Pak. J. Engg. & Appl. Sci. Vol. 13, July, 2013 (*p.* 37-53)

A Metric Based Evaluation of Unit Tests as Specialized Clients in Refactoring

W.Basit [1], F. Lodhi[1], F.Ahmed[1] and M.U. Bhatti[2]

1. *Department of Computer Science, National University of Computer and Emerging Sciences, Faisal Town, Lahore, Pakistan. {wafa.basit, fakhar.lodhi, farooq.ahmed}@nu.edu.pk*
2. *Rmod Team, Inria Lille - Nord Europe, France. muhammad.bhatti@inria.fr*

Abstract

In the context of refactoring, a unit test significantly differs from an ordinary client. A unit test is the only safety net available to verify the impact of refactoring. In addition, tight coupling and stronger association with the refactored class are its key discriminating characteristics. Hence, any change in the code readily affects the behavior and quality of the test code. But if test code is adapted and refactored along the production code, its behavior shall be preserved and quality may improve. In this paper with the help of quality metrics, we establish the fact that unit test is a different type of client that needs "special" handling in the context of refactoring. We demonstrate through most commonly used refactorings on an open source project that there is a need to enhance the existing refactoring support for Java to include the specific adaptation mechanism for unit tests that eradicates the effect of refactoring and also improves the internal structure of test code.

Key Words: Clients; Unit tests; Refactoring Guidelines; Adaptation; maintenance;

1. Introduction

Refactoring is defined as equivalence transformation that does not change the external behavior of the software system, yet it improves the internal structure of the code [1, 28]. When it is said *"code"*, there rises a misconception which has lead to a considerable gap between the definition and the implementation of refactoring process [13, 37-40].

Code in a program or a software system is composed of multiple classes having various relationships. A class may be a parent, a child, a client or a unit test class. Each class is a client in a system which interacts with the other classes to perform its function. Therefore, any change in the interface leads to change in the clients. For instance, a refactoring performed on a public entity in the system can affect all locations where it is accessible and has been used, including the owner class, its parent or subclasses, clients and test classes. Thus, the maximum scope of the refactored entity can encompass the whole software system or program. If any of the affected parts in the system is not updated, the idea of complete behavior preservation shall not be fulfilled.

Therefore, principally the intent of refactoring should be (1) behavior preservation (2) improvement in the internal structure and (3) appropriate resolution of syntactic errors caused due to refactoring, in all the components of a software system including clients and unit tests. We will refer to these three conditions for refactoring as essential conditions for refactoring in this paper. The existing state of art generally considers these conditions for production code only and the unit tests are not taken into account. Hence, behavior is usually not preserved and quality of the unit tests deteriorates.

Fowler's refactoring catalog [1] is a most commonly used source for understanding different kind of refactorings. Based on Fowler's guidelines many tools for Java have been developed to support refactoring, a few of the most commonly used are: Eclipse, IntelliJ IDEA, JBuilder, NetBeans [23-26] etc. Among these Eclipse and NetBeans do not fulfill any of essential conditions of refactoring for the clients and unit tests. Whereas, IntelliJ IDEA and JBuilder in some cases fulfill the conditions 1 and 3 but as a consequence, the clients specifically unit tests get *infected* with *bad smells,* a condition 2 violation.

For instance, the purpose of *Move Method* [1] (see Figure 1) is to move the related behavior to its appropriate home in the system. Principally, along the movement of method to 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. But the existing tools [23-26] and guidelines [1] do not provide any support on such adaptations therefore violate the second essential condition by introducing the *indirect test* [9] smell. Also, tools like intelliJ IDEA and JBuilder do not make appropriate replacement of source object with the target object in clients including unit tests. Thus, *Move Method* refactoring [1] eventually deteriorates the quality of test code if existing support for java is used [40]. Whereas, the overall affect of refactoring should be improved quality in all the components of the software system including unit tests.

The objective of this paper is to prove that unit tests need special handling in the context of refactoring. Hence, there is a need to highlight the fact that there is a difference between the significance of unit tests during original coding vs. during refactoring. Unit tests are more critical during refactoring process because programmers rely on unit testing to determine whether a refactoring was Applie d correctly and the behavior of the production code is kept unchanged [1, 28, 29]. Proving that a refactoring is behavior preserving is non-trivial and therefore reliance on test suite executions is generally considered the best choice as formal approaches are hard to employ. But the cost and effort involved in adapting unit tests to keep them consistent with the refactored code is huge [29]. However, unit tests represent a significant software effort and investment. Therefore, it is important to keep them aligned with the refactored code. In order to exhibit our approach, we have used *LanSimulation* [35], an open source software project, which is frequently used for teaching the refactoring process at graduate level. The production and test code in this project are written in an ad hoc manner leading to various opportunities for refactoring. Quality [33] and code coverage metrics [36] have been used to demonstrate the change in quality of production and test code with and without test adaptation after refactoring. Our analysis shows that refactoring impacts unit tests in various different ways in comparison to an ordinary client therefore they require additional adaptation. The *LanSimulation* project refactored with another dimension of test adaptation can significantly help software engineering teachers to explain the complete process of refactoring.

	After Refactoring		
Before Refactoring	**Tools**	**Fowler's**	**Ideal Approach**

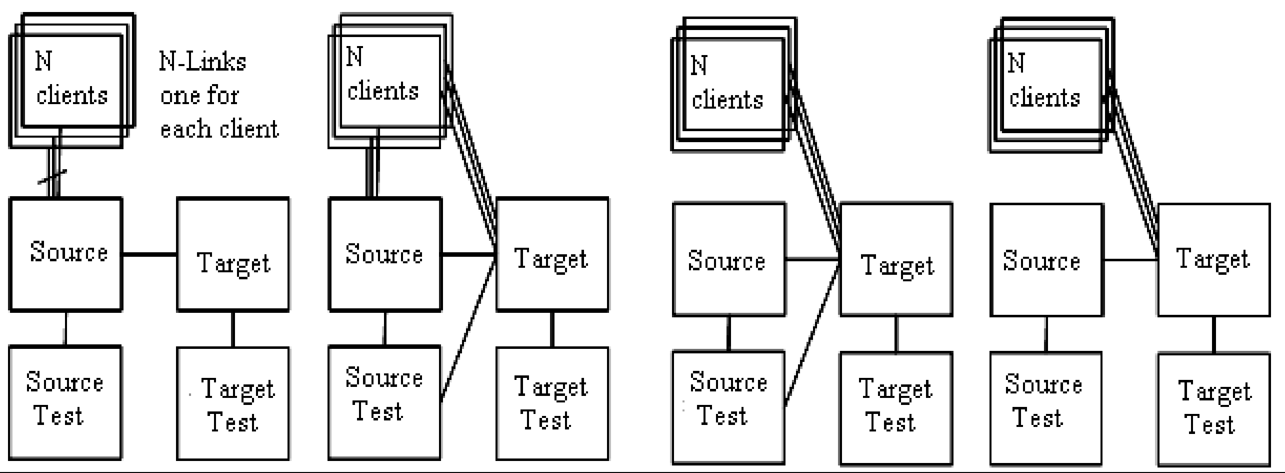

Fig.1 A comparative view of a subsystem after move method refactoring using different approaches

This paper is organized in 6 sections: section 2 surveys current literature on refactoring with respect to client and unit test adaptation. In section 3, we differentiate between a unit test and an ordinary client. In section 4, we present the extended solution of the *LanSimulation* [35]. In section 5, we lay down our analysis using various metrics. In the end we draw our conclusions.

