorings, what did you need to know that was hard to find out?</b>
Move Method, Extract Class, Extract Method, Rename Entities, Remove Method, Hide Variable	DRY Principle-6, Law of Demeter-4, Open Close Principle-1	Identification- 9, Implementation-2	Readability and Maintainability-7, Performance- 5	Dependencies – 3, Refactoring types – 2, Application Knowledge -1, Code Documentation-1

## Appendix C

Table 16: Survey results

In the source files that you refactored, how many lines of code were there when you started refactoring?	How many lines of code were there after refactoring	List the main refactorings that you performed & estimate the number of instances of each.	Which object-oriented design principles (e.g., DRY principle, Law of Demeter) have you followed while refactoring?	What is more challenging and time consuming - identification of where to apply refactoring, or carrying out the refactoring? Why?	In order to finish your refactorings, what did you need to know that was hard to find out?
~350	Around 250	<p>The main refactoring included moving the unrelated methods to a different class to provide maximum cohesion.</p> <p>Define a module and move appropriate methods to the module. Write tests.</p>	DRY.	<p>Identification of where to apply refactoring.</p> <p>We need to consider various dependencies and make a decision as to if it is really required to refactor it.</p> <p>Also it generally requires a lot of follow ups and approvals from the instructors.</p>	The dependencies.
1000	1100	delete unnecessary code that violate DRY principle...	DRY	identification	The dependencies.
150	120	creating	Law of	Carrying out	The

In the source files that you refactored, how many lines of code were there when you started refactoring?	How many lines of code were there after refactoring	List the main refactorings that you performed & estimate the number of instances of each.	Which object-oriented design principles (e.g., DRY principle, Law of Demeter) have you followed while refactoring?	What is more challenging and time consuming - identification of where to apply refactoring, or carrying out the refactoring? Why?	In order to finish your refactorings, what did you need to know that was hard to find out?
		modules that were mixed in different classes	Demeter, Open close principle.	the refactoring, because even after refactoring, there were some dependencies that failed, like SQL commands which didn't change as we refactored the schema.	dependencies
~120	~100	We were working on logging functionality for expertiza and the controller was not RESTful. we added RESTfulness to the controller.	DRY principle	identification of where to apply refactoring. The refactoring is fairly mechanical, but to identify the exact code/module	
	The lines of code were the same before and	Refactored some code into modules.	Law of Demeter	Identification. As carrying out the refactoring	What kind of refactoring to do

In the source files that you refactored, how many lines of code were there when you started refactoring ?	How many lines of code were there after refactoring	List the main refactorings that you performed & estimate the number of instances of each.	Which object-oriented design principles (e.g., DRY principle, Law of Demeter) have you followed while refactoring?	What is more challenging and time consuming - identification of where to apply refactoring, or carrying out the refactoring? Why?	In order to finish your refactorings, what did you need to know that was hard to find out?
	after. We just moved the code around.	And others like renaming.		was easy with the tools.	
490	300	Most of the times I used DRY principle. Creating partials I was able to reduce the repeated codes. I also created a new helper method to tidy up the messy code. The method handled managing the questions for custom rubric.	DRY	Identifying the portions for refactoring was pretty straightforward because the project definition provided that for us, The challenge was in actually refactoring and determining a proper way to refactor.	
354	401	in leaderboard, we refactor the functions to use	Law of Demeter	it is time consuming to find out for example participant id or response	

In the source files that you refactored, how many lines of code were there when you started refactoring ?	How many lines of code were there after refactoring	List the main refactorings that you performed & estimate the number of instances of each.	Which object-oriented design principles (e.g., DRY principle, Law of Demeter) have you followed while refactoring?	What is more challenging and time consuming - identification of where to apply refactoring, or carrying out the refactoring? Why?	In order to finish your refactorings, what did you need to know that was hard to find out?
		scorecaches .		may type located to which class. we should debug to see which class included these parameters we can passed into the functions.	
1300	2000		Law of Demeter.	carrying out the refactoring	
~600	approx 400	move functions to different models , make new modules and move functionality into them, make new models and controllers	dry	Identification and making sure that refactoring does not break the existing code	knowing the entire code
				identification of where to apply refactoring	Refactoring type
~670	around 500	Moving a method from one class to	DRY	Identification of where to apply refactoring	What each method does in a particular

<b>In the source files that you refactored, how many lines of code were there when you started refactoring?</b>	<b>How many lines of code were there after refactoring</b>	<b>List the main refactorings that you performed &amp; estimate the number of instances of each.</b>	<b>Which object-oriented design principles (e.g., DRY principle, Law of Demeter) have you followed while refactoring?</b>	<b>What is more challenging and time consuming - identification of where to apply refactoring, or carrying out the refactoring? Why?</b>	<b>In order to finish your refactorings, what did you need to know that was hard to find out?</b>
		other - around 6 methods Removing a method - 2 methods Removing unnecessary code - around 10-15 lines Removing duplication of code - around 3-4 places Changing an instance variable to local variable - around 10 Renaming a variable - 1		was challenging. There were a lot of opportunities for refactoring but identifying which were proper and which were not according to object-oriented design principles took some effort and time.	controller and relationships between some models were some things that were difficult to understand. A proper documentation for the source code was not present.
~350	Around 250	The main refactoring included moving the unrelated methods to a different class to provide			

<b>In the source files that you refactored, how many lines of code were there when you started refactoring?</b>	<b>How many lines of code were there after refactoring</b>	<b>List the main refactorings that you performed &amp; estimate the number of instances of each.</b>	<b>Which object-oriented design principles (e.g., DRY principle, Law of Demeter) have you followed while refactoring?</b>	<b>What is more challenging and time consuming - identification of where to apply refactoring, or carrying out the refactoring? Why?</b>	<b>In order to finish your refactorings, what did you need to know that was hard to find out?</b>
		maximum cohesion.  Define a module and move appropriate the module. methods to Write tests.			

