DOI: 10.15514/ISPRAS-2025-37(5)-11

Designing Refactoring Tool for Object-Oriented Code Based on Metrics

[1] *A.O. Korznikov, ORCID: 0009-0006-3941-9214 <artemkorz@mail.ru>*
[2,3] *N.N. Datsun, ORCID: 0000-0001-8560-7036 <nndatsun@inbox.ru>*

[1] *Perm State University,*
15, Bukireva st., Perm, 614068, Russia.

[2] *HSE University, Perm,*
38, Studencheskaya st., Perm, 614070, Russia.

[3] *PSHPU, Perm,*
24, Sibirskaya st., Perm, 614990, Russia.

Abstract. Currently, the information technologies industry is a leader in growth rate among the main economic sectors. However, the most important components of the development process, such as estimation and refactoring of program products, still remain without generic tools. Therefore, our main goal is to design a mean of unified modification and formal evaluation for code in object-oriented programming languages. We use refactoring patterns to define code modifications, and code metrics calculation to assess its characteristics. Our tool should help developers to make decisions connected with code quality and its modification necessity, automatize that change. Actually, it may be used in organizations and educational institutions. We have developed a domain specific language to unify the specification of object-oriented languages. Furthermore, a research prototype of the tool has been created. 3 object-oriented languages descriptions and 6 diverse refactoring patterns have been developed to demonstrate capabilities of the approach.

Keywords: refactoring; domain specific language; code metrics calculation; object-oriented language; refactoring patterns.

For citation: Korznikov A.O., Datsun N.N. Designing Refactoring Tool for Object-Oriented Code Based on Metrics. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 5, 2025, pp. 143-156. DOI: 10.15514/ISPRAS-2025-37(5)-11.

Проектирование инструмента для рефакторинга объектно-ориентированного кода с использованием расчета метрик

[1] *А.О. Корзников, ORCID: 0009-0006-3941-9214 <artemkorz@mail.ru>*
[2,3] *Н.Н. Дацун, ORCID: 0000-0001-8560-7036 <nndatsun@inbox.ru>*

[1] *Пермский государственный национальный исследовательский университет, Россия, 614068, г. Пермь, ул. Букирева, д. 15.*

[2] *НИУ ВШЭ, Пермь, Россия, 614070, г. Пермь, ул. Студенческая, д. 38.*

[3] *Пермский государственный гуманитарно-педагогический университет, Россия, 614990, г. Пермь, ул. Сибирская, д. 24.*

Аннотация. На данный момент отрасль информационных технологий (ИТ) занимает лидирующую позицию по темпам роста среди основных отраслей экономики. Однако, не существует универсальных и стандартизованных инструментов для важнейших компонентов процесса разработки: оценки и рефакторинга программных продуктов. Поэтому нашей основной целью является проектирование средства для унифицированного изменения и формальной оценки кода на объектно-ориентированном языке программирования. Для описания изменений кода используются шаблоны рефакторинга, для оценки его характеристик – расчет метрик кода. Целью нашего инструмента является помощь программистам в принятии решений, связанных с качеством кода и необходимостью внесения изменений в код, автоматизация этих изменений. Приложение может использоваться в организациях и образовательных учреждениях. Был разработан предметно-ориентированный язык, чтобы унифицировать описание объектно-ориентированных языков. Кроме того, был создан исследовательский прототип инструмента. Для демонстрации возможностей предложенного подхода были созданы 3 описания объектно-ориентированных языков и 6 различных шаблонов рефакторинга.

Ключевые слова: рефакторинг; предметно-ориентированный язык; расчет метрик кода; объектно-ориентированный язык; шаблоны рефакторинга.

Для цитирования: Корзников А.О., Дацун Н.Н. Проектирование инструмента для рефакторинга объектно-ориентированного кода с использованием расчета метрик. Труды ИСП РАН, том 37, вып. 5, 2025 г., стр. 143–156 (на английском языке). DOI: 10.15514/ISPRAS–2025–37(5)–11.

1. Introduction

At present a rapid rise of IT industry occurs. Despite this, a number of significant issues that inhibit the effective work and development of organizations are detected. Firstly, there are no standardized tools for evaluating computer programs. Secondly, refactoring is a key and integral part of software maintenance, but there is no common tool for it. Thirdly, a rate of technology substitution is high, and it is vital for organizations to use effective ones to stay competitive.

Metrics can be considered as a tool for formal evaluation of code characteristics. However, the recent researches show that applications based on metrics have a low true positive rate [1]. The authors believe that lack of an actual context is the main reason [2]. In our work, the metrics are interpreted by a programmer, who uses our tool. Developer as an expert assesses a code quality using personal professional experience and recommended limits for metrics values provided by the tool. Moreover, it is a useful practice to compare the characteristics of different program projects.

Refactoring is a mean of the uncontrollable growth prevention of a program code length, number of errors hidden, and design issues (technical debts). Therefore, the significance of refactoring cannot be underestimated, but its implementation is connected with high complexity and time cost, especially when changing the whole project [3]. Modern IDEs include means of metrics calculation as well as rapid global refactoring while coding (floss refactoring) [4: 163]. However, they may be inconvenient in some cases due to massiveness. Furthermore, IDEs focus on supporting a small number of programming languages and specific refactoring operations. In addition, researches show that many developers are cautious about these tools and prefer to perform refactoring manually [4: 163, 5: 4, 6: 1]. Our proposed approach and a research prototype based on it are independent of object-oriented programming (OOP) language.

Actually, different organizations use various versions and extensions for programming languages, some of them develop the own languages to solve particular tasks in a specific domain. On the one hand, it helps to accelerate the development and simplify understanding, but on the other hand, refactoring and metrics calculation may become complicated. Thus, extra financial resources are required to develop appropriate tools for automatic execution of those actions.

To implement the tool for refactoring object-oriented code using metrics calculation, the following tasks must to be solved to:

- analyze requirements;
- analyze existing software;
- design the program tool;
- design a domain specific language (DSL);
- implement the tool and test it.

