UDC 004.43:004.02
DOI: 10.15587/2706-5448.2022.267160

Andrii Tkachuk,
Bogdan Bulakh

USAGE OF FORMALIZED KNOWLEDGE ABOUT SOURCE CODE FOR REFACTORING ACTIONS IN SWIFT

The object of research in the paper is the source code of a software product written in the Swift programming language. Most programs as a product of a certain project during the life cycle require changes and modifications, which is costly or impossible to implement in the case of poor code quality. One of the options for solving the problem of poor code quality is the timely application of refactoring principles. The existing problem is that implementation of high-level refactoring must be done manually by the developer without the use of automated tools as built-in solutions cannot fulfill the need due to the architectural complexity of the product.

To reduce the number of errors made during refactoring, to simplify the process of performing routine actions, it is suggested to use a new software product for refactoring. It works with high-level user commands based on a formalized description of the source code together with a knowledge base containing a description of code entities and their properties (what specific actions can be performed with them). In the work, the refactoring of the source code was carried out using the example of the Swift programming language. The proposed approach of component architecture (knowledge base, software engine) further allows to expand the functionality of the software product to other programming languages.

The work was directed to the development of a prototype of a software product using the proposed approach to check and compare the results with other refactoring tools. A command line utility has been developed that accepts a verbal command as an input and outputs the results of processing and analysis of the source code (search for complex structures in the code) or applies the proposed change. As a result of the conducted testing, it was established that the use of the proposed approach allows performing complex refactoring tasks with the help of a simple verbal formalized command. Accomplishing the same task using only the built-in refactoring tools requires significantly more time and effort or is impossible at all.

Keywords: *Swift programming language, knowledge base, analysis and refactoring tools, source code.*

Received date: 26.09.2022
Accepted date: 14.11.2022
Published date: 17.11.2022

How to cite
Tkachuk, A., Bulakh, B. (2022). Usage of formalized knowledge about source code for refactoring actions in Swift. Technology Audit and Production Reserves, 6 (2 (68)), 6–10. doi: https://doi.org/10.15587/2706-5448.2022.267160

1. Introduction

The use of a standard approach (the one that is offered as an integral part of the language) for refactoring in Swift is limited [1]. Adding new refactoring actions is carried out strictly according to one principle, which makes it impossible to add handling of non-standard situations.

To write a refactoring action, it is necessary to have a compiler, because refactoring does not work with «raw» code, but with its representation – an abstract syntax tree. The use of the added actions is possible only for their «main» purpose without the possibility of parameterization and changing properties while working with the utility. Writing new actions is difficult, which slows down the process of the emergence of new refactoring possibilities [2]. It is these listed drawbacks of refactoring that cause the lack of variety of actions that can be applied and narrow focus of those actions that already exist (they could be applied only to a certain part of the code with a specific goal). To be able to perform full-fledged refactoring using the methods offered in the catalog of code antipatterns (code smells catalog), it is necessary to have a large set of tools, which is impossible due to the complexity of their development [3]. In this case, developers performing refactoring are forced to do a lot of work during refactoring manually, which nullifies the task of automatic refactoring and increases the number of errors in the code [4, 5].

To avoid all the limitations described above, it is suggested to use entities, concepts, and the knowledge base for refactoring. The user must formulate descriptive task for refactoring [6, 7].

The proposed software product must accept as input a request that operates with code concepts and carry out the appropriate refactoring.

The product is innovative, as there are no analogues on the market, and the need to use its functionality is apprehensible.

The object of research in the work is the refactoring of the source code of a software product written in the Swift programming language.

The aim of research is to develop an approach for code refactoring of a source code written in the Swift language, which would be based on the formalization of knowledge about the code.

2. Material and Method

During the research, study was carried out and problems that could not be solved by means of standard refactoring were discovered [8–10].

