Tkachuk, Andrii; Bulakh, Bogdan

Article

Research of possibilities of default refactoring actions in Swift language

Technology audit and production reserves

Provided in Cooperation with:
ZBW OAS

Reference: Tkachuk, Andrii/Bulakh, Bogdan (2022). Research of possibilities of default refactoring actions in Swift language. In: Technology audit and production reserves 5 (2/67), S. 6 - 10. http://journals.uran.ua/tarp/article/download/266061/262233/614012. doi:10.15587/2706-5448.2022.266061.

UDC 004.43
DOI: 10.15587/2706-5448.2022.266061
Article type «Reports on Research Projects»

Andrii Tkachuk,
Bogdan Bulakh

RESEARCH OF POSSIBILITIES OF DEFAULT REFACTORING ACTIONS IN SWIFT LANGUAGE

The object of research in the paper is a built-in refactoring mechanism in the Swift programming language. Swift has gained a lot of popularity recently, which is why there are many new challenges associated with the need to support and modify the source code written in this programming language. The problem is that the more powerful refactoring mechanism that can be applied to Swift is proprietary and cannot be used by other software. Moreover, even closed-source refactoring software tools are not capable of performing more complex queries.

To explore the possibilities of expanding the built-in refactoring, it is suggested to investigate the software implementation of the sourcekit component of the Swift programming language, which is responsible for working with «raw» source code, and to implement new refactoring action in practice. To implement the research plan, one refactoring activity that was not present in the refactoring utilities (adding an implementation of the Equatable protocol) was chosen. Its implementation was developed using the components and resources provided within the sourcekit component. To check the correctness and compliance with the development conditions, several tests were created and conducted.

It has been discovered that both refactoring mechanisms supported by the Swift programming language have a limited context and a limited scope and application. That is why the possibility of expanding the functionality should not be based on the local level of code processing, but on the upper level, where it is possible to combine several source files, which often happens in projects. The work was directed to the development of the own refactoring action to analyze and obtain a perfect representation of the advantages and disadvantages of the existing component. As a result, a new approach to refactoring was proposed, which will allow solving the problems described above.

Keywords: *Swift programming language, refactoring, open-source code, sourcekit component.*

Received date: 06.09.2022
Accepted date: 17.10.2022
Published date: 24.10.2022

How to cite
Tkachuk, A., Bulakh, B. (2022). Research of possibilities of default refactoring actions in Swift language. Technology Audit and Production Reserves, 5 (2 (67)), 6–10. doi: https://doi.org/10.15587/2706-5448.2022.266061

1. Introduction

In the Swift programming language, there are two ways to specify a piece of code for refactoring: by to the current position of the cursor in the code and by a selected range of code. Local refactoring takes place within a single file. Extracting a part of an existing method to a new one and combining duplicate expressions (duplicates consolidation) are examples of local refactoring [1]. Global refactoring that changes code in several files (e. g., global renaming) currently demands particular coordination of the Xcode IDE (whose source code is proprietary) and cannot currently be implemented relying solely on Swift source code. Only local refactoring actions are considered in this paper.

Refactoring is determined by the position of the cursor in the editor or by the selected area [2, 3]. Based to how they are initialized, refactoring actions are classified as cursor-based or range-based [4]. For example, the aim of refactoring at a given cursor position in a Swift source file can be renaming of the current token. Range-based refactoring requires both start and end position of the cursor to indicate its aim, such as extracting part of an existing method and declaring it as a new method. To accelerate the development of these two categories of refactoring actions, the sourcekit handler provides preparsed results called ResolvedCursorInfo and ResolvedRangeInfo which contain information about the cursor position or range in a Swift source file.

For example, ResolvedCursorInfo can provide information about whether the cursor position in the Swift source file points to the beginning of an expression and, if so, provide the corresponding abstract syntax tree node for that expression. Also, if the cursor points to a token representing the name of an entity (variable, class, method), ResolvedCursorInfo contains a declaration corresponding to that name. Similarly, ResolvedRangeInfo contains information about a given range of source code, such as whether the code in selected range has several entry or exit points.

The object of research in the paper is the built-in refactoring mechanism in the Swift programming language.

The aim of research is to analyze the limitations of the built-in refactoring mechanism of the Swift language and to find ways to improve this mechanism [5].

