Investigating the Effect of Refactoring on Software Testing Effort

Karim O. Elish and Mohammad Alshayeb
Information and Computer Science Department
King Fahd University of Petroleum & Minerals
Dhahran 31261, Saudi Arabia
{kelish, alshayeb}@kfupm.edu.sa

Abstract—Refactoring, the process of improving the design of existing code by changing its internal structure without affecting its external behavior, tends to improve software quality by improving design, improving readability, and reducing bugs. There are many different refactoring methods, each having a particular purpose and effect. Consequently, the effect of refactoring on software quality attribute may vary. Software testing is an external software quality attributes that takes lots of time and effort to make sure that the software performs as intended. In this paper, we propose a classification of refactoring methods based on their measurable effect on software testing effort. This, in turn, helps the software developers decide which refactoring methods to apply in order to optimize a software system with regard to the testing effort.

Keywords—refactoring; software metrics; testing effort.

I. INTRODUCTION

Any software that is related to a real-world problem domain, must continuously adapt to changes as the problem domain changes [1]. During the development of software and its maintenance phase, it is very likely that the code will be modified or improved. This can happen due to many reasons, including changes in the requirements, the addition of new requirements, or bug fixes. In object-oriented paradigm, the process of making changes to software that affect its internal structure without altering its behavior is called “refactoring” [2-3].

Refactoring is a technique that reduces software complexity by improving its internal structure. There are many different refactoring methods, each having a particular purpose and effect. Consequently, the effect of refactoring methods on software quality attribute may vary [4].

Software testing is an important activity in any software development. Testing activities can take between 30 and 50 percent of the total effort spent in the development of software [5]. Software testability, as defined by ISO/IEC Std. 9126, is a set of attributes of software that bear on the effort needed to validate the software product [6]. In this paper, we adopt this definition.

Therefore, the objective of this paper is to propose a classification of refactoring methods based on their measurable effect on software testing effort in order to help the software developers decide which refactoring methods can help in optimizing a software system with regard to the testing effort.

The rest of this paper is organized as follows. Section 2 reviews the related work. Section 3 describes how the testing effort can be assessed from internal quality metrics. Section 4 describes the research method. Finally, Section 5 concludes the paper and gives directions for future work.

II. RELATED WORK

A number of research studies have investigated the effect of refactoring on internal and external software quality attributes. Stroggyllos and Spinellis [7] analyzed source code version control system logs of four popular open source software systems to detect changes marked as refactorings and examine how the software metrics are affected by refactorings. Du Bois and Mens [8] proposed formalism based on abstract syntax tree representation of the source-code, extended with cross-references to describe the impact of refactoring on internal program quality.

Geppert et al. [9] empirically investigated the impact of refactoring of a legacy system on changeability. They considered three factors for changeability: customer reported defect rates, effort, and scope of changes. Wilking et al. [10] investigated the effect of refactoring on maintainability and modifiability through an empirical evaluation. Maintainability was tested by randomly inserting defects into the code and measuring the time needed to fix them. Modifiability was tested by adding new requirements and measuring the time and LOC metric needed to implement them. Du Bois et al. [11] developed practical guidelines for applying refactoring methods to improve coupling and cohesion characteristics and validated these guidelines on an open source software system.

Kataoka et al. [12] proposed coupling metrics as a quantitative evaluation method to measure the effect of refactoring on the maintainability of the program by comparing the coupling before and after the refactoring. Moser et al. [13] proposed a methodology to assess if refactoring improves reusability and promotes ad-hoc reuse in an XP-like development environment.

We can observe the limitations of the existing research studies reviewed above. The researchers in [7, 9-10, 13] did not investigate the direct effect of each kind of refactoring...
methods proposed by Fowler [2] on internal or external software quality attributes. In addition, [7-8] were limited to studying the effect of refactoring on internal quality metrics only, they did not consider any external quality attributes. Moreover, [8, 11] have classified refactoring methods based on internal quality metrics, but used a limited number of refactoring methods. Additionally, none of the existing research studies have classified refactoring methods based on their effect on software testing effort which is one of the external software quality attributes. In this study, we focus on classifying refactoring methods based on their effect on software testing effort.