2. Requirements

To solve these problems, a tool independent of a specific object-oriented languages set should be developed. It must:

- provide means of a unified description of various languages and refactoring operations;
- use terms of the procedural and OOP paradigms to calculate the respective metrics;
- describe languages and refactoring operations in a similar way to simplify the tasks of programmer;
- apply refactoring to the whole project or its individual physical (files) and logical (hierarchies of classes) parts;
- allow developers to evaluate the formal characteristics of program code, compare metrics of refactored code with initial values.

3. Related works

3.1 Technical debt and code smells

In practice, developers often use refactoring to remove “technical debt”, particularly “code smells” [7]. Technical debt means a decrease of code quality in its development [1]. In the long term, it leads to such serious consequences as an increase in cost of defects correction, further development and making changes [1, 8]. Code smells are the most studied and recognizable type of technical debt related to design problems [8]. The term is firmly entrenched in the context of refactoring and combines the problems encountered in object-oriented code [1].

Code metrics can also be used to determine this kind of drawback [1], especially Chidamber and Kemerer set is often used. According to various works, metrics in the context of refactoring are applied for identification of low-quality code parts [9], comparing a source code with refactored one [10] and estimating a cost of refactoring application [11].

3.2 Refactoring tools

The study of current refactoring tools is presented in a systematic mapping study [12]. According to the data obtained, there is a set of refactoring applications that recommend changes, perform refactoring and detect it [12]. The most commonly considered language is Java, and there are also tools working with code in C, JavaScript, and various DSL (e.g., CSS). The denoted tools are focused on applying specific refactoring operations within a certain set of languages.

In the review [13], Eclipse, Xcode are named among the popular refactoring tools. Moreover, some of the applications solve similar issues, such as refactoring tests, performed using DARTS [10] and B-refactoring [12]. Other systems are highly specialized and work with a particular domain: RIdiom automatically replaces all code fragments that do not correspond to Python idioms [14]; ReSwither modifies structure of a switch operator in Java [15]; Android Studio plugin works with energy efficiency [16]. Besides, a lot of the presented tools are either unavailable in Russia or are not free: Synchronizer, Asyncdroid, Xll and others [12].

3.3 Tools for code metrics calculation

Systematic mapping studies (SMS) [17-19] examine the possibilities of tools for automated metric calculation. It is worth noting a similar study, the authors of which indicated that SourceMeter and Metrics are most often used to calculate program metrics; PMD and JDeodorant are commonly used for detection and removing bad smells; JDeodorant and Eclipse are most frequently applied for refactoring [20: 929]. However, the applications for refactoring, metrics calculation, and tools proposed by IDEs, which are discussed in these papers, are not universal.

4. Describing approach and designing tool

4.1 Framework

We propose a refactoring approach described in Fig. 1. An iteration of the refactoring loop [21] requires to identify (step 3) code fragments, to recommend (steps 2, 5) metrics comparison evaluation, and to apply (steps 4, 6) code modifications. A green outline shows a preparatory step performed by the programmer manually. The blue frames depict steps performing semi-automatically, when a decision is made by user. Other steps are done automatically. Thus, in refactoring loop we suggest a user to choose the refactoring pattern and to decide whether to apply the modification using results of the metrics comparison.

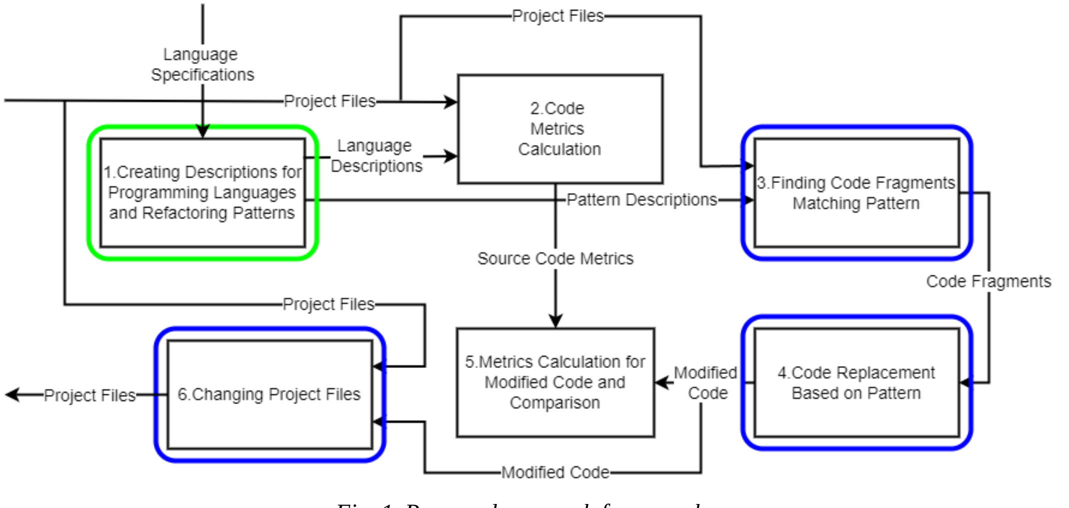


Fig. 1. Proposed approach framework.

4.2 Designing tool

The main purpose of our tool is to obtain values of the formal code metrics, perform refactoring, and compare calculated parameters. As the determined indicators, Chidamber and Kemerer, Lorenz and Kidd metrics sets were selected. Additionally, Halstead metrics, lines of code (LOC) and program style evaluation which are independent of a paradigm, were included in the set used.

The tool consists of 4 modules (Fig. 2): (1) an analyzer, (2) metrics calculator, (3) explanation and (4) refactoring units. The analyzer includes lexical, syntactic and semantic code analysis. As we develop a generic tool, only the general semantics of object-oriented languages is considered and the rest of it must be specified in a particular language description. The metrics calculation is performed both for individual elements of object-oriented languages and for a whole project by calculating average values. To explain the results, a dictionary that includes namespaces and classes is used, as well as a comparison between the calculated metrics values and numbers recommended by their authors.

