2015 IEEE 39th Annual International Computers, Software & Applications Conference

Analyzing Refactorings' Impact on Regression Test Cases

Yuan Gao, Hui Liu*, XiaoZhong Fan, ZhenDong Niu, Bridget Nyirongo
School of Computer Science and Technology
Beijing Institute of Technology
Beijing, China
Email: alexandagy@gmail.com, liuhui2005@gmail.com, fxz@bit.edu.cn, zniu@bit.edu.cn, bnyirongo@cc.ac.mw

***Abstract*—Software refactoring is to improve readability, maintainability and expansibility of software by adjusting its internal structure, whereas the external behaviors of software are not changed. Although software refactoring should not change the external behaviors of software systems, they might make a regression test case obsolete (with syntax and runtime errors) or fail. People have investigated which refactorings had an influence on regression test case. However, how test cases are influenced by refactorings and what kind of errors might be introduced remain unknown. To this end, in this paper, we proposed an approach to analyze refactorings' impact on regression test cases. On one hand, we analyzed why regression test cases failed. On the other hand, we analyzed the influence of refactorings on software interfaces. Based on the analysis, we built up a mapping between refactorings and test case failure. Such a mapping can be used to guide test case repair automation where test cases are made obsolete by refactorings. The approach was evaluated on five open-source applications. Evaluation results suggest that the precision of the approach is greater than 80%.**

I. Introduction

Software refactoring [1][2][3] is to improve readability, maintainability and expansibility of software by adjusting its internal structure whereas the external behaviors of software are not changed. The source code becomes complex, difficult to read or debug, and even harder to expand because of software modification caused by evolving requirements. Software refactoring has been widely used to delay the degradation effects of software aging and facilitate software maintenance [4][5][6].

Test case execution is a usual way to ensure software behavior preserving [7]. Test case is crucial for that it is relied on to determine whether a refactoring is applied correctly and the behavior of the production code is kept unchanged [3][8]. In Test Driven Development (TDD), it requires that test cases should be created or modified before writing or modifying source code. Software refactoring is a kind of special software modification. In TDD, software refactoring should follow the rule of "test first". It means that a test case should be modified before refactoring could be finally applied to the corresponding source code. When a test case is modified, programmers need to refactor the source code to adapt to the test case. It ensures the correctness for continuous but small refactoring activities [9].

*Corresponding author

According to the definition, software refactoring requires that the external behaviors of software should be unchanged when modifying software internal elements. Nonetheless, they might make a test case obsolete (with syntax and runtime errors) or fail [10]. Test case is difficult to be repaired, especially in Test Driven Refactoring (TDR) [11]. Traditionally, after we perform refactoring, we can repair test case according to messages returned by compiler or debugger. But, in TDR, we can only revise a test case based on hypothesis how it would be affected by the upcoming refactoring. Thus, it makes the repair of the corresponding test case more difficult.

If we know how software refactoring impact on a test case, refactoring inference can be inferred according to primary modification on test cases. Consequently, refactoring can be accomplished and the test case can be repaired automatically.

To address the issue, published research [12][13][14] have analyzed the influence of refactoring on test cases. But, people only made some foundational studies on the issue such as which refactoring had an influence on regression test case. How test cases are influenced by refactorings and what kind of errors might be introduced remain unknown. What's more, the research findings tend to rely on expert's subjective conjecture. They lacked empirical evidence support.

To this end, we proposed an approach to analyze how refactoring impact on regression test cases. First, we analyzed why a test case failed. Second, we analyzed the impact of refactorings on software interfaces. Based on the analysis, we built up a mapping relationship graph which can be used to guide test case repair automation where test cases are made obsolete by refactorings. The approach was evaluated on five open-source applications. Evaluation results suggest that the precision of the approach is greater than 80%.

The rest of the paper is structured as follows: We illustrate a motivating example to show how refactoring impact on test case in section II. Section III proposes an approach to analyze the impact of refactorings on regression test cases. We carry out an evaluation of the approach on open-source applications in section IV. Section V presents an overview of related research. The conclusions drawn are provided in section VI.

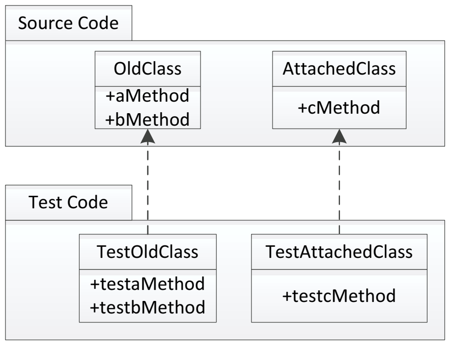


Fig. 1. **Before Refactoring**

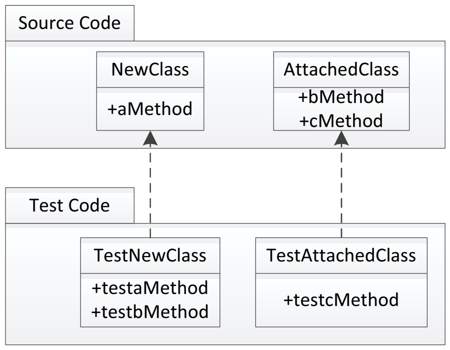


Fig. 2. **After Refactoring**

II. Motivation Example

To demonstrate that software refactoring may lead to failure of the test case, we present a motivating example, as depicted in Fig.1 and Fig.2. Fig.1 shows the state of source code and test code before refactoring. The source code consists of two classes, ”OldClass” and ”AttachedClass”. Corresponding test classes are ”testOldClass” and ”testAttachedClass”. Public methods (”aMethod”, ”bMethod” and ”cMethod”) in source code have corresponding test methods (”testaMethod”, ”testbMethod” and ”testcMethod”) in test classes.

We perform refactoring operation on source code as follows steps. (1) Rename Class: Change Class name ”OldClass” to ”NewClass”; (2) Move Method: Move ”bMethod” from ”NewClass” to ”AttachedClass”. The state of source code is changed, as depicted in Fig.2.

