Arab J Sci Eng (2011) 36:1241–1251
DOI 10.1007/s13369-011-0111-3

RESEARCH ARTICLE - COMPUTER ENGINEERING AND COMPUTER SCIENCE

Mohammad Alshayeb

The Impact of Refactoring to Patterns on Software Quality Attributes

Received: 25 June 2009 / Accepted: 23 June 2010 / Published online: 19 October 2011

Abstract The search for techniques to improve software quality and achieve robust, reliable, and maintainable software is ongoing. Refactoring, an approach that improves the internal structure of software without affecting its external behavior, is one method that aims to achieve better quality. Refactoring to patterns is another. The goal of this paper is to investigate whether refactoring to patterns improves software quality. This is done empirically by examining the metric values of external quality attributes for different software systems before and after refactoring to patterns is applied. We found no consistent improvement trends in the software quality attributes. This is because each refactoring to patterns technique has a particular purpose and effect, and hence affects software quality attributes differently.

Keywords Software refactoring · Software metrics · Refactoring to patterns · Quality improvement · Empirical study

الخلاصة

البحث عن أساليب لتحسين جودة البرمجيات وتحقيق برمجيات قوية موثوق بها قابلة للصيانة لا تزال مستمرة. إن إعادة الهيكلية، وطريقة تحسن البنية الداخلية للبرمجيات دون التأثير على سلوكه الخارجي، هي إحدى الطرق التي تهدف إلى تحقيق نوعية أفضل. إن إعادة الهيكلية لأنماط التصميم طريقة أخرى. والهدف من هذه الورقة هو التحقق إن كانت إعادة الهيكلية لأنماط التصميم تحسن جودة البرمجيات. ويتم ذلك من خلال دراسة تجريبية عن طريق قياس متريات البرمجيات لعناصر الجودة الخارجية لانظمة برمجيات مختلفة قبل وبعد تطبيق إعادة الهيكلية لأنماط التصميم. ولم نجد أي نزعة تحسن ثابت في عناصر جودة البرمجيات. وذلك لأن كل أسلوب لإعادة الهيكلية لأنماط التصميم له غرض وتاثير معين، وبالتالي له تأثير مختلف على عناصر جودة البرمجيات.

1 Introduction

Refactoring is the process of improving the design of existing code by changing its internal structure without affecting its external behavior [1–3]. Refactoring is concerned with restructuring the internal code (attributes and methods) across the existing classes and should be applied when adding a function, fixing a bug, or when doing a code review. It should also be done when developers encounter "bad smells" (symptoms or warning signs in the source code of the program that could indicate a potential problem). These "bad smells" are indicators that the code should be refactored. A catalog of refactorings has also been identified [3]. The

M. Alshayeb (✉)
Information and Computer Science Department,
King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia
E-mail: alshayeb@kfupm.edu.sa

main elements in the catalog include composing methods, moving features between objects, organizing data, simplifying conditional expressions, making method calls simpler, and dealing with generalization.

A design pattern is a description of a recurrent problem in design and a possible high-quality solution. Patterns describe the communicating objects and classes that are customized to solve a general design problem in a particular context. Design patterns, the building blocks of the system architecture, are abstract descriptions of object-oriented designs that appear repeatedly for a possible high-quality solution [4]. The use of design patterns can provide reusable solutions, establish common terminology between team members, and make software more modifiable. Researchers also claim that design patterns improve software quality by making object-oriented (OO) software more flexible, modular, and understandable [4].

There is a natural relation between patterns and refactorings. Patterns are where you want to be, whereas refactorings are ways to get there from somewhere else [5]. This idea agrees with the observation made by the four authors of the classic book, *Design Patterns: Elements of Reusable Object-Oriented Software*: "Our design patterns capture many of the structures that result from refactoring. . . . Design patterns thus provide targets for your refactorings." [4]

Kerievsky [5] introduced the concept of refactoring to patterns in his book *Refactoring to Patterns*. Refactoring to patterns explains the relation between refactoring and design patterns. According to Kerievsky [5], refactoring to patterns is the marriage of refactoring (the process of improving the design of existing code) with patterns (the classic solutions to recurring design problems). Woolf defined refactoring to patterns as "a revolutionary approach to applying patterns that combines the top-down utility of design patterns with the bottom-up discovery of iterative development and continuous refactoring" [5]. Refactoring to patterns allows designers to move their designs safely towards design patterns by applying sequences of low-level refactorings.

