Empirical Evaluation of Syntactic and Semantic Defects Introduced by Refactoring Support

Wafa Basit[1], Fakhar Lodhi[2], and Usman Bhatti[3]
[1]Department of Computer Science, National University of Computer and Emerging Sciences, Pakistan
[2]Department of Computer Science, GIFT University, Pakistan
[3]Rmod Team, Inria Lille-Nord Europe, France

Abstract: *Software maintenance is a major source of expense in software projects. A proper evolution process is a critical ingredient in the cost-efficient development of high-quality software. A special case of software evolution is refactoring that cannot change the external behaviour of the software system yet should improve the internal structure of the code. Hence, there is always a need to verify after refactoring, whether it preserved behaviour or not. As formal approaches are hard to employ, unit tests are considered the only safety net available after refactoring. Refactoring may change the expected interface of the software therefore unit tests are also affected. The existing tools for refactoring do not adequately support unit test adaptation. Also, refactoring tools and guidelines may introduce semantic and syntactic errors in the code. This paper qualitatively and quantitatively analyses data from an empirical investigation involving 40 graduate students, performed against a set of semantic and syntactic defects. Findings from the expert survey on refactoring support have also been shared. The analysis in this paper shows that there are notable discrepancies between preferred and actual definitions of refactoring. However, continued research efforts are essential to provide Guide Lines (GL) in the adaptation of the refactoring process to take care of these discrepancies, thus improving the quality and efficiency of the software development.*

Keywords: *Refactoring, unit testing, pre-conditions, semantic defects, maintenance.*

Received June 2, 2013; accepted March 29, 2014; published online March 8, 2015

1. Introduction

Software refactoring as a concept has been widely accepted in the industry and is considered a practice that can provide significant improvements in software quality. However, during the entire history of software development, automated refactoring techniques have not been able to fully evolve to provide complete and reliable support in all software development paradigms. There are multiple reasons for the lack of use of refactoring support. Some being deficient usability, efficiency and reliability. The effects of refactoring have to be analyzed to access the program locations that require change. But, if the preconditions are not adequately evaluated and the required adaptations are not performed, refactoring not only negatively affects the artifacts from the phases in the development life cycle but also affects clients in the production code. Nonetheless, many times the unit tests which are specialized clients [3, 4] in the context of refactoring are crippled due to the refactoring process, leaving the developer with no way to verify the behavior preservation after refactoring. The research question answered in this paper is: Are the professionals, academics and students satisfied with certain features of refactoring support they are using and what are the obstacles for software development practitioners? This paper covers the following:

1. Evaluation of three commonly used refactoring including Move Method (MM), Pull Up Method (PUM) and Push Down Method (PDM) provided by Eclipse, Netbeans, IntellijIDEA and JBuilder [14, 15, 19, 31].
2. Empirical investigation of fowler's refactoring guidelines involving 40 graduate students.
3. Discussion on the results from our expert survey on refactoring support.

This study shall assist in pointing out directions for future research within software refactoring and unit testing.

2. Related Works

The outcome of a refactoring process should be preserved behaviour, improved software quality and consistency between the refectories code and other software artefacts including documentation, design documents, tests, etc., [24]. However, in practice in addition to breaking the production code, the evolution of the other artefacts is generally not taken into account. Unit testing is a fundamental component of the refactoring process. Fowler [16] is of the view that every class should have a main function that tests the class or separate test classes should be built that work in a framework to make testing easier, which implies that test code cannot be separated from the production code. Therefore, any process affecting the production code should readily adapt the associated clients and the test code [11, 12, 23]. Zaidman *et al.* [32] are also of the view that there is a need for tools and methods that

can help the co-evolution of source and test code. In our earlier work [1, 2, 3, 4, 5, 17, 22] the state of art and practice that addresses or should address client and unit test adaptation while refactoring has been discussed in detail the. It has been summarized in Table 1.

Table 1. Summary of the work related to client and test code adaptation after refactoring.