2. Related Work

Refactoring process should evaluate the software quality and maintain the consistency between the refactored code and other software artifacts including documentation, design documents, tests, etc [14]. However, in practice the evolution of code along the other artifacts is generally not taken into account.

Unit testing is a fundamental component of the refactoring process. Fowler [1] 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 [2, 11]. Zaidman *et al.* 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 [19]. In our earlier work [13, 37-40] we have discussed in detail the state of art and practice that addresses or should address client and unit test adaptation while refactoring. We summarize the existing state of art in Table 1

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

Researcher	**Research Summary**
Fowler [1]	• 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 [2]	Presented a test taxonomy that categorizes refactorings based on their effect on test code. These are: compatible, backwards compatible, make backwards compatible, and incompatible.
Counsell et al. [4-6]	• Assessed the test taxonomy presented in [2]. • In our previous work [13] we have shown that the categorization used by [2-6] has various loop holes. • A refactoring dependency graph is developed for Fowler‘s catalogue [1] and a shorter list of compatible refactorings is suggested by excluding all the other refactorings that may use those refactorings that break unit tests. • This approach essentially rejects the use of many important refactorings that are necessary for improving the program structure.
H. C. Jiau and J. C. Chen [8], Pipka [11]	• Test Driven Refactoring (TDR) [8] and Test-first Refactoring (TFR) [11] 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 [28]. • Do not provide guidelines to adapt test code according to the targeted refactoring.
Soares et al. [32]	• Soares et al. [32] 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 refactorings. • They have also evaluated the refactoring support provided by Eclipse, IntelliJ IDEA, JBuilder, NetBeans. They observe that program refactorings in IDEs are commonly implemented in an ad hoc way and the semantic aspects of behavior are several times not preserved.
Basit et al [13,37-40]	• In [13] a mutually exclusive categorization of refactoring

	guidelines has been presented based on the impact of refactoring on clients and unit tests. • In [37] the problems with Fowler's refactoring catalog and java refactoring tools including NetBeans, Eclipse, Intellij IDEA and JBuilder have been discussed. These tools introduce semantic errors in the refactored 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 [38] 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. • TAPE (Test Adaptation Plugin for Eclipse) [39] makes easier for the developer to organize the unit tests along the changes in the refactored code. • In [40] it has been demonstrated with the help of various examples that unit test is a specialized client in the context of refactoring.
Daniel et al.[41]	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.
Tools	**Salient Features**
TestCareAssistant [15]	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 [16]	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]	CatchUp! adapts clients of the evolving Application Programming Interfaces (API's) [18]. It provides full support for three types of refactorings Rename Type, Moving Java Elements and changing method signature. This tool takes care of only compilation errors that can appear in the clients due to a subset of refactorings performed on any API, and therefore ignore the semantic errors that could be caused due to refactoring process.
Kaba [20]	KABA [20] also includes all clients in the refactoring process. It guarantees preservation of behavior for the clients either through static analysis or all test runs (dynamic analysis) for any input.
Reba [21]	ReBA instead of adapting the clients of the evolving API, creates compatibility layers between new library APIs and old clients [21].This layer is created in the form of an adapted version that supports both versions of the API.

3. Unit Test: A Specialized Client

Unit testing is performed by developers to ensure that no individual unit or a class in the system makes it error prone. Unit tests, test code at its lowest level of granularity which is the method level in Object Oriented (OO) domain. But nowadays, the meaning of the term "unit testing" seems to have been lost. Unit tests are written by developers in many different ways. One class may have multiple test classes testing it or vice versa. Similarly, one method or set of methods may be tested by multiple methods. Several times the unit test suites are much larger than the production code itself, so managing test code and ensuring its completeness becomes extremely difficult. Unit tests written in such an ad hoc manner are usually infected with *test smells* [9].

It is usually said that "code is code", and so test code that exercises a given class is usually not special in any significant way from other clients of that class. There is a need to correct this misconception about code and test code in the context of refactoring. Unit tests have a few discriminating characteristics from that of an ordinary client.

3.1 Unit test: A safety net

In refactoring, execution of unit tests has a very high significance. Fowler suggests several times during the course of refactoring to "*Compile and Test*" [1]. This is because after refactoring, unit tests serve as a *safety net* that verifies the preservation of software behavior. However, this is not the case with the ordinary clients.

3.2 Tight Coupling

Unit tests are tightly coupled with the classes they test [9]. The more coupled two components are, the more difficult it gets to keep the two consistent. This coupling between code and unit tests is unavoidable and is intended to be this way. Therefore, in this particular situation it is not considered a bad design. But on the contrary if this level of coupling is seen between classes in the production code, it leads to refactoring.

3.3 Parallel hierarchy of classes

Implementing unit tests often leads to a *parallel hierarchy of classes,* where every class has a corresponding test class [12, 27]. Consequently, there emerges an inheritance chain of corresponding unit test classes, paralleling the classes being tested. While it is by no means necessary, it makes things much simpler if unit tests are set up in a parallel hierarchy in the software under test [27]. The inheritance relationship between test classes maintained in parallel with production code has several advantages [30]:

1) *Reuse of test code* - The inheritance relationship between test classes for production classes that are also related via inheritance facilitates the reuse of individual test cases.

2) *Separation of production and test code* - There are several reasons for keeping the two types of code separate. First, the production code remains smaller and thus less complex. Second the executible code also remains smaller requiring fewer resources. Finally, these two pieces are sometimes written by different groups and the physical separation becomes necessary.

3) *Maintenance of test cases* - In an iterative development process, the tests should be easy to apply repeatedly across the development iterations. They must also be easy to maintain as the production code changes. The inheritance relationship in an object-oriented language supports the development of code with these chara\cteristics

4) *Easy Testing:* The test software is organized around the same architecture as the production code. The architecture of the test code never coincides with the production code but it always stays the same distance away. If a developer studies the production architecture, they automatically understand the architecture of the test software. This organization of the test classes makes it easier to test a method in the context of a class and to overcome information hiding in order to observe state.