III. SOFTWARE TESTING EFFORT ASSESSMENT

Testing effort can be assessed from internal quality metrics [14]. Bruntink and van Deursen [14] identified and evaluated a set of object-oriented metrics that can be used to estimate the testing effort of the classes of object-oriented software. Lines of Code for Class (LOCC) and Number of Test Cases (NOTC) are used to represent the size of a test suite, while Depth of Inheritance Tree (DIT), Fan out (FOUT), Lack of Cohesion on Methods (LCOM), Lines of Code (LOC), Number of Children (NOC), Number of Fields (NOF), Number of Methods (NOM), Response for a Class (RFC), and Weighted Methods per Class (WMC) are used as predictors to predict/estimate the size of a test suite. In their experiment, they used five case studies of Java systems for which JUnit test cases exist. The systems include four commercial systems and one open source system (Apache Ant). They were able to find a significant positive correlation between object-oriented metrics (RFC, FOUT, WMC, NOM, and LOC) and the size of a test suite (LOCC and NOTC). This means that RFC, FOUT, WMC, NOM, and LOC are indicators for testing effort.

A. Object-Oriented Metrics Suite

We used a suite of metrics that measures the structural quality of object-oriented code and design. More specifically, we consider two metrics from Chidamber and Kemerer metrics suite [15] i.e. Response for a Class (RFC) and Weighted Methods per Class (WMC). In addition, we include one metric which measures class coupling (Fan out) and two metrics which measure class size i.e. Number of Methods (NOM) and Lines of Code (LOC). We chose these metrics because they have been used by previous empirical study to assess software testing effort and they were found to be good indicators for software testing effort [14]. The metrics we investigated are the following:

- **Response for a Class (RFC):** it is defined as the number of methods that can potentially be executed in response to a message being received by an object of that class.
- **Weighted Methods per Class (WMC):** it is defined as the number of methods defined (implemented) in a given class. Traditionally, it measures the complexity of an individual class (weighted sum of all the methods in a class).

- **Fan out (FOUT):** it is defined as the number of classes that a given class uses, not the classes it is used by.
- **Number of Methods (NOM):** it is defined as the number of methods implemented in a given class.
- **Lines of Code (LOC):** it is defined as the total source lines of code in a class excluding all blank and comment lines.

IV. RESEARCH METHOD

This section describes the research method to investigate the effect of selected refactoring methods on software testing effort.

A. Selected Refactoring Methods

Fowler [2] defined more than 70 different kinds of code refactorings in his refactoring catalog. Each one of them includes the motivation of why the refactoring should be performed and step-by-step description of how to carry out the refactoring. As an initial set of refactoring methods to investigate, we composed a list of five refactoring methods. We have chosen some refactorings that redistribute responsibilities within classes or between classes and some refactorings that operate at the method/field level. In addition, most of the selected refactorings are supported in popular widely used refactoring tools such as IntelliJ IDEA [16], RefactorIT [17], and Eclipse [18]. The refactorings we investigated are the following:

- **Consolidate Conditional Expression:** combines sequence of conditional tests with the same result into a single conditional expression and then extracts it.
- **Encapsulate Field:** encapsulates public field by making it private and provides accessor (getter and setter methods) to it.
- **Extract Class:** creates a new class and moves the relevant fields and methods from the source class into the new class.
- **Extract Method:** extracts group of statements into a new method.
- **Hide Method:** changes the visibility of a method by making the method private.

B. Software Systems Background

The software systems used in this study are three course projects and three open source projects. The course projects (Project 01, Project 02, and Project 03) were developed by undergraduate students taking “ICS 102 Introduction to Computing” course offered by the Department of Information and Computer Science at King Fahd University of Petroleum and Minerals. The open source projects were downloaded from SourceForge.net [19]. These projects are: JLOC [20], J2Sharp [21], and JNFS (Java Net File Sender) [22]. The subject systems were selected from different domains. Additionally, they were developed by different programmers having different programming abilities and expertise (students/beginners and professionals). Table 1 summarizes some main characteristics of the studied software projects.
Table 1. The characteristics of studied software projects