Researcher	Research Summary
Fowler [16]	• A widely adopted extensive catalog of 68 refactoring guidelines. • Informal and inconsistent level of detail. • Do not provide guidelines on the adaptation of unit tests. In most cases, steps on client adaptation are also missing.
Deursen *et al.* [11]	Presented a test taxonomy that categorizes refactoring based on their effect on test code. These are: compatible, backwards compatible, make backwards compatible and incompatible.
Counsell *et al.* [6, 7, 8]	• Assessed the test taxonomy presented in [11]. • In our previous work [3] it has been shown that the categorization used by [6-8] has various loop holes. • A refactoring dependency graph is developed for Fowler's catalogue [16] and a shorter list of compatible refactoring is suggested by excluding all the other refactoring that may use those refactoring that break unit tests. • This approach essentially rejects the use of many important refactoring that are necessary for improving the program structure.
Jiau *et al.* [20, 28]	• Test Driven Refactoring (TDR) [20] and Test-first Refactoring (TFR) [28] involve adaptation of associated unit tests before the refactoring process takes place. • These approaches fit well in Extreme Programming paradigm but are not general enough to be used in all development environments where testing first is not always possible. • Do not provide guidelines to adapt test code according to the targeted refactoring.
Soares *et al.* [29]	• Soares *et al.* [29] propose a technique for generating a set of unit tests that can be useful for detecting semantic errors after a sequence of object-oriented program refactoring. • They have also evaluated the refactoring support provided by Eclipse, IntelliJ IDEA, JBuilder, NetBeans. They observe that program refactoring in IDEs are commonly implemented in an ad hoc way and the semantic aspects of behavior are several times not preserved.
Basit *et al.* [1, 2, 3, 4, 5, 22]	In [3] a mutually exclusive categorization of refactoring guidelines has been presented based on the impact of refactoring on clients and unit tests. In [2] the problems with Fowler's refactoring catalog and java refactoring tools including Net Beans, Eclipse, Intellij IDEA and JBuilder have been discussed. These tools introduce semantic errors in the refectories code. It has also been shown that the quality of the unit tests is also deteriorated if existing approaches for refactoring are used. In order to prove the effectiveness of extended refactoring guidelines, the results from an experiment have also been shared. In [1] the extended refactoring guidelines for pull up method have been presented. The semantic issues that can be introduced due to this refactoring have been discussed through examples. Test Adaptation Plug-in for Eclipse (TAPE) [22] makes easier for the developer to organize the unit tests along the changes in the refectories code. In [5] it has been demonstrated with the help of various examples that unit test is a specialized client in the context of refactoring. The formal specification of the extended refactoring guidelines and an adaptation framework is presented in [4].
Daniel *et al.* [9]	Proposed an approach to check whether refactoring tools introduce compilation errors or not. This work ignores detection of semantic errors that could be introduced through existing refactoring tools.
TestCareAssistant [26]	This tool is implemented as a Java prototype that provides automated guidance to developers for repairing test compilation errors caused due to changes such as adding, removing or changing types of parameters and return values of methods.
ReAssert [10]	Reassert repairs assertions in test code by traversing the failure trace. It performs dynamic and static analysis to suggest repairs to developers. Again this tool does not help in fixing the semantic errors introduced through refactoring
CatchUp! [18]	Catch-up! adapts clients of the evolving Application Programming Interfaces (API's). This tool takes care of only compilation errors that can appear in the clients due to a subset of refactoring performed on any API, and therefore ignore the semantic errors that could be caused due to refactoring process.
Kaba [28]	KABA also includes all clients in the refactoring process. It guarantees preservation of behaviour for the clients either through static analysis or all test runs (dynamic analysis) for any input.
Reba [13]	ReBA instead of adapting the clients of the evolving API, creates compatibility layers between new library APIs and old clients .This layer is created in the form of an adapted version that supports both versions of the API.

3. Comparative Analysis of Refactoring Tools and Guidelines

Before code transformation takes place, an early check should be performed to evaluate preconditions of the refactoring; if all preconditions are satisfied, refactoring mechanics are executed. These mechanics include Guide Lines (GL) for restructuring and corrective transformations required to preserve externally observable behaviour of the program. However, there are gaps between the definitions and the actual implementation of refactoring.

Fowler's refactoring catalog [16] is widely used both in the industry and academia for training and pedagogical purposes. Based on Fowler's GL many refactoring tools for Java have been developed including the most commonly used: Eclipse, IntelliJ IDEA, JBuilder and NetBeans etc., [14, 15, 19, 31]. These tools do not completely address the issues related to behaviour preservation including client/test adaptation. One of the reasons is lack of refactoring GL that include all necessary pre and post conditions for the preservation of program behaviour [3]. Fowler's refactoring GL (FGL) give a good starting point to developers in order to re-factor a program but lack focus on various aspects of behaviour preservation. Using these GL [16] refactoring process can not only flag semantic but also syntactic violations in the code. Syntactic errors are cheap to fix but identification and resolution of semantic violations is hectic as well as expensive [5]. The missing detail in the refactoring GL is also inherited by various refactoring tools for Java [5]. Therefore, Fowler's catalog is also included in this analysis. Because, with the help of extended refactoring GL, existing tools can be improved and new tools can be developed that preserve program behaviour. In this section, a few semantic as well as syntactic defects that are introduced by refactoring tools in various locations throughout the software system have been defined and demonstrated.

Three commonly used refactoring: PUM, PDM and MM [16] have been discussed. PUM refactoring is used by software designers to generalize and eliminate the duplicate code present in subclasses, as a result the cohesion of the super class is increased and the coupling of the subclasses is reduced. Similarly, PDM increases cohesion of the subclasses by specializing certain behaviour of a super class that is relevant only for some of its subclasses. MM is used for moving features between classes such that coupling in the system is loosened [16]. These refactoring are most commonly used refactoring. The automated support for these refactoring is provided by almost all commonly used Java refactoring tools [14, 15, 19, 31]. But, the precondition evaluation and adaptation is implemented in an Ad-Hoc way ignoring many aspects of behaviour preservation. Table 2 highlights the semantic as well as syntactic errors that could appear in the refactored code by performing these refactoring either manually through GL or with the use of refactoring tools.

Table 2. Comparative analysis of refactoring tools and GL.

		Tools				GL
Preconditions		**Eclipse**	**NetBeans**	**IntelliJ**	**JBuilder**	**FGL**
Invalid Access to Super Object	**PUM**	×	×	☑	☑	×
	MM	×	N/A	×	⦸	×
	PDM	×	×	☑	☑	×
Overriding Wrongly Enabled	**PUM**	×	×	×	⦸	×
	MM	×	N/A	⦸	⦸	×
	PDM	×	×	×	⦸	×
Invalid Access to Duplicate Variable	**PUM**	⦸	⦸	⦸	⦸	×
	MM	×	N/A	⦸	⦸	×
	PDM	⦸	⦸	⦸	⦸	×
Constructor	**PUM**	×	×	☑	☑	×
	MM	×	N/A	☑	☑	×
	PDM		×	☑	☑	×
Main Method	**PUM**	×	×	☑	⦸	×
	MM	×	N/A	×	⦸	×
	PDM		×	☑	⦸	×
Checks Due to Static Import	**PUM**	×	×	×	×	×
	MM	×	N/A	×	×	×
	PDM	×	×	×	×	×
Relocating Test Code	**PUM**	×	×	×	×	×
	MM	×	N/A	×	×	×
	PDM	×	×	×	×	×
Appropriate Replacement of Calls to Candidate Method in Clients/ Unit Tests	**PUM**	×	×	×	×	×
	MM	×	N/A	×	×	×
	PDM	×	×	×	×	×

⦸ means the system warns the developer but allows to continue.
× means the system does not recognize the issue.
☑ means the system recognizes the issue and handles it appropriately.