It is known that *parallel class hierarchies* [1] indicate tight coupling, duplication, a lack of abstraction and usually result in hard to change production code including clients. Since tight coupling is the very purpose of unit testing (as we want to test every public method of a specific class) and since we usually want to test concrete implementations instead of abstractions, parallel class hierarchies do not seem to be a problem in the context of unit testing.

3.4 Stronger Association

The unit test classes as defined in JUnit [22] contain test method/s for each method to be tested. These test methods directly call the functions they test. The association between a class and its unit test/s is much *stronger* than the client's *association*. For instance, if a method M is moved from class A to B, its client C starts referring to B and the association ends. Whereas A_T the unit tests of class A shall always have a link with A.

3.5 Textual versus organizational adaptations

We define *Textual Adaptations* as changes done at the statement level inside any class in the system affected due to refactoring. Whereas *Organizational Adaptations* can be defined as transformations that are related to changes in the structure of any class impacted with refactoring. Generally refactoring results in textual adaptations in both Client and unit test classes. *Rename Method* [1] requires changing the method calls both in clients and unit tests. The organizational changes are made in unit test classes only. *Move Method* [1] requires movement of test method to the target's test class whereas clients requires only replacement of object (source to target) in the calls to the moved method.

3.6 Number of ordinary clients versus unit test classes

There is no limitation on the number of clients for any class in the system. But in unit testing it is suggested to have one unit test class for each class in the system. If any class is too big or complicated to be tested by one class, it should be refactored.

The additional adaptations required by the unit tests after refactoring to eliminate the problems and smells are displayed in Table 2. In our earlier work we have explained in detail each of the refactoring and its impact on unit tests [13, 37-40].

Refactorings that are done at statement level or inside a method body have no impact on the test suites because unit tests are normally black-box tests [12]. But refactorings involving the interface of the class can lead to broken clients and tests if proper adaptive actions are not taken.

1) Demonstration

In order to exhibit our approach, we have used *LanSimulation* [35] an open source software that is frequently used for teaching the refactoring process. The production and test code in this project are written in an ad hoc manner leading to various opportunities for refactoring (see Figure 2).The refactored source code using our proposed approach is available at:

http://code.google.com/p/lansim-refactoring/source/checkout

In order to determine the effectiveness of test adaptation, initially we refactored the production code but did not restructure or extend the test code and only fixed the compiler errors in the *LanTests* class. Later, the production code and test code were evolved together. The final revision of the *LanSimulation* project is modeled in Figure 3 (Production code) and Figure 4 (Test Code)

After refactoring the production code, number of classes has visibly increased. *Network* class that was acting as a god class in the system earlier, has been cut to proper size and its functionality has been distributed to other classes as *Node*, *Printer*, *Workstation*, etc. - thus, resulting in the usual benefits of refactoring. We present 10 revisions of the *LanSimulation* project, each revision was created by applying one or more refactorings on the production and test code.

In Table 3 the effect of each refactoring on the production code is described. The changes performed on the production code ensure fulfillment of all three essential conditions for the production code.

Table 2 Test problems/smells introduced due to various refactorings

Type of Refactoring	**Test smell/ problem introduced**	**Unit test adaptation required**
Statement level	None	None
Renaming program entities	Bad test name	Rename test method,, test class, test package
Change in method signature	Invalid tests	Update references, make return types compatible
Extraction	Eager tests, Missing tests	Extract test method, test class, sub test class
Inlining	Invalid tests	Remove test method or test class exclusively testing the inlined method or class
Pull Up/push down	Test code duplication, Indirect tests, Invalid tests	Pull up/push down test method
Moving	Indirect tests, Invalid tests, Test code duplication	Move test method, update references