## Appendix D

**Table 17: List of OSS projects in Expertiza in fall '12**

<b>Project</b>	<b>Project Name</b>	<b>Classes</b>
E201	Break up assignment.rb	assignment.rb
E202	Refactor assignment_controller	assignment_controller.rb
E203	Refactor assignment_participant.rb	assignment_participant.rb, assignment_team.rb
E204	Remove Individual Assignments	grades_controller.rb response_controller.rb review_mapping_controller.rb sign_up_sheet_controller.rb student_task_controller.rb submitted_content_controller.rb dynamic_review_mapping.rb (helper) assignment_participant.rb assignment.rb leaderboard.rb metareview_response_map.rb participant.rb review_response_map.rb sign_up_topic.rb (as well as several views: please search for "team_assignment")
E205	Users	models/user.rb (211 lines)
E207	Questionnaires and advice	controllers/questionnaire_controller.rb (242 lines)
E208	Changes to permissions	assignment_controller.rb sign_up_sheet_controller.rb student_review_controller.rb student_task_controller.rb assignment.rb views/assignment/_due_dates.html.erb views/assignment/edit.html.erb views/student_task/list.html.erb
E209	Refactor signup code	sign_up_sheet_controller.rb, sign_up_topic.rb, signed_up_user.rb
E210	Refactor review_mapping_controller	review_mapping_controller.rb
E211	E-mailer improvements	-
E212	Wiki spider	models/wiki_type.rb (272 lines)

<b>Project</b>	<b>Project Name</b>	<b>Classes</b>
E216	Refactor TeamController and ParticipantChoicesController	<a href="https://github.com/expertiza/expertiza/issues/114">https://github.com/expertiza/expertiza/issues/114</a>
E219	Miscellaneous controller cleanup	Any controller class not mentioned above

The detailed description of all projects can be found at the link:

[https://docs.google.com/a/ncsu.edu/document/d/1zZ-a\\_tkLGrbYJbG-2QBCRqe\\_Y9XLR4AZMH0rc6KvrYU/edit](https://docs.google.com/a/ncsu.edu/document/d/1zZ-a_tkLGrbYJbG-2QBCRqe_Y9XLR4AZMH0rc6KvrYU/edit)

## Appendix E

This is an example of a project specification given to the students for their assignment. The below project description shows Root-Canal refactoring project.

### **Project E201: Break up assignment.rb**

**Contact:** Ed Gehringer (efg@ncsu.edu)

**Classes:** assignment.rb

**What it does:** This is the model class for assignments.

**What is wrong:** The class is 677 lines long, which is way out of proportion. It contains functionality for adding and removing participants for this assignment, which should really be in a `participant` class, for assigning reviewers, which should probably be in a `reviewer` class, and for computing the maximum score possible on a questionnaire. Functions like `compute_scores`, and `candidate_topics_to_review`, among others, should be moved to other classes.

#### **How to fix**

Move functionality out of `assignment.rb` to other classes, preferably other existing model classes corresponding to tables in the database.

#### **Testing**

Provide functional tests for the actions that have moved from `assignment.rb` to another class.

## Appendix F

**Table 18: Revised branch statistics for each project**

<b>Project</b>	<b>Number of Files</b>	<b>Number of Commits</b>	<b>Number of Additions/Deletions</b>
E 201	12	10	1042/848
E 202	31	59	1223/621
E 203	4	3	88/81
E 204	20	3	243/476
E 205	23	20	315/113
E 207	8	7	1067/104
E 208	25	5	1154/649
E 209	1	4	759/24
E 210	17	17	332/265
E 211	24	3	226/109
E 212	11	7	192/78
E 216	15	20	270/431
E 219	18	6	1449/150
<b>Total</b>	<b>276 (not distinct); 67 distinct files</b>	<b>164</b>	<b>8360/3949</b>

## Appendix G

A code snippet showing *missed refactorings* in a method.

Introduce temporary  
Explaining variable for  
params[:error\_msg]

Introduce temporary  
Explaining variable for  
params[:msg]

```
def redirection
  flash[:error] = params[:error_msg] unless params[:error_msg] and
  params[:error_msg].empty?
  flash[:note] = params[:msg] unless params[:msg] and params[:msg].empty?

  @map = ResponseMap.find(params[:id])
  if params[:return] == "feedback"
    redirect_to :controller => 'grades', :action => 'view_my_scores', :id =>
@map.viewer.id
  elsif params[:return] == "teammate"
    redirect_to :controller => 'student_team', :action => 'view', :id => @map.viewer.id
  elsif params[:return] == "instructor"
    redirect_to :controller => 'grades', :action => 'view', :id => @map.assignment.id
  else
    redirect_to :controller => 'student_review', :action => 'list', :id => @map.viewer.id
  end
end
```