Using Software Quality Attributes to Classify Refactoring to Patterns

Karim O. Elish
Department of Computer Science
Virginia Tech
Email: kelish@vt.edu

Mohammad Alshayeb
Information and Computer Science Department
King Fahd University of Petroleum & Minerals
Email: alshayeb@kfupm.edu.sa

***Abstract*—Refactoring to patterns allows software designers to safely move their designs towards specific design patterns by applying multiple low-level refactorings. There are many different refactoring to pattern techniques, each with a particular purpose and a varying effect on software quality attributes. Thus far, software designers do not have a clear means to choose refactoring to pattern techniques to improve certain quality attributes. This paper takes the first step towards a classification of refactoring to pattern techniques based on their measurable effect on software quality attributes. This classification helps software designers in selecting the appropriate refactoring to pattern techniques that will improve the quality of their design based on their design objectives. It also enables them to predict the quality drift caused by using specific refactoring to pattern techniques.**

***Index Terms*—refactoring to patterns, software metrics, software quality.**

I. INTRODUCTION

In object-oriented paradigm, the process of making changes to software that affect its internal structure without altering its behavior is called "refactoring" [1, 2]. Refactoring is a practice that reduces software complexity by improving its internal structure.

The concept of refactoring to patterns was introduced in Joshua Kerievsky's book "Refactoring to Patterns" [3]. Refactoring to patterns captures the relation between refactoring and design patterns. Kerievsky defined refactoring to patterns as *marriage of refactoring (the process of improving the design of existing code) with patterns (the classic solutions to recurring design problems)* [3]. In other words, refactoring to patterns allows software designers to safely move their designs towards design patterns by applying sequences of low-level refactorings [3]. The main advantage of refactoring to patterns is the introduction of a higher abstraction level. There are many different refactoring to pattern techniques, each having a particular purpose and effect. Consequently, the effects of refactoring to pattern techniques on software quality attributes may vary [4]. This means that when applying refactoring to patterns, some quality attributes can be improved and some others can be impaired. Moreover, software designers usually design for specific design goals, of which the means of achieving may sometimes contradict. It is so far imprecise for software designers how to use refactoring to pattern techniques to improve specific quality attributes.

The objective of this paper is to propose a classification of refactoring to pattern techniques based on their measurable effect on internal and external software quality attributes. This classification aims to help software designers in choosing appropriate refactoring to pattern techniques that will improve the quality of their design based on their design objectives. It also enables them to predict the quality drift caused by using specific refactoring to pattern techniques.

The rest of this paper is organized as follows. Section 2 reviews the related work. Section 3 describes internal and the external software quality attributes, and means of assessing external quality attributes based on internal quality attributes. Section 4 describes our research methodology. Section 5 presents the classification of refactoring to pattern techniques based on software quality attributes. Section 6 discusses possible threats to the validity of our approach. Finally, Section 7 concludes the paper and gives directions for future work.

II. SOFTWARE QUALITY ATTRIBUTES

Software quality is the degree to which software possesses a desired combination of attributes (e.g. adaptability and reusability) [5]. This means that defining software quality for a system is equivalent to defining a list of software quality attributes for that system [5]. ISO/EIC 9126 standard [6] defines software quality characteristics as "a set of attributes of a software product by which its quality is described and evaluated". The factors that affect software quality can be classified into two groups [7]: (*i*) factors that can be directly measured i.e. internal quality attributes (e.g. length of program as lines of code) and (*ii*) factors that can be measured only

doi:10.4304/jsw.7.2.408-419

indirectly i.e. external quality attributes (e.g. maintainability and reliability). In the following subsections, we define the internal and the external software quality attributes used in our classification and describe how external quality attributes can be assessed using internal quality attributes.

A. Internal Quality Attributes under Study

Software metrics are typically used as internal quality attributes [8]. A software metric is a characteristic (attribute) of software product, process, or resource [7, 8]. In order to classify refactoring to pattern techniques based on internal quality attributes, we used a suite of metrics that measures the structural quality of object-oriented code and design. More specifically, we consider the Chidamber and Kemerer metrics suite [9] which consists of Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling between Objects (CBO), Response for a Class (RFC), Weighted Methods per Class (WMC), and Lack of Cohesion on Methods (LCOM). In addition, we include one metric which measures class coupling (Fan out) and two metrics which measure class size (Number of Methods and Lines of Code). We chose these metrics because of their wide acceptance among the software engineering community. Moreover, they capture important dimensions of object-oriented design characteristics: inheritance by (DIT, NOC), coupling by (CBO, RFC, FOUT), size/complexity by (WMC, NOM, LOC), and cohesion by (LCOM). Additionally, they have been used by several previous empirical studies such as [10-16] to investigate their correlation with external software quality attributes. In particular, these metrics were investigated by (i) Dandashi [14] to assess adaptability, completeness, maintainability, understandability, and reusability and (ii) Bruntink and van Deursen [15] to assess testability. The metrics we investigate are the following:

- **Depth of Inheritance Tree (DIT)**: defined as the length of the longest path from a given class to the root class in the inheritance hierarchy.
- **Number of Children (NOC)**: defined as the number of classes that inherit directly from a given class.
- **Coupling between Objects (CBO)**: defined as the number of distinct non-inheritance related classes to which a given class is coupled. A class is coupled to another if it uses methods or attributes of the coupled class.
- **Response for a Class (RFC)**: 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)**: 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).
- **Lack of Cohesion on Methods (LCOM)**: defined as the number of pairs of member functions without shared instance variables, minus the number of pairs of member functions with shared instance variables. However, the metric is set to zero whenever this subtraction is negative.
- **Fan out (FOUT)**: defined as the number of classes that a given class uses, not the classes it is used by.
- **Number of Methods (NOM)**: defined as the number of methods implemented in a given class.
- **Lines of Code (LOC)**: defined as the total source lines of code in a class excluding all blank and comment lines.