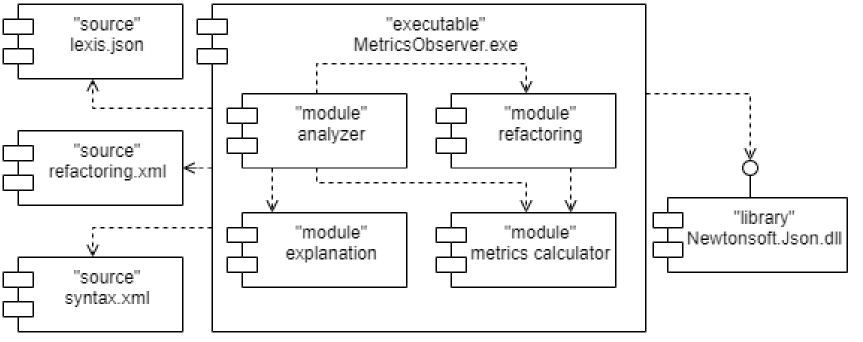


Fig. 2. Tool architecture.

4.3 Designing DSL

To provide an opportunity of analyzing the code in various languages, a textual DSL was developed. It allows a programmer to describe lexis and syntax of the OOP languages. The language is based on terms such as class, namespace, operator, and operand. BNF was defined for the DSL.

A pattern description method was selected to implement refactoring [22] and code parts. Therefore, it provides a possibility to work similarly with the language syntax and the refactoring pattern. The refactoring pattern structure that consists of 4 parts (Fig. 3): variables, search, replace, and references section was proposed. Firstly, a list of entities required and their initial values is described. Secondly, elements of a language and its context must be defined as the syntax is. That definition could be placed separately as a code fragment description.

<table>
<tr><td><Refactorings> ::= <Pattern> {<SubPattern> | <Fragment>}</td></tr>
<tr><td><SubPattern> ::= <Pattern></td></tr>
<tr><td><Pattern> ::= “<Refactoring xsi:type="Pattern">” <PatternBody> <Next> “</Refactoring>”</td></tr>
<tr><td><PatternBody> ::= <Name> <Variables> <Search> <Replace></td></tr>
<tr><td><Name> ::= “<Name>” <String> “</Name>”</td></tr>
<tr><td><Variables> ::= “<Variables>” <PosEntity> {<PosEntity> | <VariableEntity>} “</Variables>”</td></tr>
<tr><td><PosEntity> ::= <PositionsVar> | <PositionVar></td></tr>
<tr><td><PositionsVar> ::= “<Variable xsi:type="Position">” <String> “</Variable>”</td></tr>
<tr><td><Search> ::= “<Search>” <DescriptionBody> “</Search>”</td></tr>
<tr><td><DescriptionBody> ::= <Instruction> {<Instruction>}</td></tr>
<tr><td><Replace> ::= “<Replace>” {<Insert>} “</Replace>”</td></tr>
<tr><td><Insert> ::= “<Entity xsi:type="” <InsertType> “">” <Position> [<Description>] “</Entity>”</td></tr>
<tr><td><Next> ::= “<Next>” <Refactor> {<Refactor>} “</Next>”</td></tr>
<tr><td><Refactor> ::= “<Refactor>” <String> <Arguments> “</Refactor>”</td></tr>
<tr><td><Fragment> ::= “<Refactoring xsi:type="Fragment">” <FragmentBody> “</Refactoring>”</td></tr>
<tr><td><FragmentBody> ::= <Name> <Variables> <Description></td></tr>
</table>

Fig. 3. Fragment of BNF for refactoring pattern structure.

Subsequently, the replacement part requires position of the code fragment to be defined. The location is set as a tuple of a first character position and symbols amount. It can be found by the code description. The code replacement requires the position and a type of change: adding or modification. Besides, it would be inevitable to perform the search again in some cases. Thus, the last part of the pattern is used for referring other ones and share information accumulated.

Visualization of replacements made is also a significant aspect. We decided to highlight the code fragments related to the first entity described. As a result, the corresponding fragments of a source and modified code are colored identically. It allowed highlighting the refactored code parts automatically. Additionally, the developer can assign a color independently. It might be useful when creating additional files while refactoring.

5. Implementation and testing

The implemented DSL was based on xml as it is conveniently serializable. It was necessary to define descriptions of OOP languages for testing. For this reason, the lexis and syntax were defined for subsets of C# 7.3, Java SE 8, and C++11 using the DSL. The main purpose of those definitions is to provide an opportunity to test the research prototype developed. Moreover, some language details are not significant for metrics calculation, because of that a complete description is not required (e.g., C++ pragma instructions).

5.1 Testing refactoring patterns

We have implemented 6 refactoring instances and they have been tested using a code in C#. Fig. 4-10 show the source code on the left and the refactored one on the right.

Fig. 4 demonstrates a sort of imports. It is performed sequentially for each word in the compared strings and started using the reserved command for the list. The corresponding lines are automatically highlighted with colors. The corresponding declarations are created.

