The Impact of Refactoring on Class and Architecture Stability

Mohammad Alshayeb

Information and Computer Science Department
King Fahd University of Petroleum & Minerals
Dhahran 31261, Saudi Arabia
alshayeb@kfupm.edu.sa

Refactoring is used to improve the internal structure of the code without affecting its external behaviour. This is done by restructuring the components of the software, i.e. changing the internal structure within classes or changing the structure between classes. However, this may have an impact on class and architecture stability. In this paper we assess the impact of refactoring on class and architecture stability and then propose a classification for refactoring methods based on the impact of refactoring on class and architecture stability.

Keywords: Class stability; architecture stability; refactoring; refactoring classification
ACM Classification: D.2

1. INTRODUCTION

Refactoring, the process of improving the design of existing code by changing its internal structure without affecting its external behaviour (Fowler *et al*, 1999; Wake, 2003), is used to improve software quality by improving its design, readability, and reducing bugs (Fowler *et al*, 1999). It is concerned with restructuring the internal code (attributes and methods) across the existing classes and between classes. Refactoring is often employed as a part of software development. Refactoring is considered a core part of the software development cycle in some software processes like Extreme Programming (XP) (Beck, 1999).

Changes are necessary to continue increasing the value of the software. Software continues to change in its life cycle due to evolution in the requirements. This evolution causes the software to be unstable. Software that has low stability may have high-amplified changes through its design (Alshayeb and Li, 2004). Consequently, the maintenance cost and effort will be higher. Therefore, there is a vital need for stable software.

Undesirable instability can exist at the architecture level; the architecture represents the higher level of a software design. Undesirable instability can also exist at lower level; which is class level. Classes in Object-Oriented (OO) systems form the basic element of software architecture.

Class stability measures the ease with which a software item can evolve while preserving its design (Grosser *et al*, 2002) while architecture stability measures the extent to which software is flexible to endure requirement and environment changes while preserving the architecture (Bahsoon, 2003). Well-designed OO software systems should be able to evolve without major changes in their architecture (Grosser *et al*, 2003). Implementing architectural changes is very expensive; therefore, software must be designed with greater architectural stability (Jazayeri, 2002).

Manuscript received: 30 April 2010
Communicating Editor: John Hosking

Classes in OO systems form the basic component of the software architecture; hence, designers should also strive to keep classes as stable as possible. Many measures for class stability (Li *et al*, 2000; Grosser *et al*, 2002; Grosser *et al*, 2003; Alshayeb *et al*, 2011) and architecture stability (Mattsson and Bosh, 1999; Bansiya, 2000; Mattsson and Bosch, 2000; Jazayeri, 2002; Ahmed *et al*, 2003; Olague *et al*, 2006; Hassan, 2007) have been defined.

There is a significant development effort and cost put into refactoring software (Alshayeb *et al*, 2001). Researchers argue that effort spent on refactoring is worthwhile since it in general improves software quality. However, each refactoring method has a different effect and impact on software quality (Alshayeb, 2009). Software designers usually design for specific design goals, which in turn may contradict each other. Therefore, in order to optimize the software design for maintainability, designers should strive to build stable software. However, there are no guidelines for software designers on which refactoring methods to use in order to maintain stable software. Therefore, we need to study the effect of refactoring methods based on their effect on software stability and provide a classification of these refactoring methods based on their measurable effect on software stability. This classification aims to help software designers in choosing appropriate refactoring methods to maintain stable software. It also enables them to predict the quality drift caused by using specific refactoring methods.

The objectives of this paper are to assess the impact of refactoring on class and architecture stability; and to propose a classification for refactoring methods based on their impact on class and architecture stability. This will give the software engineers insight into which refactoring methods to use in order to maintain stable software.

2. LITERATURE REVIEW

This section introduces the different definitions and measures of class and architecture stability and software refactoring.

2.1 Class Stability

Many researchers proposed measures to measure class stability. Li *et al* (2000) presented an empirical study of OO software evolution and examined how the implementation of a class can affect its design. They proposed the Class Implementation Instability (CII). The CII metric measures the evolutionary changes in the implementation of a class by measuring the percentage of changes of LOC in design N that is affected by changes in design N+1 (in terms of lines of code added, deleted and changed). Grosser *et al* (2003) proposed a case-based reasoning approach for predicting the stability of Java classes from relevant metric data. The approach uses the stress factor to measure class stability. The stress, the result of a major change in the requirements, is defined as the degree of increase in the responsibilities of the class itself or the classes to which it is related. The key assumption behind the stability measurement is that the class is stable whenever its interface remains unchanged between versions. The stability of a class interface depends on the design (structure) of the class and the stress produced by applying the new requirements between two versions. Rat *et al* (2004) proposed a measure for class stability that considers a class is stable between version *i-1* and version *i* if there was no change in the number of methods of a class.

Alshayeb *et al (*2011) proposed a new metrics to measure class stability. They identified eight class properties that affect class stability. These properties are: inherited class name, class interface name, class access-level, class variable, class variable access-level, method signature, method access-level and method body (code).