If we don’t revise the test case, test framework would return a message like ”OldClass is undefined”. Only if the instance type of ”OldClass” is replaced by ”NewClass”, the test case can pass. Then, another message will be returned like ”bMthod does not exist”, which is due to moving ”bMethod” from ”NewClass” to ”AttachedClass”. Therefore, ”testbMethod” cannot invoke ”bMethod” by the instance of ”NewClass”. The way we correct the error is to relocate the method call of ”bMethod”. After these modifications, source code and test case can run correctly. But, from Fig.2 we notice that the mapping relationship between source code and test case is not renewed yet, which would affect subsequent evolution and maintenance of the test code. For example, developers may not locate corresponding test cases by class names. Therefore, if we know how refactorings affect test cases, it can be used to guide test case repair automation.

III. Approach

A. *Test Case Failure Analysis*

Test case is used to validate source code. Test case runs in three steps in sequence: setting input data, calling and executing tested code and judging whether output is consistent with the expected result. Input data are one of the most important parts of the test case. Different data represent that the tested code runs in diverse scenarios. It includes parameters, local or global variables and IO data etc.. The expected result is the return value of tested code according to input data in ideal condition. Running test case is actually the process of comparing expected results with real output according to different input in turn and judging whether they are equal.

Software refactoring requires that the external behaviors of software should be unchanged. Therefore, the same test case should pass when it is used to test refactored code. But, as the example given in section 2, software refactoring may lead to failure of test case.

Information that is returned from testing framework reveals why the test case failed. We analyzed the collected information of test case failure and classified them according to external manifestation: (1) test case runs with an error or exception. The main cause of running error or exception is that refactoring alters the state of the relevant element (class, method, attribute etc.). These changes include name changes, location changes, type changes etc.. Therefore, test case cannot find or identify the changed element which leads to running or compiling error or exception. In motivation example, after performing Move Method refactoring, ”bMethod” is moved from ”NewClass” to ”AttachedClass”. Test case cannot find ”bMethod” by instance of ”NewClass” which leads to the compiling error. Other refactorings like Rename, Remove Parameter will also lead to the error; (2) the expected result and output are not consistent with each other. The main cause of the situation is that the form of the relevant element has changed. For instance, Replace Array with Object changes array into an object. The element in the array is replaced as a field of the object. We analyzed different condition of test case failure and specify the classification which is illustrated in Fig.3.

B. *Refactoring Influence on Software Interface*

In this section, we focus on refactoring influence on software interface. Software refactoring provides an efficient and controlled code arranging technology. Although software refactoring requires that the external behaviors of software should not be changed, many of the refactorings do change an interface[3] as declared by Fowler.

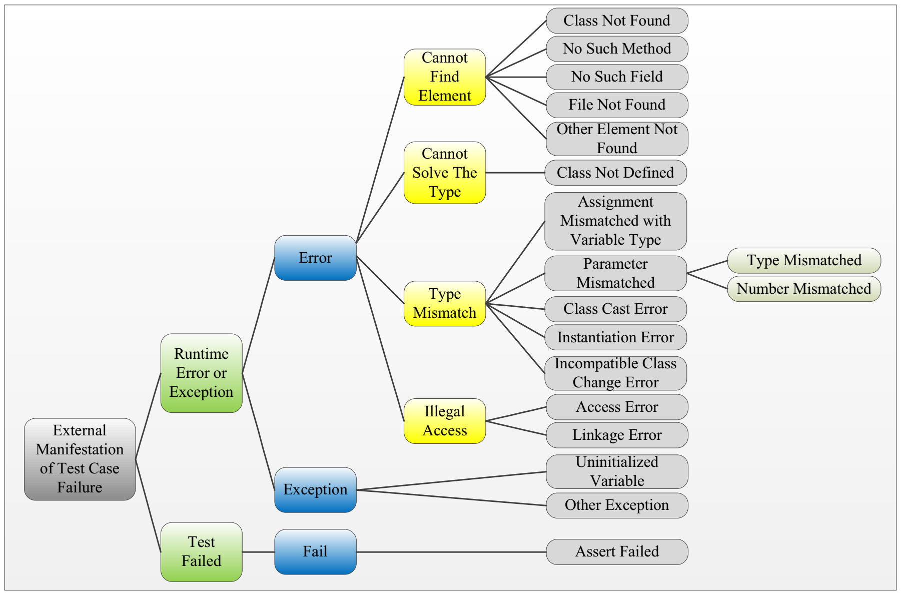


Fig. 3. **Classification of Test Case Failure External Manifestation**

In Java programming, interface is an abstract type. It cannot be instantiated and must be implemented by classes, thus conceals the implementation of itself. Interfaces realize multiple inheritance. A Java class may implement, and an interface may extend, any number of interfaces. Therefore, when a interface is changed, all the classes who implement the interface would be affected.

In regression test, interface changes are the primary causes of test case failure. A very small change in interface will affect many test cases. As mentioned above, refactorings would change interfaces. When we refactor, we change source code in three ways: adding element, removing element and adjusting element. In this place, element refers to software element, including field, method, class, package, even part of the source code. These operations affect the way of accessing software elements, thus affect software interface.

Fowler [3] proposed 72 refactorings in his book. Big refactorings, including *Convert Procedural Design to Objects*, *Separate Domain from Presentation*, *Tease Apart Inheritance*, and *Extract Hierarchy*, are composed of other smaller refactorings. The impact of big refactorings on software interface can be evaluated by the effect of a series of other smaller refactorings. Therefore, we do not take account of these big refactorings.

We analyzed the description and examples of refactorings in [3]. The way in which source code changes are specified into 14 categories according to refactoring intention, change form and range of software elements. Refactorings are classified based on these characteristics. The categories are outlined in Fig.4. The classified refactorings are listed in Table I.

As can be seen from Fig.4 and Table Ithe proportion of refactorings which change source code only by adding or removing elements is small. Most of refactorings use more complex operation to adjust source code. However, in fact, some refactorings which belong to adjusting element are kinds of adding or removing element synchronously.

