Enhancing Unit Tests in Refactored Java Programs

Anna Derezińska[a] and Olgierd Sobieraj
Warsaw University of Technology, Institute of Computer Science, Nowowiejska 15/19, Warsaw, Poland

Keywords: Code Refactoring, Test Maintenance, Unit Tests, Eclipse, IntelliJ IDEA, Java, JUnit.

Abstract: Refactoring provides systematic changes to program code in order to improve its quality. These changes could also require modifications of unit tests associated with a refactored program. Developer environments assist with many code refactoring transformations, which also support some modifications of the tests. Two popular environments for Java programs have been found to be unable to update these tests for all refactoring in a satisfactory way. The flaws in refactoring, the adaptation of the tests after refactoring, and possible improvements were discussed. A tool extension has been introduced to integrate with a refactoring in the Eclipse environment and maintain the corresponding tests. For selected refactorings, additional test cases could also be created to increase code coverage and improve the testing of a refactored program. Experiments have been conducted to evaluate the proposed solutions and verify their limitations.

1 INTRODUCTION

Refactoring of programs is one of the important activities in software development and maintenance (Fowler, 2018), (Mens and Tourwe, 2004).

Refactoring is associated with program testing in different ways. First, tests are essential for verifying the refactoring. However, refactoring a program can cause changes in the structure of the program. Therefore, unit tests associated with the refactored area can become outdated (Gao, et al., 2015). They require modification according to the completed refactorings. Moreover, new tests can supplement the refactored code if it was not covered by tests. In this paper, we focus on the latter problem: the maintenance of tests in refactored programs.

We have compared the realization of code refactoring in two commonly used environments that support refactoring of Java programs: IntelliJ IDEA (IntelliJrefactoring, 2023) and Eclipse (Eclipse – Refactoring, 2023). Many refactorings have been found to update the tests in a satisfactory way. However, in several cases, the tests were not modified or could provide errors. For those refactorings that do not modify tests or make insufficient adaptations, refactoring enhancements were proposed.

To solve this problem, we developed a prototype tool, called RefactorPlugin, that extends the Eclipse IDE. It has been designed to act transparently for a user and automatically update tests after a refactoring, if necessary. Besides, in selected refactorings, additional tests are created to cover the refactored code and improve the program testing. We conducted experiments to evaluate the test enhancements and discuss the limitations of the solution.

The paper is structured as follows. The next section gives an overview of the background and related work. In Section 3, we discuss the impact of refactorings in two Java IDEs on tests and propose modifications to the test cases. The main features of RefactorPlugin are presented in Section 4. In Section 5, we discuss the experimental evaluation of the approach. Finally, Section 6 concludes the paper and presents future work.

2 BACKGROUND AND RELATED WORK

We can identify four different ways in which the refactoring can be combined with a program tests.

1. A subset of tests can be used to verify a refactoring transformation (Fowler, 2018).
2. Refactoring of the code has an impact on the code of several tests that should be adjusted in the corresponding way (Gao, et al., 2015).

[a] https://orcid.org/0000-0001-8792-203X

Derezińska, A. and Sobieraj, O.
Enhancing Unit Tests in Refactored Java Programs.
DOI: 10.5220/0011997800003464
In *Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2023)*, pages 734-741
ISBN: 978-989-758-647-7; ISSN: 2184-4895

3. During refactoring, new code units (classes, fields, and methods) could be created, and therefore, new unit tests could be built that are devoted directly to those code units.
4. Specialized refactoring transformations can be designed and applied directly to the code of test cases (Garousi and Kucuk, 2018).

Experimental verification of any transformation could be based on checking some invariant conditions. These invariants could be specified as tests that a program should pass before and after a refactoring transformation. Only selected tests of a program satisfy this requirement. Moreover, such tests could be designed aimed at a specific refactoring (Walter and Pietrzak, 2004).

Code refactoring can treat tests as any other code module of the project. This kind of test modification could be sufficient in some cases, e.g., for the *rename* refactoring. However, in many cases, additional maintenance of unit tests is necessary to adjust to the change of code structure; otherwise, the tests become obsolete (Gao, et al., 2015).