Since Alshayeb *et al (*2011) used the stability definition of Grosser *et al* (2003) and the aim was

to measure stability for the *base* version with respect to the new version, the measurement scope was focused on the *base* version. Therefore, they considered the stable or unchanged class properties and ignored other class changes between the base version and the other versions. A property is considered unchanged if it has not been changed between versions *n* and version *n*+1.

To calculate the stability of an OO class they assumed that all class properties have the same weight and each property is handled separately. Therefore, they calculated the stability of each property then summed all results to produce OO class stability. The value of 1 means the class has full stability while the value of 0 means the class has full instability. This metric will be used in this study to measure class stability.

2.2 Architecture Stability

Researchers have also measured architecture stability. Jazayeri (2002) evaluated the architecture stability by examining the amount of change applied in successive releases of telecommunication software and showed how retrospective analysis may be performed. Jazayeri applied three types of retrospective analysis to twenty releases of a large telecommunication software system. In the first, he compared simple measures such as module size, number of modules changed and number of modules added in the different releases. In the second, he measured coupling among modules to discover if there are certain modules that always change during the same release. In the third, he used colour visualization to outline the system evolution.

Bansiya (2000) introduced a methodology to evaluate framework architecture structural characteristics and stability. Bansiya applied a suite of OO metrics for system and class level on successive versions of frameworks. After data collection, metric values were computed and compared to determine the *extent-of-change* in structural characteristics between releases. To compute the *extent-of-change*, metric values of the framework are normalized, i.e. dividing the metric value of a version by the metric value of the previous version. The normalized metrics values were summed to produce an '*aggregate-change*'. The *extent-of-change* value of any version is found by calculating the deference of the *aggregate-change(Vi)* of a version (*i*, with *i*>0) with the *aggregate-change(V*1*)* value of the first version. This approach will be used in this study to measure architecture stability.

Mattsson and Bosch (1999) assessed the stability of frameworks. An experiment was carried out on one telecommunication framework and two commercial GUI frameworks. They adopted Bansiya's approach (Bansiya, 2000) to find the stability of framework by calculating *extent-of-change* from different metrics data for each framework.

Ahmed *et al* (2003) introduced an approach to measure architectural stability of an OO system by using similarity metrics. These metrics compare the base version of a system with the next versions. Then a regression line is generated from these similarity values. A higher value represents a stable architecture.

Tonu *et al* (2006) presented a metric-based approach to evaluate architecture stability. The approach proposed applying both retrospective and predictive analysis. Tonu *et al,* by using the metric-based approach, were able to identify where the architecture of the systems used in the experiment become stable.

2.3 Refactoring

The term "refactoring" was first introduced by Opdyke in his PhD dissertation (Opdyke, 1992). Refactoring refers to "the process of changing an [object-oriented] software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure" (Fowler

et al, 1999). In other words, refactoring improves the internal structure of software without adding new functionality.

Fowler *et al* (1999) identified a number of benefits for refactoring. These include: improving the design of software making software easier to understand, helping to find bugs and helping to program faster.

Fowler *et al* (1999), in their refactoring catalogue, defined more than 70 different kinds of code refactorings over six categories. Each one of them includes the motivation of why the refactoring should be performed and step-by-step description of how to carry out the refactoring. After that many other refactorings have been discovered and used (Fowler, 2010).

2.4 Refactoring Effect Assessment using Software Quality Attributes

Many researchers assessed the refactoring effect on software using the internal quality attributes used (DuBois and Mens, 2003; Tahvildari and Kontogiannis, 2003; Stroggylos and Spinellis, 2007). Others used the external quality attributes. DuBois *et al* (2005) 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. Geppert *et al* (2005) empirically investigated the impact of refactoring of a legacy system on changeability. Wilking *et al* (2007) investigated the effect of refactoring on maintainability and modifiability through an empirical evaluation.

Researchers have also mapped the refactoring effect on internal software quality to external software quality. Kataoka *et al* (2002) proposed coupling metrics as a quantitative evaluation

Study	Internal Quality	External Quality	Refactorings Classification?
DuBois *et al*, 2005 (DuBois *et al*, 2005)	–	Understandability	No
Geppert *et al*, 2005 (Geppert *et al*, 2005)	–	Changeability	No
Wilking *et al*, 2007 (Wilking *et al*, 2007)	–	Maintainability Modifiability	No
Kataoka *et al*, 2002 (Kataoka *et al*, 2002)	Coupling	Maintainability	No
DuBois *et al*, 2004 (DuBois *et al*, 2004)	Coupling, Cohesion	Maintainability	Yes, based on internal quality.
Moser *et al*, 2006 (Moser *et al*, 2006)	C&K metrics, MCC, LOC	Reusability	No
Elish and Alshayeb (Elish and Alshayeb, 2010)	C & K metrics, FOUT, NOM, LOC	Adaptability Completeness Maintainability Understandability Reusability Testability	Yes
This Work	–	Class & Architectural Stability	Yes

Table 1: Summary of the related work

method to measure the effect of refactoring on the maintainability of the program by comparing the coupling before and after the refactoring. Du Bois *et al* (2004) developed practical guidelines for applying refactoring methods to improve coupling and cohesion characteristics and validated these guidelines on an open source software system. Moser *et al* (2006) proposed a methodology to assess if refactoring improves reusability and promotes ad-hoc reuse in an XP-like development environment. This work aims to assess the impact of refactoring on class and architecture stability; and to propose a classification for refactoring methods based on their impact on class and architecture stability. Recently, a classification of refactoring methods based on software quality attributes such as adaptability, completeness, maintainability, understandability, reusability, and testability has been proposed (Elish and Alshayeb, 2010). Table 1 shows a summary of the related work.