As an example for comparison purposes, the analysis action that looks for all methods that have more than 3 parameters was chosen. The built-in refactoring tool was tested to obtain a result for future comparison (the task turned out to be overwhelming). The same test was performed for additional refactoring utilities that work on regular expressions. The result showed that they are not able to solve such a task.

The structure of the software knowledge base, the architecture of the software engine and their implementation were developed.

The developed software utility accepts as input a line like the following: *Test.swift find func paramsCount greaterThanOrEquals 3*.

For all requests to be resolved successfully, it is necessary to describe the entities and concepts and correctly program their processing [11]. This is what the knowledge base is for. In this project, the knowledge base is a formalized description of the source code written in the Swift language. It contains its properties, correspondences between «raw» code and code entities, concepts (classes, structures, string literals), their properties (name, identifier, number of parameters) and actions that can be performed (search, rename). Such a description can be formalized, for example, by means of descriptive logics (and corresponding languages such as OWL-DL), but in this work, for simplicity, a meta-description is created using the same Swift programming language.

In Fig. 1 a class diagram is shown. It describes a part of the knowledge base that contains knowledge about the String entity type and what properties it has. Thanks to the protocol organization of the code, any entity can be described using formalized methods, which will allow expanding the functionality of the product effectively.

In Fig. 2 the class diagram of the software engine is shown. The engine is used when there is a request for analysis, such as a «search», or a change (application of refactoring). It reads the source code, sends it to the knowledge base module for evaluation, and returns the result to the user in the form of a text response or modified source code.

Fig. 3 shows a class diagram describing the interaction of the engine classes with the classes representing the knowledge base.

The developed prototype of the software product is a command line utility [12]. Creating a graphical user interface is not appropriate, because the program can act as a sub-process for an integrated development environment or a special text editor.

To check all the requirements set for the product, several test runs were carried out and the boundary conditions of execution were verified.

3. Results and Discussion

An important trait of a developed software product prototype is that the commands that are given are not «strict». For example, the command *rename class name equals Test Renamed* should not be programmed in this specific appearance. It should work successfully even if the entity type is changed to a structure: *rename struct name equals Test Renamed*. That is, the program supports logical deduction from a combination of available (implemented) entities, concepts, actions and performs tasks correctly. It is thanks to this trait that a high rate of scaling of the software product is achieved, since at the initial stage an engine that processes the command and executes it is developed, and then only the knowledge base is supplemented with a description of the entities, their properties, and actions that can be performed.

