Proceedings of the 18th Conference on Computer Science and Intelligence Systems pp. 937–941
DOI: 10.15439/2023F7277
ISSN 2300-5963 ACSIS, Vol. 35

Review of Automated Code Refactoring of C# Programs

Anna Derezińska
0000-0001-8792-203X
Warsaw University of Technology
Institute of Computer Science
Nowowiejska 15/19,
00-665 Warsaw, Poland
Email: A.Derezinska@ii.pw.edu.pl

Dawid Sygocki
Warsaw University of Technology
Institute of Computer Science
Nowowiejska 15/19,
00-665 Warsaw, Poland,
Email:
dawid.sygocki.stud@pw.edu.pl.

***Abstract*—Code refactoring is supported by many Integrated Development Environments. This paper is focused on the automated code refactoring of C# programs. We have analyzed more than sixty refactorings available in three popular IDEs. We cataloged different restrictions, defects, and other quality concerns associated with the implementation of the refactorings, taking into account both modification of the production code and of the corresponding test cases. An extension to automate selected refactoring improvements has been developed for the ReSharper platform and experimentally verified.**

***Index Terms*—code refactoring, unit testing, code quality, code and test maintenance, C#.**

I. Introduction

REFACTORING techniques have been proposed to improve code quality [1][2]. Their effective application can be assisted with automated tools incorporated into Integrated Development Environments (IDEs). We have examined automated refactoring that can be used in programs written in the C# programming language. We have reviewed three environments commonly used for C# program development, namely: Microsoft Visual Studio, JetBrains Rider, and Visual Studio Code. The following research questions were addressed:

1. Can we rely on automated refactoring, i.e. is the final code always correct?
2. Are the unit tests corresponding to the refactored area adjusted together with the refactoring completed?
3. Can we, in an automated way and transparently to a developer, fix malfunctions detected in refactoring or add any improvements?

To perform the study, a benchmark program was developed covering code variants consistent with these environments. We have found that some refactorings can be correctly applied only under certain restrictions, while others can produce invalid code. Moreover, the test cases related to the refactored area could be improved.

The main contributions of the paper are the following:

- A review of over 60 automated refactorings of C# programs supported by three popular environments.
- Identification and classification of different restrictions, extensions, defects, and quality concerns associated with the refactoring implementation.
- As a proof of concept, development of Refix, a prototype that fixes defects of selected refactorings of the ReSharper tool-(used in JetBrains Rider 2022 and also as a plugin applied to the Microsoft Visual Studio [3]).

The paper is structured as follows: In the next Section we discuss the related studies. In Section 3, we give an overview of the refactoring consequences for code and tests in three popular environments. A developed tool (Refix) is briefly presented in Section 4. In Section 5, we conclude the paper.

II. Related Work

There exist many tools that help in automated refactoring in software development and maintenance. Though, their usage still causes difficulties for developers, as reported in [4].

Deficiencies in the refactoring tools were mainly studied for Java and C programs. Testing of refactoring engines [5] found 1.4% of refactoring tasks failing for Java and 7.5% for C. In [6], the problems of name binding and accessibility rules in refactoring are discussed. Some problems could be similar, but there are no studies of C#.

Another problem of refactoring implementation is its impact on the test cases [7]. Many experiments related to the test maintenance of refactored programs were performed in Java [8], [9], [10]. In [8], it was shown that tests often require additional handling when the production code is refactored. To handle this, a prototype developed in Eclipse was discussed in [9]. In [10], refactoring of Java programs with JUnit tests was examined. Various flaws in updating tests were identified. RefactorPlugin was developed to correct selected defects in tests and create additional tests in accordance with the refactoring performed.

Refactoring in C# programs was studied as one of the aspects to explore similarities and differences between test and production classes [11]. It was found that while production classes underwent more changes, the maintenance of tests

caused major problems. In the thesis [12], an extension to Visual Studio was developed, but the author focused on a new kind of refactoring, not on fixing the existing ones. To the best of our knowledge, improvements in the C# code and tests after refactoring were not considered.

The impact of using the ReSharper tool on the results of test runs, builds, and version control commands was examined in [13]. Experiments on Enriched Event Stream Dataset led to a higher rate of failure builds and higher percentage of commits. ReSharper was also studied in some research and was claimed to be popular among developers [14].