3. REFACTORING METHODS UNDER STUDY FOR CLASS STABILITY

This section explains the refactoring methods that are used in this research; an example is shown in Figure 1. The methods presented here are adopted from Fowler (2010) and Fowler *et al* (1999) which cover the different levels: field, method and class. The examples of these methods can also be found in Fowler (2010) and Fowler *et al* (1999). The used methods include the following:

- **Consolidate Conditional Expression:** when there is a sequence of conditional tests with the same result, then combine them into a single conditional expression and extract it into a new method. Figure 1 shows an example.
- **Extract Class:** when a class is doing work that should be done by two, then extract a new class from the original class and move the relevant fields and methods from the old class into the new class (Fowler *et al*, 1999). As a result, each class will have clear responsibilities.
- **Encapsulate Field:** for each public field (attribute) in a class, encapsulate this field by making it private and provide methods for accessing (getting) and updating (setting) its value (Fowler *et al*, 1999). In addition, replace all direct references to the attribute by calls to these methods.

```
// Before
double disabilityAmount() {
  if (seniority < 2) return 0;
  if (monthsDisabled > 12) return 0;
  if (isPartTime) return 0;
  // compute the disability amount
  . . .
}

// After
double disabilityAmount() {
  if (isNotEligableForDisability())
      return 0;
  // compute the disability amount
  . . .
}
boolean isEligibleForDisability() {
  return ((seniority < 2) || (monthsDisabled > 12)
          || (isPartTime));
}
```

Figure 1: Consolidate conditional expression: before and after refactoring

- **Hide Method:** if a method is not used by any other class, then change its visibility by making the method private (Fowler *et al*, 1999).
- **Extract Method:** this technique extracts a set of statements that can be grouped together into a new method whose name explains the purpose of the method (Fowler *et al*, 1999). Then, it replaces the extracted statements with a call to a new method. "Extract Method" allows the extracted method to be reused in other places, and makes the code more readable.
- **Inline Class:** when there is a class which is not doing very much, move all its attributes and methods into another class and delete it. "Inline Class" is the reverse of "Extract Class" (Fowler *et al*, 1999).
- **Inline Method:** this technique allows the placement of a method's body into the body of its callers and remove the method (Fowler *et al*, 1999). It can be used when a method's body is as clear as its name.
- **Inline Temp:** when there is a temp variable that is assigned once with a simple expression and it is getting in the way of other refactoring, then replace all references to that temp with the expression (Fowler *et al*, 1999).
- **Remove Setting Method:** for any field that should be set at creation time and never altered, remove any setting method for that field (Fowler *et al,* 1999). This means that if you do not want a field to change once the object is created, then do not provide a setting method for that field.
- **Replace Assignment with Initialization:** since, in general, the programmers first declare many variables that will be used in the program, and then, assign values to them. Instead, make it into a direct initialization (Fowler, 2010).
- **Replace Magic Number with Symbolic Constant:** sometimes you have numbers (magic numbers) with special values that are difficult to understand their meaning. To solve this problem, create a constant with a useful name and replace the number with it (Fowler *et al*, 1999).
- **Reverse Conditional:** this technique allows to reverse the sense of the conditional and reorder the conditional's clauses (Fowler, 2010). This will make the conditional easier to understand.
- **Consolidate Duplicate Conditional Fragments:** when the same fragment of code is in all branches of a conditional expression, move it outside of the expression (Fowler, 2010).
- **Decompose Conditional:** when you have a complicated conditional (if-then-else) statement, extract methods from the condition, then part, and else parts (Fowler, 2010).
- **Remove Assignments to Parameters:** when the code assigns to a parameter, use a temporary variable instead (Fowler, 2010).
- **Remove Control Flag:** when you have a variable that is acting as a control flag for a series of boolean expressions, use a break or return instead (Fowler, 2010).
- **Replace Nested Conditional with Guard Clauses:** when a method has conditional behaviour that does not make clear what the normal path of execution is, use Guard Clauses for all the special cases (Fowler, 2010).
- **Add Parameter:** when a method needs more information from its caller; add a parameter for an object that can pass on this information (Fowler, 2010).
- **Encapsulate Collection:** when a method returns a collection, make it return a read-only view and provide add/remove methods (Fowler, 2010).

- **Remove Parameter:** if a parameter is no longer used by the method body, remove it (Fowler, 2010).
- **Rename Method:** when the name of a method does not reveal its purpose, change the name of the method (Fowler, 2010).