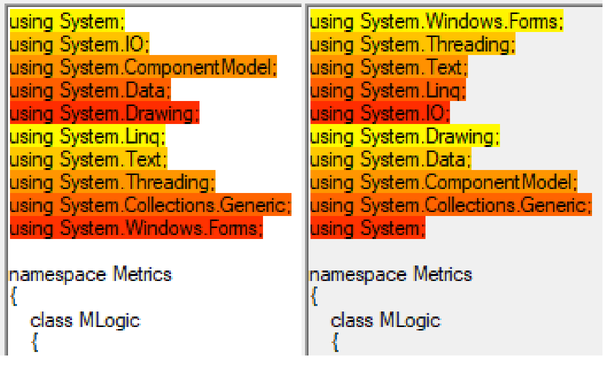
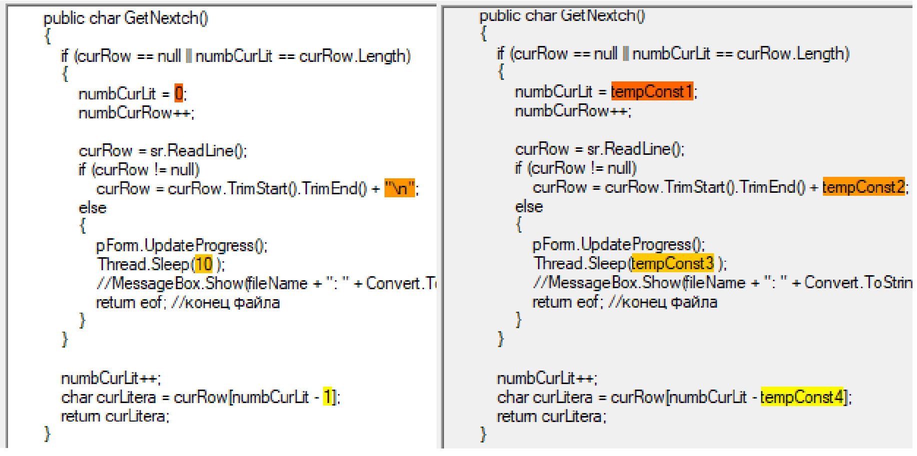


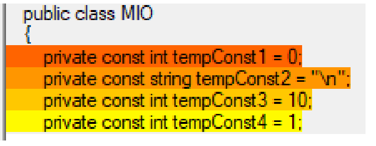
Fig. 4. Sorting imports.

Fig. 5 (*a*) shows an example of moving literals from class methods to constants using the stated name. Actually, they are numbered automatically and the same values are considered as the same constant. Code fragments are skipped until an operand that is a literal is encountered. The unique values of variables are stored using sets, whereas positions and values are contained with stacks. The corresponding changes in code have the identical color, and declarations of the constants are highlighted with the last change color (Fig. 5, *b*). Fig. 6 illustrates an example with C++ code.

Fig. 7 depicts an example of the following pattern. It allows a programmer to move the code located between comments of a certain type, defined by a regular expression. For that reason, a method with described modifiers and name is created. Although the comments shown are convenient for processing with that pattern, they are practically meaningless and do not contribute to documentation of an application. After performing this type of pattern, the documentation refactoring proposed in [23] should be performed additionally.



(a)



(b)

Fig. 5. Transforming literals into constants (C#):
a) source and refactored code; b) declarations of constants in refactored code.

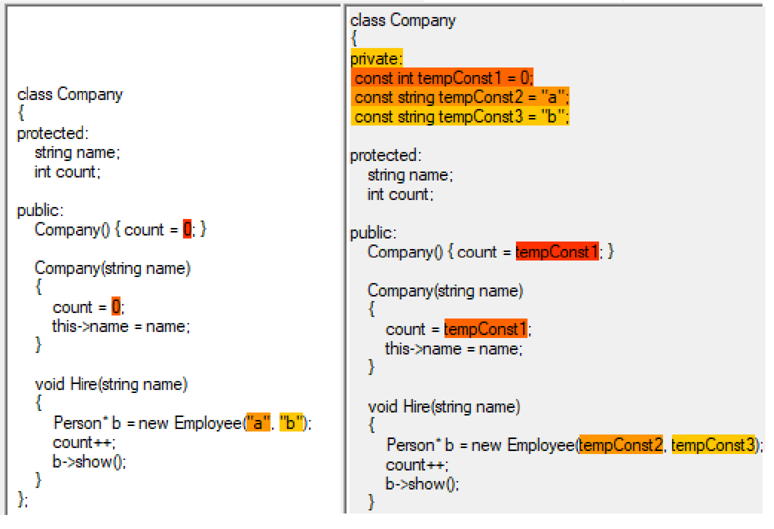


Fig. 6. Transforming literals into constants (C++).

Fig. 8 demonstrates an example of using a pattern when a developer requires creating a corresponding property for a public field.

Fig. 9 (*a*) shows an example of refactoring that extracts an interface from a class into new separated file. After public methods are found, their signatures are transferred to the extracted interface in the created file (Fig 8, *b*). Furthermore, inheritance code related to the interface is added to the class. The color design is performed manually for inheritance, as it is a new code fragment added to the existing set.

Fig. 10 depicts an example of move method refactoring that is often used in practice [24]. The idea is to find a calls number of the specified method for each class and move it to the one with the highest value.

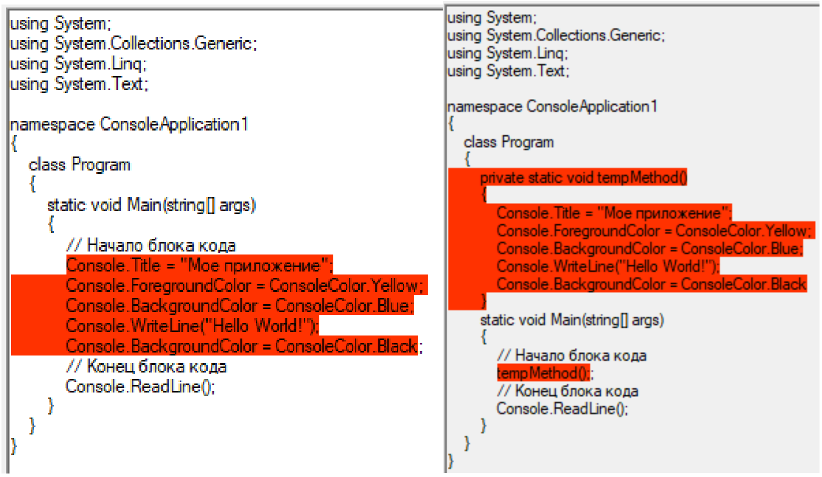


Fig. 7. Moving code into method using comments.