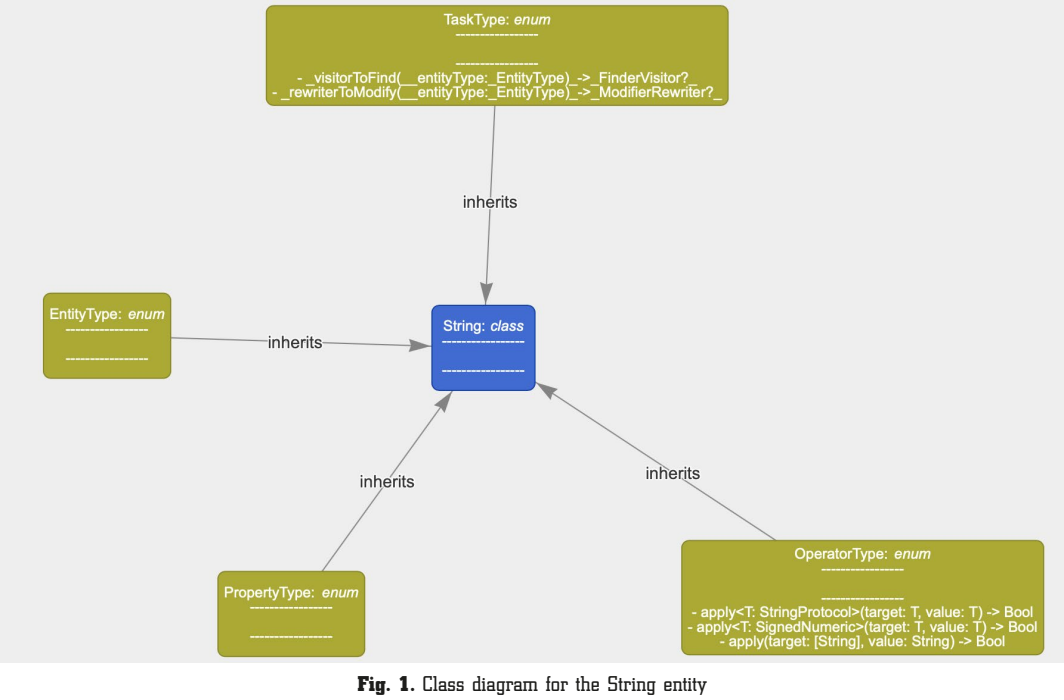

Fig. 1. Class diagram for the String entity

ISSN 2664-9969

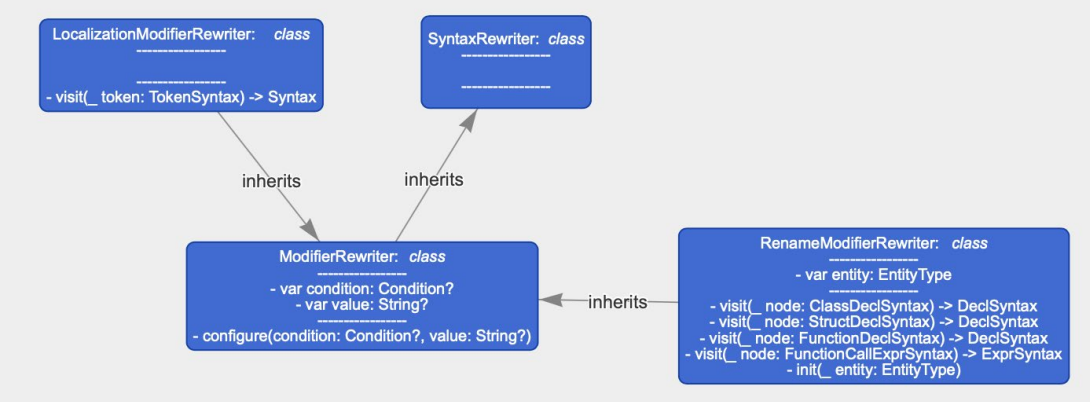

Fig. 2. Class diagram for the software engine

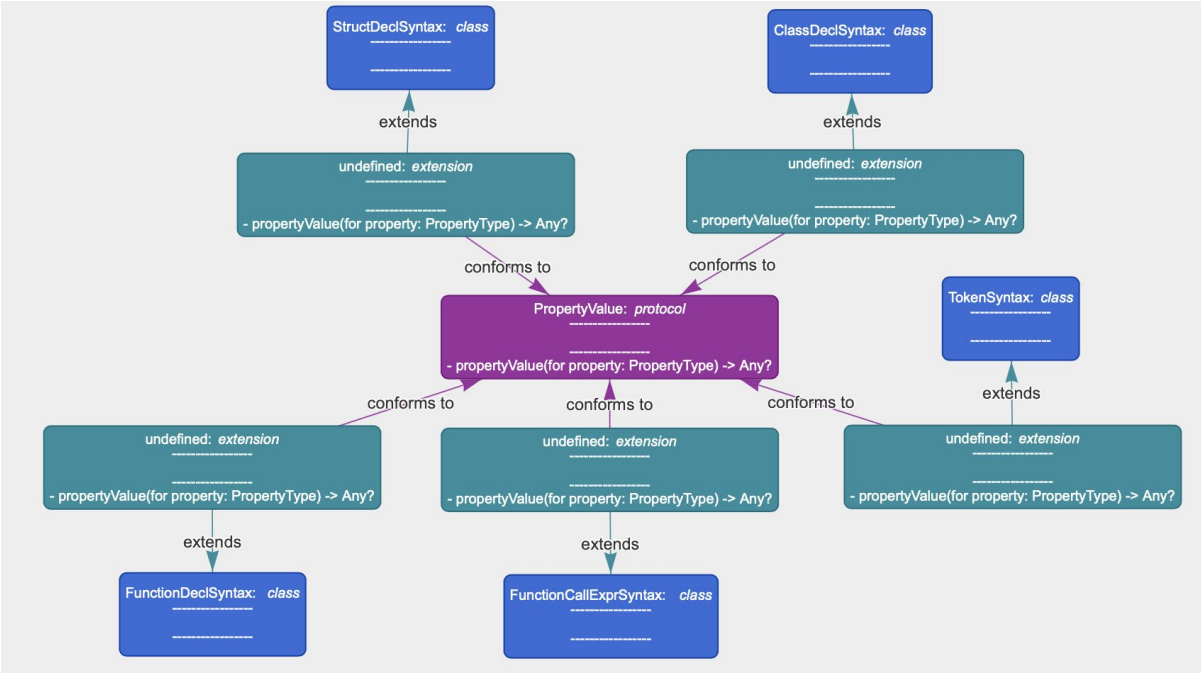

Fig. 3. Class diagram in the part of interaction of the engine with the database

The main obstacle in the development of the software product was getting the ability to work with an abstract syntax tree. Processing the «raw» source code and formalizing it manually is an inefficient task, because it is solved by the initial stages of the Swift language compiler. That is, in the case of own implementation of the processing function, it would be writing the compiler «from scratch». The expediency of such work remains in great doubt [13].

To access the abstract syntax tree generated by the compiler, it is necessary to have access to the lexer and tokenizer, which are components of the compiler from the libSyntax library written in C++. The SwiftSyntax library, which is a high-level wrapper for the libSyntax library [14–16], was used to develop the prototype.

The block diagram of the algorithm of the program is shown in Fig. 4.

Interaction with the program is done through the terminal. To start work, it is necessary to enter a command in the terminal. After that, the program is executed independently and does not require the input of additional data from the user.

After command is received, it is checked for correctness.

If the command turns out to be incorrect, the program terminates execution and displays the corresponding error. If the command is correct, the program continues.