4. REFACTORING METHODS UNDER STUDY FOR ARCHITECTURE STABILITY

This section explains the refactoring methods that are used for architecture stability, two examples from Fowler (2010) and Fowler *et al* (1999) are given. The examples of the other methods are given in Fowler (2010) and Fowler *et al* (1999). These methods are:

- **Extract Subclass:** when a certain class contains a subset of features or members only used in some instances, a subclass is created for that subset (Fowler, 2010). Figure 2 shows an example.
- **Extract Super-Class:** the extraction of a super-class from a set of classes with similar features or members to contain the common features and members (Fowler, 2010).
- **Hide Delegate:** when a client is calling a delegate class of an object, methods are created on the server to hide the delegate (Fowler, 2010).
- **Introduce Foreign Method:** when a server class needs an additional method but you cannot modify the class, a method is created in the client class with an instance of the server class as its first argument (Fowler, 2010).
- **Move Field:** if a field in a class is used by another class more than its class, a new field is created in the other class (Fowler, 2010).
- **Move Method:** if a method in a class is used by another class more than its class, a new method with similar body is created in the other class and the old method is either turned into a simple delegation, or removed (Fowler, 2010).
- **Pull Up Field:** when two or more subclasses have the same fields, the fields are moved to the super-class (Fowler, 2010).
- **Pull Up Method:** when a set of subclasses have methods with identical results, the methods are moved to the super-class (Fowler, 2010).
- **Push Down Field:** when a field in a super-class is used only by some subclasses, the field is moved to those subclasses (Fowler, 2010).
- **Push Down Method:** when a behaviour or a method on a super-class is only relevant for some of its subclasses, then the method is moved to those subclasses (Fowler, 2010).

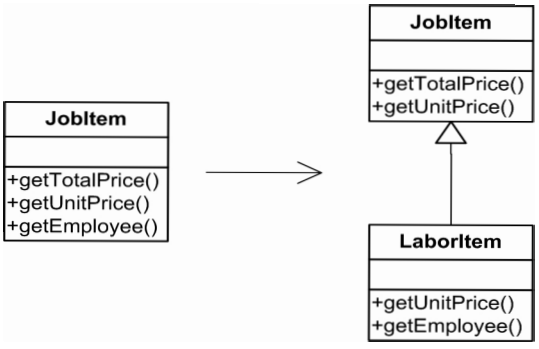

Figure 2: Extract Subclass: before and after refactoring

- **Replace Conditional with Polymorphism:** if a condition chooses different behaviour based on the type of an object, each clause of the condition is moved to an overriding method in a subclass and the original method is converted to abstract (Fowler, 2010).
- **Replace Delegation with Inheritance:** when many simple delegations are used for the entire interface, then the delegating class can be extracted as a subclass of the delegate (Fowler, 2010).

5. REFACTORING IMPACT ON STABILITY

In order to achieve the objectives of the paper, we empirically evaluate and assess the impact of class and architecture stability. The empirical validation can provide evidence of the impact. In this section we present the experiments we conducted and their results.

5.1 Data Used for Refactoring

Data used for refactoring is collected from different sources including student projects, open source systems and from Fowler's catalogue of refactorings (Fowler, 2010; Fowler *et al*, 1999). The selection of the source code has no effect on the results since the refactoring should be applied in the same way in any code as long as the code contains the "bad smell".

For our experiments, we selected small size code, in fact, the smaller the code the better it is because small size code can clearly show the impact of refactoring on class stability with no other side effects. The following section details the process followed to measure the impact of refactoring on class stability.

5.2 Refactoring Impact on Class Stability

The objective of this experiment is to assess the effect of refactoring on class stability; therefore, we applied the selected refactoring techniques, listed in Section 3, on source code. We used the class stability metric proposed by Alshayeb *et al* (2011) in which they identified a set of eight class properties/factors that affect class stability. These properties represent the key elements of the class structure; hence the stability of these properties affects the stability of the whole class. These properties were used to measure the overall class stability. These eight properties are: class access-level, class interface name, inherited class name, class variable, class variable access-level, method signature, method access-level, method body. Changes of these properties yield to unstable class.

5.2.1 Assessing and Classifying Refactoring Methods Based on Class Stability

Since our focus is class stability, we focused on refactoring techniques that are applied within classes (or use only one class); these refactorings also cover field (attribute), method, and class levels. We investigated only two refactoring methods that generate or combine two classes: extract class and inline class. They seem to generate very unstable classes. The reason for the high instability is that only the source class is measured and the other class (that is either created or deleted) was not considered. Therefore, it is expected that the source class will have many changes which will highly affect its stability.

Each refactoring technique is applied independently from the other techniques, no other changes are made on the code except those related to the refactoring applied. Once the refactoring is applied, the class stability between the two versions (before and after refactoring) is then measured and the effect on each class property identified by Alshayeb *et al* (2011) is recorded. A Class Stability Metric Tool (CSMT) (Naji, 2008) is used to automate the analysis of class stability.

The value of stability is not our concern. Our main concern is how many class properties have

```
//before refactoring
    void printOwing() {
          printBanner();
          //print details
          System.out.println ("name:      " + _name);
          System.out.println ("amount    " + getOutstanding());
    }

// after refactoring
    void printOwing() {
          printBanner();
          printDetails(getOutstanding());
    }
    void printDetails (double outstanding) {
          System.out.println ("name:      " + _name);
          System.out.println ("amount    " + outstanding);
    }
```

Figure 3: Example of applying Extract Method refactoring (Fowler *et al*, 1999)

been affected. This is because the fact that the code may contain more than one instance of the same property (for example more than one class variable); however, only one of these instances may change. Therefore, our classification will consider the number of properties that have changed and not the value of the change.