B. External Quality Attributes under Study

To classify refactoring to pattern techniques based on external quality attributes, we identified a set of external quality attributes as follows:

- **Adaptability**: defined as the ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed [17].
- **Completeness**: defined as the degree to which the component implements all required capabilities [18].
- **Maintainability**: defined as the ease with which a component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment [17].
- **Understandability**: defined as the degree to which the meaning of a software component is clear to a user [18].
- **Reusability**: defined as the degree to which a component can be used in more than one software system, or in building other components, with little or no adaptation [18].
- **Testability**: defined as a set of attributes of software that bear on the effort needed to validate the software product [6].

C. Assessing External Quality Attributes Using OO Metrics

There is no direct way to measure external software quality attributes, which are also called indirect attributes for this reason. Nonetheless, software metrics can be used as indicators (estimators/predictors) for external software quality attributes. Several empirical studies such as [10-16] used software metrics as indicators to assess external quality attributes. These empirical studies provide some empirical evidence indicating that most of object-oriented metrics can be useful quality indicators.

In order to assess adaptability, completeness, maintainability, understandability, and reusability from internal quality metrics, we rely on the results of an existing research study. Specifically, we rely on the findings of Dandashi [14] that showed a correlation between the specified external attributes and internal quality metrics. Dandashi demonstrated a method for assessing indirect quality attributes (adaptability, completeness, maintainability, understandability, and reusability) of object-oriented systems from the direct (internal) quality attributes (McCabe’s Cyclomatic Number, Halstead’s Volume, WMC, DIT, NOC, RFC, CBO, and LOC). A case study of the feasibility of applying direct measurements to assess the indirect quality attributes was conducted using C++ code components taken from the object-oriented Particle-In-Cell Simulations (PICS) problem domain. A survey was conducted to gather data about the PICS indirect quality attributes. The results showed that indirect quality attributes measured by human analysis with direct quality attributes measured by the automated tool provide empirical evidence that direct and indirect quality attributes do correlate. More specifically, Complexity, Volume, WMC, and LOC are proportional (positively correlated) to the levels of adaptability, completeness, maintainability, and understandability while NOC, DIT, RFC, and CBO are inversely proportional (negatively correlated) to the levels of adaptability, completeness, maintainability, and understandability. In addition, reusability can be estimated from adaptability, completeness, maintainability, and understandability.

In order to assess testability from internal quality metrics, we rely on the findings of another research study that showed a correlation between testability and internal quality metrics. Bruntink and van Deursen [15] 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. LOCC (Lines of Code for Class) and NOTC (Number of Test Cases) were used to represent the size of a test suite, while DIT, FOUT, LCOM, LOC, NOC (Number of Children), NOF (Number of Fields), NOM (Number of Methods), RFC, and WMC were 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 existed. The systems included four commercial systems and one open source system (Apache Ant). They were able to find a significant 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.

Table1 summarizes relationships between the internal quality attributes (DIT, NOC, CBO, RFC, FOUT, WMC, NOM, LOC, and LCOM) and the external quality attributes (adaptability, completeness, maintainability, understandability, reusability, and testability), where “+ve” sign represents positive correlation, “-ve” sign represents negative correlation, “0” sign represents no correlation at all, and “NA” sign represents that the correlation between an internal quality attribute and an external quality attribute was not studied.

TABLE 1. THE RELATIONSHIP BETWEEN INTERNAL AND EXTERNAL SOFTWARE QUALITY ATTRIBUTES

External Quality Attribute	Study	Internal Quality Attribute								
		DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Adaptability	Dandashi [14]	-ve	-ve	-ve	-ve	*NA*	+ve	*NA*	+ve	*NA*
Completeness	Dandashi [14]	-ve	-ve	-ve	-ve	*NA*	+ve	*NA*	+ve	*NA*
Maintainability	Dandashi [14]	-ve	-ve	-ve	-ve	*NA*	+ve	*NA*	+ve	*NA*
Understandability	Dandashi [14]	-ve	-ve	-ve	-ve	*NA*	+ve	*NA*	+ve	*NA*
Reusability	Dandashi [14]	-ve	-ve	-ve	-ve	*NA*	+ve	*NA*	+ve	*NA*
Testability	Bruntink and van Deursen [15]	0	0	*NA*	+ve	+ve	+ve	+ve	+ve	0

III. RELATED WORK

Software refactoring has been studied extensively by the software engineering research community. However, only a few studies have investigated the effect of refactoring on software quality attributes. In this section, we provide an overview of the studies that have investigated the effect of refactoring with respect to the quality attributes.

Stroggylos and Spinellis [19] analyzed source code version control system logs of four popular open source software systems. They detected changes marked as refactorings and examined how those changes affected software metrics. The metrics examined included the C&K metrics suite [9], Ca (Afferent Coupling), and NPM (Number of Public Methods). Their results indicated significant negative changes in certain metric values. Specifically, refactoring methods caused an increase in metrics such as LCOM, Ca, and RFC; this indicated that the used refactoring methods caused classes to become less coherent. Du Bois and Mens [20] proposed a formalism to describe the impact of refactoring on internal program quality. They based their formalism on the abstract syntax tree representation of source-code, extended with cross-references. They focused on three refactoring methods: “Encapsulate Field”, “Pull up Method”, and “Extract Method” and analyzed their impact on internal program quality metrics (NOM, NOC, CBO, RFC, and LCOM). Boshnakoska and Mišev [21] investigated the correlation between Object-Oriented metrics and refactoring. They used an extended C&K metrics suite and concluded that object-oriented metrics can be used to identify classes that require immediate attention

Kosker et al. [22] proposed a machine learning based model, using Weighted Naive Bayes, to predict classes to be refactored by analyzing the code complexity.