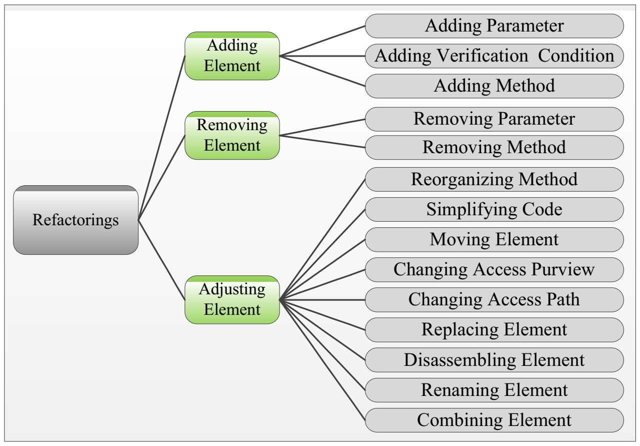


Fig. 4. **Causes of Interface Changes**

For example, refactorings, belonging to moving element, remove element from original position and add such element to objective position at the same time; Refactorings, belonging to disassembling element, split one element into two or more elements, which increases the element number; On the contrary , refactorings, belonging to combining element, combine two or more elements into one element, which decreases the element number. To simplify the analysis, these refactorings are no longer considered in other categories repeatedly.

According to the detailed explanation of the refactoring and accompanying examples, we classify the refactorings into

TABLE I
Category of Refactorings according to characteristic

Form	**Characteristics**	**Refactoring**
Adding Element	Adding Parameter	Add Parameter(II)
	Adding Verification Condition	Introduce Assertion(I)
	Adding Method	Introduce Foreign Method(II), Introduce Local Extension(II)
Removing Element	Removing Parameter	Remove Parameter(II)
	Removing Method	Remove Setting Method(II)
Adjusting Element	Reorganizing Method	Extract Method(I), Replace Temp with Query(I), Replace Method with Method Object(I) Introduce Explaining Variable(I), Substitute Algorithm(I),Split Temporary Variable(I) Remove Assignments to Parameters(I), Inline Method(I), Remove Control Flag(I)
	Simplifying Code	Decompose Conditional(I), Consolidate Conditional Expression(I), Introduce Null Object(I) Consolidate Duplicate Conditional Fragements(I), Replace Conditioanl With Polymorphism(I) Introduce Assertion(I),Replace Nested Conditional With Guard Clauses(I), Inline Temp(I)
	Moving Element	Move Method(II), Inline class(II), Pull Up Method(II), Extract class(II), Move Field(II) Form Template Method(II), Push Down Method(II), Pull Up Constructor Body(II), Pull Up Field(II) Extract Superclass(II),Push Down Field(II), Extract Interface(II), Extract Subclass(II)
	Changing Access Purview	Self Encapsulate Field(II), Encapsulate Downcast(II), Replace Constructor with Facotory Method(II) Change Unidirectional Association to Bindirectional(II), Encapsulate Collection(II) Change Bindirectional Association to Unidirectional(II), Hide Method(II), Encapsulate Field(II)
	Changing Access Path	Hide Delegate(II), Replace Delegation with Inheritance(II), Replace Inheritance with Delegation(II) Remove Middle Man(II)
	Replacing Element	Replace Type code with State/Startegy(II), Replace Magic Number with Symbolic Constant(II) Replace subclass with Fields(II), Replace Type code with Class(II), Replace Error Code with Exception(II) Replace Array with Object(II), Replace Data value with Object(II), Replace Record with Data Class(II) Change Reference to Value(II), Replace type code with subclasses(II), Change Value to Reference(II)
	Disassembling Element	Duplicate Observed Data(II), Separate Query from Modifier(II) Replace Parameter with Explicit Methods(II)
	Renaming Element	Rename Method(II)
	Combining Element	Parameterize Method(II), Introduce Parameter Object(II), Collapse Hierarchy(II) Preserve Whole Object(II), Replace Parameter with Methods(II)

two types based on whether they have an impact on software interfaces. Refactorings are labeled in Table I where type I shows that it has no effect on the interface. While type II means it can affect the interface.

From Table I, eighteen types of refactorings do not affect interface. They fall within three categories: adding verification condition, reorganizing method and simplifying code. Most of these refactorings only adjust the code inside a method and won't affect the output of the method. Therefore, they won't affect the interface. Other refactorings involve method invocation, such as extract method. When part of the code was encapsulated and moved to another position, a reference of the new method would be remained at the original position. Thus has no influence on the interface. Except for these, refactorings in other characteristic categories would affect the interface.

Characteristics of categories where refactoring would affect the interface are explained as below:

(1) Adding parameter. The number of parameter changes which lead to the change of interface.
(2) Adding method. New method needs a new access interface.
(3) Removing parameter. It is the same as (1).
(4) Removing method. The original method is deleted. The corresponding interface cannot be used.
(5) Moving element. Element is moved from original position to objective position. The original interface cannot be used.
(6) Changing access purview. The visibility of the element is changed, which can be accessed indirectly.
(7) Changing access path. The access relationship between two elements is altered. The original interface cannot be used.
(8) Replacing element. The original element is replaced by other types. The original interface cannot be used.
(9) Disassembling element. One element is split into two or more elements. The original interface cannot access several elements at the same time.
(10) Renaming element. The name of the element is changed, which cannot be found through original interface.
(11) Combining element. Two or more elements are combined into one element, which cannot be accessed through original interface.

C. Mapping Relationship

According to analysis mentioned above, we got a set of test case failures (Fig.3) and a set of ways by which refactoring affect software interfaces (Fig.4). We combine the two parts and built up a mapping between refactorings and test case failure. Such a mapping can be used to guide test case repair automation where test cases are made obsolete by refactorings.

We have accomplished the mapping relationship graph to show all 68 refactorings and their impacts on regression test cases. However, the size of the graph precludes its inclusion in this paper. It is attached in the appendix. Characteristics of refactorings are used as a link to associate refactorings with test case failure. Connections between refactorings and test case failure indicates "a refactoring may lead to what kind of test case failure."

According to the mapping relationship graph, when we need to revise test cases before refactoring, we should follow the steps below: First, confirm the type of refactoring being performed. Second, find the characteristic of the refactoring and how it affects the software interface. Then, find out possible external manifestation of the test case failure in term of the characteristic. Last, revise test cases based on the causes of test case failure.