While the general objective of refactoring and refactoring to patterns is to improve the internal structure of the code without changing its behavior, the two approaches achieve this objective in different ways. In regular refactoring it is achieved by cleaning up the code in a disciplined way [3], while in refactoring to patterns it is done by introducing higher abstraction levels [5]. Because these two approaches are different, researchers have identified a different catalog for each of these approaches [3,5], and thus, there are refactoring methods that can be applied to produce patterns [5] and refactoring methods that are used to restructure the existing code without producing patterns [1–3]. It is true that refactoring techniques do not alter the external behavior; however, the resulting internal structure from these techniques is different, and hence, the quality impact also differs. This comes naturally because the focus of each technique is different; refactoring to patterns focuses on generating more abstractions, while regular refactoring concentrates on cleaning the code without focusing on the abstraction level.

Software quality is the degree to which software possesses a desired combination of attributes (e.g., adaptability, reusability) [6]. The ISO/EIC 9126 standard [7] 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 [8]: factors that can be directly measured, i.e., internal quality attributes (e.g., program lines of code) and factors that can be measured only indirectly, i.e., external quality attributes (e.g., understandability).

Significant development effort and cost has been invested in refactoring software [9]. Researchers argue that this effort and cost are worthwhile, because refactoring tends to improve software quality. Many researchers claim that refactoring improves quality by improving design and readability, and reducing bugs [3,10].

In his refactoring to patterns catalog [5], Kerievsky listed the benefits and liabilities of each refactoring to pattern, however, he did not provide any quantitative assessment of the effect of each refactoring to pattern on software quality. Therefore, the objective of this paper is to quantitatively assess, using software metrics, the effect of refactoring to patterns on different quality attributes such as Adaptability, Maintainability, Understandability, Reusability, and Testability to decide whether the cost and time invested in refactoring to patterns are worthwhile.

2 Related Work

Many software design patterns have been proposed. Coad [11,12] described seven simple patterns in object-oriented analysis and design. Schmidt wrote many patterns in communications systems and distributed applications [13]. Pree proposed patterns for framework development [14–16], while Buschmann and his colleagues classified software patterns into architectural patterns, design patterns, and idioms [17].

Design patterns and their advantages have also been investigated by Beck et al. [18]. They found that design patterns provide an effective "shorthand" for communicating complex concepts effectively between designers.

Design patterns can be used to record and encourage the reuse of "best practices" and capture the essential parts of a design in compact form [18].

Researchers have also studied applying refactoring to produce design patterns. Kerievsky in his book claimed that using patterns to improve existing code is better than using patterns early in a new design [5]. He also explained how to combine refactoring and patterns, how to improve the design of existing code with pattern-directed refactorings, and how to identify areas of code in need of pattern-directed refactoring [5]. Various researchers have investigated how design patterns should be applied to refactor object-oriented designs. Muraki and Saeki proposed a kind of objective criterion to determine how the suite of four design patterns should be applied to refactor object-oriented designs [18]. Focusing on specific software measures that characterize specific aspects of software design structures, they developed 20 measures for conditional statements and inheritance trees. These measures help determine which design patterns should be used and where they can be applied to improve poor designs [18]. Cinneide presented a methodology for the development of design pattern transformations. Key elements in their approach are the ability to apply the transformations to existing code and their emphasis on demonstrating behavior preservation for each transformation [19].

Many refactoring methodologies have been proposed. Usha et al. proposed a model for software development using refactoring. They conducted experiments on the refactoring process and proposed a refactoring methodology consisting of three phases, namely, the Identification phase, Proposal and Application phase, and Evaluation phase [20]. Tsantalis and Chatzigeorgiou [21] proposed a methodology to automatically identify Extract Method refactoring opportunities and present them as suggestions to the designer of an object-oriented system. They also proposed a methodology for the identification of Move Method refactoring opportunities, which constitute a way of solving many common Feature Envy bad smells [22].

Many researchers have evaluated the effect of applying refactoring on software quality [23–30]. Some of these efforts have assessed the effect on quality using internal software quality attributes. Kataoka et al. [28] investigated the refactoring effect on maintainability using coupling metrics. Bois and Mens [27] analyzed the impact of refactoring on internal program quality metrics as indicators of quality factors. Stroggylos and Spinellis [29] analyzed system logs of four popular open source software systems to detect refactorings and examined how software metrics are affected. Other researchers assessed the refactoring effect on external software quality attributes. Geppert et al. [23] empirically investigated the impact of refactoring on changeability, while Wilking et al. [24] investigated the effect thereof on maintainability and modifiability. Reddy and Rao [31] quantitatively evaluated the quality enhancement by refactoring using dependency oriented complexity metrics. They found that the metrics successfully indicate quantitatively the presence of defects and the improvement in the quality of designs after refactoring. Higo et al. proposed a method to estimate refactoring effect. The method was implemented as a software tool, and a case study was conducted. They found that the estimation of the tool helped developers of the target software system perform an appropriate refactoring [32]. In a previous study, we also evaluated the impact of regular refactoring on software quality. The results of the study show that refactoring in general does not necessarily improve software quality [26]. In the earlier study, we only considered regular refactoring methods proposed by Fowler et al. [3], whereas in this study, we have shifted our focus towards considering the effects of applying the refactoring to pattern methods proposed by Kerievsky [5] on software quality. We have moved our research in this direction because the production of higher abstraction levels associated with refactoring to patterns generates a different quality impact than regular refactoring.