Figure 3 shows an example of source code before and after applying "Extract Method" refactoring. The source code of the example is adopted from Fowler *et al* (1999).

We notice from the example that only "method body" class property has changed. The other seven class properties remain unchanged since "extract method" does not affect them as shown in Table 2.

Table 2 shows the stability value of each of the eight properties identified in Alshayeb *et al* (2011) and the overall class stability after applying the refactoring techniques on the data adopted from student projects, open source systems and from Fowler's catalogue of refactorings. The number in each cell represents the stability of that class property after applying the refactoring; 1 means fully stable (no changes made for that property), 0 means fully unstable (the class property is removed or completely changed) and a value between 0-1 represents the stability of that class property.

Table 3 shows the number of properties that each refactoring affects, i.e. make it unstable. From Tables 2 and 3 we observe that two refactoring methods make the class the least stable. These two methods are: extract class and inline class. These two methods mainly affect the "class-level" since "Extract Class" generates a new class and "Inline Class" combines two classes into one. These two methods affect two class level properties (class variable and class variable access level) in addition to three method properties. The reason for the high instability is that only the source class is measured and the other class (that is either created or deleted) was not considered. Therefore, it is expected that the source class will have many changes which will highly affect its stability. These two refactorings also affect architecture stability as will be demonstrated in the next sections.

Refactoring methods that mainly affect the "method level" make the class somewhat unstable. These refactoring methods mainly affect method access level, method signature and method body.

Many refactoring methods affect only method body without any effect on the other class properties. Therefore, these refactoring methods have the least impact on stability; only two methods affect method access level and method signature without affecting the method body. These refactoring methods mainly affect field and are applied within the methods.

We also notice that class access level, inherited class name and class interface name have not

Refactoring Method	Class Access-level	Inherited class name	Class interface name	Class variable	Class variable access-level	Method access-level	Method signature	Method body	Class Stability	Class Stability
1. Extract Class	1	1	1	0.3334	0.3334	0.3334	0.3334	0.166	**0.56245**	**0.375**
2. Inline Class	1	1	1	0.5	0.5	0.6667	0.6667	0.3334	**0.70835**	
3. Remove Setting Method	1	1	1	1	1	0.6	0.6	0.4	**0.825**	**0.625**
4. Inline Method	1	1	1	1	1	0.6667	0.6667	0.3334	**0.83335**	
5. Add Parameter	1	1	1	1	1	0	0	0	**0.625**	
6. Encapsulate Collection	1	1	1	1	1	0.5	0.5	0.5	**0.8125**	
7. Remove Parameter	1	1	1	1	1	0	0	0	**0.625**	
8. Rename Method	1	1	1	1	1	0	0	0	**0.625**	
9. Consolidate Duplicate Conditional Fragments	1	1	1	1	1	1	1	0.667	**0.958375**	**0.875**
10. Replace Assignment With Initialization	1	1	1	1	1	1	1	0	**0.875**	
11. Reverse Conditional	1	1	1	1	1	1	1	0.75	**0.96875**	
12. Inline Temp	1	1	1	1	1	1	1	0	**0.875**	
13. Extract Method	1	1	1	1	1	1	1	0	**0.875**	
14. Replace Magic Number with Symbolic Constant	1	1	1	1	1	1	1	0	**0.875**	
15. Consolidate Conditional Expression	1	1	1	1	1	1	1	0.66667	**0.95833375**	
16. Decompose Conditional	1	1	1	1	1	1	1	0	**0.875**	
17. Remove Assignments To Parameters	1	1	1	1	1	1	1	0	**0.875**	
18. Remove Control Flag	1	1	1	1	1	1	1	0.5	**0.9375**	
19. Replace Nested Conditional with Guard Clauses	1	1	1	1	1	1	1	0.8	**0.975**	
20. Hide Method	1	1	1	1	1	0.1667	1	1	**0.8958375**	
21. Encapsulate Field	1	1	1	1	0.5	1	1	1	**0.9375**	

Table 2: Class stability and each property value after applying the refactoring technique

Refactoring Method	Unstable properties
Extract Class	5
Inline Class	
Remove Setting Method	3
Inline Method	
Reverse Conditional	
Add Parameter	
Encapsulate Collection	
Remove Parameter	
Rename Method	
Consolidate Duplicate Conditional Fragments	1
Replace Assignment With Initialization	
Inline Temp	
Extract Method	
Replace Magic Number With Symbolic Constant	
Consolidate Conditional Expression	
Decompose Conditional	
Remove Assignments To Parameters	
Remove Control Flag	
Replace Nested Conditional With Guard Clauses	
Hide Method	
Encapsulate Field	

Table 3: Number of unstable properties

been affected by any investigated refactoring method as we only focused on the refactoring methods that are applied within classes.