After the command is checked, the class which is used to traverse the file containing the source code is initialized. It is selected and configured according to the task written in the command.

For each node of the program that falls under the conditions specified in the command, the condition is evaluated, and the application of the task is performed, which includes the output of the work result in one form or another.

After processing all the nodes, the program terminates the execution.

To begin with, the proposed software product must support the entities, properties and tasks described in Table 1.

During the evaluation of the results, several tests were conducted. The most indicative tests are presented below.

Fig. 5 shows the source code written in Swift. To check the ability of the created software product to find objects in the code that cannot be found using standard refactoring tools, the following command is executed: *refactor Test. swift find class conforms to FirstProtocol*. The result of the

execution is a range of text of the source code (starting line and character, ending line and character).

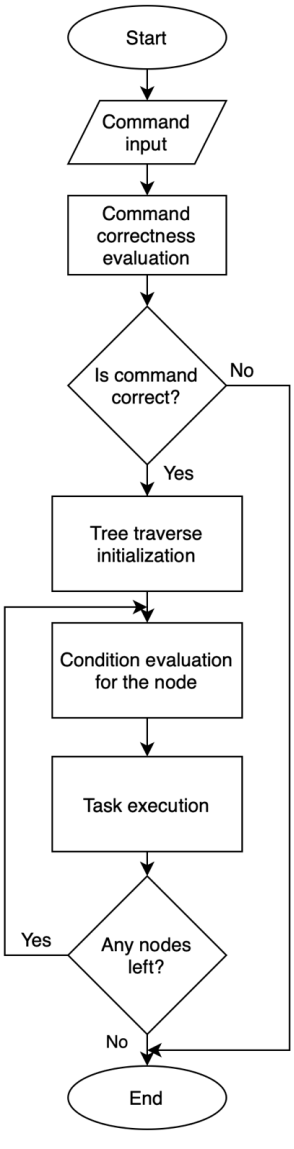

Fig. 4. Block diagram of the program algorithm

Table 1

Proposed functionality

Parameter type	Available alternatives
Tasks	find, rename, localize
Entities	class, struct, string, func
Properties of entities	name, paramsCount, conforms, inherits
Operators	equals, contains, from, to, greaterThanOrEquals