Few research studies have been directed towards the effect of refactoring to patterns on software quality attributes [27,28,33]. Bois et al. [34] performed a controlled experiment to empirically investigate differences in program comprehension between the application of Refactor to Understand and the traditional Read to Understand pattern. Their study showed that refactoring can be used to improve the understanding of software. Masuda et al. described a case study on refactoring of existing software using design patterns. Using C&K metrics, they quantitatively evaluated the effectiveness of applying design patterns to software and showed that it improves flexibility and extensibility of a piece of software [33].

Many researchers have investigated the relation between internal quality attributes and external quality attributes. Bois et al. [27] 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 that are generally recognized as indicators for software maintainability. Moser et al. [30] proposed a methodology to assess if 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) considered to be relevant to reusability. However, all these studies are limited to a few external quality attributes.

Dandashi [35] demonstrated a method for assessing indirect quality attributes such as adaptability, maintainability, understandability, and reusability of object-oriented systems from the direct quality attributes. Bruntink and Deursen [36] identified and evaluated a set of object-oriented metrics that can be used to estimate the testing effort of classes of object-oriented software.

The following are the external quality attributes that are used in this study.

- *Adaptability* 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 [37].
- *Maintainability* the ease with which a component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment [37].
- *Understandability* the degree to which the meaning of a software component is clear to a user.
- *Reusability* 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.
- *Testability* a set of attributes of software that indicate the effort needed to validate the software product [7].

In this research, we try to consider more external quality attributes to obtain a more precise indication of whether refactoring to patterns does indeed improve software quality.

3 Data Description

The data used in this research comes from three different open source systems. All of the used systems are written in Java and therefore, can run on all platforms with support for Java (including Windows, Linux, and MAC). The details of these systems are given below.

Profile Profile is a Java application that manages the profiles of students. It allows users to specify the department, division, section, address, and various other information. It is free software; however, the author is unknown.

TerpPaint TerpPaint is a free, image manipulation program that uses AWT/Swing. TerpPaint contains all the functionality of Microsoft Paint with the addition of an advanced color chooser and color inversion. It provides support for multiple image formats (JPEG, GIF, PNG, and BMP). Having been developed in Java by Software Engineering students at the University of Maryland, it is published under the GNU Public License (http://sourceforge.net/projects/terppaint/).

NaparanSoft is an integrated package of software and hardware used in warehouse operations to monitor the quantity, location, and status of inventory as well as the related shipping, receiving, and picking processes. The application provides a simple storage mechanism for contacts, sales representatives, and suppliers. Stock adjustments, expenses, purchase orders, and other simple inventory operations can be done using the software. It is written in Java and uses Microsoft Access for the database (http://www.naparansoft.cjb.net).

Table 1 summarizes the numbers of classes, methods, and source lines of code in each version of the software.

4 Refactoring Practices

The three systems underwent a refactoring phase. The refactoring was performed manually by the designers within the development IDE and without using tool support to automate the refactoring process. Different refactoring to patterns techniques were applied when needed (i.e., when a bad smell was encountered). Therefore, not all techniques were applied to each system. The following are the refactoring to patterns techniques used.

Chain constructors It is used when you have constructors that contain duplicate code. Then, chain the constructors together to obtain the least duplicate code [5].

Table 1 Summary of numbers of classes, methods, and source lines of code in the three systems, before and after refactoring to patterns

	Before refactoring			After refactoring		
	Number of classes	Number of methods	Source lines of code	Number of classes	Number of methods	Source lines of code
System 1: Profile	10	30	318	9	61	303
System 2: TerpPaint	58	961	11,920	59	943	11,498
System 3: NaparanSoft	26	80	3,682	27	89	3,710

Table 2 Relationship between refactoring methods and the three systems

	System 1	System 2	System 3
Chain Constructors	X	X	
Compose Method	X	X	X
Replace Constructors with Creation Methods	X		
Unify Interfaces	X	X	
Move Accumulation to Collecting Parameter		X	
Extract Composite		X	
Replace Implicit Tree with Composite			X
Introduce Polymorphic Creation with Factory Method			X
Replace Multiple Instances with Singletons			X

Compose Method It is used when you cannot rapidly understand a method's logic. Then, transform the logic into a small number of intention-revealing steps at the same level of detail [5].

Replace Constructors with Creation Methods It is used when the constructors on a class make it hard to decide which constructor to call during development. Then, replace the constructors with intention-revealing Creation Methods that return object instances [5].