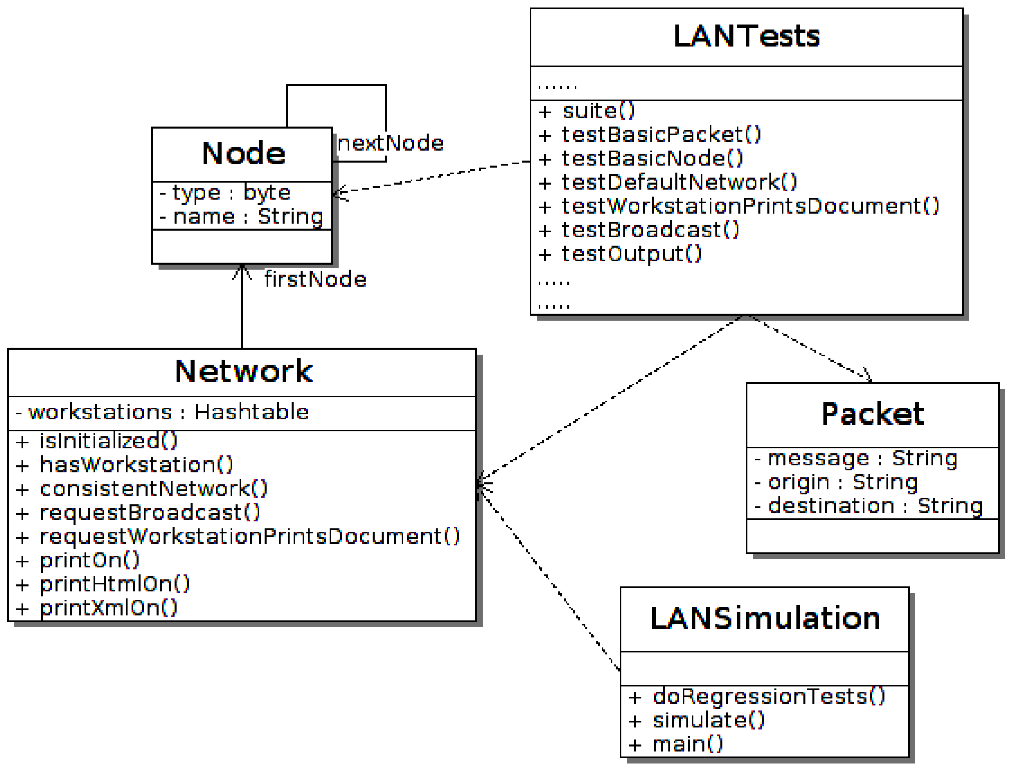

Fig.2. Lan Simulation project prior to refactoring

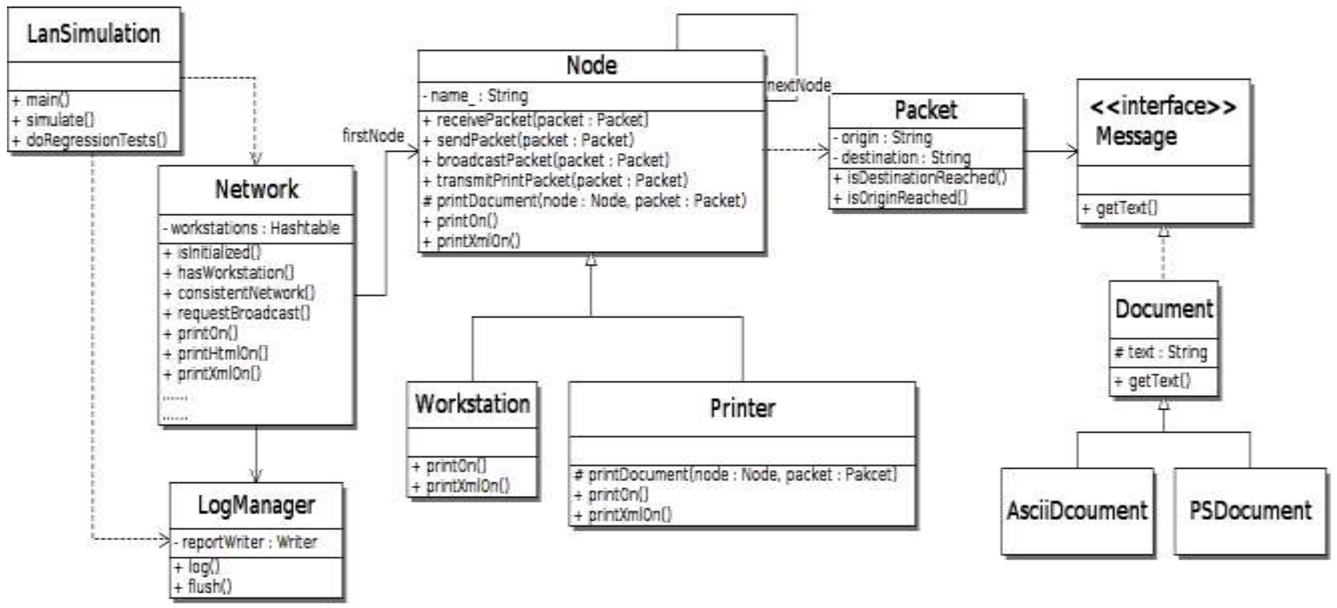

Fig.3 Lan Simulation project (production code) after refactoring

Table 3 Effects of refactorings on production code in the *LanSimulation* project

R#	Refactoring Applied	Description of Changes in the production code
r0	-	Initial commit; verified that everything is properly working; all tests verified
r1	Extract Test Class	
r2	Extract Method	Refactored method Network.consistentNetwork: removed long method smell
r3	Introduce Class, Introduce Field, Introduce Parameter	Refactored *Network.requestBroadcast*: a new class *LogManager* was introduced. This prompted further changes including introducing a new field *logManager* in the *Network* class that consequently led to introduction of a new parameter in *Network* constructor and *DefaultNetwork* method. This introduction of new parameter resulted in broken tests and client code.
r4	Introduce Method, Introduce Field, Introduce Parameter	Refactored *Network.requestBroadcast*: introduced new methods *sendPacket* and *receivePacket* in the *Node* class and moved the corresponding code from the *Network* class to the *Node* class. This required introducing a new field *logManager* in *Node* class as well and required introducing a new parameter to *Node* constructor that resulted in broken test code.
X	Introduce Method, Move Method	Refactored *Network.requestWorkstationPrintsDocument*: introduced a new method *transmitPrintPacket* to the *Node* class and also moved *printDocument* from *Network* to *Node*
r6	Move Method	Refactored *Network*: moved *printOn*, *printHtmlOn* and *printXmlOn* from *Network* to *Node* class
r7	Introduce Method	Refactored *Node.transmitPrintPacket*: introduced two methods *isDestinationReached* and *isOriginReached* and moved the necessary logic to *Packet* class
r8	Replace type code with state/strategy Remove Field, Change Parameter, Introduce Method,	Refactored *Node*: made several changes - the most important being the extraction of subclasses *Printer* and *Workstation* through application of State pattern. Other changes included removing the field *type* from *Node* as after extracting the subclasses, there was no need left for keeping *type* information. This again resulted in changing the *Node* constructor. A method *equalsType* was introduced. The changes made broke the client and test code but this time the reason was the introduction of subclasses as now the client code had to instantiate appropriate objects of the child classes rather than the parent *Node* class.
r9	Change Parameter	Refactored *Network*: change parameter in several methods that resulted in broken client and test code
r10	Extract Class / Interface	Refactored *Printer*: extracted interface *Message* and classes *Document* and its subclasses *AsciiDocument* and *PSDocument* (through applying State Pattern) – resulted in broken tests and client code

In contrast to the existing approaches of refactoring, test adaptation and reorganization helps in the fulfillment of essential condition 3 that requires improvement in the internal structure of the test code along production code. In Table 4, the adaptation steps performed on unit tests and the test smell removed due to these steps have been described. Using the existing methods and techniques of refactoring, the quality of unit tests is negatively affected. Not only new test smells are introduced but the existing ones also remain. On the contrary as demonstrated in Table 4, a number of test smells were removed in each revision of *LanSimulation* project by applying the test adaptation mechanism along refactoring. As we can see in Figure 4, the responsibilities of the *LanTests* class have been properly distributed to the unit tests corresponding to each class in the production code.

Table 4 Test smells removed after performing test adaptations in the LanSimulation project

R#	Broken Tests / Client?	Test Adaptation	Test Smell Removed
R1	No	Extract Test Classes: created separate test classes for the corresponding classes in source i.e. Network, LanSimulation, Packet and Node. Move Test Methods • testBasicPacket to PacketTest Class. • testBasicNode to NodeTest Class. • testDefaultNetworkToString(),testWorkstationPrintsDocument(), testBroadcast(),testOutput(), PreconditionViolationTestCase and testPreconditionViolation() to NetworkTest Class.	God Test Class, Indirect Test
r2	No	There is not any specific unit test for Network.consistentNetwork method therefore this refactoring does not lead to Extract test method refactoring. The newly created methods are private: • verifyRegisteredWorkstations(), • verifyWorkstationsCount(), • verifyPrinterExists(), • verifyTokenRing(). Hence, no new test methods are created.	
r3	Yes	1. Create unit test for LogManager 2. The elaboration of the objects of Network class and calls to *DefaultNetwork* method are updated to include the new parameter.	Missing Test Class
r4	Yes	1. Create new test methods for *sendPacket* and *receivePacket* in the *Node* Test class. 2. The elaboration of the objects of Node class and calls are updated to include the new parameter.	Eager Test
r5	No	1. Create new test methods for *transmitPrintPacket* 2. The test method for *printDocument* does not exist, therefore instead of moving, create a test method in the *NodeTest* class.	Missing Test
r6	No	Create test methods for *printOn*, *printHtmlOn* and *printXmlOn* in the *NodeTest* class.	Eager/ Indirect Test,
r7	No		
r8	Yes	1. Create test subclasses of *NodeTest* class, namely *PrinterTest* and *WorkstationTest*. 2. The elaboration of the objects of Node class .	Missing test class
r9	Yes	Calls to the methods whose parameters are changed are updated	
r10	Yes	1. Create test classes for *Document* and its subclasses *AsciiDocument* and *PSDocument.*	Missing Test class