Fig. 6 highlights the range that the program found as a result of execution of the received request. The expected result is the location of the *SecondTestClass* class declaration. As it can be seen, the program successfully coped with the task, despite the fact that the searched class is declared as internal to another class.

In the second test, the same source code as in the first test will be used (Fig. 5). The search for objects based on their characteristics will be done. To do this, it is needed run the command: *refactor Test.swift find func paramsCount equals 2*. The expected result is that the program will find the *shouldReturnTrue* method.

```
class FirstTestClass {
    class SecondTestClass: FirstProtocol {
        var name = "SecondTestClass"

        func shouldReturnTrue(for item: String, secondItem: String) {
            return true
        }
    }

    var name = "FirstTestClass"

    func returnTrue() -> Bool {
        return true
    }
}

protocol FirstProtocol {
    func shouldReturnTrue(for item: String, secondItem: String)
}
```

Fig. 5. Initial source code

```
class FirstTestClass {
    class SecondTestClass: FirstProtocol {
        var name = "SecondTestClass"

        func shouldReturnTrue(for item: String, secondItem: String) {
            return true
        }
    }

    var name = "FirstTestClass"

    func returnTrue() -> Bool {
        return true
    }
}

protocol FirstProtocol {
    func shouldReturnTrue(for item: String, secondItem: String)
}
```

Fig. 6. The result for the first test

In this case, the program produced two results. The first one (Fig. 7) is the declaration of the method, which has two parameters, in the protocol.

```
protocol FirstProtocol {
    func shouldReturnTrue(for item: String, secondItem: String)
}
```

Fig. 7. Result 1 for the second test

The second one (Fig. 8) is the implementation of the method in the class that implements the aforementioned protocol.

```
class FirstTestClass {
    class SecondTestClass: FirstProtocol {
        var name = "SecondTestClass"

        func shouldReturnTrue(for item: String, secondItem: String) {
            return true
        }
    }

    var name = "FirstTestClass"

    func returnTrue() -> Bool {
        return true
    }
}
```

Fig. 8. Result 2 for the second test

The advantage of the proposed software product is that it can perform refactoring tasks of a non-standard type, adapt to the user's needs. However, the main advantage

ISSN 2664-9969

is the speed and flexibility of adding new functionality. To do that, it is only necessary to add description to the knowledge base (special files in the program) of how exactly to associate the code with properties of a certain entity and add keywords to the command handler.

A limitation of this study is that the possibility of obtaining an abstract syntax tree for other programming languages and the possibility of its integration with the developed scheme of the knowledge base was not considered.

During the analysis of the results, it was found that the program has prospects for development and further commercialization. Such directions should include support of multiple files, combining multiple conditions in one command, and support of other programming languages. Also, as a future development of the idea, it is advisable to consider the description of knowledge about the code not in the form of a hierarchy of classes of a specific programming language, but in the form of a hierarchy of concepts and entities. They can be described in OWL-type languages using the appropriate logical deduction tools for those languages.

4. Conclusions

Based on the results of experimental studies, it was confirmed that the automatic analysis and refactoring systems that already exist do not fully satisfy the needs of the end user and are not capable of performing relatively non-trivial refactoring tasks. To solve problems of this kind, knowledge-based refactoring using the concepts and entities of the source code was proposed.

As a result, knowledge-based refactoring allows to form a command descriptively in the form of a task that operates with code concepts – entities, their properties, and actions that can be applied on them. Then the developer who performs refactoring will concentrate only on the formulation of the task, and the program will decide exactly how to apply the described refactoring command to the source code based on what properties the outlined entities have and how to process them.