Unify Interfaces It is used when you need a superclass and/or interface to have the same interface as a subclass. Then, find all public methods on the subclass that are missing on the superclass/interface. Add copies of these missing methods to the superclass, altering each one to perform null behavior [5].

Move Accumulation to Collecting Parameter It is used when you have a single bulky method that accumulates information to a local variable. Then, accumulate results to a Collecting Parameter that gets passed to extracted methods [5].

Extract Composite It is used when Subclasses in a hierarchy implement the same Composite. Then, extract a superclass that implements the Composite [5].

Replace Implicit Tree with Composite It is used when you implicitly form a tree structure, using a primitive representation, such as a String. Then, replace your primitive representation with a Composite [5].

Introduce Polymorphic Creation with Factory Method It is used when classes in a hierarchy implement a method similarly except for an object creation step. Then, make a single superclass version of the method that calls a Factory Method to handle the instantiation [5].

Replace Multiple Instances with Singleton This is used when the code creates multiple instances of an object that consumes too much memory or takes too long to instantiate [5].

Table 2 shows the relationship between the refactoring method applied and the system it was applied to.

5 Analysis and Results

Our aim in this study was to investigate whether refactoring to patterns improves software quality. To focus our study, we set up the following hypotheses, where, for each hypothesis, H0 represents the null hypothesis and H1 represents the alternative hypothesis.

Hypothesis 1

H_0: refactoring to patterns does not improve software adaptability
H_1: refactoring to patterns improves software adaptability

Hypothesis 2

H_0: refactoring to patterns does not improve software maintainability
H_1: refactoring to patterns improves software maintainability

Hypothesis 3

H_0: refactoring to patterns does not improve software understandability
H_1: refactoring to patterns improves software understandability

Hypothesis 4

H_0: refactoring to patterns does not improve software reusability
H_1: refactoring to patterns improves software reusability

Table 3 Summary of the relationship between the internal and external quality attributes

External quality	Study	Internal quality								
		Inheritance		Coupling			Size			Cohesion
		DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Adaptability	Dandashi [35]	−ve	−ve	−ve	−ve	0	+ve	0	+ve	0
Maintainability	Dandashi [35]	−ve	−ve	−ve	−ve	0	+ve	0	+ve	0
Understandability	Dandashi [35]	−ve	−ve	−ve	−ve	0	+ve	0	+ve	0
Reusability	Dandashi [35]	−ve	−ve	−ve	−ve	0	+ve	0	+ve	0
Testability	Bruntink and Deursen [36]	0	0	0	+ve	+ve	+ve	+ve	+ve	0

Hypothesis 5

H_0: refactoring to patterns does not improve software testability
H_1: refactoring to patterns improves software testability

We used previously published research that correlates software metrics with the external quality attributes under investigation. The work of Dandashi [35] was used to assess adaptability, maintainability, understandability, and reusability, while that of Bruntink and Deursen [36] was used to assess testability. These two studies were chosen because they had significant correlation levels.

- *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, and 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 LOC in a class excluding all blank and comment lines.

Table 3 summarizes the relationships between the internal quality attributes (DIT, NOC, CBO, RFC, FOUT, WMC, NOM, LOC, LCOM) and the external quality attributes (adaptability, maintainability, understandability, reusability, and testability), where "+ve" represents positive correlation, "−ve" represents negative correlation, and "0" represents no correlation at all.

We collected the metric values for the two versions (before and after refactoring to patterns) of the three software systems using the Metamata metrics tool [38]. We considered the classes that are common in the two versions because we measured the changes in the metric values at class level and observed their evolution with respect to the software metrics identified in Table 3 as indicators of the external qualities considered.

Tables 5 and 6 show the classes and changes in metric values with respect to the original version of each of the three systems. A dashed line indicates no change in the metric value; an upward arrow indicates an increase after refactoring, while a downward arrow indicates a decrease after refactoring.

Hypothesis testing was done by combining the metric value changes in Tables 4, 5, and 6 and the metric to external attribute correlations in Table 3, and indicating whether each of the five external attributes is improved by refactoring to patterns. To do this, we first examined, for each class in each system, the effect on the class' five attributes. Then we studied overall trends in changes in the five attributes among the classes in each system. An overall judgment whether each of the five qualities had been improved by the refactoring to patterns could then be deduced.

We investigated the effect of the metric changes on the five designated qualities for each class in the three systems, with the results given Tables 7, 8, and 9, respectively. For classes where metric changes had a pure increase effect on a quality, we indicated the end effect on the quality as an increase (upward arrow).