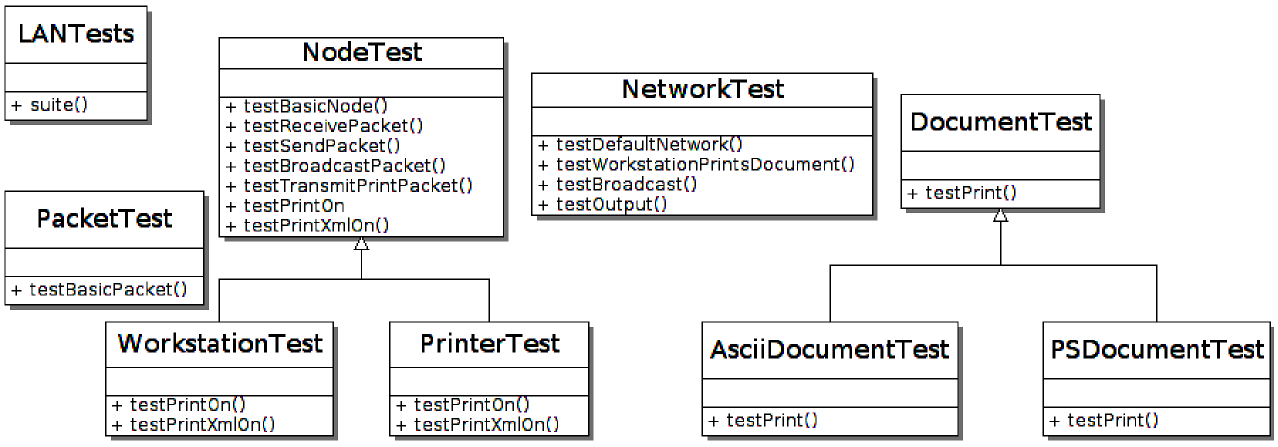

Fig.4 LanSimulation project (test code) after refactoring with test adaptation

2) Measurements

In this section we have provided the detailed comparative analysis of the conventional and our proposed approach using metrics. In Table 5 and 6 we list down metrics for size, complexity, cohesion and duplications. These metrics have been calculated using Sonar [33] for test code, production code and also complete code of *LanSimulation* project for the existing and the proposed approaches of refactoring. In this section following abbreviations have been used:

BR: Before Refactoring, ARW/OTA: After Refactoring Without Test Adaptation, ARWTA: After Refactoring With Test Adaptation, NCLC: Non-Commented Lines of Code, STMTS: Statements, CC: Cyclomatic Complexity, LCOM: Lack of Cohesion in Methods, RFC: Response For Class, NOM: Number of methods. Our proposed approach for test adaptation not only focuses on syntactic adaptation (essential condition 3) of test code but also helps in restructuring of test code, so that the test code resides in its right home (essential condition 2).

Table 5 Size metrics of the LanSimulation project before/after refactoring with/without test adaptation

		Size				
		Lines	**NCLC**	**NOM**	**Public API**	**STMTS**
Test Code	**BR**	338	281	12	12	147
	ARW/OTA	343	277	12	12	150
	ARWTA	605	400	33	35	206
Production code	**BR**	848	481	19	28	331
	ARW/OTA	1059	538	55	65	293
	ARWTA	1059	538	55	65	293
Complete code	**BR**	1186	762	31	40	478
	ARW/OTA	1402	815	67	77	443
	ARWTA	1664	938	88	100	499

Table 6 Quality metrics of the LanSimulation project before/after refactoring with/without test adaptation

		Complexity				**Duplications**	
		CC	**CC / method**	**LCOM**	**RFC**	**Blocks**	**Lines**
Test Code	**BR**	52	4.3	2	43	4	50
	ARW/OTA	52	4.3	2	48	2	24
	ARWTA	51	1.2	1.333333	133	0	0
Production Code	**BR**	19.5	2.65	1.75	76	12	123
	ARW/OTA	9.09	1.73	1	126	2	22
	ARWTA	9.09	1.73	1	126	2	22
Complete Code	**BR**	130	2.98	1.8	119	16	173
	ARW/OTA	12.67	1.95	1.08	174	4	46
	ARWTA	7.55	1.495	1.05	259	2	22

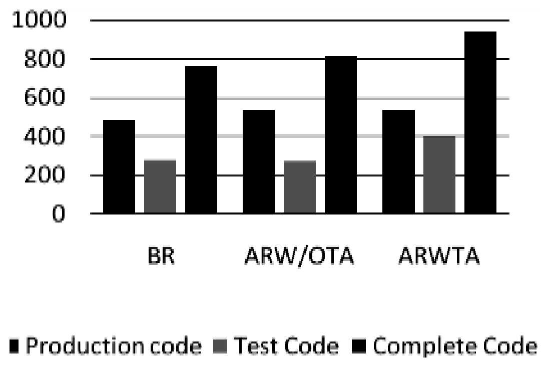

Fig.5(a) Comparative view of NCLC metrics after/before refactoring with/without test adaptation

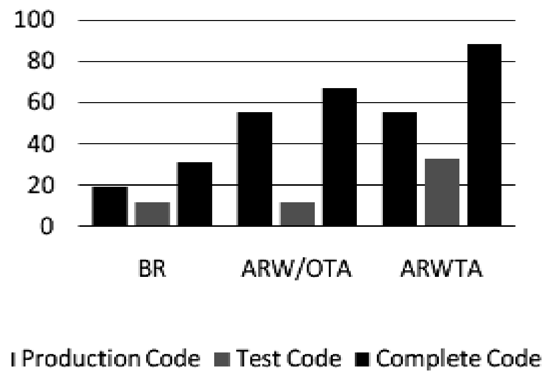

Fig.5(b) Comparative view of NOM metric after/before refactoring with/without test adaptation

We also focus on increasing test coverage, for instance, on extracting a class, a test class would be extracted or created, which would be later extended by the developer. IntelliJ IDEA and JBuilder take care of the first step in adaptation but the other subprocesses are not handled by any of the java refactoring tools [23-26].

In Figure 5(a) and 5(b) we can see that after refactoring with test adaptation, the size of test code has tremendously increased. This mainly happened because we applied “Replace Type Code with state/strategy” [1] refactoring on the *LanSimulation* project, which lead to creation of many more test classes and the test methods as well. Therefore, in this particular case we may conclude that our approach for test adaptation would significantly increase the size of test code, if the test code does not already cover the complete production code (as is the case in the *LanSimulation* project). In Figures 5(a) and 5(b) the increase in size has been demonstrated using the aggregate values of NCLC and NOM metrics for before/after refactoring with/without test adaptation.