Based on our observation, we can classify the refactoring techniques into three categories:

Category 1: Refactorings in this category make classes unstable; they affect five class properties. The stability of the class can be 0.375 (assuming no other changes are made to the code). This category is for the classes that combine/generate two classes. The measurement of stability is done for the original class. The newly generated classes are not measured.

Category 2: Refactorings in this category make classes somewhat unstable, they affect three class properties. The stability of the class can be 0.625 (assuming no other changes are made to the code). Refactorings in this category are the refactoring methods that mainly affect the method access level, method signature and method body.

Category 3: Refactorings in this category have the least impact on stability. They affect the class stability; however, they maintain reasonable stability since they affect only one class property. The stability of the class can be 0.875 (assuming no other changes are made to the code). Refactoring methods in this category are mainly the refactoring methods that affect fields and are applied within the methods.

	Design Size			Hierarchies			Single Inheritance			Multiple Inheritance			Depth of Inheritance			Width of Inheritance			Number of Parents			Number of Methods			Class Coupling		
	Before	After	Change	Before	After	Change	Before	After	Change	Before	After	Change	Before	After	Change	Before	After	Change	Before	After	Change	Before	After	Change	Before	After	Change
1. Replace Conditional with Polymorphism	1	4	+3	0	3	+3	0	3	+3	0	3	+3	0	3	+3	0	3	+3	0	3	+3	4	7	+3	0	0	0
2. Extract Subclass	2	3	+1	0	1	+1	0	1	+1	0	0	0	0	1	+1	0	1	+1	0	1	+1	3	4	+1	0	0	0
3. Replace Delegation with Inheritance	2	2	0	0	1	+1	0	1	+1	0	0	0	0	1	+1	0	1	+1	0	1	+1	2	1	-1	1	0	-1
4. Extract Super Class	2	3	+1	0	2	+2	0	2	+2	0	0	0	0	2	+2	0	2	+2	0	2	+2	6	6	0	0	0	0
5. Extract Class	1	2	+1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6	8	+2	0	1	+1
6. Inline Class	2	1	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8	6	-2	1	0	-1
7. Hide Delegate	3	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	1	-1	2	1	-1
8. Introduce Foreign Method	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	6	+1	1	1	0
9. Pull Up Method	3	3	0	2	2	0	2	2	0	0	0	0	2	2	0	2	2	0	2	2	0	2	1	-1	0	0	0
10. Push Down Method	3	3	0	2	2	0	2	2	0	0	0	0	2	2	0	2	2	0	2	2	0	1	1	0	1	0	-1
11. Move Field	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12. Move Method	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
13. Pull Up Field	3	3	0	2	2	0	2	2	0	0	0	0	2	2	0	2	2	0	2	2	0	0	0	0	0	0	0
14. Push Down Field	3	3	0	2	2	0	2	2	0	0	0	0	2	2	0	2	2	0	2	2	0	0	0	0	0	0	0

Table 4: The architecture stability and each property

We consider methods that affect 0, 1 or 2 properties to be in Category 3, methods that affect 3, 4 or 5 properties to be in Category 2 while methods that affect 6, 7 or 8 properties to be in Category 1.

5.3 Refactoring Impact on Architecture Stability

The objective of this experiment is to assess the effect of refactoring on architecture stability; therefore, we applied the selected refactoring techniques, listed in Section 4, on source code. We used the architecture stability metrics proposed by Bansiya (2000) in which he identified a set of eight properties/factors that affect architecture stability. These nine properties are: design size, hierarchies, single inheritance, multiple inheritance, depth of inheritance, width of inheritance, number of parents, number of methods and class coupling. Changes of these properties yield to unstable architecture.

5.3.1 Assessing and Classifying Refactoring Methods Based on Architecture Stability

Since our focus in this experiment is architecture stability, we focused on refactoring techniques that affect multiple classes, the methods that combine, affect or generate two or more classes.

Each technique is applied independently from the others, no other changes are made on the code except those related to the retracting applied. Once the refactoring is applied, the stability between the two versions (before and after refactoring) is then measured and the effect on each property identified by Bansiya (2000) is recorded. As in class stability, the value of stability is not our concern. Our main concern is how many properties have been affected.

Table 4 shows the architecture stability and each property value after applying the refactoring techniques while Table 5 shows the number of unstable properties.

From Tables 4 and 5 we can observe that four of the investigated refactoring methods (replace conditional with polymorphism, extract subclass, replace delegation with inheritance and extract super class) make the architecture unstable. These refactoring methods mainly affect the "class

Refactoring Method	Unstable properties
Replace Conditional With Polymorphism	8
Extract Subclass	7
Replace Delegation With Inheritance	7
Extract Super Class	6
Extract Class	3
Inline Class	3
Hide Delegate	2
Introduce Foreign Method	1
Pull Up Method	1
Push Down Method	1
Move Field	0
Move Method	0
Pull Up Field	0
Push Down Field	0

Table 5: Number of unstable properties

level" hierarchy. These methods were not considered in the classification of refactoring methods that affect class stability. However, they will also produce unstable class as they may affect the class-level properties of the class stability metric (class access-level, inherited class name and class interface name).