Al Dallal and Briand [23] proposed a formula that precisely measures the degree of interaction between each pair of methods which is used to introduce a low-level design class cohesion metric (LSCC); the metric is used as indicator for refactoring weakly cohesive classes. Cinnéide et al. [24] used the LSCC metric proposed by Al Dallal and Briand [23] to propose an automated refactoring approach to improve the cohesion properties of the program to improve the program testability.

Several empirical investigation efforts relate to this study. Du Bois et al. [25] performed a controlled experiment to empirically examine differences in program comprehension between the application of Refactor to Understand and the traditional Read to Understand patterns. Refactor to Understand is a reverse engineering pattern that is used to improve the code and the maintainers' understanding of the code. Geppert et al. [26] empirically explored the impact of refactoring a legacy system on changeability. They considered three factors for changeability: customer reported defect rates, effort, and scope of changes.

Wilking et al. [27] 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. The direct effect of an improved maintainability or modifiability caused by refactoring could not be shown within this controlled experiment.

Kataoka et al. [28] proposed coupling metrics as a quantitative evaluation method to measure the effect of refactoring on the maintainability of a program by comparing the coupling before and after refactoring. Du Bois et al. [29] developed practical guidelines for applying refactoring methods to improve coupling and cohesion characteristics and validated these guidelines on an open source software system. They assumed that coupling and cohesion are quality attributes generally recognized as indicators for software maintainability. The set of refactoring methods used in their study included: "Extract Method", "Move Method", "Replace Method with Method Object", "Replace Data Value with Object", and "Extract Class". The results indicated that these guidelines can be used to improve coupling and cohesion and consequently improve maintainability. "Extract Method" was excluded; it did not help in improving neither coupling nor cohesion. Bryton and Abreu [30] provided an initial proposal of a method for refactoring with quantitative and experimental grounds to evaluate the effect of refactoring on maintainability. Moser et al. [31] proposed a methodology to assess whether or not refactoring improves reusability and promotes ad-hoc reuse in an XP-like development environment. They focused on internal software metrics (CBO, LCOM, RFC, WMC, DIT, NOC, LOC, and McCabe's cyclomatic complexity) that are considered to be relevant to reusability. They conducted a case study on a software project developed using an agile, XP-like methodology and analyzed the impact of refactoring on internal quality metrics. Their results indicated that refactoring has a positive effect on reusability and thus promotes ad-hoc reuse in XP-like environments.

Elish and Alshayeb [32] classified a set of refactoring methods [1] based on their effect on a set of internal and external software quality attributes. Furthermore, Alshayeb [33] conducted an empirical study to assess the effect of refactoring proposed by Fowler [1] on software quality. He found that refactoring as a whole practice does not necessarily improve all software quality attributes. This motivated us to investigate the effect of refactoring to patterns and classify them based on their measurable effect on software quality attributes.

A summary of the related work is presented in Table 2. The existing research studies reviewed above were limited to the effects of refactoring methods proposed by Fowler [1] on internal or external quality attributes. They did not investigate the effect of refactoring to pattern methods proposed by Kerievsky [3] on internal or external quality attributes. Additionally, Kerievsky in his refactoring to patterns catalog stated the benefits and liabilities of each refactoring to pattern, however, he did not provide any quantitative assessment for the effect of each refactoring to pattern on software quality. Accordingly, the originality of this research focuses on classifying refactoring to pattern methods based on their measurable effect on several software quality attributes including internal and external quality attributes.

IV. RESEARCH METHOD

The main research question addressed by this study is *how do refactoring to pattern techniques affect software quality attributes?*

To tackle this research question, we study the measurable effects of refactoring to pattern methods on software quality attributes. Then we classify different refactoring to pattern methods based on the quality attributes they improve. This, in turn, allows us to improve the quality of a software system by applying the appropriate refactoring to pattern techniques.

The remainder of this section is organized as follows. First, we present the selected refactoring to pattern techniques. Then, we provide a description of software systems used in our study. Finally, we describe our methodology to collect the data from the subject systems and provide an example of this methodology.

A. Refactoring to Pattern Techniques under Study

Kerievsky [3] defined twenty-seven pattern-directed refactorings in his book. For each one of them, the motivation of why the refactoring should be performed and step-by-step description of how to carry out the refactoring were described. As an initial set, we composed a list that covers the categories of the refactoring to patterns catalog and consists of refactoring to pattern techniques among the most popular and most widely used. The chosen methods for investigation are the following [3]:

- **Chain Constructors**: chains the constructors together to get the least amount of duplicate code.
- **Compose Method**: transforms a method's logic into a small number of steps at the same level of detail.
- **Form Template Method**: generalizes the methods in subclasses by extracting their steps into methods with identical signatures, and then pulls up the generalized methods to form a "Template Method".
- **Introduce Null Object**: replaces the null logic with a null object which provides the appropriate null behavior.
- **Replace Conditional Dispatcher with Command**: creates a command for each action to execute, stores the commands in a collection, and replaces the conditional logic with code to fetch and execute commands.
- **Replace Constructors with Creation Methods**: replaces the constructors with intention-revealing creation methods that return object instances.
- **Unify Interfaces**: provides a superclass/interface with same interface as a subclass.