3) Complexity

The existing approaches [1, 23-26] for refactoring ignore the quality of test code and only focus on the quality improvement of the production code. Complexity is another attribute of quality.

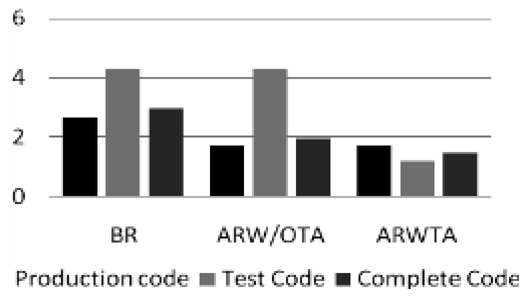

Fig.6(a) Comparative view of cc /METHOD after/before refactoring with/without test adaptation

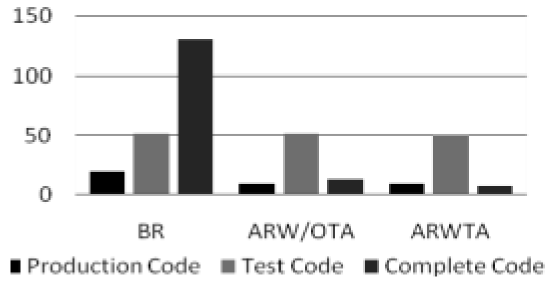

Fig.6(b) Comparative view of cc after/before refactoring with/without test adaptation

We calculated *Cyclomatic Complexity* [33] for the LanSimulation project before and after refactoring, with and without test adaptation. This metric represents the complexity of a method and complexity of a class. The metric value should be as low as possible. Higher values (more than 20) indicate that the software is hard to maintain, understand and that the degree of readability is low [33]. In the case of *LanSimulation* project the CC values prior to refactoring were 52, 19.5 and 130 for the test code, production code and complete code respectively. The detailed results can be seen in Table 6 and Figures 6 (a & b). We see that the complexity has evidently reduced after refactoring but the effect of test adaptation is extremely positive.

4) Duplications

If the same code structure is seen in more than one locations in a system it is called code duplication. It is considered a better choice to unify such code, either through *Extract Method*, *Extract Class*, *Pull Up Method* or *Form Template Method* [1]. In the *LanSimulation* project there was considerable code duplication (see Table 6 and Figure 7). Using the existing approaches the code duplication was reduced in the production code but duplication in the test code remained. Therefore, using our proposed approach the test code is also refactored and adapted, this lead to decrease in number of duplicated lines from 173 to 22 after refactoring. The left over duplicate code can be removed through further refactoring of the system.

5) Cohesion

Cohesion is a good indicator of whether a class meets the Principal of Single Responsibility. We have used LCOM (Lack of Cohesion Of Methods) metric to evaluate the cohesiveness of the *LanSimulation* project after employing the conventional and the proposed approach for refactoring.

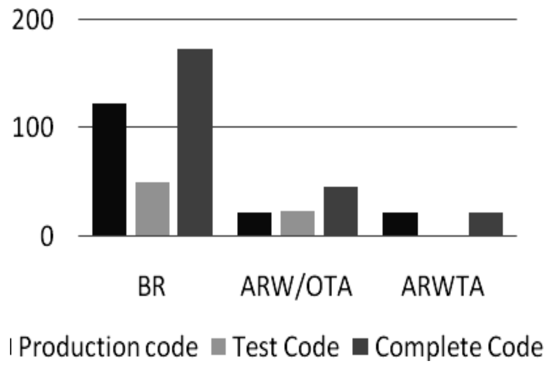

Fig.7 Comparative view of duplicated lines after/before refactoring with/without test adaptation

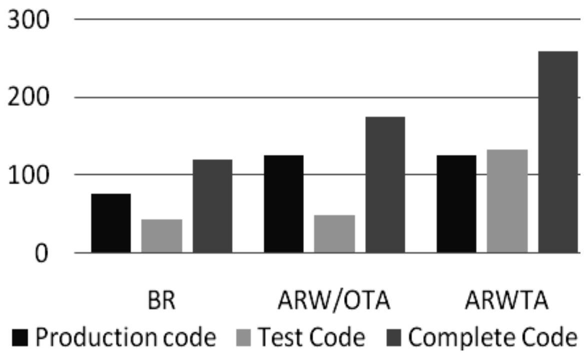

Fig.8 Comparative view of LCOM metric after/before refactoring with/without test adaptation

Sonar's [33] definition of LCOM is different from the conventionally used metric. It measures the number of "connected components" in a class. A connected component is a set of related methods and fields. There should be only one such component in each class. If there are 2 or more components, the class should be split into so many smaller classes [33]. A class that is totally cohesive will have an LCOM of 1. A class that is non-cohesive will have an LCOM greater than 1. The closer to 1 it approaches, the more cohesive, and maintainable, a class is.

LCOM prior to refactoring was higher both in the production code and test code because of Network and LanTests class. These classes were acting as God Classes. After refactoring without test adaptation the average LCOM value for the production code was reduced but the LCOM for test code remained high which contributed to the LCOM > 1 for the complete code. On the contrary after refactoring with test adaptation the LCOM for test code, production code and eventually complete code was reduced from 1.8 to 1.05 (See Table 6 & Fig.8).

6) Response for a class

The response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class [33]

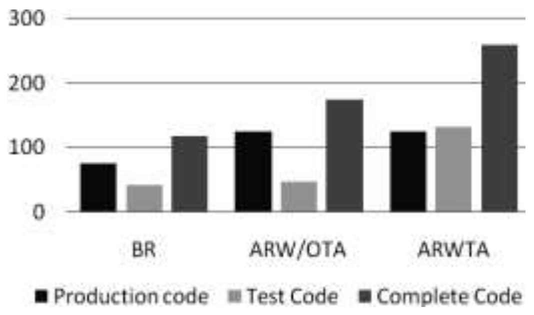

Fig. 9 Comparative view of aggregate RFC after/before refactoring with/without test adaptation

The aggregate RFC for both the test code and production code has reasonably increased (see Table 6 and Figure 9). With the increase in the number of classes the messaging between the classes has also raised and so has the RFC. Prior to refactoring there were 5 classes in the system including one test class and aggregate RFC for complete code was 119. After refactoring with test adaptation the classes were increased to 11(production code) and 9 (test code) and aggregate RFC for complete code was increased to 259. While, without test adaptation there were 11 classes in the production code and one test class and aggregate RFC for complete code was 174. With the increase in RFC the maintenance effort also increases

7) Coupling

High coupling is generally considered as a bad design characteristic. But talking about the unit tests and their coupling with the production code, it is supposed to be high. After the introduction of testing frameworks, the test code is kept separated from the production code physically, While, the logical bonding between these two cannot be reduced or broken. So in this very case refactoring with test adaptation would output production and test code with higher coupling.