Refactoring methods that mainly affect class-level (extract class and inline class) make the architecture somewhat unstable. These methods were considered in the class stability and found to be the most methods that make class unstable.

The refactoring techniques that have the least impact on architecture stability are the ones that mainly affect the field and method levels. These refactoring methods were not considered in the class stability.

Based on our observation we can classify the refactoring techniques into three categories:

Category 1: Refactorings in this category make classes unstable; they affect 6 or 7 or 8 properties. These methods are mainly the methods that deal with generalization that move methods around a hierarchy of inheritance.

Category 2: Refactorings in this category (extract class and inline class) make classes somewhat unstable; they affect 3 or 4 or 5 properties. These methods mainly deal with classes and moving features between objects. These methods were used in the class stability and found to produce unstable classes.

Category 3: Refactorings in this category have the least impact on stability; they affect 2 or 1 or no properties. These methods mainly affect field and method levels.

We consider methods that affect 0, 1 or 2 properties to be in Category 3, methods that affect 3, 4 or 5 properties to be in Category 2 while methods that affect 6, 7, 8 or 9 properties to be in Category 1.

5.4 Summary of the Results

Software designers who want to optimize their design for class stability should avoid refactoring methods that deal with class level (such as Extract Class and Inline Class). They are not recommended to apply refactoring methods that are applied at the method level or affect the method signatures. However, refactoring methods that are applied within the methods and affect only method body can be used as they have the least impact on class stability.

On the other hand, software designers who want to optimize their design for architecture stability should avoid using refactoring methods that affect the class-level hierarchy. They are not recommended to use refactoring methods that affect class-level while they can use the refactoring methods that affect field and method levels.

In general, designers should strive to apply refactoring methods that are at the lowest levels (in both class and architecture level) as they have the least impact on stability. Therefore, refactoring methods at field and method levels have the least effect on class and architecture stability and thus can be applied when stability is the main concern for the software designers.

Threat to validity

This study presents an empirical experiment to assess and classify refactoring methods based on their impact on class and architecture stability. However, there are two main limitations to the extent to which these results can be generalized. First, our methodology to investigate the effect of refactoring methods on architecture stability is based on available research study presented in Bansiya (2000) without validation on our behalf. Second, we used the refactoring code presented in

Fowler (2010) and Fowler *et al* (1999), however, when the code is not available we performed the refactorings by ourselves. This may have introduced a bias while performing the refactorings.

The small size of the used code does not present a threat to the validity for this study. This study is limited to investigating the effect of refactoring methods within class or between classes and not at the system-level. Therefore, the size of the systems is not crucial.

6. CONCLUSION AND FUTURE WORK

The purpose of this research is to assess the impact of refactoring on class and architecture stability and then propose a classification for refactoring methods based on their impact on class and architecture stability. Therefore, a number of refactoring methods were selected and used in this study.

The impact of each refactoring technique on class and architecture levels is assessed individually. Based on the percentage of impact on class properties, refactoring methods were classified into three categories (highly unstable, unstable, and fairly stable).

For class stability, refactoring methods that affect "class-level" were found to have the highest impact on class stability. Refactoring methods that mainly affect the "method-level" were found to have less impact on class stability while refactoring methods that affect fields and are applied within the methods were found to have the least impact on stability.

With regard to architecture stability, refactoring methods that affect the class-level hierarchy were found to have the highest impact on architecture stability. Refactoring methods that affect class-level were found to have less impact on architecture stability while refactoring methods that affect field and method levels were found to have the least impact on architecture stability.

This study helps software designers in selecting appropriate refactoring methods when stability is their main concern. This study also enables them to predict the quality drift on class and architecture stability caused by using specific refactoring methods.

Future research includes examining more refactoring techniques and assessing the impact of refactoring to patterns techniques on class and architecture stability and other quality attributes.

7. ACKNOWLEDGEMENT

The author 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

AHMED, M., RUFAI, R., ALGHAMDI, J. and KHAN, S. (2003): Measuring architectural stability in object oriented software. Dhahran, Saudi Arabia, King Fahad University of Petroleum and Minerals.

ALSHAYEB, M. (2009): Empirical investigation of refactoring effect on software quality. *Information and Software Technology Journal* 51(9), 1319–1326.

ALSHAYEB, M. and LI, W. (2004): An empirical study of system design instability metric and design evolution in an agile software process. *Journal of Systems and Software,* 74(3): 269–274.

ALSHAYEB, M., LI, W. and GRAVES, S. (2001): An empirical study of refactoring, new design, and error-fix efforts in extreme programming. In *Proceeding: 5th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2001)* 323–325. Orlando, Florida.

ALSHAYEB, M., NAJI, M., ELISH, M. and AL-GHAMDI, J. (2011): Towards measuring object-oriented class stability. IET Software, in press.

BAHSOON, R. (2003): Evaluating software architectures for stability: A real options approach. In *Proceedings of the 25th International Conference on Software Engineering, Doctoral Symposium,* 765. Portland, USA.

BANSIYA, J. (2000): Evaluating framework architecture structural stability. *ACM Computing Surveys* (CSUR).

BECK, K. (1999): *Extreme Programming Explained: Embracing Change*. Addison Wesley.