IV. Evaluation

To evaluate the proposed approach, we conducted an evaluation with open-source applications.

A. Experimental Setup

TABLE II
Basic Information of Evaluation Projects

Project	Version	Number of Class (NOC)	Line of Code (KLOC)	Number of Test case (NOC)
Ant	1.7.0	778	93.8	287
	1.8.4	841	105.1	308
Maven	2.0.11	266	24.8	95
	2.2.1	284	105.1	102
Struts	2.3.1	1520	109.3	367
	2.3.4	1535	110.2	369
Lucene	3.1.0	1220	141.9	610
	3.6.0	1642	234.7	870
Spring Framework	3.0.2	3181	177.0	977
	3.0.7	3240	183.1	1000

Subject applications are shown in Table II. These applications are open-source applications available at Apache [15] or SourceForge [16].

Ant[*] is a Java library and command-line tool whose mission is to drive processes described in build files as targets and extension points dependent upon each other.

Maven[†] is a software project management and comprehension tool based on Project Object Model(POM).

Struts[‡] is an open source framework for building Java web applications, and can be integrated with other technologies to provide model and view.

Lucene[§] is a high-performance, full-featured text search engine library written entirely in Java.

Spring Framework[¶] is an open source application framework and inversion of control container for the Java platform.

For each application, two major release versions were selected (Ant_version 1.7.0/1.8.4, Maven_version 2.0.11/2.2.1, Struts_version 2.3.1/2.3.4, Lucene_version 3.1.0/3.6.0 and Spring Framework_version 3.0.2/3.0.7). Application introductions reveal that the five applications come from different domains, and were developed by different developers. The size of applications varies from 24.8KLOC to 234.7KLOC. All versions of the applications are stable released. Selected applications may help generalize the conclusions.

A well-known and publicly available refactoring identification tool, RefactoringCrawler[17], is used in the evaluation. RefactoringCrawler is used to identify refactorings between two versions of production code. It has been reported that the recall of the tool on EclipseUI and Struts reached to 86% [17]. RefactoringCrawler can identify seven types of refactorings, including Rename Class(RC), Rename Method(RM), Rename Package(RP), Move Method(MM), Pull Up Method(PUM), Push Down Method(PDM) and Change Method Signature(CMS).

The evaluators are doctoral students who major in software engineering. For each application, the evaluation follows the following process:

(1) RefactoringCrawler was applied to mine the refactoring history from the application code.
(2) Selecting relevant test case according to mined refactorings.
(3) Applying test case in former version to refactored source code.
(4) Recording execution results and comparing it with mapping relationship graphs.
(5) Counting the comparison results and calculating accuracy.

In the evaluation, when evaluator finished running one test case, the testing result showed whether and why the test case failed. Evaluator recorded the result and compared it with mapping relationship graph to find whether the result was consistent with the expected result. The comparison result was classified into two groups: (1) The test result is consistent with the expected result (marked Cons*); (2) The test result is inconsistent with the expected result (marked InCons*).

B. Result and Analysis

TABLE III
Information of Discovered Refactoring

Project	RC	RM	RP	MM	PUM	PDM	CMS
Ant	1	2	0	9	31	0	30
Maven	1	0	2	0	0	0	15
Struts	3	3	56	5	0	0	14
Lucene	0	0	0	3	7	0	18
Spring Framework	0	1	0	2	1	0	49

Information of refactoring mined from the applications by RefactoringCrawler is listed in Table III. Seen from Table III, six kinds of refactoring data were collected. Only PDM was not found in the applications. In addition, we separated two kinds of refactoring: Add Parameter(APM), Remove Parameter(RPM), from CMS. The comparison between test case running result and expected result is listed in Table IV. As we can see from Table IV, the total number of refactoring mined from five applications is 64, 26, 81, 28 and 51 respectively. The corresponding consistent result is 56, 13, 70, 23 and 46. The precision is 87.5%, 81.3%, 86.4%, 82.1% and 90.2% respectively.

[*]Ant. http://ant.apache.org/.
[†]Maven. http://maven.apache.org/.
[‡]Struts. http://struts.apache.org/.
[§]Lucene. http://lucene.apache.org/.
[¶]Springframework. http://www.springsource.org/spring-framework/.

TABLE IV
Comparison between Test Running Results and Expected results

	Project									
	Ant		**Maven**		**Struts**		**Lucene**		**Spring Framework**	
Refactoring	Cons*	InCons*	Cons*	InCons*	Cons*	InCons*	Cons*	InCons*	Cons*	InCons*
RC	1	0	1	0	2	1	0	0	0	0
RM	2	0	0	0	3	0	0	0	1	0
RP	0	0	2	0	50	6	0	0	0	0
MM	8	1	0	0	4	1	3	0	2	0
PUM	28	3	0	0	0	0	6	1	1	0
PDM	0	0	0	0	0	0	0	0	0	0
APM	10	2	4	1	5	1	9	3	25	3
RPM	7	2	6	2	6	2	5	1	17	2
Total	56	8	13	3	70	11	23	5	46	5
Precision	87.5%		81.3%		86.4%		82.1%		90.2%	