<table>
<thead>
<tr>
<th>Project</th>
<th>Language</th>
<th># of Classes</th>
<th>Lines of Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Course Projects</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project 01</td>
<td>Java</td>
<td>3</td>
<td>406</td>
<td>A program for managing the cars in a car rental agency</td>
</tr>
<tr>
<td>Project 02</td>
<td>Java</td>
<td>2</td>
<td>299</td>
<td>A program for managing a computer software store</td>
</tr>
<tr>
<td>Project 03</td>
<td>Java</td>
<td>2</td>
<td>334</td>
<td>A program for managing a computer software store</td>
</tr>
<tr>
<td>Open Source Projects</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JLOC</td>
<td>Java</td>
<td>6</td>
<td>308</td>
<td>An application for counting comment, blank, and source code lines</td>
</tr>
<tr>
<td>J2Sharp</td>
<td>Java</td>
<td>4</td>
<td>434</td>
<td>An application for converting Java code into C# code</td>
</tr>
<tr>
<td>JNFS</td>
<td>Java</td>
<td>8</td>
<td>431</td>
<td>An application for sending a file from client to server via the internet</td>
</tr>
</tbody>
</table>

In this study, we are concerned with the nature of change and not with the amount of change; therefore, the size of the system does not affect the results.

C. Data Collection

In this section, we describe the methodology used to collect data from subject systems. The methodology includes the following steps [23]:

1. Collect the internal quality metrics for a studied software system before applying a refactoring method. The Metamata metrics tool [24] was used to collect the internal quality metrics.
2. Perform the refactoring method in the system.
3. Collect the internal quality metrics for the system after applying the refactoring method.
4. Report the changes in the internal quality metrics for each class in the system.
5. Map the changes in the internal quality metrics to the external quality attributes.
6. Repeat steps 1 to 5 for all the investigated refactoring methods.

D. Example

We will show an example of how to study the effect of refactoring methods on the testing effort using the “Extract Method” refactoring. This refactoring technique extracts a set of statements that can be grouped together into a new method and replaces the extracted statements with a call to the new method. Figure 1 shows the source code before applying “Extract Method”. This example is taken from Course Project 02.

In this example, “Extract Method” can be performed to extract the statements in “case 7” from method `showActions()`. To achieve this, we create a new method called `toExit()` in the `StoreManagement` class and then move the statements in “case 7” from `showActions()` method to `toExit()` method.

```java
class StoreManagement{
    ....
    private static void showMenu(){...}
    public static void showActions(){
        int choice;
        do{
            showMenu();
            choice = Integer.parseInt(input.readLine());
            switch(choice){
                case 1: displayAllInventory();
                case 2: findCompanyName();
                case 3: findProductName();
                case 4: modifyPrice();
                case 5: addItem();
                case 6: purchase();
                case 7: saveToFile();
                    update();
                    System.out.println("Thanks");
                System.exit(0);
            }
        }while(choice !=7);
    }
    ...... //other methods
}
```

Figure 1. Before applying “Extract Method”

After that, we replace the statements in “case 7” with a method call to `toExit()`. Figure 2 shows the source code after applying “Extract Method”.

```java
class StoreManagement{
    ....
    private static void showMenu(){...}
    public static void showActions(){
        int choice;
        do{
            showMenu();
            choice = Integer.parseInt(input.readLine());
            switch(choice){
                case 1: displayAllInventory();
                case 2: findCompanyName();
                case 3: findProductName();
                case 4: modifyPrice();
                case 5: addItem();
                case 6: purchase();
                case 7: toExit();
                    update();
                    System.out.println("Thanks");
                System.exit(0);
            }
        }while(choice !=7);
    }
    ...... //other methods
}
```

Figure 2. After applying “Extract Method”

Before the “Extract Method” is applied, the internal quality metrics for the above example (`StoreManagement` class) are collected. They are collected again after “Extract Method” is applied. This enables us to observe the changes in the internal quality metrics caused by applying “Extract Method”. As a result, “Extract Method” increases the RFC, WMC, NOM, and LOC metrics values for `StoreManagement` class. It does not have effect on FOUT metric value.
class StoreManagement{
    ....
    private static void showMenu(){...}
    public static void showActions(){
        int choice;
        do{
            showMenu();
            choice = Integer.parseInt(input.readLine());
            switch(choice){
                case 1: displayAllInventory();
                case 2: findCompanyName();
                case 3: findProductName();
                case 4: modifyPrice();
                case 5: addItem();
                case 6: purchase();
                case 7: toExit();
            }
        }while(choice !=7);
    }
    public static void toExit(){
        saveToFile();
        update();
        System.out.println("Thanks");
        System.exit(0);
    }
    .... //other methods
}

Figure 2. After applying “Extract Method”

To demonstrate the effect of “Extract Method” on the testing effort, we map the changes in the internal quality metrics to the testing effort based on [14]. For example, to see whether “Extract Method” improves the testing effort or not, we can observe from the previous example that “Extract Method” increases the metrics value of RFC, WMC, NOM, and LOC. The changes in these metrics values are mapped to the testing effort based on [14] that shows a positive correlation between the testing effort and RFC, FOUT, WMC, NOM, and LOC metrics. Therefore, “Extract Method” increases (impairs) testing effort as RFC, WMC, NOM, and LOC metrics increase. This means that “Extract Method” increases the testing effort for StoreManagement class.