Afferent couplings of a class measure the number of other classes that use the specific class. Efferent couplings measure the number of different classes used by the specific class. The ratio of efferent coupling (Ce) to total coupling (Ce + Ca) such that I = Ce / (Ce + Ca) is called the *Instability Index*. This metric is an indicator of the class's resilience to change. The range for this metric is 0 to 1, with I=0 indicating a completely stable class and I=1 indicating a completely instable class [41].

In Table 7 and Figure, it can be seen that average instability index for refactoring without test adaptation has increased and for refactoring with test adaptation the instability index has reduced to 0.71. This has happened because in the former approach the increase in classes of production code has not affected the number of test classes and there is only one test class for 11 production classes therefore it is highly instable with respect to maintenance.

Table 7 Comparative view of coupling metrics after refactoring with/without test adaptation

		Sum of Afferent Couplings	**Sum of Efferent Couplings**	**Average Instability Index**
Production Code	**BR**	9	6	0.38
	ARW/OTA	40	30	0.47
	ARWTA	53	30	0.420671
Test Code	**BR**	1	3	0.75
	ARW/OTA	1	8	0.888889
	ARWTA	13	33	0.71
Complete Code	**BR**	10	9	0.43
	ARW/OTA	57	38	0.50
	ARWTA	66	63	0.55

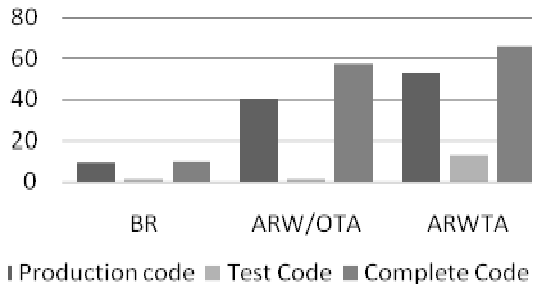

Fig. 10 Aggregate Afferent Couplings after/before refactoring with/without test adaptation

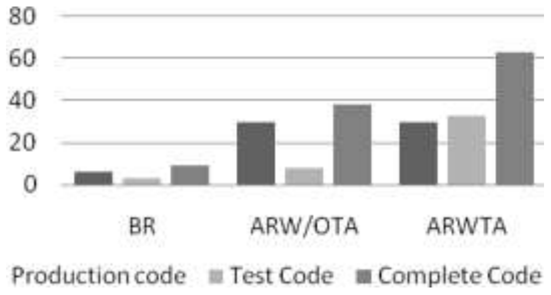

Fig. 11 Aggregate Efferent Couplings after/before refactoring with/without test adaptation

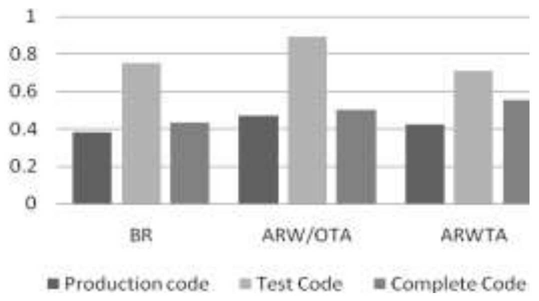

Fig. 12 Aggregate Instability Index after/before refactoring with/without test adaptation

In Figures 10,11 and 12 it is obvious that afferent and efferent couplings for the test code and production code have increased significantly specifically after refactoring with test adaptation. The reason for this increase is, that prior to refactoring most of the code was residing in *God Classes* namely *Network* and *LanTests*. When our approach was applied on the code the overall number of classes increased in a bigger proportion as compared to refactoring without test adaptation and hence, this gave rise to the couplings as well. It is interesting to note that the instability index was reduced for the test code in spite of the increase in couplings. When it is said that low coupling represents better quality, this is a very subjective statement. Using our results we also establish the fact that in the case of relationship between the production code and test code, high coupling is required in order to ensure maximum code coverage. In conclusion our approach has performed better in terms of adequately associating the production and test code. It has also made the system more stable in terms of maintenance

7) Code Coverage

Code coverage describes the extent to which the source code of a program has been tested. In our study we have employed Line Coverage and Branch Coverage for measuring code coverage using *Cobertura* [36]. Line coverage implies lines of code that are executed during unit test execution divided by the total number of executable lines, while, branch coverage measures the percentage of conditionals that are evaluated at least once divided by the total number of branches. As elicited in Table 4, with the evolution of production code, we identified the missing tests and created them such that the total coverage increased. The results are apparent in Table 8 and Figures 13 (a & b). The test extension along refactoring of production code results has increased branch coverage to almost 50%, whereas line coverage has risen to almost 97% after refactoring with test adaptation.

4. Conclusions

Refactoring is a structured and disciplined process of code transformation that should not invalidate behavior or deteriorate quality of any component in the software system including clients and unit tests. Unit tests are clients that require additional adaptations as compared to ordinary clients. If these changes are not performed, test code can get infected with various test smells including Eager Test, Indirect Test, Test Code Duplication etc.

Unit tests owe high significance in the refactoring process because they determine the validity of software behavior after refactoring. The existing state of art and practice on refactoring generally does not address appropriate client adaptation specifically test code adaptation. Java

Table 8 Comparative view of test coverage metrics after/before refactoring with/without test adaptation

	Branch Coverage (BC)%	**Line Coverage (LC)%**	**# of Un-covered Branches (UB)**	**# of Un-covered Lines (UL)**
Before Refactoring	27.75	79.9	76	38
After Refactoring without Test Adaptation	55.6	86.27778	61	28
After Refactoring With Test Adaptation	78.925	96.62222	53	8

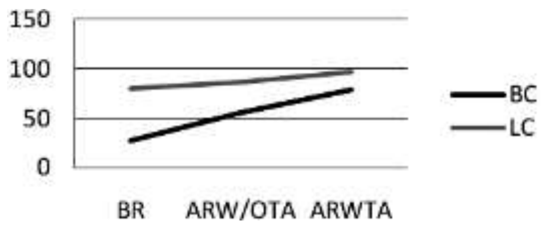

Fig.13(a) Comparative view aggregate branch and line coverage metrics after/before refactoring with/without test adaptation

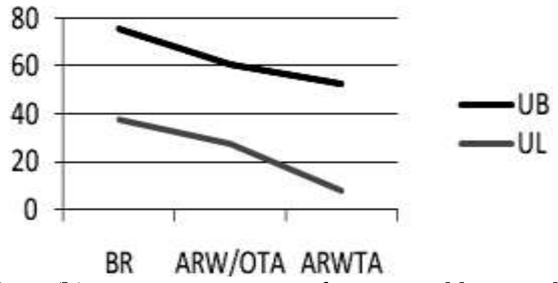

Fig.13(b) Comparative view of uncovered lines and branches metric after/before refactoring with/without test adaptation

development tools like JBuilder and IntelliJ syntactically adapt the clients and unit tests, such that externally observable behavior is preserved but they instead of improving the overall quality of the system, worsen it by inducing test smells. Similar is the case with Fowler's guidelines, which do not provide any mechanics for restructuring the test code.