Table 4 Differences in metric values for System 1 before and after refactoring to patterns

	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Construct	–	–	–	–	–	–	–	–	–
Address.java	–	–	–	▲	–	▼	▲	▲	▲
Department.java	–	–	–	▼	–	▼	▼	▼	▼
Division.java	–	–	–	▼	–	▼	▼	▼	▼
Graduate.java	▼	–	–	▲	–	▲	▲	▲	▲
Instructor.java	–	–	▲	▲	▲	▲	▲	▲	▼
Person.java	–	–	▼	▼	▼	▼	▼	▼	▲
Profile.java	▲	–	▲	▲	▲	▲	▲	▲	▲
Section.java	–	–	–	–	–	–	–	▼	▲
UserAccount.java	–	–	–	–	–	–	–	–	–

Table 5 Differences in metric values for System 2 before and after refactoring to patterns

	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Construct	–	–	–	–	–	–	–	▼	–
Fill.java	–	–	–	–	–	–	–	–	–
GifDecoder.java	–	–	–	▲	–	▲	▲	▲	–
NeuQuant.java	–	–	–	▲	▲	▲	▲	▲	▼
TerpPaintFileFilter.java	–	–	–	–	–	–	–	▲	▲
Text.java	▲	▼	▼	▼	▼	▼	▼	▼	▼
blur.java	▲	▼	▼	▼	▼	▼	▼	▼	▼
brightness.java	–	–	–	▲	–	▼	▲	▼	–
brushTool.java	–	–	–	▲	–	▲	▲	▲	▲
ellipseTool.java	–	–	–	–	–	–	–	▼	–
eraserTool.java	–	–	–	–	–	–	–	▼	–
magicSelectTool.java	–	–	–	▲	–	▲	▲	▲	–
polygonTool.java	–	–	▼	▼	▼	▼	▼	▼	▼
rectTool.java	▲	–	–	▼	▼	▼	▼	▼	▲
roundedRectTool.java	–	–	–	–	–	–	–	–	–
selectTool.java	▲	–	–	▼	▼	▼	▼	▼	▲
selectallTool.java	–	▼	–	–	–	–	–	▼	–
splash.java	–	–	–	–	–	–	–	▼	–

Table 6 Differences in metric values for System 3 before and after refactoring to patterns

	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Construct	▲	▼	▼	▼	▼	▼	▼	▼	–
FrmCustomer.java	–	–	–	–	–	–	–	–	–
FrmInvoice.java	–	–	–	–	–	–	–	–	–
FrmProduct.java	▲	▼	▼	▼	▼	▼	▼	▼	–
FrmSalesRep.java	–	–	–	–	–	–	–	–	–
FrmSearchCustomer.java	–	–	–	–	–	–	–	–	–
FrmSearchProduct.java	–	–	–	–	–	–	–	–	–
FrmSearchSalesRep.java	–	–	–	–	–	–	–	–	–
FrmSearchWarehouse.java	–	–	–	–	–	–	–	–	–
FrmSplash.java	▲	▼	▼	▼	▼	▼	▼	▼	–
FrmSupplier.java	–	–	–	–	–	–	–	–	–
FrmWarehouse.java	–	–	–	▲	–	–	–	▼	–
MainForm.java	–	–	–	–	–	–	–	–	–
XPStyleTheme.java	–	–	–	–	–	–	–	–	–
clsPublicMethods.java	–	–	–	–	–	–	–	–	–
clsPublicVariables.java	–	–	–	▲	–	–	–	▲	–
frm_add_edit_customer.java	–	–	–	–	–	–	–	–	–
frm_add_edit_product.java	–	–	–	–	–	–	–	–	–
frm_add_edit_salesrep.java	–	–	–	–	–	–	–	–	–
frm_add_edit_supplier.java	–	–	–	–	–	–	–	–	–
frm_add_edit_warehouse.java	–	–	–	–	–	–	–	–	–

For classes where metric changes had a pure decrease effect on a quality, we indicated the end effect on the quality as a decrease (downward arrow). Some classes had metric values that did not change at all after refactoring to patterns. In these classes, no changes in the metric values meant no effect on the correlated external qualities. We indicated these unaffected qualities with "none".

Table 7 Impact of refactoring on external qualities for System 1

	Adaptability	Maintainability	Understandability	Reusability	Testability
Construct	None	None	None	None	None
Address.java	Unknown	Unknown	Unknown	Unknown	Unknown
Department.java	Unknown	Unknown	Unknown	Unknown	▼
Division.java	Unknown	Unknown	Unknown	Unknown	▼
Graduate.java	Unknown	Unknown	Unknown	Unknown	▲
Instructor.java	Unknown	Unknown	Unknown	Unknown	▲
Person.java	Unknown	Unknown	Unknown	Unknown	▼
Profile.java	Unknown	Unknown	Unknown	Unknown	▲
Section.java	▼	▼	▼	▼	▼
UserAccount.java	None	None	None	None	None