DUBOIS, B., DEMEYER, S. and VERELST, J. (2005): Does the "Refactor to understand" reverse engineering pattern improve program comprehension? In *9th European Conference on Software Maintenance and Reengineering (CSMR'05),* 334–343.

DUBOIS, B. and MENS, T. (2003): Describing the impact of refactoring on internal program quality. In *International Workshop on Evolution of Large-scale Industrial Software Applications,* 37–48.

DUBOIS, B., DEMEYER, S. and VERELST, J. (2004): Refactoring – Improving coupling and cohesion of existing code. In *11th Working Conference on Reverse Engineering (WCRE'04),* 144–151.
ELISH, K. and ALSHAYEB, M. (2011): A classification of refactoring methods based on software quality attributes. *Arabian Journal of Science and Engineering*, 36(7): 1253–1267.
FOWLER, M. (2010): Refactoring Website: http://www.refactoring.com/. Accessed 12-Apr-2010.
FOWLER, M., BECK, K., BRANT, J., OPDYKE, W. and ROBERTS, D. (1999): *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley.
GEPPERT, B., MOCKUS, A. and ROBLER, F. (2005): Refactoring for changeability: A way to go? In *11th IEEE International Software Metrics Symposium (METRICS'05)*.
GROSSER, D., SAHRAOUI, H. and VALTCHEV, P. (2002): Predicting software stability using case-based reasoning. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering,* 295–298. Edinburgh, UK.
GROSSER, D., SAHRAOUI, H. and VALTCHEV, P. (2003): An analogy-based approach for predicting design stability of java classes. In *Proceedings of the Ninth International Software Metrics Symposium,* 252–262.
HASSAN, Y.S. (2007): *Measuring Software Architectural Stability Using Retrospective Analysis*. King Fahd University of Petroleum & Minerals.
JAZAYERI, M. (2002): On architectural stability and evolution. In *Proceedings of the seventh Ada-Europe International Conference on Reliable Software Technologies,* 13–23.
KATAOKA, Y., IMAI, T., ANDOU, H. and FUKAYA, T. (2002): A quantitative evaluation of maintainability enhancement by refactoring. In *International Conference on Software Maintenance (ICSM'02),* 576–585.
LI, W., ETZKORN, L., DAVIS, C. and TALBURT, J. (2000): An empirical study of object-oriented system evolution. *Information and Software Technology,* 42(6): 373–381.
MATTSSON, M. and BOSCH, J. (2000): Stability assessment of evolving industrial object-oriented frameworks. *Journal of Software Maintenance: Research and Practice,* 12(2): 79–102.
MATTSSON, M. and BOSH, J. (1999): Characterizing stability in evolving frameworks. In *Proceedings of Technology of Object-Oriented Languages and Systems,* 118–130.
MOSER, R., SILLITTI, A., ABRAHAMSSON, P. and SUCCI, G. (2006): Does refactoring improve reusability? In *9th International Conference on Software Reuse (ICSR'06),* 287–297.
NAJI, M. (2008): *Measuring Class Stability*. King Fahd University of Petroleum & Minerals.
OLAGUE, H., ETZKORN, L., LI, W. and COX, G. (2006): Assessing design instability in iterative (agile) object-oriented projects, *Journal of Software Maintenance and Evolution: Research and Practice,* 18(4): 237–266.
OPDYKE, W. (1992): *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. Univ. of Illinois at Urbana-Champaign.
RAT, D., DUCASSE, S., GIRBA, T. and MARINESCU, R. (2004): Using history information to improve design flaws detection. In *Proceedings of the Eighth European Conference on Software Maintenance and Reengineering (CSMR'04)*.
STROGGYLOS, K. and SPINELLIS, D. (2007): Refactoring – Does it improve software quality? In *5th International Workshop on Software Quality (WoSQ'07: ICSE Workshops),* 10–16. ACM Press.
TAHVILDARI, L. and KONTOGIANNIS, K. (2003): A metric-based approach to enhance design quality through meta-pattern transformations. In *Proceedings of 7th European Conference On Software Maintenance And Reengineering (CSMR'03),* 183–192.
TONU, S., ASHKAN, A. and TAHVILDARI, L. (2006): Evaluating architectural stability using a metric-based approach. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR'06),* 261–270.
WAKE, W.C. (2003): *Refactoring Workbook*. Addison Wesley.
WILKING, D., KHAN, U. and KOWALEWSKI, S. (2007): An empirical evaluation of refactoring. *e-Informatica Software Engineering Journal,* 1(1): 27–42.

BIOGRAPHICAL NOTES

Dr Mohammad Alshayeb received his MS and PhD in computer science and certificate of software engineering from the University of Alabama in Huntsville in 2000, 2002 and 1999 respectively. He received his BS in computer science from Mutah University, Jordan in 1995. Dr Alshayeb worked as a senior researcher and software engineer and managed software projects in the United States and the Middle East. Dr Alshayeb taught and coordinated industrial training courses. He provided consulting services to major industrial and educational institutes. He is a certified project manager (PMP). Dr Alshayeb's research interests include software refactoring, software quality, software measurement and metrics, object-oriented design and empirical studies in software engineering.


Mohammad Alshayeb