3.1. Invalid Access to Super Object

In object oriented programming, access to parent class's methods is a common practice. The super object in Java gives child classes access to parent's overridden constructors and methods. But movement of methods up and above in the inheritance hierarchy can create a semantic violation which will not be flagged by the compiler. For example, if a refactoring is performed using Fowler's GL, where method foo is implemented in class A and class B, in each class the method foo returns a different value. Before refactoring, the clients of method m get return value equal to '23' (by calling method in class B) but, after refactoring the same method returns '43'(by calling method in class A). This simple scenario highlights the fact that by ignoring the context sensitive constructs like Super, semantic errors can be injected in the production and test code.

Table 3. Behaviour altered due to invalid access to super object.

	Before Refactoring	**After Refactoring**

MM	Class A{ int foo() { return 43; }}
	Class B { int foo(){ return 23; }}
	Class C extends B{ int m(){ super.foo(); }}
	Class D extends A{ int n(){
	super.foo()+1; }}
	// n() returns 44
		Class A{ int foo(){ return 43; }}
		Class B { int foo(){ return 23; }}
		Class C extends B{ int m(){ super.foo();}
		int n(){ super.foo()+1; }}
		Class D extends A{ }
		// n() returns 24

PUM	Class A{ int foo(){ return 43; }} Class B extends A{ int foo(){ return 23; }} Class C extends B{ int n(){ super.foo(); }} // n() returns 23	class A{ int foo(){ return 43; }} class B extends A{ int foo(){ return 23; }int n(){ super.foo(); }} class C extends B{} // n() returns 43

PDM	Class A{ int foo(){ return 43; } }
	Class B extends A{ int foo(){ return 23; }
	int n(){ super.foo(); }}
	Class C extends B{}
	// n() returns 43
		Class A{ int foo(){ return 43; }}
		Class B extends A{ int foo(){ return 23; }
		}
		Class C extends B{ int n(){ super.foo(); }}
		// n() returns 23

Also, if the calls to super are not replaced by appropriate method calls, the program shall not compile, specifically when the call to a non existing method is made using super object in the pulled up method. The refactoring support provided by commonly used tools including NetBeans [31] and Eclipse [14] do not take care of this aspect while refactoring. Similarly MM and PDM can also introduce semantic and syntactic errors if the preconditions related to presence of calls to Super object are not evaluated as shown in Table 3 above.

3.2. False Overriding

The code below highlights a problem of false overriding that can be caused if the inheritance hierarchy of the target class for candidate method is not completely analyzed for uniqueness. Generally, the tools check for the uniqueness of the method in the target class but, do not take into account its ancestor classes. The movement of the method may result in false overriding if any of the ancestors has a method with the same signature as that of the candidate method. The scenarios for three refactoring are given in Table 4.

Table 4. Scenarios showing invalid overriding after refactoring.

	Before Refactoring	**After Refactoring**

MM	Class Calculator{
	int Add(int a,int b){ return a+b; }
	int multiply(int a,int b){ return a*b; }}
	Class Pay { int multiply(int p,int s){ return 2* p *s;}}
	Class Pay_Calc extends Calculator{ }
	Class Client{
	int CalculatePay(){
	Calculator c= new Pay_Calc();
	return c.multiply(2000,5);
	}}
	// CalculatePay() returns 10000
		Class Calculator{
		int Add(int a,int b){ return a+b; }
		int multiply(int a,int b){ return a*b; }}
		Class Pay { }
		Class Pay_Calc extends Calculator{
		int multiply(int p,int s){ return 2* p *s; }}
		Class Client{
		int CalculatePay(){
		Calculator c= new Pay_Calc();
		return c.multiply(2000,5); }}
		// CalculatePay() returns 20000

PUM	Class Employee{ int getPay(){ return 43; }}
	Class Regular extends Employee{
	int getPay(int grd){ return grd *23; }
	}
	Class Engineer extends Regular{
	int getPay(){ return 60; }}
	Class Client{
	int CalculatePay(){
	Employee emp=new Regular();
	return emp.getPay();
	}} // CalculatePay() returns 43
		Class Employee{ int getPay(){ return 43; }
		}
		Class Regular extends Employee{
		int getPay(int grd){ return grd *23; }
		int getPay(){ return 60; }
		}
		Class Engineer extends Regular{ }
		Class Client{
		int CalculatePay(){
		Employee emp=new Regular();
		return emp.getPay();
		}} // CalculatePay() returns 60

PDM	Class Employee{ int getPay(){ return 43; } }
	Class Regular extends Employee{
	int getPay(int grd){ return grd *23; }
	int getPay(){ return 60; }}
	Class Engineer extends Regular{ }
	Class Client{
	int CalculatePay(){
	Employee emp=new Regular();
	return emp.getPay();
	}} // CalculatePay() returns 60
		Class Employee{ int getPay(){ return 43; } }
		Class Regular extends Employee {
		int getPay(int grd){ return grd *23;}}
		Class Engineer extends Regular{
		int getPay(){ return 60; }}
		Class Client{
		int CalculatePay(){
		Employee emp=new Regular();
		return emp.getPay();
		}} // CalculatePay() returns 43

3.3. Invalid Access to a Duplicate Field

Generally refactoring tools check for the uniqueness of the candidate method in the target class prior to refactoring. The fields from the source class are either accessed through source object in the target class or are moved to the target class after the method is moved. But, in case of duplicate fields both in the source and

target, developer may not get a compiler error, resulting in invalid software behaviour. In Table 5, before and after states of the refactored code have been demonstrated each showing different results after refactoring.