In general, test generation can give quite satisfactory results (Ramler, Klammer, and Buchgeher, 2018), (Olsthoorn, 2022). Though, application of generated tests can be associated with certain drawbacks, such as a higher maintenance cost, a larger number of tests than those manually created, and difficulties in including expert knowledge (Shamshiri et al., 2018). However, the number of generated tests is limited, their creation is triggered by a refactoring, and they are bound to the elements involved in the refactoring.

Unit test cases, like any other code, could be analyzed and refactored. There is wide research on the identification of specialized smells in tests and on the improvement of tests (Garousi and Kucuk, 2018).

In this paper, we aim only at the second and third ways of combining refactoring with program tests.

There are different tools that support code refactoring in an automated way. However, as shown also in recent research (Eilertsen and Murphy, 2021), developers still have problems with their basic usability. One of these directions is automation, both in the identification of refactoring needs and in refactoring itself, as proposed with Spartanizer developed in Eclipse (Gil and Orrù, 2017).

On the other hand, developers often disregard simple refactoring of test cases that could be completed by existing tools (Aniche, Treude, and Zaidman, 2022).

The impact of refactoring on tests was investigated in experiments (Peruma, et al., 2022). In a study on three Java projects and regression tests conducted by (Rachatasumrit and Kim, 2012) was reported that only 22% of refactored methods and fields were covered by regression tests. The need for automated refactoring validation and test update was suggested due to failed tests after refactoring.

Refactoring in eight open source projects was identified using RefactoringMiner, and logs of test execution were investigated (Kashiwa, et al., 2021). It was observed that most refactoring operations do not break tests, but occasionally small fixes are required. The problems mainly referred to *add parameter*, *change parameter type*, and *change return type*, i.e., changing a method signature.

There were attempts to extend the refactoring tools to improve tests influenced by refactoring.

In (Kiran, 2011), TAPE: Test Code Adaptation Plug-in for Eclipse was reported. It supports only four transformations (*move method, inline method, pull up method,* and *rename method*) and helps in synchronizing tests after refactoring. Client code adaptation is not incorporated in TYPE. Moreover, the tool works on legacy versions of Eclipse.

An approach based on the detection of changes in the AST was presented in (Passier, Bijlsma, and Bockisch 2016). It was implemented in a prototype for Eclipse. However, the performance could be questionable if many other changes in the code are performed, not just refactoring.

Another tool combined with Eclipse was presented in (Jaradat and Qusef, 2019). A current version of the tool GreenRef supports three types of refactoring: *rename method*, *add parameters*, and *remove parameters*. It was shown that it noticeably saved development time compared to manual test maintenance. The automated maintenance of tests was based on running tests after refactoring, detection of misused tests, and their recovery.

3 HOW CODE REFACTORING DOES INFLUENCE TESTS?

In common practice, a set of unit tests is associated with a program. After a refactoring transformation had been realized, the program code would not only be changed, but in many cases, the corresponding tests could be automatically modified.

When using tool support, a preview of the code transformation could be presented to a user before performing a refactoring. In the interactive mode, there are usually windows that show the code before and after refactoring, where the changes can be observed and accepted. However, the implied test modifications are not shown altogether, even though, they are also completed immediately after the refactoring has been accepted.

3.1 Modifications of Tests

IntelliJ IDEA and Eclipse are popular IDEs that support refactoring of Java programs. We compared all available refactorings and investigated their impact on unit tests. In the analysis and experimental verification of the impact, unit tests implemented in JUnit5 were used (JUnit, 2023).

The general findings of our research are summarized in Table 1. In columns 'Eclipse' and 'IntelliJ_Idea', a refactoring is indicated by '+' if it is supported in the IDE, accordingly. The column 'Test modified' includes 'Yes' if the corresponding unit tests were automatically modified after completion of the refactoring mentioned in this row. If a refactoring is implemented in both IDEs, then both tools also modify the tests. If a refactoring is implemented only in one IDE, then 'Yes' confirms the modification of the test using this tool only. The value 'No' stands for tests that have not been modified in the tool(s) that implement the refactoring. An 'Err' sign indicates situations that cause errors in tests.

Based on our research, we list the details of the automated modifications in tests that are caused by code refactorings in the IDEs (1-17).

While considering the potential influences on tests, we found that, for selected refactorings, the tests they affect need some improvements. These cases are denoted with Roman numerals in the last column of Table 1. Below, we present our recommendations for test improvements (I-IX) that should be complemented. Some recommendations were evaluated using our tool (Sects 4 and 5).