TABLE 2. SUMMARY OF RELATED WORK

Study	Studied Internal Quality Attributes	Studied External Quality Attributes	Refactoring to Patterns Classification?
Stroggylos and Spinellis [19]	C & K metrics, Ca, NPM	-	No
Du Bois and Mens [20]	NOM, NOC, CBO, RFC, LCOM	-	No
Boshnakoska and Mišev [21]	LCOM, DIT, IFANIN CBO, NOC, RFC, NIM, NIV, WMC	-	No
Kosker et al. [22]	Complexity metrics	-	No
Al Dallal and Briand [23]	Cohesion	-	No
Cinnéide et al. [24]	Cohesion	-	No
Du Bois et al. [25]	-	Understandability	No
Geppert et al. [26]	-	Changeability	No
Wilking et al. [27]	-	Maintainability Modifiability	No
Kataoka et al. [28]	Coupling	Maintainability	No
Du Bois et al. [29]	Coupling, Cohesion	Maintainability	No
Bryton and Abreu [30]	Cohesion Size metrics Complexity metric	Maintainability	No
Moser et al. [31]	C & K metrics, MCC, LOC	Reusability	No
Elish and Alshayeb [32]	C & K metrics, FOUT, NOM, LOC	Adaptability Completeness Maintainability Understandability Reusability Testability	No
Alshayeb [33]	C & K metrics, FOUT, NOM, LOC	Adaptability Maintainability Understandability Reusability Testability	No
This Work	C & K metrics, FOUT, NOM, LOC	Adaptability Completeness Maintainability Understandability Reusability Testability	Yes

B. *Software Systems Background*

The software systems used in this study are four open source systems. They were downloaded from SourceForge.net[1]. The open source systems are: HTML Parser[2], Java Class Browser[3], Java Neural Network Trainer[4], and JMK (Make in Java)[5].

Table 3 summarizes some main characteristics of these systems. The subject systems were selected from different domains (Java parser, class browser, neural networks and file manager) and different system sizes (number of classes range from 9-202). In this study, we focus on studying the effect of refactorings at the class-level not at the overall system-level [34]. Therefore, the size of the system is not crucial.

C. *Data Collection*

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

1. Collect the internal quality metrics for each class in a studied software system before applying a refactoring to pattern technique. The *Metamata*[6] metrics tool was used to collect the internal quality metrics.
2. Manually perform the refactoring to pattern technique on the classes of the system (where applicable).
3. Collect the internal quality metrics for each class affected in the system after applying the refactoring to pattern technique.
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 to pattern techniques.

D. *An Example*

This section shows a simple example of how to study the effect of refactoring to patterns on the software quality attributes using the "Unify Interfaces" refactoring. This refactoring method allows you to provide an interface to a superclass that is identical to the subclass's interface. To achieve this, find all public methods in the subclass that are missing in the superclass. Then, add copies of these missing methods to the superclass and change each one to perform null behavior [3]. Figure 1 shows the UML class diagram of the illustrated example before and after the application of "Unify Interfaces". Figure 2 and Figure 3 show the corresponding source codes before and after the application of "Unify Interfaces" respectively. This example is taken from Java Neural Network Trainer system.

[1] http://sourceforge.net/index.php
[2] http://sourceforge.net/projects/htmlparser/
[3] http://sourceforge.net/projects/classbrowser/
[4] http://sourceforge.net/projects/javanntrain/
[5] http://sourceforge.net/projects/jmk/
[6] www.metamata.com

In this example, "Unify Interfaces" can be performed to provide the *Trainer* superclass with the same interface as its *QuickProp* subclass. To achieve this, find all public methods in the *QuickProp* subclass that are missing in the *Trainer* superclass. We can observe that the *addMomentum (double momentum)* method exists in *QuickProp* subclass and it is missing in the *Trainer* superclass. Hence, we add a copy of the missing method *addMomentum (double momentum)* to the *Trainer* superclass and then alter this method to perform null behavior. This refactoring makes the classes of the objects share a common interface.

Before "Unify Interfaces" is applied, the internal quality metrics for the above example are collected. They are collected again after "Unify Interfaces" is applied. This enables us to observe the changes in the internal quality metrics caused by applying "Unify Interfaces". Table 4 presents the changes in the internal quality metrics caused by "Unify Interfaces" for the *Trainer* class, where "↑" symbol represents an increase in a metric value, "↓" symbol represents a decrease in a metric value, and "-" symbol represents no change in a metric value.

To demonstrate the effect of "Unify Interfaces" on the investigated external quality attributes, we map the changes in the internal quality metrics to the external quality attributes based on [14, 15]. For example, to see whether "Unify Interfaces" improves the testability or not, we can observe from Table 4 that "Unify Interfaces" increases the metrics value of RFC, WMC, NOM, LOC, and LCOM. The changes in these metrics values are mapped to testability based on [15] that shows a positive correlation between testability and RFC, FOUT, WMC, NOM, and LOC metrics. Therefore, "Unify Interfaces" increases (impairs) testability (testing effort) as RFC, WMC, NOM, and LOC metrics increase. This means that "Unify Interfaces" increases the testing effort for *Trainer* class (superclass).

TABLE 4. CHANGES IN OO METRICS CAUSED BY "UNIFY INTERFACES": TRAINER CLASS

DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
-	-	-	↑	-	↑	↑	↑	↑

V. A CLASSIFICATION OF REFACTORING TO PATTERNS

In Table 5, we summarize the changes in the studied internal quality metrics caused by applying the investigated refactoring to pattern techniques using the empirical results from all the studied software systems. A "↑" symbol represents an increase in a metric value, a "↓" symbol represents a decrease in a metric value, and a "-" symbol represents no change in a metric value.