Table 5. Scenarios showing invalid access to a duplicate field in the target class.

	Before Refactoring	After Refactoring

MM	Class Comparator{
	int adjust=2;
	int getMax(int a,int b){ return Math.max(a,b) + adjust ;}}
	Class Adder{
	int adjust=1;
	int add(int a,int b){ return a+b + adjust;}
	int getMin(int a,int b){ return Math.min(a,b)-adjust;}
	} // getMin(2,4) returns 1
		Class Comparator{
		int adjust=2;
		int getMax(int a,int b){ return Math.max(a,b) + adjust; }
		int getMin(int a,int b){ return Math.min(a,b)-adjust;}}
		Class Adder{
		int adjust=1;
		int add(int a,int b){ return a+b + adjust;
		}
		} // getMin(2,4) returns 0

PUM	Class Parent{
	int attribute=1;
	int pmethod(){ return attribute; }}
	Class Child extends Parent{
	int attribute=2;
	int cmethod(){ return attribute +1; }
	} // cmethod() returns 3
		Class Parent{
		int attribute=1;
		int pmethod(){ return attribute; }
		int cmethod(){ return attribute +1; }}
		Class Child extends Parent{
		int attribute=2;
		} // cmethod() returns 2

PDM	Class Parent{
	int attribute=1;
	int pmethod(){ return attribute; }
	int cmethod(){ return attribute +1; }}
	Class Child extends Parent{
	int attribute=2;}
	// cmethod() returns 2
		Class Parent{
		int attribute=1;
		int pmethod(){ return attribute; }}
		Class Child extends Parent{
		int attribute=2;
		int cmethod(){ return attribute +1; }}
		// cmethod() returns 3

3.4. Static Import

Static import is a feature provided by Java Development Kit (JDK) which allows unqualified access to static members without inheriting from the type containing the static members. Refactoring can introduce semantic errors in the code if this feature has been used as demonstrated in the following examples in Table 6. Before PUM refactoring, the method *getEmployeeN*o returns "4", whereas after *valueOf* method is pulled up to the Employee class the method returns '5', clearly a semantic error. This happens because the *getEmployeeNo* method in the Employee class was making an unqualified access to *valueOf* method of *Java.lang.String* but after a method with the same name is pulled up to Employee class; *getEmployeeNo* method calls the local *valueOf* method which returns "5". Similarly, MM and PDM refactoring in these scenarios alter the externally observable software behaviour.

Table 6. Scenarios showing invalid use of function when using Static Import.

	Before Refactoring	After Refactoring

MM	class Employee{
	static String getEmployeeNo(int i) {
	return valueOf(i) ; }}
	class Pay { static String valueOf(int pay) { return pay * 3;}}
	// getEmployeeNo(4) returns 4
		class Employee{
		static String getEmployeeNo(int i) {
		return valueOf(i) ;}
		static String valueOf(int pay) {return pay * 3;}}
		class Pay {} // getEmployeeNo(4) returns 12

PUM	class Employee{
	static String getEmployeeNo(int i) {
	return valueOf(i) ;}}
	class RegularEmployee extends Employee{
	static String valueOf(int i) { return i +1; }
	} // getEmployeeNo(4) returns 4
		class Employee{
		static String getEmployeeNo(int i) { return valueOf(i) ;}
		static String valueOf(int i) { return i +1;}}
		class RegularEmployee extends Employee{}
		// getEmployeeNo(4) returns 5

PDM	class Employee{
	static String getEmployeeNo(int i) {
	return valueOf(i) ; }
	static String valueOf(int i) { return i +1;}}
	class RegularEmployee extends Employee{
	} // getEmployeeNo(4) returns 5
		class Employee{
		static String getEmployeeNo(int i) {
		return valueOf(i) ; }}
		class RegularEmployee extends Employee{
		static String valueOf(int i) { return i +1; }
		} // getEmployeeNo(4) returns 4

3.5. Main Method

A method is generally moved to another class if it is the right home for it. For example, *FixEngine*() logically belongs to the *Engine* class, if it is not there it should be moved. Whereas, a main method is an entry point to the software system, it does not exhibit the behaviour of its owner class. Moving a main method can lead to runtime exception where the IDE may not be able to detect the entry point unless altered by the developer.

3.6. Constructor

A constructor is a specialized method that initializes the data members of the class, creates an object and has the same name as that of the class. If a method is moved to any other class, it loses its meaning; it becomes an ordinary method that initializes a set of variables. It is not principally correct to move, pull up or push down a constructor. It can be seen in Table 2 that a few tools allow such operations on constructor, resulting in invalid or broken clients or unit tests.