Seen from Table IV, different number of refactorings suggest that refactoring emphasis in each project is different. We analyzed the instance where the testing result is inconsistent with the expected result. The analysis results show that two main cases lead to the disagreement between testing result and the expected result: composite refactoring and chain refactoring. (1) Composite refactoring refers that two or more refactorings are applied to the same class or code. For example, in project Ant, mined refactoring shows that method *isFilesystemOnly()* is moved from *org.apache.tools.ant.types.resources.CompressedResource* to *org.apache.tools.ant.types.resources.MappedResourceCollection* . *MappedResourceCollection* is a new class in the latter application version and does not exist before refactoring. The class is created as a result of performing refactoring extract class, while move method is just one step of extract class. Similarly, in the same project, mining result shows that refactoring add parameter is performed on method *doReport(PrintStream)* in *org.apache.tools.ant.Diagnostics*. In fact, the method body is first extracted from *doReport(PrintStream)*. Based on the newly created method, refactoring add parameter is performed. (2) Chain refactoring means that two kinds of refactorings are connected with each other, namely, one refactoring will affect the other refactoring. For example, in project Maven, two methods: *createLocalArtifact(String, String)* and *createArtifact(Artifact, ArtifactRepository)* were in *org.apache.maven.artifact.AbstractArtifactComponent*. Method *createArtifact(Artifact, ArtifactRepository)* was invoked by *createLocalArtifact(String, String)*. In process of refactoring, the two methods were refactored. *createArtifact(Artifact, ArtifactRepository)* was changed to *createArtifact(Artifact, ArtifactRepository, String)* and *createLocalArtifact(String, String)* was changed to *createLocalArtifact(String, String, String)*. The relationship of invocation is not changed, viz. , method *createArtifact(Artifact, ArtifactRepository, String)* is still invoked by *createLocalArtifact(String, String, String)*. In the same way, in project Lucene, two methods: *retrieveInterestingTerms(Reader)* and *retrieveTerms(Reader)* were in *org.apache.lucene.search.similar.MoreLikeThis*. Method *retrieveTerms(Reader)* was invoked by *retrieveInterestingTerms(Reader)*. In process of refactoring, the two methods were refactored. *retrieveTerms(Reader)* was changed to *retrieveTerms(Reader, String)* and *retrieveInterestingTerms(Reader)* was changed to *retrieveInterestingTerms(Reader, String)*. The relationship of invocation is not changed, viz. , method *retrieveTerms(Reader, String)* is still invoked by *retrieveInterestingTerms(Reader, String)*. Due to the composite refactoring and chain refactoring, the effect of refactoring on test case is complicated which makes the testing result is inconsistent with the expected result.

To validate the effectiveness and extensibility of the experimental result to other applications and refactorings, we used statistical methods to analyze the experimental result. We calculated precision of each kind of refactoring (precision = Consistent / (Consistent + Inconsistent)). First, we analyzed whether the experimental result was applicable to other refactorings. Second, we analyzed whether different applications would affect the experimental result. Statistics are presented in Table V$\sim$VIII.

TABLE V
Statistics of Individual Refactoring Accuracy

Group	**N**	**Mean**	**Std. Deviation**	**Std. Error**
1	40	.936425	.0898548	.0142073

Table V shows statistics of individual refactoring accuracy. As showed in Table V, refcatorings mined from five applications are regarded as 40 samples according to different types. The average precision of a single type refactoring is 0.936425. Table VI shows the single sample test of the data. From Table VI, we can find that p = 0.000<0.05. The result shows that there are no significant differences between different type of refactorings. Thereforethe experimental result is applicable to other refactorings.

Table VII shows the basic descriptive statistics of experimental result in each application. The average precision calculated by the experimental result where refactorings are mined from Spring Framework is the highest, 0.973500. The

TABLE VI
Single Samples Test

				95% Confidence Interval for Mean	
t	**df**	**Sig. (2-tailed)**	**Mean Deviation**	**Lower Bound**	**Upper Bound**
65.912	39	.000	.9364250	.06797934	.28196632

TABLE VII
Descriptives

					95% Confidence Intervaln for Mean			
	N	**Mean**	**Std. Deviation**	**Std. Error**	**Lower Bound**	**Upper Bound**	**Minimum**	**Maximum**
1.00	8	.925375	.0881167	.0311540	.851708	.999042	.7780	1.0000
2.00	8	.943750	.1050085	.0371261	.855961	1.031539	.7500	1.0000
3.00	8	.909500	.1044140	.0369159	.822208	.996792	.7500	1.0000
4.00	8	.930000	.1011632	.0357666	.845425	1.014575	.7500	1.0000
5.00	8	.973500	.0490714	.0173494	.932475	1.014525	.8930	1.0000
Total	40	.936425	.0898548	.0142073	.907688	.965162	.7500	1.0000

TABLE VIII
ANOVA

			Sum of Squares	**df**	**Mean Square**	**F**	**Sig.**
Between Groups		(Combined)	.019	4	.005	.547	.702
	Linear	Contrast	.005	1	.005	.643	.428
		Deviation	.013	3	.004	.515	.674
Within Group			.296	35	.008		
Total			.315	39			

average precision calculated by the experimental result where refactorings are mined from Struts is the lowest, 0.909500. From Table VIII, the observation value F = 0.547, p = 0.702>0.05. Result shows that refactoring data collected from different applications have no significant difference in statistics. Therefore, the experimental result is applicable to other applications. Different applications would not affect the experimental result.

C. Threats to Validity

A threat to internal validity is that analysis for each refactoring is based on the description and accompanying examples in [3]. As we know that people have also proposed some other refactorings, but refactorings which were proposed by Fowler have been widely accepted in industry and academe. Therefore, we mainly discussed the 68 refactorings in Fowler's book. Fowler described the refactorings with motivation and mechanic. most of the refactorings are accompanied by examples. But the examples cannot include all the code condition. Refactoring may lead to different influence on the interface according to different code environment. For example, inline method, as described in [3], it only needs to move the method's body into the body of its callers and remove the method which won't affect the interface. But, if the method is also used by other methods outside the class, the interface will be affected. The corresponding test case will be a failure. Therefore, the influence of refactoring should be analyzed according to actual code environment.

Another threat to internal validity is that we regard that the failure of test case is only caused by refactorings but not software errors. In the evaluation, the selected projects are well-known applications in Java. The evaluated versions of projects are stable released and must be fully tested before publication. This can ensure the correctness of the refactorings which would not lead to software errors. Furthermore, evaluators only applied test code of first version to the latter version of source code and did not revise the code. Therefore, we regard that the failure of test case is only caused by refactorings but not software errors.

The last but not least threat to internal validity is that the evaluation was carried out by doctoral students. Although they are familiar with refactoring, their experience in software engineering is not rich enough. However,, in the evaluation, most of the work was accomplished by the tool, which prevents the results from being affected by subjective factors.

A threat to external validity is that different projects may have certain specific characteristics (such as specific refactoring emphases). It is possible that different results could be obtained from different sets of software elements. To reduce the effect, we carried out the evaluation on five applications from different domains, developed by different developers. The size of the projects varies from 24.8KLOC to 234.7KLOC. It helped generalize the conclusions.