TABLE 3. THE CHARACTERISTICS OF STUDIED SOFTWARE SYSTEMS

System	Language	# of Classes	Lines of Code	Description
HTML Parser	Java	202	25992	An application used to parse HTML in either a linear or nested fashion. Primarily used for transformation or extraction.
Java Class Browser	Java	9	647	An application used to view the class file names in a Java archive or Directory with multiple archives.
Java Neural Network Trainer	Java	11	1171	An application used to add new training algorithms and training patterns.
JMK	Java	47	5016	An application used to ensure that a set of files is in a consistent state.

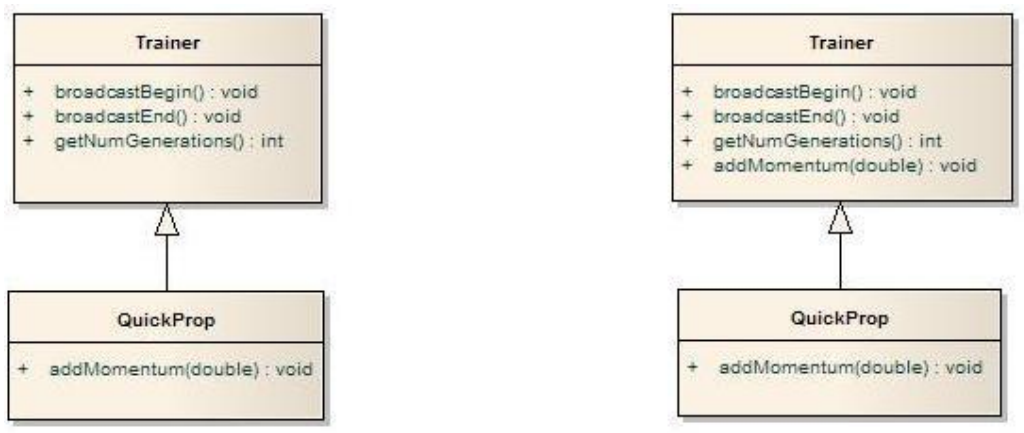
Trainer
+ broadcastBegin() : void
+ broadcastEnd() : void
+ getNumGenerations() : int
QuickProp
+ addMomentum(double) : void
Trainer
+ broadcastBegin() : void
+ broadcastEnd() : void
+ getNumGenerations() : int
+ addMomentum(double) : void
QuickProp
+ addMomentum(double) : void

Figure 1. Example of UML class diagram before and after applying “Unify Interfaces”

```
class Trainer{
   .....
   public void broadcastBegin(){...}
   public void broadcastEnd(){...}
   public int getNumGenerations(){...}
 }

 class QuickProp extends Trainer{
   .....
   public void addMomentum(double momentum){

       //... implementation details
   }
 }
```

Figure 2. Before applying “Unify Interfaces”

```
class Trainer{
   .....
   public void broadcastBegin(){...}
   public void broadcastEnd(){...}
   public int getNumGenerations(){...}
   public void addMomentum(double momentum){}

 }

 class QuickProp extends Trainer{
   .....
   public void addMomentum(double momentum){

       //... implementation details
   }
 }
```

Figure 3. After applying “Unify Interfaces”

TABLE 5. CLASSIFICATION OF REFACTORING TO PATTERNS BASED ON INTERNAL SOFTWARE QUALITY ATTRIBUTES

Refactoring to Patterns	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Compose Method	-	-	-	↑	-	↑	↑	↑	↑
Form Template Method	-	-	-	↑	-	↑	↑	↑	↑
Replace Constructors with Creation Methods	-	-	-	↑	-	↑	↑	↑	↑
Unify Interfaces	-	-	-	↑	-	↑	↑	↑	↑
Chain Constructors	-	-	-	-	-	-	-	↓	-
Introduce Null Object	-	↑	-	-	-	-	-	↓	-
Replace Conditional Dispatcher with Command	-	-	↑	↑	↑	↑	↑	↓	↑

In order to explain why the internal quality metric increases or decreases as a result of applying a refactoring to pattern technique, we need to analyze the effect of each refactoring to pattern technique on the internal quality metrics as follows:

- **Chain Constructors**: chains the constructors together to reduce the amount of duplicate code. Therefore, it does not affect any of the investigated internal quality metrics except that it reduces the source code statements (LOC).
- **Compose Method**: does not affect the DIT and NOC metrics because it neither inherits a class nor creates subclasses. It does not use methods or attributes of other classes; consequently, it does not have an effect on CBO and FOUT metrics. It increases the values of RFC, WMC, NOM, LOC, and LCOM metrics because it composes methods (extracts several groups of statements into new methods). In summary, "Compose Method" increases the size of the class in terms of number of methods and source code statements (RFC, WMC, NOM, and LOC), and makes the class less cohesive as it assigns more responsibilities to it.
- **Form Template Method**: does not affect the DIT and NOC metrics because it neither inherits a class nor creates subclasses. It does not use methods or attributes of other classes; consequently, it does not have an effect on CBO and FOUT metrics. It increases the values of RFC, WMC, NOM, LOC, and LCOM metrics since the parent class generalizes the methods in the subclasses by extracting their steps into methods with identical signatures and then pulls up the generalized methods to form a Template Method. In summary, "Form Template Method" increases the size of the parent class in terms of number of methods and source code statements (RFC, WMC, NOM, and LOC), and makes the parent class less cohesive as it assigns more responsibilities to it.
- **Introduce Null Object**: replaces the null logic with a null object which provides the appropriate null behavior. The null object is created by extracting the subclass in the source class. The NOC increases as the source class has a new subclass and the number of classes in the system is increased as well. The LOC metric of the target class is reduced as the method removes the alternative actions if the object is null. In summary, "Introduce Null Object" increases the number of subclasses (NOC) of the source class, reduces the size of the target class in terms of source code statements (LOC), and increases the number of classes in the system.
- **Replace Conditional Dispatcher with Command**: creates a command for each action by creating a new class (concrete command) for each command to handle the action. Then, it creates an interface or abstract class that declares an execution method. After that, in the class that contains the conditional dispatcher, it defines and populates a command map that contains instances of each concrete command. Therefore, "Replace Conditional Dispatcher with Command" does not affect the DIT and NOC metrics because it does not inherit a class and is not inherited by other classes. It increases the values of CBO and FOUT metrics since the class that contains the conditional dispatcher is coupled to the abstract class that declares an execution method. It increases the values of RFC, WMC, NOM, and LCOM metrics of the class that contains the conditional dispatcher as it defines and populates a command map. It reduces the LOC metric of the class that contains the conditional dispatcher as it moves the actions implementation to the concrete command. The number of classes in the system is increased since a new class is created for each command to handle the action. In summary, "Replace Conditional Dispatcher with Command" increases the size of the class in terms of number of methods (RFC, WMC, and NOM), increases the coupling between the classes (CBO, FOUT), reduces the source code statement (LOC), makes the class less cohesive as it assigns more responsibilities to it, and increases the number of classes in the system.
- **Replace Constructors with Creation Methods**: does not affect the DIT and NOC metrics because it neither inherits a class nor creates subclasses. It does not use methods or attributes of other classes; consequently, it does not have an effect on CBO and FOUT metrics. It increases the values of RFC, WMC, NOM, LOC, and LCOM metrics since it replaces the constructors with creation methods that

return object instances. In summary, "Replace Constructors with Creation Methods" increases the size of the class in terms of number of methods and source code statements (RFC, WMC, NOM, and LOC), and makes the class less cohesive as it assigns more responsibilities to it.

- **Unify Interfaces**: does not affect the DIT and NOC metrics because it neither inherits a class nor creates subclasses. It does not use methods or attributes of other classes; consequently, it does not have an effect on CBO and FOUT metrics. It increases the values of RFC, WMC, NOM, LOC, and LCOM metrics of the superclass; this is because it adds to the superclass copies of all public methods in the subclass that are missing from the superclass. In summary, "Unify Interfaces" increases the size of the superclass class in terms of number of methods and source code statements (RFC, WMC, NOM, and LOC), and makes the superclass class less cohesive as it assigns more responsibilities to it.

To classify the refactoring to pattern techniques according to their effects on external quality attributes, we rely on the findings of the existing research studies that show a correlation between external quality attributes and internal quality metrics. More specifically, we rely on the results found by (*i*) Dandashi [14] to assess adaptability, completeness, maintainability, understandability, and reusability and (*ii*) Bruntink and van Deursen [15] to estimate the testing effort. The classification is done by mapping the changes in the internal quality metrics to the external quality attributes based on these studies.

Table 6 presents the classification of the investigated refactoring to pattern techniques based on the external quality attributes using empirical results from the studied software systems. The "↑" symbol represents an increase (improvement) in the investigated external quality attributes except for testability (testing effort) for which it means impairment. The "↓" symbol represents a decrease (impairment) in the investigated external quality attributes except for testability (testing effort) for which it means improvement. And the "-" symbol represents no change in the external quality attribute.

A. *Overall Observations*

This study provides a number of interesting results which can be observed as follows:

- The change in internal software quality metrics was observed in all cases of studied software systems.
- We can observe from Table 5 that "Compose Method", "Form Template Method", "Replace Constructors with Creation Methods", and "Unify Interfaces" increase the class size in terms of WMC, NOM, and LOC metrics as they introduce new methods.
- The empirical results support the earlier explanation that "Introduce Null Object" and "Replace Conditional Dispatcher with Command" increase the number of classes in the system.
- One of the objectives of good software design is to reduce the coupling where possible and increase the cohesion where possible [35]. We can observe from Table 5 that "Replace Conditional Dispatcher with Command" increases the coupling (CBO and FOUT). Additionally, "Compose Method", "Form Template Method", "Replace Constructors with Creation Methods", "Unify Interfaces", and "Replace Conditional Dispatcher with Command" reduce the cohesion (increase LCOM). It is thus possible to make a decision regarding which of the above refactorings should be used in design according to the desired effects on cohesion and coupling.
- Some notable effects can be observed from Table 5 like "Introduce Null Object" and "Replace Conditional Dispatcher with Command" can have varying effects on internal quality metrics. These situations need further analysis:
  a) "Introduce Null Object": it increases the NOC metric value by one as the source class has a new subclass. However, it reduces the LOC metric value of the target class by more than one as it removes the alternative actions if the object is null. Overall, we can conclude that the effect of NOC is not dominant compared with LOC. Consequently, "Introduce Null Object" impairs (decreases) adaptability, completeness, maintainability, understandability, reusability, and improves testability (decreases testing effort) as it increases the value of the LOC metric.
  b) "Replace Conditional Dispatcher with Command": it reduces the value of the LOC metric of the class that contains the conditional dispatcher since it moves the actions' implementation to the concrete commands classes. However, it increases the values of the CBO and FOUT metrics of the class that contains the conditional dispatcher since this class is coupled to the abstract class that declares an execution method. Additionally, this method increases the values of RFC, WMC, NOM, and LCOM metrics of the class that contains the conditional dispatcher as the command map is defined and populated. Overall, we can conclude that the effect of LOC metric is not dominant compared with the other investigated metrics. Consequently, "Replace Conditional Dispatcher with Command" impairs (decreases) adaptability, completeness, maintainability, understandability, reusability, and impairs testability (increases testing effort) as it increases the values of CBO, FOUT, RFC, WMC, NOM, and LCOM metrics.