3.7. Incorrect Replacement of Calls

JBuilder and IntelliJ IDEA [15, 19] in few cases syntactically adapt the clients and unit tests after refactoring, such that externally observable behaviour is preserved but, the overall quality of the system deteriorates. For instance, the purpose of MM is to decrease the coupling and increase cohesion of the overall system including source and target classes. But, these refactoring tools for Java instead of removing the association of source with client use the target class's object in the source class to call the moved method. Which actually keep the clients coupled with the source, even after the refactoring. In this manner externally observable behaviour is preserved but this is definitely not improvement in the internal structure of the software system. In order to perform MM using these tools, the target object is either sent as a parameter of the candidate method or in the other case target class's object is declared in the source class. Here, the consequences of the later scenario are discussed.

Ideally, along the movement of method to the target, its corresponding test code should also be moved to the target's test class, by doing so, the association between the target and test source can be removed. None of the tools including JBuilder, Eclipse and IntelliJ IDEA take care of these aspects of refactoring (NetBeans does not support MM). As a consequence the Coupling Between Objects (CBO) is increased. This observation can be proven through the use of *CBO* metric.

$$CBO = \frac{Numberoflinks}{Numberofclasses} \quad (1)$$

By putting values in Equation 1 from Figure 4, with N number of clients of source class and assuming one test class for source and target CBO becomes.

$$CBO = \frac{(4+2N)}{(4+N)} > \frac{(4+N)}{(4+N)} > \frac{(3+N)}{(4+N)} \geq \frac{(3+N)}{(4+N)} \quad (2)$$

By putting values in Equation 2 with $N=2$, number of clients for source class and assuming one test class for source and target each, Equation 2 takes form of Equation 3.

$$CBO = \frac{8}{6} > \frac{6}{6} > \frac{5}{6} \geq \frac{5}{6} \quad (3)$$

Looking at the Equation 3 it can be seen that refactoring tools lead to the worst *CBO*. Fowler's approach is better but as it does not take into account the test code restructuring phenomenon, association between target and source test is created resulting in increased coupling. Last but not the least it is suggested that the code refactoring should be followed by test restructuring also. Here, it is important to note that the intent of MM refactoring is to reduce the CBO in the overall software system. On the contrary coupling is increased instead of decreasing or remaining stable if existing tools are used. Also, by using Fowler's GL for MM the *CBO* metric value increases because it misses steps related to test code restructuring due to which the target is made wrongly associated to source test as shown in Figure 1.

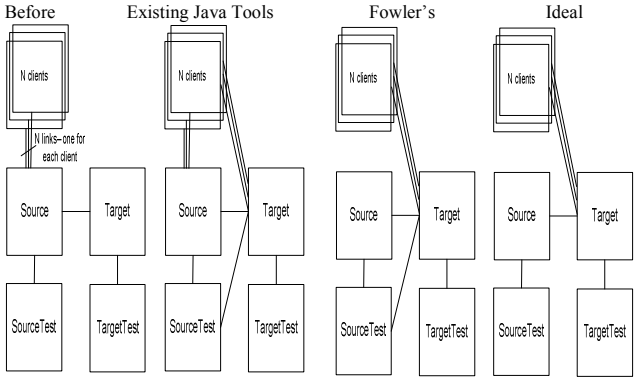


Figure 1. A comparative view of a subsystem after MM refactoring using different approaches.

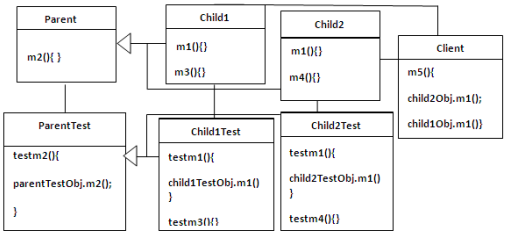
Additionally, in the case of PUM refactoring if the child objects are not replaced by the parent object in the clients, the violation of good design principles would occur resulting in unnecessary association and therefore high coupling. However, with PDM, incorrect replacement of parent object may end up in parent accessing the children in the client classes, which is not a valid relationship.

3.8. Relocating Test Code

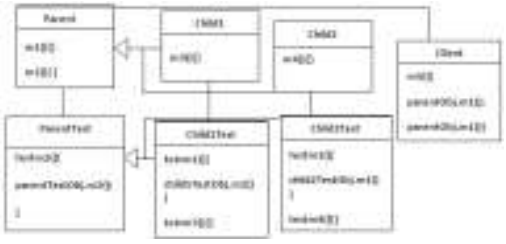
Refactoring and unit tests go together. Whether, a refactoring was behaviour preserving is usually confirmed through the use of unit tests.

However, in practice it is not always possible, because there are certain refactoring that change the software interface and therefore clients accessing this interface are affected including the unit tests [21, 22, 30]. The existing literature and the refactoring support do not differentiate between an ordinary client and the unit tests.

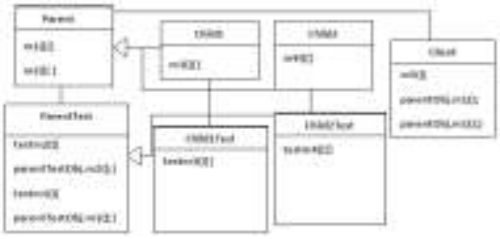
Consequently, the additional adaptations required by unit tests after refactoring are not supported by the refactoring tools. As can be seen in Figure 2-b, after method *m*1 is pulled up to the parent class, the test methods for method *m*1 remains in the child classes resulting in *duplicated test code*. Similarly, when a method is moved from a source to target class, and the test code is not moved from source's test to the target's test, this leads to *indirect testing*. Also, in case of PDM, when the test code is not moved to its right home it leads to an invalid association between the child classes and the parent's test class.



a) Software system before PUM refactoring.



b) Software system after PUM refactoring (Existing approaches).



c) Software system after PUM refactoring (improved GL).