2. Research methodology

There is no need to start work with an unprocessed representation of cursor positions or a range to introduce the new refactoring action in Swift programming language. Instead, ResolvedCursorInfo and ResolvedRangeInfo may be utilized; they can be used to retrieve preparsed data relevant for the refactoring.

To write the functionality for refactoring, it is necessary to consider the specifics of the compilation process as a whole [6].

For refactoring, only the first few basic steps of compilation, which transform the source code into an abstract syntax tree (AST), are important.

An Abstract Syntax Tree (AST) is a graph whose main elements are operators (i. e., vertices of the graph) and operands (i. e., leaves).

All nodes of Swift's abstract syntax tree can be divided into three types: Decl (declarations), Expr (expressions), and Stmt (statements).

They correspond to the three entities used in the Swift language. Names of functions, structures, parameters are declarations. Expressions are entities that return values, such as a function call. Statements are parts of the language that define the control flow and execution of code, but do not return a value (for example, if or do-catch). Fig. 1 shows part of an abstract syntax tree that describes the declaration of class members.

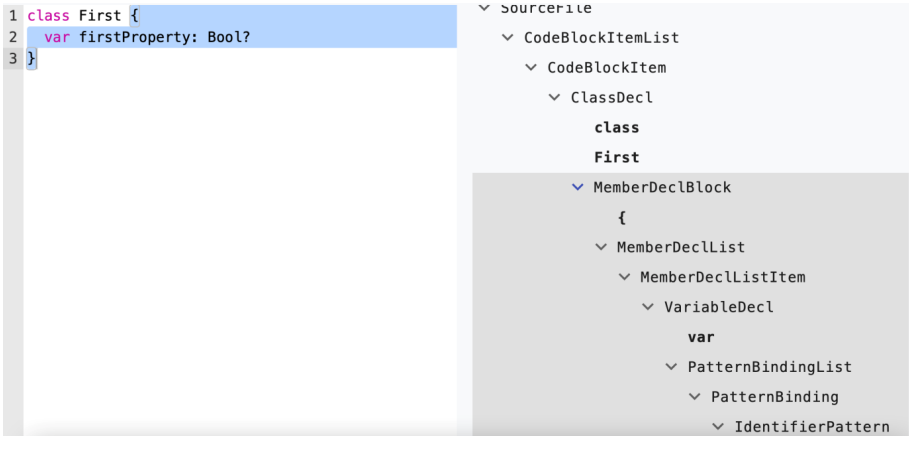

Fig. 1. Visual representation of an abstract syntax tree

```
CURSOR_REFACTORING(AddEquatableConformance, "Add Equatable Conformance", add.equatable.conformance)
```

Fig. 2. Declaration of refactoring

After the AST is available, refactoring embedded in the compiler may be executed (that is, code from the compiler describing possible refactoring methods may be applied to the source code). For refactoring of source code, according to the programmer's needs, an API that allows working with AST is used. The code describing the refactoring action is added to a special file that is later compiled and itself becomes a part of the compilation tools of the given language.

To create a refactoring tool, it is necessary to know how to work with Swift's AST. An important characteristic of the Swift AST is that it is directly linked to the source code because it derives from tokens. This means that it is possible to get a reference to the location of raw code in the source file represented by a specific AST node. Without this information, it would be impossible to rename identifiers, perform movements, and simplify invocation – in general, to perform refactoring.

The AST in its final form has defined types and links to the source code. Only for such an AST is it possible to carry out refactoring in terms of processing the code (checking the implementation of rules, the presence of repetitions, necessary invocations, etc.) and replacing the source code with the necessary one.

To find opportunities to improve refactoring, it is need to:

– learn more about the principles of adding a refactoring action in Swift language;

– familiarize yourself with the principle of Swift source code compilation and construction;

– assess the complexity of developing new actions for refactoring and their integration with the development environment.

To fulfill the goals, it is advisable to complete one of the open proposals in the Swift defect tracking system [7] regarding adding a new refactoring action by implementing it. The task is to add the possibility to automatically complete code with the method required by the Equatable protocol.

To implement refactoring based on cursor position (Add Equatable Conformance), it must be declared in the RefactoringKinds.def file with the entry shown in Fig. 2.