1. *Rename* – a name is changed in all places where a code item is declared, defined, or referenced; so also in the test code is fixed automatically.
2. *Move – Move field/method* moves a code element and automatically adjusts the test code to the movement of the element.
3. *Pull up* (I) - if a field or a method is moved to a base class, a test is automatically modified. However, there is a problem with duplicated fields or methods being pulled up to the upper classes in the inheritance tree. The tools give an opportunity to block such transformations, but this could be skipped by a developer. These duplicates cause errors in tests. Therefore, the situation should be identified and avoided.
4. *Push down* (II) - in order to modify tests, moved methods and fields have to be located, and their classes should be changed into the current ones. Additionally, fields and methods in subclasses could be duplicated due to Push down. The tools generate warnings about this, but they could be ignored by a user. Therefore, in tests, a class that uses a field or method should be substituted by a current one.
5. *Extract* (III) - extracted methods do not have their own dedicated tests; hence, they could not have to be modified. But, new tests could be created to cover these new methods.

Table 1: Update of tests after code refactoring.

Reafactoring		Eclipse	IntelliJ_Idea	Tests modified	Recommended test update
1	Rename	+	+	Yes	-
2	Move	+	+	Yes	-
3	Pull up	+	+	No/Err	I
4	Push down	+	+	No	II
5	Extract	+	+	No	III
6	Change method signature	+	+	No	IV
7	Convert local variable to field	+	-	No	V
8	Inline method	+	+	Yes/Err	VI
9	Encapsulate fields	+	+	Yes	-
10	Introduce parameter	+	+	No	VII
11	Find and replace code duplicates	-	+	Yes	-
12	Generalize declared type	+	-	No	VIII
13	Replace constructor with builder	-	+	Yes	-
14	Infer generic type arguments	+	-	Yes	-
15	Replace temp with query	-	+	Yes	IX
16	Remove middleman	-	+	Yes	-
17	Wrap method return value	-	+	Yes	-

6. *Change method signature* (IV) - after changes to method arguments, the method calls in tests are not changed accordingly. Null values are substituted instead of values with specific types. Therefore, a test should be modified in two cases: (i) the return type of a method was changed, and it has to be modified consistently in tests; (ii) the number and types of arguments of a method were changed; then the method in tests should be called with the consistent types of arguments and random values.
7. *Convert local variable to field* (V) - based on a local variable, a field is created. Local variables are not used in tests, that is way, they are not spoiled. But new fields could be created that are not covered by the tests. Therefore, the tests should be extended or additional tests created.
8. *Inline method* (VI) - the automated modification of tests could provide errors in both IDEs. Inline method results in substituting calling a method by the method body. Therefore, the code could be simplified, and some unnecessary methods could be removed. The refactored tests could be erroneous when private fields encounter; hence, they should be corrected in order to compile the tests.
9. *Encapsulate fields* – the getter and setter methods of fields are created. The direct access to these fields is then substituted by the methods. This kind of substitution is also made in the tests in an automated way.
10. *Introduce parameter* (VII) – a new parameter is introduced to a method signature. Therefore, a new undeclared parameter can appear in tests. After refactoring, calls of the refactored method should be localized in tests and modified by extending the method with a new parameter and assigning a random value.
11. *Find and replace code duplicates* – the refactoring removes duplicated code, which is replaced by a method that has the same body. As a result, the method signatures remain unchanged, and the modifications in tests are not provided and not required.
12. *Generalize declared type* (VIII) – a type is generalized in the given code place only. When a generalized field appears in a test or a method returns a generalized type, then this field or type should be adapted to the generalized type.
13. *Replace constructor with builder* – this replacement of a constructor follows changes in all places where the considered objects are created. In IntelliJ IDEA, the appropriate modifications are automatically provided in any code, including the test cases.
14. *Infer generic type arguments* – the generic type is automatically introduced in all corresponding code places, including the tests.
15. *Replace temp with query* (IX) - creation of a method that substitutes an equation could cause problems. Therefore, the arguments of the method used in the tests should be adjusted to the refactored code.
16. *Remove middleman* –apart of the refactoring item, other classes are adjusted accordingly. Therefore, the tests are automatically modified.
17. *Wrap method return value* – an additional wrap class is created during refactoring. This class is also automatically used in the corresponding places in the code, including tests.