Figure 2. comparison of refactoring approaches applied on *m1*().

Therefore, every step in refactoring should be succeeded by appropriate adaptation in the test code to avoid deterioration of test code quality resulting in test smells.

4. Empirical Evaluation

In this section, results from a controlled experiment involving 40 graduate students have been reported. This experiment was performed to judge the loop holes in Fowler's refactoring GL. According to Fowler, refactoring are based on "human intuition" [16], but in reality every human is different and so is everyone's intuition. Every phase of refactoring starting from detection of bad smells to actual refactoring process requires rigorous support and GL. As discussed in earlier sections, in spite of available automated support and an extensive catalog of refactoring, mistakes can still occur.

4.1. Experiment Settings

In order to prove our claim and check the effectiveness of the extended GL, an experiment involving 40 graduate students was conducted with a major in software engineering. It was a prerequisite to have adequate knowledge of Java language to participate in the exercise. 55% students had industrial experience above 3 years, 45% students had experience ranging from 0-2 years. 10 students were without any industrial development experience but they were included in our experiment because these students had done their graduate level final year projects using Java and they had the expertise required to perform this exercise. Prior to the experiment the students had attended 3 lectures on refactoring as a part of their *Advanced Software Engineering* course. A four hours session was conducted to analyze the student's capability to re-factor using Fowler's GL. The code for a *Pay Roll System* was used for refactoring experiment. There were 16 classes in total in this system, 8 classes in the production code and a parallel hierarchy of 8 test classes developed in JUnit. The size metrics are given in Table 7.

Table 7. Size metrics for pay roll system.

	Lines of Code	Methods	Classes
Production Code	402	19	8
Test Code	241	12	8
Complete Code	643	31	16

The code was shared with the students two days prior to the experiment. The experiment was conducted in a four hours session. In the first hour, students were explained the domain and the task they had to perform. Each step of the GL was also explained to ensure that all students understand every step. The next three hours were divided into two equal halves. They were given two problems one by one. The problems were of average complexity that students could understand in the given amount of time. The defects had been distributed in different locations, so that the identification of one does not affect the other. Students were required to refer to every step of refactoring with the appropriate step number in the Fowler's GL in their solutions. If they did any step without using the GL, they explicitly specified. They were provided with the unit tests along production code.

4.2. Hypotheses

In this paper results for MM, PUM and PDM refactoring have been shared. The hypotheses of this study are based on the following proposition:

- P: There is no need to augment Fowler's GL.
- P': Fowler's GL should be extended. Find below the alternate hypotheses.
- $H0_1$: Core refactoring cannot be performed successfully by most students with existing refactoring GL.
- $H1_1$: Semantic errors shall be introduced by most students if the existing refactoring GL are used.
- $H2_1$: Test code restructuring cannot be done successfully by most students using existing refactoring GL.
- $H3_1$: Client adaptation cannot be done successfully by most students using existing refactoring GL.

Alternate hypotheses for failure rate greater or equal to 50% are accepted. It can be inferred from the results as shown in Table 8 that in most cases the developers were not able to detect the semantic errors introduced due to refactoring. In spite of the small sized project (only 402 lines of code) and limited time. All students performed the core refactoring steps correctly. But most of the subjects did not check the preconditions required for avoiding semantic defects and also did not properly adapted unit tests. The results show that, steps on prevention of semantic defects and test code restructuring should be added to existing refactoring GL.

Table 8. Results from the refactoring experiment.

	Tasks	Success			Failure			Failure Rate (%age)			Hypotheses		
		MM	PUM	PDM	MM	PUM	PDM	MM	PUM	PDM	MM	PUM	PDM
$H0_1$	Core Refactoring (**CR**)	40	40	40	0	0	0	0	0	0	R	R	R
$H1_1$	Replaced Super Object with Appropriate Code (**RSO**)	18	15	10	22	25	30	55	52.5	75	A	A	A
	Overriding not Wrongly Enabled (**OE**)	1	3	7	39	37	33	97.5	92.5	82.5	A	A	A
	Did not Access the Duplicate Variable (**DV**)	25	10	15	15	30	25	37.5	75	62.5	R	A	A
$H2_1$	Relocating Test Code (**RTC**)	15	13	17	25	27	23	62.5	67.5	57.5	A	A	A
	Test Code Adaptation (**TCA**)	20	19	15	20	21	25	50	52.5	62.5	A	A	A
$H3_1$	Appropriate Replacement of Method Calls in Clients (**RC**)	24	5	30	16	35	10	40	87.5	25	R	A	R

4.3. Threats to Validity

As in any empirical study, the external validity of this experiment is limited by the choice of study subject. *Pay Roll System* was used because it was a small sized system, its source and test code were available and it had quite a number of opportunities for refactoring. Similarly, the subjects of the case study are heterogeneous (students with no industrial experience and students with 3-6 years of professional experiment).

This could confound the findings, as for example students with no experience may behave very different from industrial developers. As regarding to internal validity one has to be aware that with a single case study it is not possible to infer whether or not Fowler's GL should be augmented. However, additional

research in larger and different contexts is needed to ensure that our results are indeed true.