Another threat to external validity is that only sever types of refactoring are discussed in the evaluation. However, they are basically subtypes of 2 categories: "moving element" and

”renaming element” in Fig. 3. The refactoring identification tool leads to the limitation. Although, there are likewise some similar tools, only RefactoringCrawler is publicly available and it can only identify seven kinds of refactoring. The simple example test result of statistics analysis shows that the experimental result is applicable to other refactorings. But, if appropriate tools can be obtained, we will further evaluate our theoretical analysis results in the future work.

V. Related Work

Opdyke[1] first proposed refactoring in his doctoral dissertation. He proposed 23 kinds of primitive refactorings and proved that if the refactoring meets the preset conditions, the program behavior would be invariant. Based on [1], Roberts[18] proposed post condition assertions and he proved that in order to ensure the legality of a particular refactoring, it must meet its behavior preserving criteria which include both pre and post conditions. Fowler [3] improved the content of refactoring and proposed 72 types of refactorings. He explained the refactoring with a detailed description and examples which make refactoring is widely adopted and used.

Deursen et al. [12] explored the relationship between unit testing and refactoring. They proposed the notion of test-first refactoring and extended the refactoring guidelines to handle test code adaptation. They proposed that the refactoring can be categorized as compatible, backwards compatible, make backwards compatible, and incompatible. They [13] composed a set of specific test refactorings to improve tests. For each test smell, they gave a solution, using either a potentially specialized variant of an existing refactoring from [3] or one of the dedicated test refactorings.

Counsell et al. [14] evaluated testing taxonomy proposed by Deursen et al. [13]. They developed a refactoring dependency graph for Fowler’s catalogue [3]. After they analyzed the interdependencies of the refactoring categories, they preferred that only 8 refactorings are ”compatible”. The rest refactorings that use ”incompatible” refactoring will affect test case. They [19] considered the composition of refactorings an under-researched area. They have also suggested an extension to the Fowler’s refactoring guidelines to address test code adaptation. They extend this work by analyzing the different paths in the dependency graph and their effect on eradicating the bad smells from the code.

Basit et al. [20] analyzed the effect of refactoring on source code and test code and preferred that the test code is special. They thought that accompany with refactoring, revising test code is special and it is different from revising source code, such as migrating test method, naming test method etc. their analysis showed that current refactoring guidelines cannot take a full guide for user to repair test code. Further studies are needed in order to improve it. They [21] later evaluated extended refactoring guidelines and the experiment result showed the judge the usability and effectiveness of the extended guidelines for Move Method refactoring.

Thies and Bodden [22] proposed a tool, RefaFlex, towards the refactoring of Java programs. The tool uses a dynamic program analysis to log reflective calls during test runs and then uses this information to proactively prevent the programmer from executing refactorings that could otherwise alter the program’s behavior.

Pipka in his work [17] focuses on test-first practice of Extreme Programming (XP). He describes the adaptation of unit tests with respect to the target refactoring prior to the refactoring process called Test-First Refactoring (TFR). Jiau and Chen[11] proposed the concept of test driven refactoring (TDR), which requires that the developer finishes test adaptation before applying refactoring. A tool, Refiner, was proposed to infer refactoring from test code changes. In both cases, the validity of refactoring is confirmed through modified unit tests. These approaches fit well in Extreme Programming and agile modes of development.

Dinh-Trong et al. [23] proposed a scope-based test coverage criterion which is based on refactoring’s scope of impact. An example is illustrated to show how to use the new test coverage criteria for assessing the adequacy of refactoring tests. Kropp and Schwaiger [24] developed a tool which allows the generation of the complete fit test code and test specification based on existing code. The tool also includes automatic refactoring of test data when refactoring production code and vice versa, when changing the fit test specification, it also updates production code accordingly. It reduces the maintenance effort of fit tests in general. Mongiovi [25] proposed a tool, called Safira, to evaluate whether a transformation is behavior preserving. The tool generates a test suite focusing on exercising only the entities impacted by a transformation. Safira was also used in the refactoring context to assist developers in refactoring activities by yielding a test case whenever it detects a behavior change.

Walter and Pietrzak [26] proposed the concept of generalization refactoring test. They proposed an approach to automatically generate test case according to the changes caused by refactoring. Chu et al. [27] proposed a four-phase approach to guide the construction of the test case refactoring for design patterns. They used some well-known design patterns and evaluate its feasibility by means of test coverage. Daniel et al. [28] proposed a white-box approach to repair test script caused by gui refactoring. They used GUI refactorings as a means to encode the evolution of the GUIs and used the recorded refactorings to change the GUI code and to repair test cases.

VI. Conclusion

Software refactoring, as an efficient method to improve software quality, has become one of the key activities for software development and maintenance. Although refactoring requires that the external behaviors should not be changed, it may lead to changes of software interface which would in turn fail the original test cases. To insure the correctness of refactoring, test case should be repaired. Published research has analyzed the influence of refactoring on a test case. But, people only made some foundational studies on the issue such as which refactoring had an influence on regression test case. How test cases are influenced by refactorings and what kind of errors might be introduced remain unknown. What's more, research findings tend to rely on expert's subjective conjecture. They were not supported by empirical evidence.

To this end, in this paper, we proposed an approach to analyze refactorings' impact on regression test cases. On one hand, we analyzed why regression test cases fail. On the other hand, we analyzed the influence of refactorings on software interfaces. Based on the analysis, we built up a mapping between refactorings and test case failure. Such a mapping can be used to guide test case repair automation where test cases are made obsolete by refactorings. The approach was evaluated on five open-source applications. Evaluation results suggest that the precision of the approach is greater than 80%.

Acknowledgments

The work is funded by the National Natural Science Foundation of China (No. 61272169, 61472034, 61370137), Program for New Century Excellent Talents in University (No. NCET-13-0041), Beijing Higher Education Young Elite Teacher Project (No. YETP1183), the 111 Project of Beijing Institute of Technology, and the National Basic Research Program of China No. 2013CB329303.

References

[1] Opdyke WF (1992) Refactoring Object-Oriented Frameworks. Ph.D. thesis, Urbana-Champaign, IL, USA.