Table 8 Impact of refactoring on external qualities for System 2

	Adaptability	Maintainability	Understandability	Reusability	Testability
Construct	▼	▼	▼	▼	▼
Fill.java	None	None	None	None	None
GifDecoder.java	Unknown	Unknown	Unknown	Unknown	▲
NeuQuant.java	Unknown	Unknown	Unknown	Unknown	▲
TerpPaintFileFilter.java	▲	▲	▲	▲	▲
Text.java	Unknown	Unknown	Unknown	Unknown	▼
blur.java	Unknown	Unknown	Unknown	Unknown	▼
brightness.java	▼	▼	▼	▼	Unknown
brushTool.java	Unknown	Unknown	Unknown	Unknown	▲
ellipseTool.java	▼	▼	▼	▼	▼
eraserTool.java	▼	▼	▼	▼	▼
magicSelectTool.java	Unknown	Unknown	Unknown	Unknown	▲
polygonTool.java	Unknown	Unknown	Unknown	Unknown	▼
rectTool.java	Unknown	Unknown	Unknown	Unknown	▼
roundedRectTool.java	None	None	None	None	None
selectTool.java	Unknown	Unknown	Unknown	Unknown	▼
selectallTool.java	Unknown	Unknown	Unknown	Unknown	▼
splash.java	▼	▼	▼	▼	▼

Table 9 Impact of refactoring on external qualities for System 3

	Adaptability	Maintainability	Understandability	Reusability	Testability
Construct	Unknown	Unknown	Unknown	Unknown	▼
FrmCustomer.java	None	None	None	None	None
FrmInvoice.java	None	None	None	None	None
FrmProduct.java	Unknown	Unknown	Unknown	Unknown	▼
FrmSalesRep.java	None	None	None	None	None
FrmSearchCustomer.java	None	None	None	None	None
FrmSearchProduct.java	None	None	None	None	None
FrmSearchSalesRep.java	None	None	None	None	None
FrmSearchSupplier.java	None	None	None	None	None
FrmSearchWarehouse.java	None	None	None	None	None
FrmSplash.java	Unknown	Unknown	Unknown	Unknown	▼
FrmSupplier.java	None	None	None	None	None
FrmWarehouse.java	▼	▼	▼	▼	Unknown
MainForm.java	None	None	None	None	None
XPStyleTheme.java	None	None	None	None	None
clsPublicMethods.java	None	None	None	None	None
clsPublicVariables.java	Unknown	Unknown	Unknown	Unknown	▲
frm_add_edit_customer.java	None	None	None	None	None
frm_add_edit_product.java	None	None	None	None	None
frm_add_edit_salesrep.java	None	None	None	None	None
frm_add_edit_supplier.java	None	None	None	None	None
frm_add_edit_warehouse.java	None	None	None	None	None

In some cases, some of the correlated metric changes affected the quality increasingly while other metric changes did so decreasingly making it difficult to specify the collective overall effect. It is of course possible when having such mixed effects that a decreasing effect may be less than the opposing increasing effect/s

Table 10 Percentage of classes with external qualities affected by refactoring to patterns in System 1

	No effect (%)	Unknown (%)	Increased (%)	Decreased (%)
Adaptability	20	70	0	10
Maintainability	20	70	0	10
Understandability	20	70	0	10
Reusability	20	70	0	10
Testability	20	10	30	40

Table 11 Percentage of classes with external qualities affected by refactoring to patterns in System 2

	No effect (%)	Unknown (%)	Increased (%)	Decreased (%)
Adaptability	11	56	6	28
Maintainability	11	56	6	28
Understandability	11	56	6	28
Reusability	11	56	6	28
Testability	11	6	28	56

Table 12 Percentage of classes with external qualities affected by refactoring to patterns in System 3

	No effect (%)	Unknown (%)	Increased (%)	Decreased (%)
Adaptability	77	18	0	5
Maintainability	77	18	0	5
Understandability	77	18	0	5
Reusability	77	18	0	5
Testability	77	5	5	14

resulting in an overall collective increasing effect. It is also possible that an increasing effect may be less than the opposing decreasing effect/s resulting in an overall collective decreasing effect. However, the studies defining these correlations did not specify a cut value or threshold that would enable us to classify a single effect as significant or slight compared to other effects. For example, in a class like selectallTool.java how much less does the NOC have to be to overcome a 20% decrease in the LOC? It is not possible to calculate such an effect because the studies only provided correlations that are all acceptably significant, but without specifying quantitatively exactly how significant each correlation is. Because of this inability to rule out one effect with another stronger effect, we decided to only claim an improvement or deterioration in an external quality if all sub-effects in the internal attributes together lead to a clear collective increase or decrease. We also decided to indicate cases where the possibility of effects overcoming each other exists with an "unknown" collective effect.