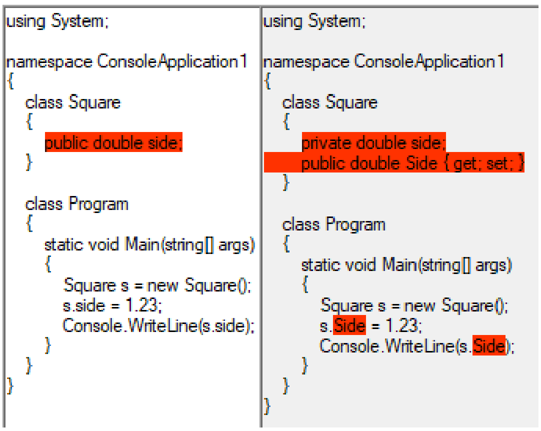


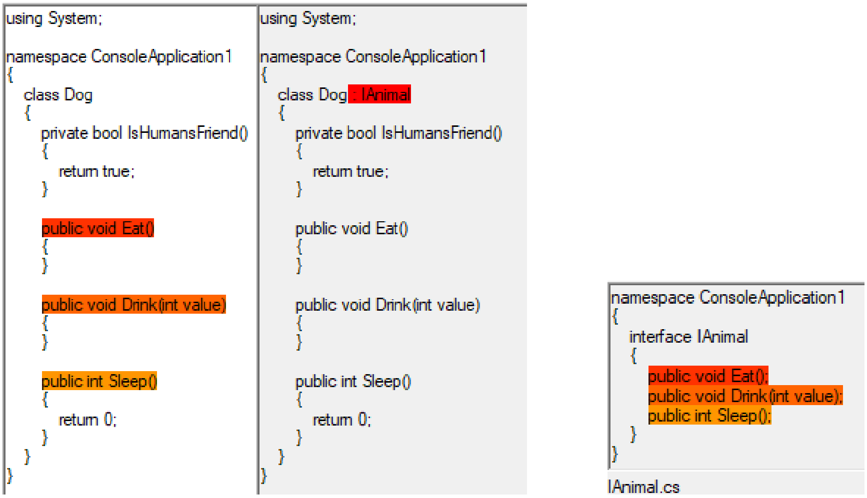
Fig. 8. Encapsulating field.

The refactoring is divided into 2 patterns: moving a method into the class and correcting names. Firstly, the number of calls is calculated, method is transferred if necessary, and the new location is highlighted. Secondly, if the current position of a method does not match the class where it is implemented, then the class name is replaced or added before the method call. The secondary pattern is referred using an instruction and the modifications performed by it are not highlighted.

As a result of the refactoring, we have achieved an improvement in response for a class (RFC) metric, but lack of cohesion in methods (LCOM) has also increased (Fig. 11).

5.2 Testing performance

A test bench has the following parameters: OS Windows 10 Pro, 64-bit system, Intel Core i5-6500 3.20GHz CPU, 16 GB RAM. Fig. 12 shows the chosen refactoring pattern, which have been used to test performance of the research prototype. We have applied that pattern to a code of Metrics Observer [25]. The project consists of 15 files written in C#.



(a) ***(b)***

Fig. 9. Extracting interface and creating new file: a) source and refactored code; b) content of created file.

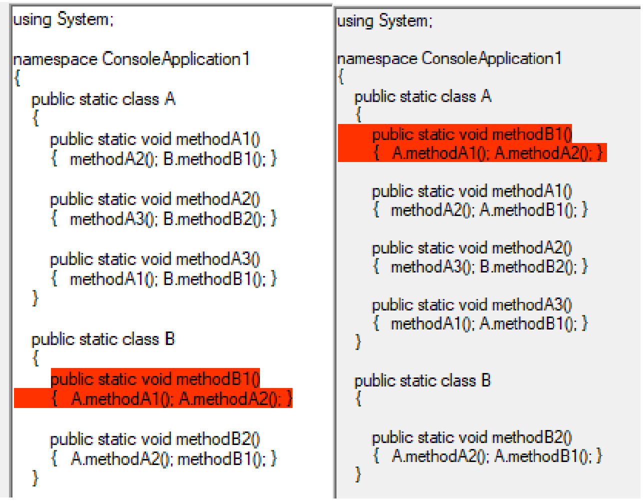


Fig. 10. Moving method and updating names: methodB1.