[2] Mens T, Tourwé T (2004) A survey of software refactoring. IEEE Trans Softw Eng 30: 126–139.

[3] Fowler M, Beck K, Brant J, Opdyke W, Roberts D (2003) Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 1 edition.

[4] Kim M, Cai D, Kim S (2011) An empirical investigation into the role of api-level refactorings during software evolution. In: Proceedings of the 33rd International Conference on Software Engineering. New York, NY, USA: ACM, ICSE '11, pp. 151–160. doi:10.1145/1985793.1985815.

[5] Kim M, Zimmermann T, Nagappan N (2012) A field study of refactoring challenges and benefits. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. New York, NY, USA: ACM, FSE '12, pp. 50:1–50:11. doi: 10.1145/2393596.2393655.

[6] Fontana FA, Spinelli S (2011) Impact of refactoring on quality code evaluation. In: Proceedings of the 4th Workshop on Refactoring Tools. New York, NY, USA: ACM, WRT '11, pp. 37–40. doi: 10.1145/1984732.1984741.

[7] Bannwart F, Müller P (2006) Changing programs correctly: Refactoring with specifications. In: Proceedings of the 14th International Conference on Formal Methods. Berlin, Heidelberg: Springer-Verlag, FM'06, pp. 492–507. doi:10.1007/11813040.33.

[8] Guerra EM, Fernandes CT (2007) Refactoring test code safely. In: Proceedings of the International Conference on Software Engineering Advances. Washington, DC, USA: IEEE Computer Society, ICSEA '07, pp. 44–. doi:10.1109/ICSEA.2007.57.

[9] Vonken F, Zaidman A (2012) Refactoring with unit testing: A match made in heaven? In: Proceedings of the 2012 19th Working Conference on Reverse Engineering. Washington, DC, USA: IEEE Computer Society, WCRE '12, pp. 29–38. doi:10.1109/WCRE.2012.13.

[10] Kim M, Rachatasumrit N (2012) An empirical investigation into the impact of refactoring on regression testing. In: Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM). Washington, DC, USA: IEEE Computer Society, ICSM '12, pp. 357–366. doi:10.1109/ICSM.2012.6405293.

[11] Jiau HC, Chen JC (2009) Test code differencing for test-driven refactoring automation. SIGSOFT Softw Eng Notes 34: 1–10.

[12] Deursen AV, Moonen L (2002). The video store revisited - thoughts on refactoring and testing.

[13] Deursen AV, Moonen L, Bergh A, Kok G (2001) Refactoring test code. In: Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001. pp. 92–95.

[14] Counsell S, Hierons RM, Najjar R, Loizou G, Hassoun Y (2006) The effectiveness of refactoring, based on a compatibility testing taxonomy and a dependency graph. In: Proceedings of the Testing: Academic & Industrial Conference on Practice And Research Techniques. Washington, DC, USA: IEEE Computer Society, TAIC-PART '06, pp. 181–192. doi:10.1109/TAIC-PART.2006.33.

[15] Apache. URL `http://www.apache.org`.

[16] Sourceforge. URL `http://sourceforge.net`.

[17] Pipka JU (2002) Refactoring in a "test First"-World. In: Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering (XP).

[18] Roberts DB (1999) Practical Analysis for Refactoring. Ph.D. thesis, Champaign, IL, USA. AAI9944985.

[19] Counsell S, Swift S, Hierons RM (2007) A test taxonomy applied to the mechanics of java refactorings. In: Sobh TM, editor, SCSS (1). Springer, pp. 497-502.

[20] Basit W, Lodhi F, Bhatti U (2010) Extending refactoring guidelines to perform client and test code adaptation. In: Sillitti A, Martin A, Wang X, Whitworth E, editors, XP. Springer, volume 48 of *Lecture Notes in Business Information Processing*, pp. 1-13.

[21] Basit W, Lodhi F, Bhatti MU (2012) Evaluating the extended refactoring guidelines. In: Proceedings of the 2012 IEEE 36th Annual Computer Software and Applications Conference Workshops. Washington, DC, USA: IEEE Computer Society, COMPSACW '12, pp. 260–265. doi: 10.1109/COMPSACW.2012.55.

[22] Thies A, Bodden E (2012) Refaflex: Safer refactorings for reflective java programs. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis. New York, NY, USA: ACM, ISSTA 2012, pp. 1–11.

[23] Dinh-Trong T, Geppert B, Li JJ, Roessler F (2008) Looking for more confidence in refactoring? how to assess adequacy of your refactoring tests. In: Proceedings of the 2008 The Eighth International Conference on Quality Software. Washington, DC, USA: IEEE Computer Society, QSIC '08, pp. 255–263. doi:10.1109/QSIC.2008.49.

[24] Kropp M, Schwaiger W (2009) Reverse generation and refactoring of fit acceptance tests for legacy code. In: Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications. New York, NY, USA: ACM, OOPSLA '09, pp. 659–664. doi:10.1145/1639950.1639961.

[25] Mongiovi M (2011) Safira: A tool for evaluating behavior preservation. In: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion. New York, NY, USA: ACM, OOPSLA '11, pp. 213–214. doi:10.1145/2048147.2048213.