III. Review of Refactoring

In the refactoring review, we have checked whether an automated refactoring provides a valid transformation of the code and whether the corresponding unit tests are correctly modified. We focused on three different IDEs that support refactoring of C# programs:

1) Refactoring embedded in Microsoft Visual Studio 2022, i.e. MVS without additional extensions.
2) JetBrains Rider 2022, and the same refactoring engine used in the ReSharper plugin applied to MVS [3][15].
3) Visual Studio Code with an addition to support the C# language (ms-dotnettools.csharp).

A. Benchmark for Refactoring Review

Analysis of the effects of refactoring was assisted by a benchmark program [16]. We have developed it to cover a set of refactoring examples and the corresponding unit tests. The production code was developed in three versions related to IDEs mentioned above.

The benchmark also encompasses three variants of a set of unit tests corresponding to the popular unit test frameworks that support C#: MSTest, NUnit, and xUnit.net.

B. Comparison of the Refactoring Capabilities

We have reviewed the way of implementation of all refactorings that were supported in the mentioned environments. As a result, we made recommendations on many refactorings. They were classified into the following four categories:

1. **Extension (E):** the refactoring implementation is extended in comparison to its basic meaning [1].
2. **Restriction (R)**: the implementation of the refactoring is restricted compared to its basic meaning.
3. **Defect (D)**: a fault was detected in the refactoring implementation that should be fixed, as it causes the project not to compile. This situation is often associated with a “conflict”, i.e. a warning reported by an IDE after an attempt of a refactoring.
4. **Quality concern (Q)**: a shortcoming occurs that does not cause a compilation error or another improvement to the code and tests could be suggested.

In Tables I and II, we summarize all refactorings implemented in three environments. The last three columns correspond to Microsoft Visual Studio - MVS, JetBrains Rider (also ReSharper) – R/R, and Visual Studio Code – VSC, accordingly. The sign ‘-’ denotes that the refactoring is not supported in the environment. If the refactoring was implemented, the character ‘+’ shows that no recommendations were related to it. Otherwise, a combination of letters (E, R, D or Q) signifies the recommendation categories associated with the refactoring. The assigned recommendations are discussed in the subsequent subsections.

C. Review of Refactoring in Visual Studio

Here, we deal with the refactorings embedded directly in MVS 2022. Those supported by the Resharper extension, often applied to MVS, are discussed in the next subsection.

Restrictions (R):

- In #43 and #44. The move method and move field refactoring is limited to static members.

Defects (D):

- In #2. In synchronization of a namespace and a folder name, the `using` directive in tests could be not updated. The situation was rare and this defect can be treated as a minor one.
- In #14. In the conversion between a property and `get` method, the refactoring does not consider references to the transformed property present in the object initialization. In this case, a warning is shown.
- In #43 and #44. It refers to moving a field or a method. A static class can be indicated as an `internal` one. Therefore, its members are not accessible in the test project. One of the popular conventions is the application of an attribute to make these internal types accessible and structural testing possible. Hence, the improvement of this defect is of low priority.

Quality improvements (Q):

- In #39 and #40. When a method or a field is pulled up to the base class, other descendant classes could be checked in terms of these member occurrences.
- In #39, #40, and #54. While a method or a field is pulled up, or when a superclass is extracted, we could consider using the base class where possible. The refactoring can be followed by another refactoring #38.

D. Review of Refactoring in ReSharper

Among the solutions discussed, the ReSharper engine, used in JetBrains Rider and as an extension in MVS, provides the largest number (64) of automated refactorings. Below, we list the recommendations passed on to these refactorings.

Extensions (E):

- In #31. The change of signature refactoring can be applied not only to a method but also properties, indexers and constructors. Moreover, two alternative forms of the transformation are available: *change signature* and *transform parameters*.

Restrictions (R):

- In #8. The refactoring to make a member static can be completed only if the method takes an instance as a parameter.