E. Refactoring Effect on OO Metrics

Table 2 summarizes the changes in the studied internal quality metrics, caused by applying the investigated refactoring methods, using the empirical results from all studied software projects. In order to explain why the internal quality metric increases or decreases as a result of applying a refactoring method, we need to analyze the effect of each refactoring method on the studied internal quality metrics.

- **Consolidate Conditional Expression**: does not use methods or attributes of other classes and consequently it does not have effect on FOUT metric. However, it reduces the value of RFC since it combines a series of conditional checks into a single conditional check. Moreover, it increases the value of RFC, WMC, and NOM, metrics as it extracts the single conditional check into a new method. In summary, “Consolidate Conditional Expression” increases the size of the class in terms of number of methods (RFC, WMC, and NOM), and reduces the number of source code statements (LOC).

- **Encapsulate Field**: does not use methods or attributes of other classes and consequently it does not have effect on FOUT metric. However, it increases the value of RFC, WMC, NOM, and LOC metrics as it provides accessors (setter and getter) methods to a field (attribute). In summary, “Encapsulate Field” increases the size of the class in terms of number of methods and source code statements (RFC, WMC, NOM, and LOC).

- **Extract Method**: does not use methods or attributes of other classes and consequently it does not have effect on FOUT metric. However, it increases the value of RFC, WMC, NOM, and LOC metrics as it extracts group of statements into a new method. In summary, “Extract Method” increases the size of the class in terms of number of methods (RFC, WMC, NOM, and LOC), and reduces the number of source code statements (RFC, WMC, NOM, and LOC).

Table 2. Summary of changes in OO metrics caused by applying refactoring methods

<table>
<thead>
<tr>
<th>Refactoring Method</th>
<th>RFC</th>
<th>FOUT</th>
<th>WMC</th>
<th>NOM</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encapsulate Field</td>
<td>↑</td>
<td>-</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>Extract Method</td>
<td>↑</td>
<td>-</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>Hide Method</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>Consolidate Conditional Expression</td>
<td>↑</td>
<td>-</td>
<td>↑</td>
<td>↑</td>
<td>↓</td>
</tr>
<tr>
<td>Extract Class</td>
<td>↓</td>
<td>↑</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
</tr>
</tbody>
</table>
of methods and source code statements (RFC, WMC, NOM, and LOC).

- **Extract Class**: increases the value of FOUT metric since the source class is coupled to the extracted class. Additionally, it reduces the value of RFC, WMC, NOM, and LOC metrics of the source class as it moves the relevant fields and methods from the source class into the extracted class. However, it increases the number of classes in the system as it extracts new class. In summary, “Extract Class” reduces the size of the source class in terms of number of methods and source code statements (RFC, WMC, NOM, and LOC), increases the coupling between the classes (FOUT), and increases the number of classes in the system.

- **Hide Method**: changes the visibility of a method from public to private if the method is not used by other classes. Therefore, it does not affect any of the investigated internal quality metrics.

**F. Refactoring Effect on Testing Effort**

In order to classify the investigated refactoring methods based on the testing effort, we rely on the findings of Bruntink and van Deursen [14] study that shows a correlation between the testing effort and the internal quality metrics. Therefore, the classification is done by mapping the changes in the internal quality metrics to the testing effort based on [14].

Table 3 summarizes the classification of the investigated refactoring methods based on the testing effort using the empirical results from all studied software projects, where “↑” symbol represents an increase in testing effort, “↓” symbol represents a decrease in testing effort, and “±” symbol represents no change in testing effort.

**G. Discussion and Observations**

This study provides a number of interesting results which can be observed as follows. First, the results obtained from software projects developed by undergraduate students are same as the results obtained from open source software projects. This strengthens our refactoring classification as we consider different programming abilities and expertise (students / beginners and professionals), as well as different project domains and development environments.