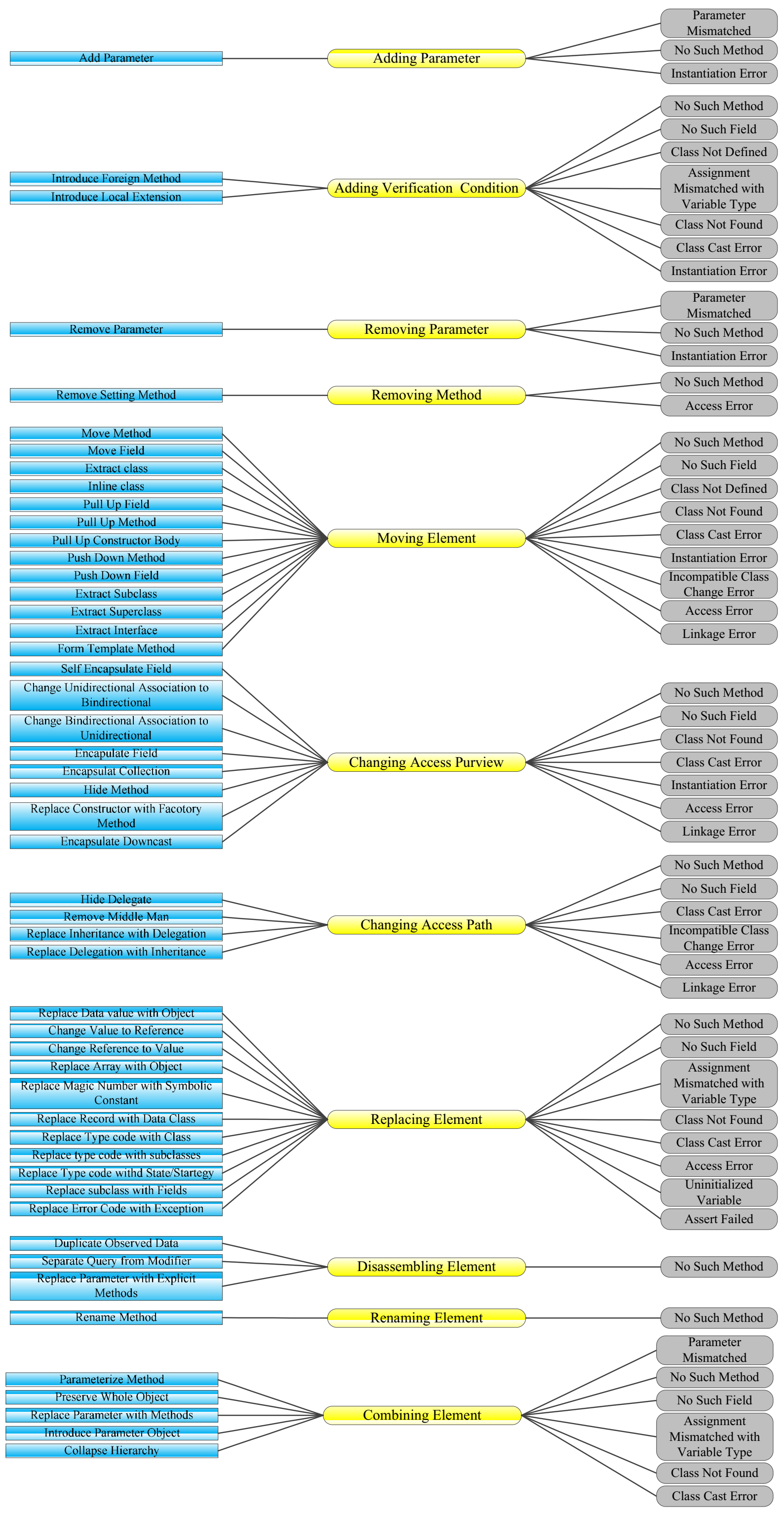
[26] Walter B, Pietrzak B (2004) Automated generation of unit tests for refactoring. In: Eckstein J, Baumeister H, editors, XP. Springer, volume 3092 of *Lecture Notes in Computer Science*, pp. 211-214.

[27] Chu PH, Hsueh NL, Chen HH, Liu CH (2012) A test case refactoring approach for pattern-based software development. Software Quality Control 20: 43–75.

[28] Daniel B, Luo Q, Mirzaaghaei M, Dig D, Marinov D, et al. (2011) Automated gui refactoring and test script repair. In: Proceedings of the First International Workshop on End-to-End Test Script Engineering. New York, NY, USA: ACM, ETSE '11, pp. 38–41. doi: 10.1145/2002931.2002937.

Appendix

Mapping Relationship between Refactoring and Test Case Failure


Add Parameter
Adding Parameter
Parameter Mismatched
No Such Method
Instantiation Error
Introduce Foreign Method
Introduce Local Extension
Adding Verification Condition
No Such Method
No Such Field
Class Not Defined
Assignment Mismatched with Variable Type
Class Not Found
Class Cast Error
Instantiation Error
Remove Parameter
Removing Parameter
Parameter Mismatched
No Such Method
Instantiation Error
Remove Setting Method
Removing Method
No Such Method
Access Error
Move Method
Move Field
Extract class
Inline class
Pull Up Field
Pull Up Method
Pull Up Constructor Body
Push Down Method
Push Down Field
Extract Subclass
Extract Superclass
Extract Interface
Form Template Method
Moving Element
No Such Method
No Such Field
Class Not Defined
Class Not Found
Class Cast Error
Instantiation Error
Incompatible Class Change Error
Access Error
Linkage Error
Self Encapsulate Field
Change Unidirectional Association to Bindirectional
Change Bindirectional Association to Unidirectional
Encapulate Field
Encapsulat Collection
Hide Method
Replace Constructor with Facotory Method
Encapsulate Downcast
Changing Access Purview
No Such Method
No Such Field
Class Not Found
Class Cast Error
Instantiation Error
Access Error
Linkage Error
Hide Delegate
Remove Middle Man
Replace Inheritance with Delegation
Replace Delegation with Inheritance
Changing Access Path
No Such Method
No Such Field
Class Cast Error
Incompatible Class Change Error
Access Error
Linkage Error
Replace Data value with Object
Change Value to Reference
Change Reference to Value
Replace Array with Object
Replace Magic Number with Symbolic Constant
Replace Record with Data Class
Replace Type code with Class
Replace type code with subclasses
Replace Type code withd State/Startegy
Replace subclass with Fields
Replace Error Code with Exception
Replacing Element
No Such Method
No Such Field
Assignment Mismatched with Variable Type
Class Not Found
Class Cast Error
Access Error
Uninitialized Variable
Assert Failed
Duplicate Observed Data
Separate Query from Modifier
Replace Parameter with Explicit Methods
Disassembling Element
No Such Method
Rename Method
Renaming Element
No Such Method
Parameterize Method
Preserve Whole Object
Replace Parameter with Methods
Introduce Parameter Object
Collapse Hierarchy
Combining Element
Parameter Mismatched
No Such Method
No Such Field
Assignment Mismatched with Variable Type
Class Not Found
Class Cast Error