To experimentally confirm this idea, a software implementation of the refactoring utility for Swift code was developed and a number of non-trivial operations on the code, unavailable in standard refactoring tools, were performed. However, the formulated statements are also valid for processing code written in other modern high-level programming languages.

Conflict of interest

The authors declare that they have no conflict of interest in relation to this study, including financial, personal, authorship, or any other, that could affect the study and its results presented in this article.

Financing

The study was conducted without financial support.

Data availability

The manuscript has associated data in the data repository.

References

1. *Swift Documentation. Swift Local Refactoring.* Available at: https://www.swift.org/blog/swift-local-refactoring/
2. Lacerda, G., Petrillo, F., Pimenta, M., Guéhéneuc, Y. G. (2020). Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software, 167.* doi: https://doi.org/10.1016/j.jss.2020.110610
3. Almogahed, A., Omar, M., Zakaria, N. H. (2022). Refactoring Codes to Improve Software Security Requirements. *Procedia Computer Science, 204,* 108–115. doi: https://doi.org/10.1016/j.procs.2022.08.013
4. *Code refactoring best practices.* Available at: https://www.altexsoft.com/blog/engineering/code-refactoring-best-practices-when-and-when-not-to-do-it/
5. Kaur, S., Singh, P. (2019). How does object-oriented code refactoring influence software quality? Research landscape and challenges. *Journal of Systems and Software,* 157. doi: https://doi.org/10.1016/j.jss.2019.110394
6. Fowler, M. (1999). *Refactoring. Improving the Design of Existing Code.* Addison-Wesley, 63.
7. Morales, R., Soh, Z., Khomh, F., Antoniol, G., Chicano, F. (2017). On the use of developers' context for automatic refactoring of software anti-patterns. *Journal of Systems and Software, 128,* 236–251. doi: https://doi.org/10.1016/j.jss.2016.05.042
8. Don, R., Brant, J. *Refactoring tools.* Available at: http://www.laputan.org/pub/patterns/fowler/Roberts-Brant.doc
9. De Nicola, R., Di Stefano, L., Inverso, O., Uwimbabazi, A. (2022). Automated replication of tuple spaces via static analysis. *Science of Computer Programming, 223.* doi: https://doi.org/10.1016/j.scico.2022.102863
10. Hammad, M., Babur, Ö., Basit, H. A., van den Brand, M. (2022). Clone-Writer: An effective editor for developing code by using code clones. S*oftware Impacts, 13.* doi: https://doi.org/10.1016/j.simpa.2022.100323
11. Al Dallal, J. (2012). Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics. *Information and Software Technology, 54 (10),* 1125–1141. doi: https://doi.org/10.1016/j.infsof.2012.04.004
12. *Swift Syntax Command Line Tool* (2019). Available at: https://www.pointfree.co/episodes/ep55-swift-syntax-command-line-tool
13. Almeida, L. (2019). *An Overview of SwiftSyntax.* Available at: https://medium.com/@lucianoalmeida1/an-overview-of-swifts-yntax-cf1ae6d53494
14. Mattt (2018). *SwiftSyntax.* Available at: https://nshipster.com/swiftsyntax/
15. *SwiftSyntax Documentation.* Available at: https://github.com/apple/swift-syntax/tree/main/Documentation
16. *A set of Swift bindings for the libSyntax library.* Available at: https://iosexample.com/a-set-of-swift-bindings-for-the-libsyntax-library/

✉***Andrii Tkachuk,*** *Postgraduate Student, Department of System Design, National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute», Kyiv, Ukraine, ORCID: https://orcid.org/0000-0002-9127-6381, e-mail: andrewtkachuk@yahoo.com*

Bogdan Bulakh, *PhD, Associate Professor, Department of System Design, National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute», Kyiv, Ukraine, ORCID: https://orcid.org/0000-0001-5880-6101*

✉*Corresponding author*