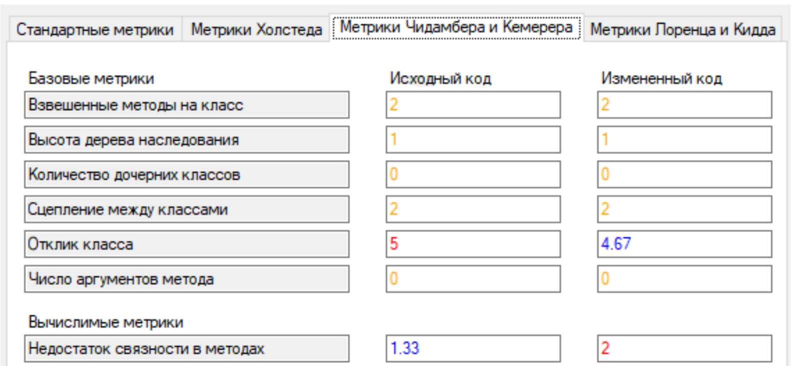


Fig. 11. Metrics comparison for methodB1.

```xml
<?xml version="1.0"?>
<ArrayOfRefactoring xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
    <Refactoring xsi:type="Pattern">
        <Name>Вынести константы</Name>
        <Variables>
            <Variable xsi:type="Positions">constant</Variable>
            <Variable xsi:type="VariablesUnique">constantValue</Variable>
            <Variable xsi:type="Position">constantPut</Variable>
            <Variable>constantName<Value>tempConst</Value></Variable>
            <Variable>constantNumber</Variable>
            <Variable>langName</Variable>
        </Variables>
        <Search>
            <!--Определить язык-->
            <Entity xsi:type="Reserved">
            <!--1 В классе-->
            <Entity xsi:type="SkipTo">classSyKw</Entity>
            <!--Куда будет помещено объявление константы-->
            <Entity xsi:type="SkipTo">mainStructureSy</Entity>
            <Entity xsi:type="FindPosition">
                <Link>constantPut</Link>
                <Of>
            </Entity>
            <!--2 из блоков кода-->
            <Entity xsi:type="SkipTo">
            <!--если не конец класса-->
            <Entity xsi:type="If">
                <Check xsi:type="CheckExpression">
                    <Verifiable xsi:type="Not">
                </Check>
                <Description>
                    <!--3 найти константу-->
                    <Entity xsi:type="While">
                        <Check xsi:type="CheckExpression">
                        <Description>
                            <Entity xsi:type="SkipTo">
                            <!--4 если нашлась константа - запомнить значение-->
                            <Entity xsi:type="If">
                                <Check xsi:type="CheckType">const</Check>
                                <Description>
                                    <Entity xsi:type="SaveToken">constantValue</Entity>
                                    <Entity xsi:type="FindPosition">
                                        <Link>constant</Link>
                                        <Of>
                                            <Entity xsi:type="AcceptType">const</Entity>
                                        </Of>
                                    </Entity>
                                </Description>
                            </Entity>
                        </Description>
                    </Entity>
                </Description>
            </Entity>
            <!--5 если следующий класс или конец файла - завершение-->
        </Search>
        <Replace>
            <Entity xsi:type="InsertAfter">
            <Entity xsi:type="While">
                <Check xsi:type="CheckValue">constantValue</Check> <!--Пока в стеке есть значения-->
                <Description>
                    <!--Добавление объявления-->
                    <Entity xsi:type="InsertAfter">
                        <Position>constantPut</Position>
                        <Description>
                            <!--Получение номера значения константы с именем constantValue-->
                            <Entity xsi:type="VariablesNumber">constantValue<Link>constantNumber</Link></Entity>
                            <Entity xsi:type="If">
                        </Description>
                    </Entity>
                    <!--Замена константы на заданное имя-->
                    <Entity xsi:type="Insert">
                        <Position>constant</Position>
                        <Description>
                            <Entity xsi:type="NameNoSpace">constantName</Entity>
                            <Entity xsi:type="NameNoSpace">constantNumber</Entity>
                        </Description>
                    </Entity>
                    <Entity xsi:type="VariablesNext">constant</Entity>
                </Description>
            </Entity>
        </Replace>
    </Refactoring>

    <Refactoring xsi:type="Fragment">
        <Name>constDef</Name>
        <Variables>
        <Description>
    </Refactoring>
</ArrayOfRefactoring>
```

Fig. 12. Refactoring pattern fragment for transforming literals into constants.

Project files were automatically sorted in lexicographic order of their names and enumerated. Code fragments matching the refactoring pattern were found in 10 files. We measured the time taken by the main steps of the refactoring loop and number of logical lines of code for each file (Fig. 13).

As a result, the most time spent was equal to 589 ms. It was required to apply refactoring pattern to syntactic analyzer class (file 7) and calculate metric values. It took 213 ms to find code fragments in the file containing 4023 logical lines of code, and 337 ms to calculate and compare metric values. Despite the size of code, it had only 14 literals to be transformed into constants. However, the most amount of time spent for code modification step was required to change a code of token dictionary (file 2), which contained the largest number of unique literals: 102.

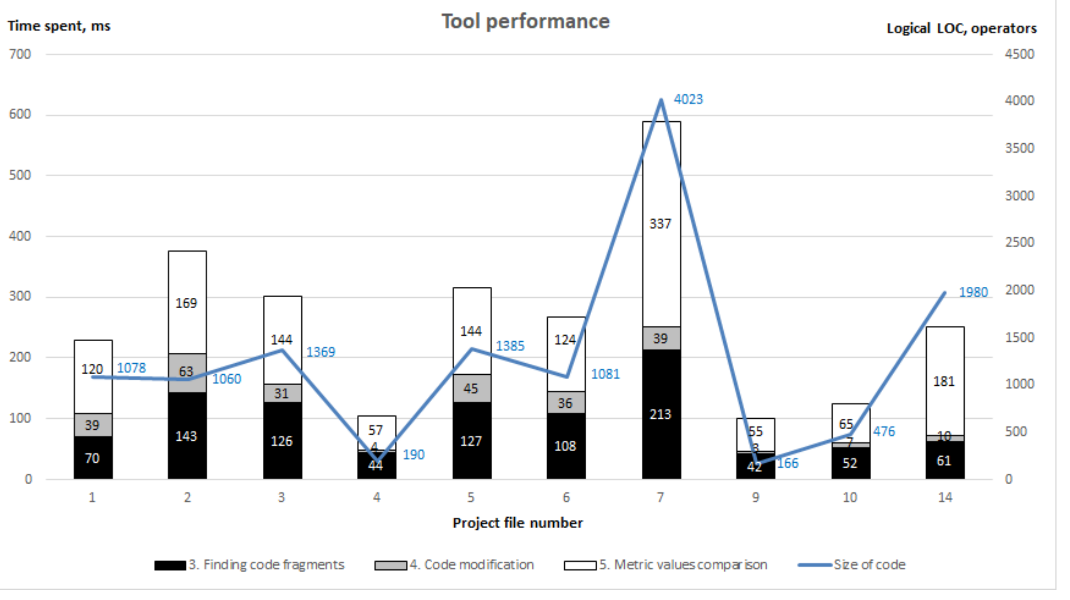


Fig. 13. Tool performance.

6. Discussion

Our tool requires language specifications in the DSL developed by a programmer. Nevertheless, it provides a generic tool for object-oriented languages and refactoring operations implementation. It may allow reducing time to deal with languages based on similar syntax and also create custom refactoring methods, make decisions regarding code modification and accumulate experience.

The developed design of a code refactoring tool that uses metric calculation and the created research prototype of our application demonstrated the possibility of:

- describing refactoring patterns using the developed DSL;
- generalized work with various object-oriented languages;
- comparing metrics when performing refactoring.

Our tool allows a developer to calculate metrics and refactor code using lexis and syntax descriptions in the DSL. Furthermore, it may be used for various versions, extensions, and new programming languages based on object-oriented paradigm.