TABLE II.
COMPARISON OF AUTOMATED REFACTORING FOR C# PROGRAMS

No	Refactoring	Environment		
		MVS	R/R	VSC
1	Safe delete	-	Q	-
2	Sync namespace and folder name	D	+	+
3	Sync a type and filename	+	+	+
4	Convert an abstract class to interface/vice versa	-	+	-
5	Convert anonymous type to class	+	+	+
6	Convert anonymous type to tuple	+	-	D
7	Convert extension method to plain static/vice versa	-	+	-
8	Make member static	+	R	-
9	Use expression body or block body for lambda expression	+	+	-
10	Convert anonymous function to local one	-	+	-
11	Convert local function to method	+	+	+
12	Make local function static	+	-	-
13	Replace constructor with factory method	-	+	-
14	Convert `get` method to property/ and vice versa.	D	D	D
15	Convert method to indexer/vv.	-	+	-
16	Convert between auto property and full property	+	+	R
17	Encapsulate field	+	+	Q
18	Replace loop with pipeline	+	+	+
19	Convert between `for` loop and `foreach` statement	+	+	+
20	Simplify LINQ expression	+	+	-
21	Convert between regular string and verbatim string literals	+	+	+
22	Simplify string interpolation	+	+	-
23	Use pattern matching	+	+	-
24	Convert `if` statement to `switch` statement or expression	+	+	R
25	Convert `switch` statement to `switch` expression	+	+	-
26	Split or merge `if` statements	+	+	+
27	Simplify conditional expression	+	+	-
28	Use explicit type	+	+	-
29	Use `new()`	+	+	-
30	Copy type	-	Q	-
31	Change signature/Transform parameters	+	ED Q	-
32	Add `null` checks of parameters	+	+	E
33	Introduce parameter	+	+	+
34	Introduce parameter object	-	+	-
35	Invert conditional expressions and AND/OR operators	+	+	+
36	Invert `if` statement	+	+	+

TABLE I.
COMPARISON OF AUTOMATED REFACTORING (CONTINUATION)

No	Refactoring	Environment		
		MVS	R/R	VSC
37	Invert Boolean	-	Q	-
38	Use base type where possible	-	+	-
39	Pull up method	Q	Q	-
40	Pull up field	Q	Q	-
41	Push down method	-	D	-
42	Push down field	-	D	-
43	Move method	RD	R	-
44	Move field	RD	+	-
45	Move a type to a matching file	+	+	+
46	Move to folder	-	+	-
47	Move type to another namespace	-	+	-
48	Remove dead code	+	+	-
49	Remove unused references	+	-	-
50	Extract method	+	Q	+
51	Inline method	+	Q	-
52	Extract class	+	R	-
53	Inline class	+	R	-
54	Extract superclass	Q	+	DQ
55	Extract interface	+	+	+
56	Extract members to partial class	-	+	-
57	Introduce field	-	+	-
58	Wrap, indent and align	+	+	+
59	Sort `using` declarations	+	+	-
60	Introduce local variable	+	+	+
61	Move declaration near reference	+	+	+
62	Rename	+	+	+
63	Change member or internal type visibility to public/ internal/ protected/ private protected/ private	-	D	-
64	Change type visibility to public/ internal	-	+	-
65	Change to virtual/ non-virtual	-	D	-
66	Change to abstract/ non-abstract	-	D	-
67	Make method override/Add `new` keyword	-	+	-

- In #43. The refactoring of move method is restricted, because the target class has to be a parameter of the moved method. It does not pertain to static methods.
- In #52. The extract class refactoring includes only a variant in which the reference to the new class remains in the old one. The test cases remain unchanged, although they could have been modified by changing the subject of the tests to the new class.
- In #53 In the inline class refactoring, the absorbed class is required to include reference to the target class.

In the following cases, a refactoring could provoke erroneous behavior.

Defects (D):

- In #14. When converting between a property and `get` method, references to the transformed property that are present in the object initialization are invalid. To signal this problem, the refactoring tool creates a warning.
- In #31. In the refactoring of transform parameters, the transformation of a method with an `out` parameter to an expression is incorrect. In the method body, an assignment of the already missing parameter exists, and therefore the program cannot compile.
- In #41 and #42. After the push down refactoring, the tests of the modified base class do not compile any more. They should also be refactored and moved to the appropriate descendant class to which a method or a field was pushed down. The analogous problem is in the pull up refactoring.
- In #63. In the refactoring that changes a member or internal type visibility, the corresponding tests are not modified accordingly. In dependence of the qualifier used, `private` in particular, the final project could not compile.
- In #65 and #66. After the addition or deletion of the `virtual` or `abstract` modifiers, the corresponding tests are not updated. They either need to be deleted or adjusted by changing their subject.

Other improvements in refactoring (Q) could be suggested:

- In #1. In the refactoring of safe delete, several lines with references to a deleted element are also removed. Therefore, it could be beneficial to remove empty or pointless corresponding tests.
- In #30. Refactoring the type copy could also require modification of the corresponding tests. They could either be duplicated or enhanced by applying parametrized tests according to the refactored type.
- In #31. Depending on the details of the change signature refactoring, we could update the tests. For example, if a list of parameters was shortened, some variables could be deleted from a test case; if a parameter was added, a variable with a default value could be introduced in a test case, etc.
- In #37. If a Boolean value is inverted, an Assert could be changed to do tests more legible, e.g., substitute `Assert.True(!value)` with `Assert.False (value)`. However, the exact behavior could be different in dependence on the test library used.
- In #39 and #40. This refers to tests after a method or a field pull up. For example, the corresponding tests could be moved to the base class if they have no dependencies on the original class and if the base class is not an abstract one.
- In #50. After an extract method refactoring, a new method appears. The creation of new test cases, e.g., automated test generation, could be considered.
- In #51. After applying an inline method, the tests of the method remain and their code is merged with the tested method. Consequently, a code duplication encounter. The useless tests could be deleted.