Table 3. Classification of refactoring methods based on software testing effort

<table>
<thead>
<tr>
<th>Refactoring Method</th>
<th>Testing Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encapsulate Field</td>
<td>↑</td>
</tr>
<tr>
<td>Extract Method</td>
<td>↑</td>
</tr>
<tr>
<td>Consolidate Conditional Expression</td>
<td>↑</td>
</tr>
<tr>
<td>Hide Method</td>
<td>-</td>
</tr>
<tr>
<td>Extract Class</td>
<td>↓</td>
</tr>
</tbody>
</table>

Second, we can observe from Table 2 that “Encapsulate Field” and “Extract Method” increase the class size in terms of WMC, NOM, and LOC metrics since they introduce new methods. On the other hand, “Extract Class” reduces the source class size in terms of WMC, NOM, and LOC metrics as it moves relevant fields and methods from the source class to the extracted class.

Third, the results support the earlier explanation that “Extract Class” increases the number of classes in the system.

Fourth, one of the objectives of good software design is to reduce the coupling where possible [25]. We can observe from Table 2 that “Extract Class” increases the coupling (FOUT). Therefore, “Extract Class” cannot help to achieve this objective.

Fifth, we can observe from Table 2 that “Extract Class” increases the value of FOUT metric while it reduces the value of RFC, WMC, NOM, and LOC metrics. This situation needs further analysis. Assume that the value of FOUT metric was $m$ before applying “Extract Class”. After applying “Extract Class” the value of FOUT metric becomes $m+1$ i.e. the value is increased by one. This is because that the source class is coupled to the extracted class. On the other hand, the value of RFC, WMC, NOM, and LOC metrics is decreased by more than one depends on the number of fields and methods moved to the extracted class. Overall, we can conclude that the effect of FOUT is not dominant compared with the other investigated metrics. Consequently, “Extract Class” reduces the testing effort needed to test the source class as it reduces the value of RFC, WMC, NOM, and LOC metrics.

Finally, the results suggest that “Encapsulate Field”, “Extract Method”, and “Consolidate Conditional Expression” increase the testing effort for a class. This is because that they add more responsibilities to the class by creating new methods. This, in turn, makes the class more complex. On the other hand, “Extract Class” reduces the testing effort needed to test the source class, because it moves some fields and methods to the extracted class. Hence, it makes the source class less complex.

**H. Threats to Validity**

There are some limitations to the extent to which these results can be generalized. The following are possible reservations:

- Our methodology to investigate the effect of refactoring methods on the testing effort was based on mapping the changes in the internal quality metrics, caused by applying refactoring methods, to the testing effort. This mapping was done based on available research study that shows a correlation between internal quality metrics and the testing effort. Moreover, the relation between the internal quality metrics and the testing effort was based on this research study without validation on our behalf of their findings regarding the correlations.

Another possible threat is the use of small-size systems. This would be important when investigating the effect at the
system-level. Nevertheless, this study was limited to investigating the effect of refactoring at the class-level not at the system-level. Therefore, the system size is not crucial as it does not alter the results significantly.

We do not validate if the refactored classes that show an improvement in the testing effort are indeed easier to test than before. Such validation should be addressed in a future study.

V. CONCLUSION

In this paper, we proposed a classification of refactoring methods based on their measurable effect on (i) internal quality metrics and (ii) software testing effort which is one of the external software quality attributes. This, in turn, helps the software developers decide which refactoring methods can help in optimizing a software system with regard to the testing effort. The classification is done by mapping the changes in the internal quality metrics, caused by applying refactoring methods, to the testing effort based on study presented in [14] that shows a positive correlation between the testing effort and RFC, FOUT, WMC, NOM, and LOC metrics. The resulting classification indicates that “Encapsulate Field”, “Extract Method”, and “Consolidate Conditional Expression” increase the testing effort, while “Extract Class” reduces the testing effort. Additionally, “Hide Method” has no effect on the testing effort at all.

One direction of future work would be classifying refactoring methods based on other software quality attributes such as maintainability, and reusability. It is also interesting to classify a more extended set of refactoring methods based on software quality attributes to form a large classification catalog.

Usually software developers are encouraged to apply multiple refactoring methods. Hence, another possible direction of future work would be investigating the effects of not single but multiple refactoring methods together on the software quality attributes of a system. This would be especially valuable when studying groups of refactoring methods that have inverse (conflicting) effects on software quality attributes.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their constructive comments. This work is supported by King Fahd University of Petroleum and Minerals under FT-2008/11.

REFERENCES