This is necessary so that the IDE, when interacting with the sourcekit, can provide information about available refactoring action and show it in the context menu. CURSOR_REFACTORING keyword indicates that specified refactoring action can be initiated by the position of cursor and will utilize ResolvedCursorInfo object in the implementation. The refactoring action definition contains a few extra things like an internal refactor ID for unique addressing, a string representation that will be displayed to users when the context menu is presented, and a stable key. This definition record allows the compiler of C++ to generate a «skeleton» description of the RefactoringActionAddEquatableConformance class for refactoring and its invocations. Taking this into account, it should be noted that one of the disadvantages of such development is that the developer should focus not directly on the implementation of the necessary functions, but on writing the correct code within the complicated infrastructure of classes for built-in refactoring [8].

After the declaration of a new refactoring action is made, it is necessary to programmatically implement two functions that will give information about:

1) when it is necessary to show the refactoring action;

2) what code change should be applied when the user invokes this refactoring action.

Both functions are automatically generated as part of the class «skeleton» after adding a new refactoring action registration entry. For the IDE to display the new refactoring action as applicable under the correct conditions, it is necessary to implement the isApplicable() function of the RefactoringActionAddEquatableConformance class in

ISSN 2664-9969

Refactoring.cpp (as shown in Fig. 3). ResolvedCursorInfo is an object that contains a description of the context (binding to the code) in which the refactoring action was invoked [9].

The isValid() method (Fig. 4) checks whether the selected declaration context (class, structure, enumeration) meets the conditions for refactoring application, namely: it has stored properties, it does not conform to the Equatable protocol, and whether the requirements of the protocol are correct. If the method returns true, then this type of refactoring will be available in the integrated development environment when the cursor is placed in the appropriate place.

Next, it is necessary to state how the code specified by the cursor position or selection should be transformed if the refactoring action is applied. For this, it is necessary to implement the performChange() method of the RefactoringActionAddEquatableConformance class. During the performChange() implementation, it is possible to access the same ResolvedCursorInfo object that was obtained in isApplicable() method [9].

In Fig. 5, in the performChange() method, the context of the invocation is obtained, based on which the insertion place for the name of the protocol to be added is searched along with the place for members of the class (function) that will also be added. In the body of the function, EditConsumer object can be used to edit the text around the statement pointed by the cursor position with the proper API calls (insertAfter).

All actions for the source code processing are implemented in other additional methods.

The developed refactoring action (after it is applied to the source code) inserts the line «: Equatable» after the token of the selected declaration, which denotes the conformance to the protocol by a class, structure, or enumeration. Then the method implementation required by the Equatable protocol is inserted into the declaration body.

Fig. 6 shows the code that obtains the name of the protocol and formation of the text buffer, which must be inserted into the source code.

Fig. 7 describes the code that receives the text of the method to be inserted. First, the protocol requirements (method name and its parameters) are searched. Next, the necessary indentation is searched, considering the place of insertion of the method body. After that, a class that formally prints the method body in the source code considering all parameters is configured.

In addition to the methods described above, it was necessary to develop a relatively large number of auxiliary functions to organize the correct operation of the main functions.

A special approach for writing tests is used [9]. It is necessary to declare the presence of a certain kind of tests for refactoring actions, create files with input and output data, apply functions, and then the testing framework will compare the obtained result with the desired one. Fig. 8 shows an example of one of these tests.

```
bool RefactoringActionAddEquatableConformance::
isApplicable(ResolvedCursorInfo Tok, DiagnosticEngine &Diag) {
  return AddEquatableContext::getDeclarationContextFromInfo(Tok).isValid();
}
```

Fig. 3. Implementation of the isApplicable() method

```
bool isValid() {
  // FIXME: Allow to generate explicit == method for declarations which
  // already have compiler-generated == method
  return StartLoc.isValid() && ProtInsertStartLoc.isValid() &&
  !conformsToEquatableProtocol() && isPropertiesListValid() &&
  isRequirementValid();
}
```

Fig. 4. Implementation of the isValid() method

```
bool RefactoringActionAddEquatableConformance::
performChange() {
  auto Context = AddEquatableContext::getDeclarationContextFromInfo(CursorInfo);
  EditConsumer.insertAfter(SM, Context.getStartLocForProtocolDecl(),
                           Context.getInsertionTextForProtocol());
  EditConsumer.insertAfter(SM, Context.getInsertStartLoc(),
                           Context.getInsertionTextForFunction(SM));
  return false;
}
```