3.2 Creation of New Tests

The improvement of tests for a refactored code could not be limited by modification of existing tests. In some refactorings, new classes, fields, or methods are created that were not tested before the refactoring. Therefore, we also suggested creation of additional basic tests that are related to selected refactorings and are based on the following principles.

A new test on a public field uses the assignment of a variable of a consistent type. Then the field value is compared to the correct value. In a unit test, the *assertEquals* assertion is used (JUnit, 2023).

In new tests focused on methods, a method with appropriate arguments is called. It is checked whether no unexpected exception(s) is thrown, using the *assertAll* or *assertDoesNotThrow* assertions.

On request, such new tests could be created for the following refactorings (the numbers match those in Table 1):

- *Pull up* (I): If a field or method moved to a super class is not tested, an object of the super class is created in a new test. In addition, the field is verified or the method is tested *(assertAll*).
- *Push down* (II)*:* Apart from correction of tests, this refactoring can be associated with new tests. An object of a subclass is created and either a field or a method is verified (*DoesNotThrow*).
- *Extract* (III): A new method is created, therefore, a dedicated test is build *(assertAll*).
- *Change method signature* (IV): If the changed method was not used in the tests, a new one is created (*DoesNotThrow*).

- *Convert local variable to field* (V)*:* After refactoring, a new field is created that is tested with a variable assignment if it is public (assertion *Equals*). Additionally, if the method with this field was not tested, a new test is built (*DoesNotThrow*).
- *Introduce parameter* (VII): If the modified method is not used in tests, a new test could be created (*DoesNotThrow*).
- *Generalize declared type* (VIII)*:* If a new public field is created in the refactoring, a new test can check the assignment of a variable to the field newly created. If the generalize refers to a type returned by a method, in a new test, the method should be called, and lack of an exception verified (*DoesNotThrow*).

4 REFACTORPLUGIN TO SUPPORT TESTING

The test modification could be combined with the refactoring in different ways. Tests of a refactored program could be maintained after a whole development session, periodically, or on demand. Alternatively, each refactoring that is accepted by a developer could trigger an automatic reaction.

In the latter approach, a test suite could be modified without disturbing the activity of a developer. A disadvantage is that the modification only takes into account a single refactoring, while sometimes a series of correlated ones is made.

Apart from the modification of existing tests, a tool could also create new tests if the refactored area is not covered by unit tests. The tests could be added after a series of refactoring transformations or just after each refactoring.

An assumption of our solution was to make the process transparent to a user who refactors a program. Therefore, the following issues were concerned in the plugin development:

1. Detection of an event that triggers the plugin.
2. Identification of the refactoring kind and its attributes.
3. Handling of different kinds of launching a refactoring that could be selected by a user.
4. Modification of the source code and tests.

The Eclipse framework has an open architecture, and its functionality can be extended with a plugin using the Eclipse Plugin Development Environment (PDE). In Eclipse, there are different events that could trigger the plugin:

a) change in the AST,
b) completion of a refactoring,
c) adding a log to the refactoring history.

A change in AST can be easily detected, but it could refer to more than just refactoring. Information about a just completed refactoring is necessary to activate a preferred reaction. This information could be extracted either from an event type detected by a listener of a refactoring execution or from the recent log in the refactoring history.

We have developed the RefactorPlugin for Eclipse, which is launched every time when information about a just completed refactoring is added to the refactoring history. The refactoring details are extracted from the argument passed to the history.

In the plugin, the following requirements were assumed. A test is modified only if a refactoring transformation has a direct impact on the fields or methods used in the test. New tests are created for these fields, methods, and classes that were used in a refactoring but were not tested before. Therefore, the tests ensure basic code coverage. Assertions are used to verify conditions in tests. The recommendations for the following cases were implemented: I, II, III, IV, and VII (the numbers as in Table 1).

The RefactorPlugin consists of three main modules:

- *refactorplugin* – for cooperation with the refactoring transformation,
- *parser* – for processing the source code and building the AST,
- *reafctortests* – responsible for the modification and creation of tests.

In order to effectively access and modify the source code, an AST of the code can be used. There are several tools that support the parsing of Java code, such as ANTLR4 (ANTLR, 2023), JavaParser, coreAST. In RefactorPlugin, parts of the source code are analyzed using the AST created by ANTL4 with the Java grammar. The source code of the tests is modified at the AST level.