To facilitate determining whether each quality was improved, we counted the classes in which a positive, negative, unknown, or no effect on the external qualities occurred. We then calculated the percentage these classes comprised of the total number of classes in the systems. These figures are summarized in Tables 10, 11, and 12.

From Table 3, we observe that adaptability, maintainability, understandability, and reusability have the same metric correlations. This means that when testing hypotheses 1–4 the refactoring to patterns effects on these external qualities should be similar.

From Tables 10, 11, and 12 we notice that adaptability, maintainability, understandability, and reusability were positively affected in none of the classes in Systems 1 and 3, and only 6% of the classes in System 2. This 6% includes only those classes where we are sure that the qualities were improved; this is because in classes where the refactoring to patterns effect is unknown, we cannot be sure if the quality is improved or not. On the other hand, these same qualities were negatively affected in 10% of the classes in System 1, 28% of the classes in System 2, and 5% of the classes in System 3. With such inconsistent percentages, we surely cannot conclude that refactoring to patterns in these software systems improves adaptability, maintainability, understandability, or reusability. Hypotheses 1, 2, 3, and 4 are thus rejected.

For hypothesis 5, we observe that testability was positively affected in 30% of the classes in System 1, 28% of the classes in System 2, and 5% of the classes in System 3. On the other hand, testability was negatively affected in 40% of the classes in System 1, 56% of the classes in System 2, and 14% of the classes in System 3. With such varying percentages, it is again not possible to conclude that refactoring in these software systems improves testability. Hypothesis 5 is thus also rejected.

6 Threats to Validity

There are possible threats to the validity of the results of this study. During the investigation of the effect of refactoring methods on external quality attributes, we relied on previous studies [36,35] that correlated internal quality metrics and external quality attributes without validating their results. Thus our findings are subject to the significance of the previous correlations.

Another possible reservation is the use of small scale systems, which would be crucial, were we investigating system-level changes. This study, however, is more focused on class-level effects, in which case system size does not alter the results significantly.

7 Conclusion and Future Work

In this research study, we tried to find consistent trends in improvements to external quality attributes as a result of applying refactoring to patterns to three systems. No such consistent trends were found. Furthermore, the effects of refactoring to patterns on the external qualities showed discrepancies and were vague and indefinite in many cases. In systems where refactoring to patterns improved a quality in a certain percentage of classes, it reversely abated that same quality in a similar or even higher percentage of classes. Accordingly, we are unable to confirm the generalization that refactoring to patterns improves software quality.

The results and conclusion above are comparable with the outcomes of a research study we conducted to investigate the effects of refactoring on the whole, and not specifically to patterns, on the selected set of quality attributes. In the latter study, we could not prove that refactoring in general improves software quality.

Based on this research effort and the previous study, we can see that studying the effect of refactoring and refactoring to patterns as a whole on software quality leads to unclear trends in quality improvement. Achieving lucid trends may be possible in future research by classifying refactoring/refactoring to patterns methods based on their assessable effects on quality attributes. Such a classification would help software designers in choosing appropriate refactoring/refactoring to patterns methods to improve certain software quality attributes.

Future research includes an attempt to validate the improvement in quality by testing more systems and different datasets in similar and different contexts. This may further confirm or invalidate the results of this empirical study.

Acknowledgments The author acknowledges the support of King Fahd University of Petroleum and Minerals. This work was carried out under grant JF-2005/07. Special thanks to Mr. Karim Elish. The author also acknowledges the valuable comments on earlier versions of this article made by the anonymous reviewers.

References

1. Opdyke, W.: Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks. PhD Thesis, Univ. of Illinois at Urbana-Champaign (1992)
2. Wake, W.: Refactoring Workbook. Addison-Wesley, Reading (2003)
3. Fowler, M.; Beck, K.; Brant, J.; Opdyke, W.; Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (1999)
4. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Reading (1994)
5. Kerievsky, J.: Refactoring to patterns. In: Addison-Wesley Signature Series. Addison-Wesley, Reading (2004)
6. Standard 1061-1992 for a Software Quality Metrics Methodology (1992)
7. ISO/IEC9126, 9126 Standard, Information technology—Software product evaluation—Quality characteristics and Guidelines for Their Use (1991)
8. Pressman, R.: Software Engineering: A Practitioner's Approach, 6th edn. McGraw-Hill, NY (2005)
9. Alshayeb, M.; Li, W.; Graves, S.: An Empirical Study of Refactoring, New Design, and Error-fix efforts in extreme programming. In: Proceedings of the 5th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2001), Orlando, FL (2001)
10. Mens, T.; Tourwe, T.: A Survey of Software Refactoring. IEEE Trans. Softw. Eng. **30**, 126–139 (2004)
11. Coad, P.: Object-oriented patterns. Commun. ACM **35**, 152–159 (1992)
12. Coad, P.; North, D.; Mayfield, M.: Object Models: Patterns, Strategies, and Applications. Prentice-Hall, Englewood Cliffs (1995)
13. Schmid, H.: Creating applications from components: a manufacturing framework design. IEEE Softw. **13**, 67–75 (1996)
14. Pree, W.: Meta patterns: a means for capturing the essentials of reusable object-oriented design. In: Proceedings of the European Conference of Object Oriented Programming, ECOOP'94, Bologna, Italy, pp. 150–162 July, 1994
15. Pree, W.: Design Patterns for Object-Oriented Software Development. Addison-Wesley, Reading (1995)