5. An Expert Survey on Practices Associated with Refactoring and Unit Testing

In order to, get the expert opinions on our findings regarding refactoring tools a controlled survey was conducted. There were in total 28 questions, distributed into three categories: Demographics (6), refactoring (15) and unit testing (7). A final question asked for feedback on the survey. Before giving it to respondents, the questionnaire was validated by 4 experts and a few amendments were made based on their suggestions. The survey was designed as a cross-sectional and controlled survey. To ensure the integrity of the results, participation tokens were created for selected individuals. They belonged to technologically advanced countries and their profiles showed that they had contributed in the research or development of refactoring and unit testing support.

Respondents provided their opinion by selecting one of the options from 5-likert scale of answers against each question. The 5-scale options represent: Strongly agree, agree, neutral, disagree and strongly disagree. 38 experts in refactoring domain were consulted. Most of the respondents used Java for development. 76% respondents used Eclipse, 39% respondents used other tools and 18% respondents performed refactoring manually. 42% respondents worked in software development organizations of all sizes, where as others were involved in research and development activities at universities. 74% participants had an advanced or expert level of refactoring knowledge. 58% respondents had above 5 years of IT experience. Unit testing was performed mostly by 66% and sometimes by 32% respondents. The data from the survey was analyzed to answer several research questions. The results are summarized in the Table 9.

Table 9. Results from survey questions on performance of refactoring tools.

Refactoring Tools	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree	No Response
Introduce Syntactic Errors in the Unit Tests	7.89%	26.32%	10.53%	23.68%	21.05%	10.53%
Introduce Semantic Errors in the Unit Tests	2.63%	39.47%	28.95%	10.53%	7.89%	10.53%
Introduce Semantic Errors in the Clients	10.53%	39.47%	18.42%	7.89%	10.53%	13.16%
Introduce Semantic Errors in Refactored Code	7.89%	21.05%	21.05%	21.05%	21.05%	7.89%
Deteriorate the Software Quality	0.00%	5.26%	18.4%	23.6%	42.11%	10.53%
Introduce Test Smells in the Unit Tests	5.26%	18.42%	21.05%	23.68%	10.53%	21.05%
Do not Support Unit Tests Reorganization	26.32%	44.74%	10.53%	5.26%	0.00%	13.16%
Do Not Fix Syntactic Errors in the Unit Tests	10.53%	18.42%	34.21%	10.53%	15.79%	10.53%

The analysis shows that, in general, respondents agree to most of the issues identified in the survey and hence desire improvement. While confirming many unidentified issues with the experts, our findings also highlight gaps between required and existing tool characteristics. It can be seen from the results that there is more agreement from the respondents regarding issues related to unit test reorganization and adaptation. Very few participants agreed that the refactoring tools deteriorate quality, but it has been shown that this is indeed true. The figures indicate that there is need for extensive research in this area. Additionally, the results for research questions related to semantic defect introduction in clients, refactored code and unit tests also show that there is a huge room for improvement.

6. Conclusions

Refactoring is a structured and disciplined process of code transformation. Software maintenance, such as refactoring should ideally improve software quality [27]. But on the contrary the existing support for refactoring may introduce semantic errors as well as bad smells in the production and test code. In this paper, three refactoring including MM, PUM and PUM refactoring have been analyzed. Additional steps have been suggested that should be included in these refactoring. In order to, judge the effectiveness of these GL an experiment using 40 graduate students was setup. The results of the study are promising and have leaded us towards extension of other refactoring GL as well. The findings of this paper have been strengthened by conducting a controlled expert survey. Most of the respondents have appreciated the identified problems. The results from this analysis have been used to extend the Eclipse refactoring plug-in. Preliminary information about TAPEcan be found in [22].

References

[1] Basit W. and Lodhi F., "Preservation of Externally Observable Behavior after Pull Up Method Refactoring," *in Proceedings of ICCIT*, pp. 309-314, 2012.

[2] Basit W., Lodhi F., and Bhatti M., "Evaluating Extended Refactoring Guidelines," *in Proceedings of the 36th Annual Computer Software and Applications Conference Workshops*, Izmir, pp. 260-265, 2012.

[3] Basit W., Lodhi F., and Bhatti M., "Extending Refactoring Guidelines to Perform Client and Test Code Adaptation," *in Proceedings of 11th International Conference on XP*, Norway, pp. 1-13, 2010.

[4] Basit W., Lodhi F., and Bhatti M., "Formal Specification of Extended Refactoring Guidelines," *in Proceedings of International Conference on Emerging Technologies*, Islamabad, pp. 353-358, 2012.

[5] Basit W., Lodhi F., and Bhatti M., "Unit Test: A Specialized Client in Refactoring," *in Proceedings of the 7th International Conference on Software Paradigm Trends*, Rome, pp. 285-290, 2012.

[6] Counsell S., "Is the Need to Follow Chains a Possible Deterrent to Certain Refactorings and an Inducement to Others?," *in Proceedings of the 2nd International Conference on Research Challenges in Information Science*, Marrakech, pp. 111-122, 2008.

[7] Counsell S., Hierons R., Najjar R., Loizou G., and Hassoun Y., "The Effectiveness of Refactoring Based on a Compatibility Testing Taxonomy and a Dependency Graph," *in Proceedings of Testing*: *Academic and Industrial Conference*, Windsor, pp. 181-192, 2006.

[8] Counsell S., Swift S., and Hierons R., "A Test Taxonomy Applied to the Mechanics of Java Refactorings," *Advances in Computer and Information Sciences and Engineering*, Springer Netherlands, 2007.