E. Review of Refactoring in Visual Studio Code

The number of refactorings supported by Visual Studio Code (VSC) was the smallest among the three environments. As refactoring variants, two restrictions and one extension were identified.

Extension (E):

- In #32. When parameters are of the `string` type, additional checks could be applied using the `IsNullOrEmpty` method.

Restrictions (R):

- In #16. In conversion between an auto and full property, only one direction of the conversion is supported. An auto property can be converted to a full property, but the vice versa transformation is not possible.
- In #24. Conversion from the `if` instruction to the `switch` instruction or expression is restricted. It can only be applied in cases where relations in consecutive `if` statements refer to the same variable.

The following defects (D) were recognized:

- In #6. In conversion from an anonymous type to a tuple, if an object table that has a transformed anonymous type is created using the shortened inscription `new[]` then a compilation error occurs. This could be avoided by using the full description in the form `new object[]` or by casting to the object type for one of the initialization elements.
- In #14. When converting between a property and a `get` method, references to a refactored property that are used in an object initialization are not updated.
- In #54. The extract superclass refactoring does not take into account dependencies on other fields or methods. If in an extracted method, a reference to a member of the original class exists, a compilation error can arise.

Two quality improvements (Q) were recommended:

- In #17. When a field is encapsulated, visibility of the resultant field is always set to `private` and of a property set to `public`. The transformation does not take into account the initial modifiers.
- In #54. After extracting a superclass, a new base type is used instead of its descendant, where possible.

F. Summary of the Review

In general, the level of refactoring correctness in the environments is similar. Calculating the ratio of the number of refactorings classified as a defect (D) to the number of all refactorings supported by the IDE we obtained 9% for the pure MVS, 11% for JetBrains Rider (and the same for ReSharper), and finally 12% for VSC.

We observed that some defects are specific to selected environments, while others are common to different IDEs. For example, the same problem of converting a property and a `get()` method in #14, occurs in all three tools. Furthermore,

similar quality issues refer to pull up refactoring (#39, #40) in the environments in which they are implemented.

The recommendations relate to the modification of the production code but also to the corresponding unit tests. Considering the defects identified in Rider/ReSharper, 5 out of 7 defects refer to test cases. Moreover, all 8 quality issues considered in this environment suggest improvements in the tests.

In summary, the first and second research questions addressed in the Introduction, have negative answers. We cannot always rely on automated refactoring, and tests associated with the refactored area are often inadequately handled.

IV. Automated Fixing of Refactoring Defects (Refix)

Based on the analysis provided, we propose an approach to extend automated refactoring. The tool should correct selected defects and improve the quality of code and tests.

We have selected the ReSharper tool to be enhanced. This platform supports a large number of refactorings, can be used in at least two popular environments, and can be extended with plugins. As a proof of concept, an extension Refix was designed and implemented [16].

The Refix extension integrates with the ReSharper platform and can react when a refactoring is executed. If required, an additional "fixing" activity is undertaken. Currently, it supports improvements of defects related to refactorings #14, #31, #41, and #42, as described in Sect.III.D.

Refix has been tested on JetBrains Rider 2022.1.2 and MVS 2022 Community Edition with ReSharper 2022.1.2.

The benchmark developed to assess refactoring in different environments was also used in the evaluation of the Refix tool (Sect.III.A). It was run with Refix in both environments mentioned above. Unit tests from all three test libraries were used in experiments. Refactoring with Refix was completed correctly, according to expectations.

Furthermore, the evaluation of Refix was based on three real programs derived from the GitHub platform [16]. The experiments were carried out using JetBrains Rider. Unit tests of the projects were run with the xUnit.net framework.

The detailed description of the prototype, and its experimental evaluation are beyond the scope of the paper.

V. Conclusion

After analyzing automated refactorings of C# supported in three popular IDEs, we have made a set of recommendations. Several malfunctions were recognized that influenced the resulting code and tests. In particular, some refactorings deliver code that does not compile. The environments significantly differ in the number of supported transformations, but their general realization quality is at a similar level. Moreover, we have recognized the same or similar problems that referred to the same refactorings in different environments.