Therefore, there is a need to extend the refactoring guidelines to address these issues. Also, automation is critical for refactoring, as manual refactoring can be very tedious and error prone. We have developed an Eclipse Plugin name TAPE (Test Adaptation Plugin For Eclipse) that extends the existing refactoring plugin. Our strategy is to provide developer assisted refactoring support such that all major actions are suggested to the developer and are peformed with his/her consent by the tool. The preliminary information about TAPE can be found in [39].

5 References

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. Refactoring: Improving the Design of Existing Code. 1999

[2] A. van Deursen and L. Moonen. The Video Store Revisited: Thoughts on Refactoring and Testing. In Proceedings of the 3rd International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2002), pp. 71-76, 2002

[3] A. van Deursen, L. Moonen, A. van den Bergh and G. Kok. Refactoring Test Code. In Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001), pp. 92-95. 2001

[4] S. Counsell , R. M. Hierons , R. Najjar , G. Loizou and Y. Hassoun, 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, p.181-192, August 29-31, 2006

[5] S. Counsell, S.Swift and R.M. Hierons, A Test Taxonomy Applied to the Mechanics of Java Refactorings, In Proceedings of SCSS (1), pp. 497--502 , 2007

[6] S. Counsell, Is the need to follow chains a possible deterrent to certain refactorings and an inducement to others? In Proceedings of second International Conference on Research Challenges in Information Science, 2008

[7] B. George and L. William, An Initial Investigation of Test Driven Development in Industry. In Proceedings of SAC ,2003

[8] H. C. Jiau and J. C. Chen , Test code differencing for test-driven refactoring automation, ACM SIGSOFT Software Engineering Notes Volume 34 , Issue no 1, January, 2009

[9] G. Meszaros and M. Fowler, xUnit Patterns: Refactoring Test Code, Addison-Wesley, 2007

[10] B. Marick, Testing for Programmers, Lecture Notes available at: http://www.exampler.com/testing-com/writings/half-day-programmer.pdf

[11] J. U. Pipka, Refactoring in a "test first"-world. In Proceedings of 3rd Int'l Conference on eXtreme Programming and Flexible Processes in Software Engineering, 2002

[12] J.Link and P. Frohlich, Unit Testing in Java: How Tests Drive The Code. Morgan Kaufmann ,2003

[13] W. Basit, F. Lodhi and U. Bhatti, Extending Refactoring Guidelines to Perform Client and Test Code Adaptation, In Proceedings of XP 2010, pp. 1-13.

[14] T. Mens and T. Tourwe´, "A Survey of Software Refactoring," IEEE Trans. Software Eng., vol. 30, no. 2, pp. 126-139, Feb. 2004.

[15] M. MirzaAghaei, F. Pastore and M. Pezz, Automatically repairing test cases for evolving method declarations. In Proceedings of ICSM 2010,pp. 1-5

[16] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. Reassert: Suggesting repairs for broken unit tests. Proceedings of the 24th IEEE/ACM international Conference on Automated Software Engineering. IEEE/ACM, 2009.

[17] B. Daniel, T. Gvero, and D. Marinov, On test repair using symbolic Execution, In Proceedings of International Symposium on Software Testing and Analysis, 2010.

[18] J. Henkel and A. Diwan. CatchUp!: Capturing and replaying refactorings to support API evolution.In Proceedings of ICSE'05, pp. 274–283.

[19] A. Zaidman, B. V. Rompaey, S. Demeyer and A. van Deursen. Mining Software Repositories to Study Co-Evolution of Production and Test Code. In Proceedings of 1st International Conference on Software Testing (ICST'08), pp. 220-229, IEEE Computer Society, 2008

[20] M. Streckenbach and G. Snelting. Refactoring class hierarchies with kaba. In Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 315–330. ACM, 2004.

[21] D. Dig, S. Negara, V. Mohindra, and R. Johnson. Reba: refactoring aware binary adaptation of evolving libraries. In Proceedings of the 30th international conference on Software engineering, pp, 441–450. ACM, 2008.

[22] JUnit, At "http://www.junit.org"

[23] Sun Microsystems, I. (2011). Netbeans ide. At http://www.netbeans.org/.

[24] Eclipse.org (2011). Eclipse project. At http://www.eclipse.org.

[25] Jet Brains, I. (2011). Intellij idea. At http://www.intellij.com/idea/.

[26] Embarcadero Technologies, I. (2011). Jbuilder. At http://www.codegear.com/br/products/jbuilder.

[27] quilt.sourceforge.net/tutorials/**junit**.htm

[28] F. Bannwart and P. Müller, Changing Programs Correctly: Refactoring with Specifications. Proceedings of FM pp, 492-507, 2007

[29] E.M. Guerra and C.T. Fernandes, Refactoring Test Code Safely. In Proceedings of the International Conference on Software Engineering Advances , 2007

[30] Different kinds of testing, At css.dzone.com/articles/different-kinds-testing accessed on 5th July, 2012

[31] B. Daniel,D.Dig,K. Garcia and D. Marinov (2007). Automated testing of refactoring

engines. In Foundations of Software Engineering, pages 185–194.

[32] G. Soares, R. Gheyi, T. Massoni, M. Corn´elio, and D. Cavalcanti, "Generating unit tests for checking refactoring safety," in SBLP, 2009, pp. 159–172

[33] http://docs.codehaus.org/display/SONAR/Metric+definitions

[34] R.C.Martin (2002). Agile Software Development: Principles, Patterns and Practices. Pearson Education. ISBN 0-13-597444-5.

[35] S.Demeyer et al. 2005. The LAN-simulation: A Refactoring Teaching Example. In *Proceedings of the Eighth International Workshop on Principles of Software Evolution* (IWPSE '05). IEEE Computer Society, Washington, DC, USA, 123-134.

[36] Cobertura, cobertura.sourceforge.net/

[37] W. Basit, F. Lodhi and U. Bhatti,Evaluating Extended Refactoring Guidelines,To appear in Proceedings of QUORS 2012.

[38] W. Basit and F. Lodhi , Preservation of Externally Observable Behavior after Pull Up 4Method Refactoring, In Proceedings of ICCIT 2012, pp. 309-314.

[39] L.Kiran, F. Lodhi and W. Basit , Test Code Adaptation Plugin for Eclipse, In Proceedings of ICCIT 2012, pp. 297-302.

[40] W. Basit, F. Lodhi and M.U. Bhatti,Unit Test: A specialized client in refactoring ,To appear in Proceedings of ICSOFT 2012.

[41] Instability, http://codenforcer.com/instability.aspx accessed on: 5th July, 2012