[9] Daniel B., Gvero T., and Marinov D., "On Test Repair using Symbolic Execution," *in Proceedings of the 19th International Symposium on Software Testing and Analysis*, Trento, pp. 207-218 2010.

[10] Daniel B., Jagannath V., Dig D., and Marinov D., "Reassert: Suggesting Repairs for Broken Unit Tests," *in Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, Auckland, pp. 433-444, 2009.

[11] Deursen A. and Moonen L., "The Video Store Revisited-Thoughts on Refactoring and Testing," *in Proceedings of the 3rd International Conference on Extreme Programming and Flexible Processes in Software Engineering*, Italy, pp.71-76 2002.

[12] Deursen A., Moonen L., Bergh A., and Kok G., "Refactoring Test Code," *in Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering*, pp. 1-4, 2001.

[13] Dig D., Negara S., Mohindra V., and Johnson R., "Refactoring Aware Binary Adaptation of Evolving Libraries," *in Proceedings of the 13th International Conference on Software Engineering*, Germany, pp. 441-450, 2008.

[14] Eclipse Project., available at: http://www.eclipse.org, last visited 2011.

[15] Embarcadero Technologies., available at: http://www.codegear.com/br/products/jbuilder, last visited 2011.

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

[17] George B. and William L., "An Initial Investigation of Test Driven Development in Industry," *in Proceedings of the ACM Symposium on Applied SAC*, pp. 1135-1139, 2003.

[18] Henkel J. and Diwan A., "CatchUp!: Capturing and Replaying Refactorings to Support API Evolution," *in Proceedings of the 27th International Conference on Software Engineering ICSE'05*, pp. 274-283, 2005.

[19] Jet Brains., Intellij idea, available at: http://www.intellij.com/idea/, last visited 2011.

[20] Jiau H. and Chen J., "Test Code Differencing for Test-Driven Refactoring Automation," *ACM SIGSOFT Software Engineering Notes*, vol. 34, no. 1, pp. 1-10, 2009.

[21] JUnit., available at: http://www.junit.org, last visited 2015.

[22] Kiran L., Lodhi F., and Basit W., "Test Code Adaptation Plugin for Eclipse," available at: http://www.agilealliance.org/files/session_pdfs/TAPE.pdf, last visited 2012.

[23] Link J. and Frohlich P., "Unit Testing in Java: How Tests Drive The Code," available at: http://booksite.elsevier.com/samplechapters/9781558608689/01Preface_and_contents.pdf, last visited 2003.

[24] Mens T. and Tourwe T., "A Survey of Software Refactoring," *IEEE Transaction Software Engneering*, vol. 30, no. 2, pp. 126-139, 2004.

[25] Meszaros G. and Fowler M., *XUnit Patterns: Refactoring Test Code*, Addison-Wesley, 2007.

[26] MirzaAghaei M., Pastore F., and Pezz M., "Automatically Repairing Test Cases for Evolving Method Declarations," *in Proceedings of IEEE International Conference on Software Maintenance ICSM*, Timisoara, pp.1-5, 2010.

[27] Misra S. and Cafer F., "Estimating Quality of JavaScript," *the International Arab Journal of Information Technology*, vol. 9, no. 6, pp. 535-543, 2012.

[28] Pipka J., "Refactoring in a Test First-World," avilable at: http://cf.agilealliance.org/articles/system/article/file/967/file.pdf, last visited 2002.

[29] Soares G., Gheyi R., Massoni T., Corn´elio M., and Cavalcanti D., "Generating Unit Tests for Checking Refactoring Safety," available at: http://www.lbd.dcc.ufmg.br/colecoes/sblp/2009/014.pdf, last visited 2009.

[30] Streckenbach M. and Snelting G., "Refactoring Class Hierarchies with Kaba," *ACM SIGPLAN Notices-OOPSLA '04*, vol. 39, no. 10, pp. 315-330, 2004.

[31] Sun Microsystems, Netbeans ide., available at: http://www.netbeans.org/, last visited 2011.

[32] Zaidman A., Rompaey B., Demeyer S., and Deursen A., "Mining Software Repositories to Study Co-Evolution of Production and Test Code," *in Proceedings of the 1st International Conference on Software Testing*, Lillehammer, pp. 220-229, 2008.

Wafa Basit is a PhD candidate in the Department of Computer Science at the National University of Computer and Emerging Sciences, Pakistan. Her research interests include empirical software engineering, formal aspects of software evolution and maintenance. She believes that the software industry does not provide adequate training to individuals so that they can apply latest techniques to reduce the development effort and improve software quality. She is committed to contribute in the area of software refactoring such that there is enough literature and guidelines for the developers to properly refactor the software.

Fakhar Lodhi is a professor and dean at GIFT university, Pakistan. Over the last 25 years, he has spent his time evenly in the industry and academia. He has been linked with some of the leading software houses in Pakistan in various capacities and his industrial experience spans almost all aspects including design and architecture, project management, and setting-up and running software houses. His areas of interest include offshore development processes, software metrics, object oriented methods, and teaching of software engineering.

Usman Bhatti is co-founder and technical lead at Synectique, France. He obtained his PhD from France and held the position of Assistant Professor at NUCES, Pakistan before returning to France in the RMod research team in INRIA for post-doctorate. His research interests include software understanding, maintenance, visualization, and software refactoring. Currently, he leads software development at Synectique, a startup offering dedicated tools for software analysis. Usman Bhatti strongly believes that the gulf between academic and industrial world should shrink so that academic world gets to work on real problems and industrial world gets its difficult problems solved.