However, despite the opportunity to deal with diverse languages and define a single refactoring pattern in several ways, it has not already proven that an arbitrary pattern could be described with the DSL. Another limitation is the complexity of working with text DSL:

- significant increase in the number of physical code lines compared to logical ones, including xml tags;
- reduced clarity due to high nesting and large amount of text;
- requirement of the DSL specification due to a large number of keywords;
- unavailability of an environment for writing code in the DSL.

Currently, the smallest pattern (Fig. 6) consists of 78 lines and 155 words, whereas the most complex one (Fig. 9) contains 434 lines and 655 words.

The performance depends on logical lines of code; complexity of code, pattern and language descriptions used; size of entities used in refactoring pattern such as set or dictionary.

7. Conclusion

Our application requires language specifications in the DSL developed by a programmer. Nevertheless, it provides a generic tool for object-oriented languages and refactoring operations implementation. It allows reducing time to deal with languages based on similar syntax and also create custom refactoring methods, make decisions regarding code modification and accumulate experience.

In this paper we have described an approach to implement code refactoring using its metrics calculation and refactoring patterns. The DSL, research prototype, 3 OOP languages specifications, and 6 typical refactoring patterns have been created. These examples have demonstrated the capabilities of this language.

Our tool may be used in organizations to refactor code and unambiguously evaluate its properties. Moreover, it may also be applied in educational institutions to verify and correct code written by students.

This study has a several possible directions for further activities. Firstly, an equivalent visual DSL and a suitable development environment for it should be created. Secondly, studying of the required language features for describing arbitrary refactoring can be necessary. Finally, proposing recommendations for pattern application based on code metrics and confirmation profitability of code modification may also be promising.

References

[1]. Качанов В.В., Ермаков М.К., Панкратенко Г.А., Спиридонов А.В., Волков А.С., Марков С.И. Технический долг в жизненном цикле разработки ПО: запахи кода. Труды ИСП РАН, том 33, вып. 6, 2021 г., стр. 95-110. DOI: 10.15514/ISPRAS-2021-33(6)-7. / Kachanov V.V., Ermakov M.K., Pankratenko G.A., Spiridonov A.V., Volkov A.S., Markov S.I. Technical debt in the software development lifecycle: code smells. Trudy ISP RAN/Proc. ISP RAS, 2021, vol. 33, issue 6, pp. 95-110 (in Russian). DOI: 10.15514/ISPRAS–2021–33(6)–7.

[2]. Sharma T., Efstathiou V., Louridas P., Spinellis D. Code smell detection by deep direct-learning and transfer-learning. Journal of Systems and Software, vol. 176, article no. 110936, 2021, pp.1-25. DOI: 10.1016/j.jss.2021.110936.

[3]. Сыромятников С. В., Бронштейн И. Е., Луговской Н. Л. Рефакторинг в рамках программного проекта. Труды ИСП РАН, том 26, вып. 1, 2014 г., стр. 395-402. DOI: 10.15514/ISPRAS-2014-26(1)-16. / Syromyatnikov S. V., Bronshteyn I. E., Lugovskoy N. L. Refactoring on the Whole Project. Trudy ISP RAN/Proc. ISP RAS, 2014, vol. 26, issue 1, pp. 395-402 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-16.

[4]. Ivers J., Nord R. L., Ozkaya I., Seifried C., Timperley C. S., Kessentini M. Industry's cry for tools that support large-scale refactoring. In Proc. of the 44th International Conference on Software Engineering: Software Engineering in Practice, 2022, pp. 163-164. DOI: 10.1145/3510457.3513074.

[5]. Almogahed A., Mahdin H., Omar M., Zakaria N. H., Alawadhi A., Barraood S. O. Empirical Investigation of the Diverse Refactoring Effects on Software Quality: The Role of Refactoring Tools and Software Size. In Proc. of the 2023 3rd International Conference on Emerging Smart Technologies and Applications, 2023, pp. 1-6. DOI: 10.1109/eSmarTA59349.2023.10293407.

[6]. Golubev Y., Kurbatova Z., AlOmar E. A., Bryksin T., Mkaouer M. W. (2021) One Thousand and One Stories: A Large-Scale Survey of Software Refactoring (online). Available at: https://doi.org/10.48550/arXiv.2107.07357, accessed 05.05.2025.

[7]. Peruma A., AlOmar E. A., Newman C. D., Mkaouer M. W., Ouni A. Refactoring Debt: Myth or Reality? An Exploratory Study on the Relationship Between Technical Debt and Refactoring. In Proc. of the 2022 IEEE/ACM 19th International Conference on Mining Software Repositories, 2022, pp. 127-131. DOI: 10.1145/3524842.3528527

[8]. Li Z., Avgeriou P., Liang P. A Systematic Mapping Study on Technical Debt and its Management. Journal of Systems and Software, vol. 101, 2015, pp. 193-220. DOI: 10.1016/j.jss.2014.12.027.

[9]. Panigrahi R., Kuanar S. K., Kumar L. Application of Naïve Bayes classifiers for refactoring Prediction at the method level. In Proc. of the 2020 International Conference on Computer Science, Engineering and Applications, 2020, pp. 1-6. DOI: 10.1109/ICCSEA49143.2020.9132849.

[10]. Lambiase S., Cupito A., Pecorelli F., De Lucia A., Palomba F. Just-In-Time Test Smell Detection and Refactoring: The DARTS Project. In Proc. of the 2020 IEEE/ACM 28th International Conference on Program Comprehension, 2020, pp. 441-445. DOI: 10.1145/3387904.3389296.

[11]. Perera J., Tempero E., Tu Y.-C., Blincoe K. Quantifying Requirements Technical Debt: A Systematic Mapping Study and a Conceptual Model. In Proc. of the 2023 IEEE 31st International Requirements Engineering Conference, 2023, pp. 123-133. DOI: 10.1109/RE57278.2023.00021.