TABLE 6. CLASSIFICATION OF REFACTORING TO PATTERNS BASED ON EXTERNAL SOFTWARE QUALITY ATTRIBUTES

Refactoring to Patterns	Adaptability	Completeness	Maintainability	Understandability	Reusability	Testability
Compose Method	↑	↑	↑	↑	↑	↑
Form Template Method	↑	↑	↑	↑	↑	↑
Replace Constructors with Creation Methods	↑	↑	↑	↑	↑	↑
Unify Interfaces	↑	↑	↑	↑	↑	↑
Chain Constructors	↓	↓	↓	↓	↓	↓
Introduce Null Object	↓	↓	↓	↓	↓	↓
Replace Conditional Dispatcher with Command	↓	↓	↓	↓	↓	↑

VI. THREATS TO VALIDITY

There are some limitations to the extent to which these results can be generalized. The external limitations are: 1) the code used was written using Java, different languages may yield to different results as they have different constructs, 2) we considered open-source systems in our experiments which may not representative all types of systems, 3) 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 to patterns at the class-level not at the system-level. Therefore, the system size is not crucial as it does not alter the results significantly.

Internal threats include: 1) our methodology to investigate the effects of refactoring to pattern techniques on external quality attributes was based on mapping the changes in the internal quality metrics to the external quality attributes. This mapping was done based on available research studies that show correlations between internal quality metrics and external quality attributes. Moreover, the relations between the internal quality metrics and the external quality attributes was based on these research studies without validation on our behalf of their findings regarding the correlations. 2) we performed the refactorings by ourselves. This may have introduced a bias on which refactoring technique to apply while performing the refactorings 3) we manually performed refactoring without using a tool; however, the code was recompiled after each refactoring 4) we do not validate if the refactored classes that show an improvement of an external quality attribute such as understandability are indeed more understandable. Such validation should be addressed in a future study.

VII. CONCLUSION AND FUTURE WORK

In this paper, we proposed a classification of refactoring to pattern techniques based on their measurable effect on (*i*) internal software quality metrics and (*ii*) external software quality attributes (adaptability, completeness, maintainability, understandability, reusability, and testability). The classification was done by mapping the changes in the internal quality metrics, caused by applying refactoring to pattern techniques, to the external quality attributes based on correlations between them shown in previous studies. Four open source software systems were used to perform this classification. The achieved classification helps software designers in choosing appropriate refactoring to pattern techniques that will improve the quality of their design based on their design objectives. It also enables them to predict the quality drift caused by using specific refactoring to patterns.

One direction of future work would be conducting additional empirical studies using other software systems written in different programming languages like C++ or C#.

Another possible direction of future work would be investigating the effect of refactoring to patterns on different sets of (*i*) internal software quality metrics such as metrics suites proposed by Briand et al. [36, 37] and MOOD metrics suite [38] and (*ii*) external software quality attributes such as performance, correctness, and portability. After studying these effects, these attributes can be used to further classify the refactoring to pattern techniques. It would also be useful to expand the proposed classification to include a more extended set of refactoring to pattern techniques based on software quality attributes to form a large classification catalog.

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

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*: Addison-Wesley, 1999.

[2] W. Opdyke, "Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks," PhD Thesis, Univ. of Illinois at Urbana-Champaign, 1992.

[3] J. Kerievsky, *Refactoring to Patterns*: Addison Wesley, 2004.

[4] T. Mens and T. Tourwe, "A Survey of Software Refactoring," *IEEE Transactions on Software Engineering,* vol. 30, pp. 126-139, 2004.

[5] IEEEStd.1061, *Std. 1061 for a Software Quality Metrics Methodology*. New York: Institute of Electrical and Electronics Engineers, 1992.

[6] ISO/IEC9126, *9126 Standard, Information technology - Software product evaluation - Quality characteristics and guidelines for their use*, 1991.

[7] R. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition ed.: McGraw Hill, 2005.

[8] N. Fenton and S. Pfleeger, *Software Metrics: A Rigorous & Practical Approach*, 2nd Edition ed.: PWS Publishing Company, 1997.

[9] S. Chidamber and C. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering,* vol. 20, pp. 476-493, June 1994 1994.

[10] V. Basili, L. Briand, and W. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering,* vol. 22, pp. 751-761, Oct. 1996 1996.

[11] W. Li and S. Henry, "Object-Oriented Metrics that Predict Maintainability," *Journal of Systems and Software,* vol. 23, pp. 111-122, 1993 1993.

[12] L. Briand, J. Wust, J. Daly, and V. Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," *The Journal of Systems and Software,* vol. 51, pp. 245-273, 2000 2000.

[13] M. Tang, M. Kao, and M. Chen, "An Empirical Study on Object Oriented Metrics," in *6th International Software Metrics Symposium*, 1999, pp. 242-249.

[14] F. Dandashi, "A Method for Assessing the Reusability of Object-Oriented Code Using a Validated Set of Automated Measurements," in *ACM Symposium on Applied Computing,* 2002, pp. 997-1003.