5 EXPERIMENTS

The given approach was experimentally evaluated.

5.1 Experiment Setup

The developed RefactorPlugin was applied to a set of programs of various origins (Table 2) and different maturity levels of test development. The three first programs were implemented using Java 13, and the last one with Java 8. The programs were selected to require some refactoring because we want to verify not an artificial but a real development activity. Each program was processed with the following steps:

1. measuring the coverage of lines, statements, and decisions in the code under test,
2. application of a series of refactoring transformations, some of which caused an automated modification or creation of tests by RefactorPlugin,
3. testing with the modified test set,
4. measuring the coverage of lines, statements, and decisions for the modified test set.

Table 2: Projects used in experiments.

Project	Origin
Blackjack	https://vdawg97.github.io/Blackjack/
Simple-Poker	https://github.com/andyxhadji/Simple-Poker
Unit test generator	A project of a course on Compiler Techniques
Inheritance-lesson	https://github.com/jversed/Lesson07_inheritance

5.2 Experiment Results

Selected program measures calculated before and after the experiments are given in Tables 3-6, for each program, accordingly. The numbers of lines and instructions provided in the top rows of a table refer to the whole project, including production code and initial tests (Before) or with all modified and added tests (After). The values of code lines (LOC) are counted without comments and white lines.

Table 3: Blackjack.

Project with tests	Before	After
LOC	319	356
Instruction number	1631	1791
Test number	2	12
Line coverage	10.0%	45.8%
Instruction coverage	5.8%	43.6%
Decision coverage	0%	13.8%
Refactored modules		
LOC	237	250
Line coverage	3.4%	43.2%
Instruction coverage	1.5%	40.9%
Decision coverage	0%	15.7%

1. Change method signature
2. Convert local variable to field
3. Introduce parameter
4. Change method signature
5. Extract method
6. Introduce parameter
7. Extract method
8. Extract method
9. Change method signature
10. Extract method

In the *Blackjack* and *Inheritance-lesson* projects, only subsets of project modules were refactored. Therefore, the bottom rows of their tables include data of only the refactored modules. In the two remaining projects, all modules were refactored.

Below each table, a sequence of refactorings is given that was executed on the program. Any sequence is a real development phase, but others could also be used, as each refactoring was a single step, which means that after it, the tests were modified and/or new ones created. The refactored program with improved tests was an input for the next step.

Table 4: SimplePoker.

Project with tests	Before	After
LOC	214	244
Instruction number	890	1014
Test number	1	9
Line coverage	1.9%	33.6%
Instruction coverage	0.9%	33.0%
Decision coverage	0%	20.3%

1. Extract method
2. Change method signature
3. Convert local variable to field
4. Change method signature
5. Change method signature
6. Extract method
7. Introduce parameter
8. Extract method
9. Extract method
10. Change method signature

Table 5: Unit test generator.

Project with tests	Before	After
LOC	1505	1548
Instruction number	7752	7950
Test number	78	89
Line coverage	93.1%	93.9%
Instruction coverage	92.4%	93.2%
Decision coverage	83.2%	84.7%

1. Introduce parameter
2. Change method signature
3. Convert local variable to field
4. Extract method
5. Change method signature
6. Extract method
7. Introduce parameter
8. Convert local variable to field
9. Convert local variable to field
10. Change method signature
11. Extract method
12. Convert local variable to field
13. Extract method
14. Change method signature
15. Change method signature

Table 6: Inheritance-lesson.

Project with tests	Before	After
LOC	96	104
Instruction number	486	515
Test number	4	19
Line coverage	33.3%	52.9%
Instruction coverage	45.3%	61.0%
Decision coverage	0%	5%
Refactored modules		
LOC	78	83
Instruction number	373	393
Line coverage	20.1%	43.4%
Instruction coverage	29.8%	49.9%
Decision coverage	0%	5%

1. Pull up
2. Push down
3. Convert local variable to field
4. Pull up
5. Change method signature
6. Pull up
7. Push down
8. Push down
9. Extract method
10. Introduce parameter

5.3 Discussion

In experiments, all modifications to the tests existing before a refactoring step were successfully completed. Both kinds of tests were modified: those initially delivered with a project and those created during previous steps of the refactoring sequence. For example, in the *Unit test generator*, 5 tests were modified, while 3 of them were added by the given refactoring sequence (e.g., after the 14th refactoring, a test that was added in the 5th step was modified).