Fig. 5. Implementation of the performChange() method

```
std::string AddEquatableContext::
getInsertionTextForProtocol() {
  StringRef ProtocolName = getProtocolName(KnownProtocolKind::Equatable);
  std::string Buffer;
  llvm::raw_string_ostream OS(Buffer);
  if (ProtocolsLocations.empty()) {
    OS << ": " << ProtocolName;
    return Buffer;
  }
  OS << ", " << ProtocolName;
  return Buffer;
}
```

Fig. 6. Obtaining the text for the protocol

```
std::string AddEquatableContext::
getInsertionTextForFunction(SourceManager &SM) {
  auto Reqs = getProtocolRequirements();
  auto Req = dyn_cast<FuncDecl>(Reqs[0]);
  auto Params = Req->getParameters();
  StringRef ExtraIndent;
  StringRef CurrentIndent =
    Lexer::getIndentationForLine(SM, getInsertStartLoc(), &ExtraIndent);
  std::string Indent;
  if (isMembersRangeEmpty()) {
    Indent = (CurrentIndent + ExtraIndent).str();
  } else {
    Indent = CurrentIndent.str();
  }
  PrintOptions Options = PrintOptions::printVerbose();
  Options.PrintDocumentationComments = false;
  Options.setBaseType(Adopter);
  Options.FunctionBody = [&](const ValueDecl *VD, ASTPrinter &Printer) {
    Printer << " {";
    Printer.printNewline();
    printFunctionBody(Printer, ExtraIndent, Params);
    Printer.printNewline();
    Printer << "}";
  };
  std::string Buffer;
  llvm::raw_string_ostream OS(Buffer);
  ExtraIndentStreamPrinter Printer(OS, Indent);
  Printer.printNewline();
  if (!isMembersRangeEmpty()) {
    Printer.printNewline();
  }
  Reqs[0]->print(Printer, Options);
  return Buffer;
}
```

Fig. 7. Obtaining the text for the protocol method

```
extension TestAddEquatable {
    func test() -> Bool {
        return true
    }
}

extension TestAddEquatable {
}

// RUN: rm -rf %t.result && mkdir -p %t.result

// RUN: %refactor -add-equatable-conformance -source-filename %s -pos=1:16 > %t.result/first.swift
// RUN: diff -u %S/Outputs/basic/first.swift.expected %t.result/first.swift
```

Fig. 8. Writing the test

The development of refactoring actions, which are based on the selected range of code, is done in a similar way:
– action declaration in the RefactoringKinds.def file;
– implementation of the isApplicable() method, which shows when the refactoring action can be applied (the ResolvedRangeInfo object is used to describe the context of the invocation);
– implementation of the performChange() method for applying changes;
– testing.

Importantly, it is possible to access in the body of the performChange function not only the original ResolvedCursorInfo or ResolvedRangeInfo objects for the location or range selected by the user, but also other important utilities such as EditConsumer and SourceManager, which makes the implementation more convenient.

3. Research results and discussion

As a result of experiments with the existing API of the sourcekit utility responsible for refactoring in the Swift language, it was found that to add just one relatively simple refactoring action it was necessary to write about 300 lines of code. And there's no reason to believe that it will be needed to write significantly fewer lines of code in other cases. Thus, there is a crucial need to improve and automate the advanced refactoring process, which would allow the programmer to avoid writing a considerable amount of «infrastructure» code and to focuse more on the aim of refactoring. The written refactoring action code is a part of the Swift language and cannot be parameterized at runtime (all new refactoring functions that are added are strictly formalized and implemented only according to the provided framework). Let's review the result of application of the developed refactoring action. The source file for refactoring is shown in Fig. 9.

```
class TestAddEquatable {
    var property = "test"
    private var prop = "test2"
    let pr = "test3"
}

extension TestAddEquatable {
    func test() -> Bool {
        return true
    }
}

extension TestAddEquatable {
}
```

Fig. 9. Output file

Refactoring action is applied to the declaration of the TestAddEquatable class. The result of the utility execution is shown in Fig. 10 – protocol name after class name and method implementation in class body using all class properties are added.