[12]. Tavares C., Ferreira F., Figueiredo E. A Systematic Mapping of Literature on Software Refactoring Tools. In Proc. of the XIV Brazilian Symposium on Information Systems, 2018, article no. 11, pp. 1-8. DOI: 10.1145/3229345.3229357.

[13]. Murphy-Hill E., Black A. P. Refactoring Tools: Fitness for Purpose. IEEE Software, 2008, vol. 25, issue 5, pp. 38-44. DOI: 10.1109/MS.2008.123.

[14]. Zhang Z., Xing Z., Xu X., Zhu L. RIdiom: Automatically Refactoring Non-Idiomatic Python Code with Pythonic Idioms. In Proc. of the 2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings, 2023, pp. 102-106. DOI: 10.1109/ICSE-Companion58688.2023.00034.

[15]. Zhang Y., Li C., Shao S. ReSwitcher: Automatically Refactoring Java Programs for Switch Expression. In Proc. of the 2021 IEEE International Symposium on Software Reliability Engineering Workshops, 2021, pp. 399-400. DOI: 10.1109/ISSREW53611.2021.00108.

[16]. Iannone E., Pecorelli F., Di Nucci D., Palomba F., De Lucia A. Refactoring Android-specific Energy Smells: A Plugin for Android Studio. In Proc. of the 2020 IEEE/ACM 28th International Conference on Program Comprehension, 2020, pp. 451-455. DOI: 10.1145/3387904.3389298.

[17]. Корзников А. О., Дацун Н. Н. Методы и средства расчета и применения метрик кода программных продуктов: систематическое картографирование литературы. Известия СПбГЭТУ «ЛЭТИ», том 17, вып. 8, 2024 г., стр. 48-64. DOI: 10.32603/2071-8985-2024-17-8-48-64. / Korznikov A. O., Datsun N. N. Methods for Calculation and Application of Software Code Metrics: A Systematic Mapping Study. LETI Transactions on Electrical Engineering & Computer Science, 2024, vol. 17, issue 8, pp. 48-64 (in Russian). DOI: 10.32603/2071-8985-2024-17-8-25-64.

[18]. Colakoglu F. N., Yazici A., Mishra A. Software Product Quality Metrics: A Systematic Mapping Study. IEEE Access, vol. 9, 2021, pp. 44647-44670. DOI: 10.1109/ACCESS.2021.3054730.

[19]. Mshelia Y. U., Apeh S. T., Edoghogho O. A comparative assessment of software metrics tools. In Proc. of the 2017 International Conference on Computing Networking and Informatics, 2017. P. 1-9. DOI: 10.1109/ICCNI.2017.8123809.

[20]. Agnihotri M., Chug, A. A Systematic Literature Survey of Software Metrics, Code Smells and Refactoring Techniques. Journal of Information Processing Systems, 16(4), 2020, pp. 915-934. DOI: 10.3745/JIPS.04.0184.

[21]. Fernandes S., Aguiar A., Restivo A. LiveRef: a Tool for Live Refactoring Java Code. In Proc. of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022, article no. 161, pp. 1-4. DOI: 10.1145/3551349.3559532.

[22]. Mooij A. J., Ketema J., Klusener S., Schuts M. Reducing Code Complexity through Code Refactoring and Model-Based Rejuvenation. In Proc. of the 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering, 2020, pp. 617-621. DOI: 10.1109/SANER48275.2020.9054823.

[23]. Luciv D. V., Koznov D. V., Shelikhovskii A. A., Romanovsky K. Yu., Chernishev G. A., Terekhov A. N., Grigoriev D. A., Smirnova A. N., Borovkov D. V., Vasenina A. I. Interactive Near Duplicate Search in Software Documentation. Programming and Computer Software, 2019, vol. 45, pp. 346-355. DOI: 10.1134/S0361768819060045.

[24]. Dallal J. Al, Abdulsalam H., AlMarzouq M., Selamat A. Machine Learning-Based Exploration of the Impact of Move Method Refactoring on Object-Oriented Software Quality Attributes. Arabian Journal for Science and Engineering, 2024, vol. 49, pp. 3867-3885. DOI: 10.1007/s13369-023-08174-0.

[25]. Корзников А. О., Дацун Н. Н. Разработка приложения для получения метрик программного продукта на языке объектно-ориентированного программирования. Вестник Пермского университета. Математика. Механика. Информатика, вып. 3 (62), 2023 г., стр. 76-84. DOI: 10.17072/1993-0550-2023-3-76-84. / Korznikov A.O., Datsun N.N. Program Realization for Code Metrics Calculation in Object-Oriented Programming Language. Bulletin of Perm University. Mathematics. Mechanics. Computer Science, 2023, issue 3(62), pp. 76-84. (in Russian). DOI: 10.17072/1993-0550-2023-3-76-84.

Информация об авторах / Information about authors

Артем Олегович КОРЗНИКОВ – магистрант физико-математического института Пермского государственного национального исследовательского университета, бакалавр прикладной математики и информатики. Сфера научных интересов: метрики кода, объектно-ориентированные языки программирования, предметно-ориентированные языки, рефакторинг кода.

Artem Olegovich KORZNIKOV – BSc (Applied Mathematics and Computer Science), Master’s student of the Department of Physics and Mathematics of PSU. Research interests: code metrics, object-oriented programming languages, domain specific languages, code refactoring.

Наталья Николаевна ДАЦУН является приглашенным преподавателем кафедры информационных технологий в бизнесе Национального исследовательского университета «Высшая школа экономики», Пермь; доцент кафедры прикладной информатики, информационных систем и технологий Пермского государственного гуманитарно-педагогического университета, кандидат физико-математических наук, доцент. Ее научные интересы включают метрики кода, объектно-ориентированный анализ и проектирование.

Natalya Nikolaevna DATSUN – Cand. Sci. (Phys.-Math,), visiting lecturer of Department of Information Technology in Business of the National Research University Higher School of Economics, Perm; Associate Professor, Department of Applied Informatics, Information Systems and Technologies, Perm State Humanitarian Pedagogical University. Her research interests include code metrics, object-oriented analysis, and design.