Three frameworks for unit tests (MSTest, NUnit, and xUnit.net) were applied in all considered environments. We have not noticed any differences in using the frameworks, as far as the problems of refactored programs are concerned.

A prototype tool has been developed to automate code repair after refactoring [16]. The Refix plugin can be used in JetBrains Rider or MVS with the ReSharper extension. The tool was evaluated using the benchmark and some real programs from GitHub. Due to the prototype developed and its preliminary evaluation, we could positively answer the third research question. In the future, the tool could be extended to cover the remaining defects and other quality concerns.

References

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

[2] A. A. B. Baqais and M. Alshayeb, "Automatic software refactoring: a systematic literature review," *Software Quality Journal*, vol. 28, 2020, pp. 459-502, http://dx.doi.org/10.1007/s11219-019-09477-y

[3] "ReSharper: The Visual Studio extension for .NET developers by JetBrains," 2023, https://www.jetbrains.com/resharper/, [Online, Accessed 20 Jan 2023]

[4] A. M. Eilertsen and G. C. Murphy, "The usability (or not) of refactoring tools," in *Proc. IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 237-248. http://dx.doi.org/10.1109/SANER50967.2021.00030

[5] M. Gligoric, F. Behrang, Y. Li, J. Overbey, M. Hafiz, and D. Marinov, "Systematic Testing of Refactoring Engines on Real Software Projects," in *Proc. ECOOP 2013 – Object-Oriented Programming*, LNCS vol 7920. Springer, Berlin, Heidelberg. 2013, pp. 629–654, https://doi.org/10.1007/978-3-642-39038-8_26

[6] M. Schäfer, A. Thies, F. Steimann, and F. Tip, "A Comprehensive Approach to Naming and Accessibility in Refactoring Java Programs," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, 2012, pp. 1233-1257, http://dx.doi.org/10.1109/TSE.2012.13

[7] Y. Kashiwa, K. Shimizu, B. Lin, G. Bavota, M. Lanza, Y. Kamei, and N. Ubayashi, "Does refactoring break tests and to what extent?" in *Proc. IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp.171-182, http://dx.doi.org/10.1109/ICSME52107.2021.00022

[8] Y. Gao, H. Liu, X. Fan, Z. Niu, and B. Nyirongo, "Analyzing refactoring' impact on regression test cases," in *Proc. IEEE 39th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2, 2015, pp. 222-231, http://dx.doi.org/10.1109/COMPSAC.2015.16

[9] H. Passier, L. Bijlsma, C. Bockisch,, "Maintaining unit tests during refactoring," in *Proc. 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, no. 18, 2016, pp.1-6. http://dx.doi.org/10.1145/2972206.2972223

[10] A. Derezinska, O. Sobieraj, "Enhancing unit tests in refactored Java programs," in *Proc. 18th Inter. Conf. on Evaluation of Novel Approaches to Software Engineering - ENASE*, Scitepress, 2023, pp. 734-741, http://dx.doi.org/10.5220/0011997800003464

[11] M. Gatrell, S. Counsell,S. Swift, R. M. Hierons, and X. Liu, "Test and production classes of an industrial C# system: a refactoring and fault perspective," in *Proc. 41st Euromicro Conference on Software Engineering and Advanced Applications*, 2015, http://dx.doi.org/10.1109/SEAA.2015.40

[12] M. Linka, "Visual Studio refactoring and code style management toolset," M.S. thesis, Charles University in Prague, 2015.

[13] E. Firouzi and A. Sami, "Visual Studio automated refactoring tool should improve development time, but ReSharper led to more solution-build failures," in *IEEE Workshop on Mining and Analyzing Interaction Histories (MAINT)*, Hangzhou, China, 2019, pp. 2-6, http://dx.doi.org/10.1109/MAINT.2019.8666936

[14] S. Amann, S. Proksch, S. Nadi, and M. Mezini, "A study of Visual Studio usage in practice," in *Proc. IEEE 23rd International Conference on Software Analysis Evolution and Reengineering (SANER)*, vol. 1, pp. 124-134, 2016.

[15] "Rider: The Cross-Platform .NET IDE from JetBrains," 2023, https://www.jetbrains.com/rider/, [Online, Accessed 20 Jan 2023]

[16] Refix, https://galera.ii.pw.edu.pl/~adr/Refix/ [Online , Accessed 30 July 2023]