16. Pree, W.: Framework Patterns: SIGS Books & Multimedia (1996)
17. Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M.: Pattern-Oriented Software Architecture: A Pattern System. Addison-Wesley, Reading (1996)
18. Muraki, T.; Saeki, M.: Metrics for applying GOF design patterns in refactoring processes. In: Proceedings of the 4th International Workshop on Principles of Software Evolution, Sept. 2001
19. Cinneide, M.: Automated refactoring to introduce design patterns. In: Proceedings of the International Conference on Software Engineering, June 2000
20. Usha, K.; Poonguzhali, N.; Kavitha, E.: A quantitative approach for evaluating the effectiveness of refactoring in software development process. In: International Conference on Methods and Models in Computer Science, pp. 1–7 (2009)
21. Tsantalis, N.; Chatzigeorgiou, A.: Identification of extract method refactoring opportunities. In: European Conference on Software Maintenance and Reengineering, pp. 119–128 (2009)
22. Tsantalis, N.; Chatzigeorgiou, A.: Identification of Move Method Refactoring Opportunities. IEEE Trans. Softw. Eng. **36**, 347–367 (2009)
23. Geppert, B.; Mockus, A.; Robler, F.: Refactoring for changeability: a way to go. In: Proceedings of 11th IEEE International Software Metrics Symposium (METRICS'05) (2005)
24. Wilking, D.; Khan, U.; Kowalewski, S.: An empirical evaluation of refactoring. e. Inf. Softw. Eng. J. **1**, 27–42 (2007)
25. Bois, B.; Demeyer, S.; Verelst, J.: Does the "Refactor to Understand" Reverse Engineering Pattern Improve Program Comprehension? In: Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR'05), pp. 334–343 (2005)
26. Alshayeb, M.: Empirical Investigation of Refactoring Effect on Software Quality. Inf. Softw. Technol. J. **51**, 1319–1326 (2009)
27. Bois, B.D.; Mens, T.: Describing the impact of refactoring on internal program quality. In: International Workshop on Evolution of Large-Scale Industrial Software Applications (ELISA), Sept. 2003
28. Kataoka, Y.; Imai, T.; Andou, H.; Fukaya, T.: A quantitative evaluation of maintainability enhancement by refactoring. In: Proceedings of the IEEE International Conference on Software Maintenance (2002)
29. Stroggylos, K.; Spinellis, D.: Refactoring—does it improve software quality? In: Proceedings of the 5th International Workshop on Software Quality (WoSQ'07: ICSE Workshops), pp. 10–16 (2007)
30. Moser, R.; Sillitti, A.; Abrahamsson, P.; Succi, G.: Does refactoring improve reusability? In: 9th International Conference on Software Reuse (ICSR'06), pp. 287–297 (2006)
31. Reddy, K.N.; Rao, A.A.: A quantitative evaluation of software quality enhancement by refactoring using dependency oriented complexity metrics. In: Proceedings of the Second International Conference on Emerging Trends in Engineering and Technology, pp. 1011–1018 (2009)
32. Higo, Y.; Matsumoto, Y.; Kusumoto, S.; Inoue, K.: Refactoring effect estimation based on complexity metrics. In: 19th Australian Conference on Software Engineering, pp. 219–228 (2008)
33. Masuda, G.; Sakamoto, N.; Ushijima, K.: Redesigning of an Existing Software using Design Patterns. In: Proceedings of the International Symposium on Principles of Software Evolution, pp. 165–169, Nov. 2000
34. Bois, B.; Demeyer, S.; Verelst, J.: Refactoring—improving coupling and cohesion of existing code. In: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04), pp. 144–151 (2004)
35. Dandashi, F.: A Method for Assessing the Reusability of Object-Oriented Code Using a Validated Set of Automated Measurements. In: Proceedings of the ACM Symposium on Applied Computing, pp. 997–1003 (2002)
36. Bruntink, M.; Deursen, A.: An empirical study into class testability. J. Syst. Softw. **79**, 1219–1232 (2006)
37. IEEE, Std. 610.12—IEEE Standard Glossary of Software Engineering Terminology: The Institute of Electrical and Electronics Engineers (1991)
38. Metamata. Metamata Metrics Tool. http://www.metamata.com