The code coverage of the refactored modules increased considerably, especially for the programs that did not have many tests before (about 20-40%). We can also observe an increase in decision coverage, although it is not as high as line coverage (from a few percent to 20%). However, the modified or new tests were not intended to maximize the decision coverage.

In the *Blackjack* project, each refactoring caused the creation of a new test. We faced a problem with fixed arrays specified in a constructor when tests with random parameters were applied. To avoid this situation, a detailed method analysis and parameter tuning in new tests were necessary.

As a result of the refactoring of the *Inheritance-lesson* project, 15 new tests were created, which is more than the number of steps in the refactoring sequence (10). It was correct due to processing the *push down* refactoring (steps 3, 7, and 8). Pushing an item down in the class hierarchy can move it into several descendant classes. For each such class, a new test was created.

5.4 Threats to Validity

As for threats to validity in the conducted studies, only four programs were evaluated with 10-15 refactorings. Therefore, the sample was quite limited, and the quantitative results could not be generalized (external validity). It also causes low statistical conclusion validity.

Each program was treated by a different refactoring scenario as the selected transformations were adjusted to the program structure and its requirements. Any scenario consists of at least 10 refactoring steps but not all transformations implemented in the RefactorPlugin were applied to each program. However, considering the whole experiment, all kinds of transformations were covered, which helped alleviate the threat to construct validity. Furthermore, all automated modifications to the tests were also manually checked. Any modification is made just after a refactoring step; therefore, we could accept the internal validity.

6 CONCLUSIONS

Analysis of the impact of the code refactoring on unit tests in two Java frameworks showed that in some cases the tests needed to be maintained manually or an additional tool support was necessary. This requirement was partially fulfilled by RefactorPlugin, which extends Eclipse. Moreover, the developed tool can create additional tests, especially focused on items that could not have tests before refactoring.

On the basis of the conducted experiments, we have made some observations. After a refactoring, the modified tests could be correctly applied to the program, assuming the refactoring had not damaged the code. The plugin is activated automatically after a refactoring, and its operation is therefore transparent to a user. The time overhead was not noticeable. Simple new tests required further enhancements in some conditions, otherwise they could fail. This poses the question of whether all such limitations were taken into account, that is, if all new tests pass. As expected, the new tests can improve code coverage.

As future work, we can consider the modification and generation of tests for other refactorings that were not developed in the current version. However, the implementation of some of them, like *Generalize declared type*, requires keeping the program AST before and after each modification, which could degrade the tool's performance.

Support for the refactoring process could be extended, e.g., enhancement of tests could be done not automatically after each refactoring, but on demand after a selected refactoring, after a manual refactoring, or once after a series of refactorings.

More sophisticated test cases could be built or integrated unit test generators, e.g., EvoSuite, Randoop, DSpot (Ramler, Klammer, and Buchgeher, 2018), (Roslan, Rojas, and McMinn, 2022).

REFERENCES

Aniche, M., Treude, C., Zaidman, A., 2022. How Developers Engineer Test Cases: An Observational Study, In: *IEEE Transactions on Software Engineering*. vol. 48, no. 12, pp. 4925-4946. doi: 10.1109/TSE.2021. 3129889.

ANTLR [Online] [Accessed 25 Jan 2023] https://www. antlr.org/, https://github.com/antlr/antlr4-tools

Baqais, B.A.A., Alshayeb, M., 2020. Automatic software refactoring: a systematic literature review. *Software Quality Journal*. 28, pp. 459-502. doi: 10.1007/s11219-019-09477-y.

Eclipse – Refactoring [Online] [Accessed 25 Jan 2023] https://www.tutorialspoint.com/eclipse/eclipse_refacto ring.htm

Eilertsen, A.M., Murphy, G. C., 2021. The Usability (or Not) of Refactoring Tools. In: *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 237-248. doi: 10.1109/SANER50967.2021.00030.

Fowler, M., 2018. *Refactoring: improving the design of existing code* (2nd ed.). Addison-Wesley.