Next, refactoring action is applied to the extension of the TestAddEquatable class, which contains some information in its body. The result of the work looks as shown in Fig. 11.

```
class TestAddEquatable: Equatable {
    var property = "test"
    private var prop = "test2"
    let pr = "test3"

    static func == (lhs: TestAddEquatable, rhs: TestAddEquatable) -> Bool {
        return lhs.property == rhs.property &&
               lhs.prop == rhs.prop &&
               lhs.pr == rhs.pr
    }
}

extension TestAddEquatable {
    func test() -> Bool {
        return true
    }
}

extension TestAddEquatable {
}
```

Fig. 10. The result of using the utility for the class

```
class TestAddEquatable {
    var property = "test"
    private var prop = "test2"
    let pr = "test3"
}

extension TestAddEquatable: Equatable {
    func test() -> Bool {
        return true
    }

    static func == (lhs: TestAddEquatable, rhs: TestAddEquatable) -> Bool {
        return lhs.property == rhs.property &&
               lhs.prop == rhs.prop &&
               lhs.pr == rhs.pr
    }
}

extension TestAddEquatable {
}
```

Fig. 11. The result of using the utility to expand with text

A limitation of this study is the focus on one specific type of refactoring that was added.

The development of methods and approaches of refactoring, which do not depend entirely on the source code of the programming language, should be considered as

ISSN 2664-9969

a promising direction for the development of this research. This is due to the fact that the new approach will not have limitations for refactoring actions that exist in the old one (such as access only to the context of the invoked action) [10].

4. Conclusions

The method of performing refactoring in the Swift language, based on the features of the sourcekit «raw» source code handler is studied in this paper. The sequence of adding a new type of refactoring to the source code of the Swift programming language is described, as well as the functionality available for writing new refactoring actions is considered.

The work of the programming language compiler and the structure of the abstract syntactic tree, which is the result of the work of several first stages of code compilation, were considered. Using binding to an abstract syntax tree is a necessity for building functional tools for non-trivial refactoring. However, experimental studies have shown that writing refactoring rules with built-in Swift language tools is possible only in a strictly defined way using classes and specific requirements of a provided framework. The refactoring functionality created in this way cannot be properly parameterized, and the process of creating the refactoring functionality itself is too time-consuming. That is why the possibility of parameterization and the flexibility of utilities for refactoring based on formalized knowledge about the source code are priority directions for further research.

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 performed without financial support.

Data availability

The manuscript has associated data in a data repository.

References

1. Sandoval Alcocer, J. P., Siles Antezana, A., Santos, G., Bergel, A. (2020). Improving the success rate of applying the extract method refactoring. *Science of Computer Programming, 195*, 102475. doi: https://doi.org/10.1016/j.scico.2020.102475
2. Kaur, G., Singh, B. (2017). Improving the quality of software by refactoring. *2017 International Conference on Intelligent Computing and Control Systems (ICICCS)*, 185–191.
3. Saca, M. A. (2017). Refactoring improving the design of existing code. *2017 IEEE 37th Central America and Panama Convention (CONCAPAN XXXVII)*. doi: https://doi.org/10.1109/concapan.2017.8278488
4. Simmonds, J. (2002). *A Comparison of Software Refactoring Tools*. Available at: https://www.researchgate.net/publication/2946408_A_Comparison_of_Software_Refactoring_Tools
5. Chouchen, M., Olongo, J., Ouni, A., Mkaouer, M. W. (2021). *Predicting Code Review Completion Time in Modern Code Review*. ArXiv. doi: https://doi.org/10.48550/arXiv.2109.15141
6. Inoue, K., Roy, C. K. (2021). *Code Clone Analysis*. Singapore: Springer. doi: https://doi.org/10.1007/978-981-16-1927-4
7. *Swift Issues*. Available at: https://github.com/apple/swift/issues/
8. Lacerda, G., Petrillo, F., Pimenta, M. S., Guéhéneuc, Y. (2020). *Code Smells and Refactoring: A Tertiary Systematic Review of Challenges and Observations*. ArXiv. doi: https://doi.org/10.48550/arXiv.2004.10777
9. *Swift Local Refactoring*. Available at: https://swift.org/blog/swift-local-refactoring/
10. *Improving Swift Tools with libSyntax*. Available at: https://academy.realm.io/posts/improving-swift-tools-with-libsyntax-try-swift-haskin-2017/

✉***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*