[15] M. Bruntink and A. vanDeursen, "An empirical study into class testability," *Journal of Systems and Software,* vol. 79, pp. 1219-1232, 2006.

[16] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction," *IEEE Transactions on Software Engineering,* vol. 31, pp. 897-910, Oct. 2005 2005.

[17] IEEEStd.610.12, *Std. 610.12 - IEEE Standard Glossary of Software Engineering Terminology.*: The Institute of Electrical and Electronics Engineers., 1991.

[18] W. Salamon and D. Wallace, "Quality Characteristics and Metrics for Reusable Software (preliminary Report)," US DoC for US DaD Ballistic Missile Defense Organization, NISTIR 5459,May 1994.

[19] K. Stroggylos and D. Spinellis, "Refactoring - Does it improve software quality?," in *5th International Workshop on Software Quality (WoSQ'07: ICSE Workshops)*, 2007, pp. 10-16.

[20] B. DuBois and T. Mens, "Describing the impact of refactoring on internal program quality," in *International Workshop on Evolution of Large-scale Industrial Software Applications*, 2003, pp. 37-48.

[21] D. Boshnakoska and A. Mišev, "Correlation between object-oriented metrics and refactoring," *ICT Innovations, Communication in Computer and Information Science (CCIS),* vol. 83, pp. 226-235, 2011.

[22] Y. Kosker, B. Turhan, and A. Bener, "An expert system for determining candidate software classes for refactoring," *Expert Systems with Applications,* vol. 36, pp. 10000-10003, 2009.

[23] J. A. Dallal and L. Briand, "A Precise method-method interaction-based cohesion metric for object-oriented classes," *ACM Transactions on Software Engineering and Methodology (TOSEM),* 2010.

[24] M. Cinnéide, D. Boyle, and I. Moghadam, "Automated refactoring for testability," presented at the Fourth International Conference on Software Testing, Berlin, 2011.

[25] B. DuBois, S. Demeyer, and J. Verelst, "Does the "Refactor to Understand" Reverse Engineering Pattern Improve Program Comprehension?," in *9th European Conference on Software Maintenance and Reengineering (CSMR'05),* 2005, pp. 334-343.

[26] B. Geppert, A. Mockus, and F. Robler, "Refactoring for Changeability: A way to go?," in *11th IEEE International Software Metrics Symposium (METRICS'05)*, 2005.

[27] D. Wilking, U. Khan, and S. Kowalewski, "An Empirical Evaluation of Refactoring," *e-Informatica Software Engineering Journal,* vol. 1, pp. 27-42, 2007.

[28] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, "A Quantitative Evaluation of Maintainability Enhancement by Refactoring," in *International Conference on Software Maintenance (ICSM'02),* 2002, pp. 576-585.

[29] B. DuBois, S. Demeyer, and J. Verelst, "Refactoring - Improving Coupling and Cohesion of Existing Code," in *11th Working Conference on Reverse Engineering (WCRE'04),* 2004, pp. 144-151.

[30] S. Bryton and F. Abreu, "Strengthening refactoring: towards software evolution with quantitative and experimental grounds," presented at the The 4th International Conference on Software Engineering Advances, Porto, 2009.

[31] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi, "Does refactoring improve reusability?," in *9th International Conference on Software Reuse (ICSR'06),* 2006, pp. 287-297.

[32] K. Elish and M. Alshayeb, "A Classification of Refactoring Methods Based on Software Quality Attributes," *The Arabian Journal for Science and Engineering,* vol. 36, 2011.

[33] M. Alshayeb, "Empirical Investigation of Refactoring Effect on Software Quality," *Information and Software Technology Journal,* vol. 51, pp. 1319-1326, 2009.

[34] M. Alshayeb, "The Impact of Refactoring to Patterns on Software Quality Attributes," *The Arabian Journal for Science and Engineering,* vol. 36, 2011.

[35] T. Lethbridge and R. Laganière, *Object-Oriented Software Engineering: Practical Software Development using UML and Java,* 2nd ed.: McGraw-Hill, 2005.

[36] L. Briand, J. Daly, and J. Wust, "A Unified Framework for Cohesion Measurement in Object-Oriented Systems," *Empirical Software Engineering,* vol. 3, pp. 65-117, 1998.

[37] L. Briand, J. Daly, and J. Wust, "A Unified Framework for Coupling Measurement in Object-Oriented Systems," *IEEE Transactions on Software Engineering,* vol. 25, pp. 91-121, Jan/Feb 1999 1999.

[38] R. Harrison, S. Counsell, and R. Nithi, "An Evaluation of the MOOD Set of Object-Oriented Software Metrics," *IEEE Transactions on Software Engineering,* vol. 24, pp. 491-496, June 1998 1998.

Karim O. Elish is a PhD candidate and research assistant in the Computer Science Department at Virginia Tech, Blacksburg, USA. He received his BS and MS in Computer Science from King Fahd University of Petroleum and Minerals (KFUPM), Dhahran, Saudi Arabia. He worked as a lecturer in the Information and Computer Science at KFUPM. His main research interests include software security, software refactoring, and software metrics and measurement.

Mohammad Alshayeb is an assistant professor in Software Engineering at the Information and Computer Science Department, King Fahd University of Petroleum and Minerals (KFUPM), Dhahran, Saudi Arabia. He received his MS and PhD in Computer Science and certificate of Software Engineering from the University of Alabama in Huntsville. He worked as a senior researcher and Software Engineer and managed software projects in the United States and Middle East. He is a certified project manager (PMP). His research interests include Software quality, software measurement and metrics, and empirical studies in software engineering.