Garousi,V., Kucuk, B., 2018. Smells in software test code: A survey of knowledge in industry and academia. *J. of Systems and Software.* Vol. 138, pp. 52-81. doi: 10.1016/j.jss.2017.12.013

Gil, J., Orrù, M. (2017). The Spartanizer: Massive automatic refactoring. In: *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering* (SANER), pp. 477-481. doi:10.1109/ SANER.2017.7884657.

Gao, Y., Liu, H., Fan, X., Niu, Z., & Nyirongo, B., 2015. Analyzing Refactoring' Impact on Regression Test Cases. In: *IEEE 39th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2, pp. 222-231. doi: 10.1109/COMPSAC.2015.16.

IntelliJ refactoring [Online] [Accessed 25Jan 2023] https://www.jetbrains.com/help/idea/refactoring-source-code.html

Jaradat, A., Qusef, A., 2019. Automatic Recovery of Unit Tests after Code Refactoring. In: *International Arab Conference on Information Technology (ACIT)*, pp.202-208. doi: 10.1109/ACIT47987.2019.8990974.

JUnit [Online] [Accessed 12 Jan 2023] Available from https://junit.org/junit5/docs/current/api/index.html

Kashiwa,Y., Shimizu,K., Lin, B., Bavota, G., Lanza, M., Kamei, Y., Ubayashi, N., 2021. Does Refactoring Break Tests and to What Extent? In: *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp.171-182, doi: 10.1109/ICSME52107. 2021.00022.

Kiran, L., Lodhi, F., Basit, W., 2011. TAPE Test Code Adaptation Plug-in for Eclipse. Corpus ID: 12838995.

Mens, T., Tourwe, T., 2004. A survey of software refactoring. In: *IEEE Transactions on Software Engineering,* vol. 30, no. 2, pp. 126-139. doi: 10.1109/TSE.2004.1265817.

Olsthoorn, M., 2022. More Effective Test Case Generation with Multiple Tribes of AI, In: *IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 286-290. doi: 10.1145/3510454.3517066.

Passier, H., Bijlsma,L., Bockisch, C., 2016. Maintaining Unit Tests During Refactoring. In: *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, no. 18, pp.1-6. doi: 10.1145/2972206.2972223.

Peruma, A., Simmons, S., AlOmar, E.A. et al., 2022. How do I refactor this? An empirical study on refactoring trends and topics in Stack Overflow. In: *Empir Software Eng* 27, 11. doi: 10.1007/s10664-021-10045-x.

Rachatasumrit, N., Kim, M., 2012. An empirical investigation into the impact of refactoring on regression testing. In: *28th IEEE International Conference on Software Maintenance (ICSM)*, pp. 357--366. doi: 10.1109/ICSM.2012.6405293.

Ramler, R., Klammer, C., Buchgeher, G. 2018. Applying automated test case generation in industry: a retrospective. In: *International Conference on Software Testing, Verification and Validation Workshops*, pp. 364-369, IEEE. doi: 10.1109/ICSTW.2018.00074.

Roslan, M.F., Rojas, J.M., McMinn, P., 2022. An Empirical Comparison of EvoSuite and DSpot for Improving Developer-Written Test Suites with Respect to Mutation Score. In: Papadakis, M., Vergilio, S.R. (eds) *Search-Based Software Engineering. SSBSE* 2022. Lecture Notes in Computer Science, vol 13711. Springer, Cham. doi: 10.1007/978-3-031-21251-2_2.

Shamshiri, S., Rojas, J. M., Galeotti, J. P., Walinshaw, N., Fraser, G., 2018. How do automatically generated unit tests influence software maintenance? In: *11th International Conference on Software Testing, Verification and Validation*, pp.250-261. IEEE Comp. Soc. doi: 10.1109/ICST.2018.00033.

Shochat M., Raz O., Farchi, E., 2009. SeeCode – A Code Review Plug-in for Eclipse. In: Chockler H., Hu A.J. (eds) *Hardware and Software: Verification and Testing*. HVC 2008. LNCS, vol 5394. Springer, Berlin, Heidelberg. doi:10.1007/978-3-642-01702-5_21.

Walter, B., Pietrzak B., 2004. Automated generation of unit tests for refactoring. In: *Extreme Programming and Agile Processes in Software Engineering*, LNCS vol. 3092, pp. 211--214. Springer. doi: 10.1007/978-3-540-24853-8_25.