Improving Nothingness: Refactorings on Whitespace

RUTGER WITMANS, University of Twente, The Netherlands

Esoteric languages are known for being unique, clunky to code in and generally not used for real-life applications. However, as an experimentation tool, they can be powerful to find out certain answers. Refactoring is a systematic process of improving code without creating new functionality that transforms a mess into clean code and simple design. This powerful process creates code which can be used long-term and is better to understand. In this paper, we are presenting which refactorings are possible in the minimal setting that the programming language Whitespace offers, where only comments describe what the program does. After showing the refactorings that are possible on Whitespace, we will then present a tool which automates some of these refactorings. Finally, we will present the tests we have performed on the tool to check its validity.

Additional Key Words and Phrases: Whitespace, Refactoring, esoteric, programming language

1 INTRODUCTION

"Refactoring is a systematic process of improving code without creating new functionality that can transform a mess into clean code and simple design" [1]. Refactoring is an important part of being a programmer. Refactoring unclutters code, creates more optimized code and refactoring gives the programmer an easier time coding later. There are over 60 refactoring methods [1] all of which can improve the code quality of programming projects.

Furthermore, if your language of choice has a small instruction set, it is then important to keep clarity in your code to maintain the code base. Some esoteric languages for instance are designed to keep the number of allowed characters in the language to a minimum. Two of these examples include Brainf*ck and Whitespace.

Brainf*ck, designed in 1993 by Urban Müller [6], was created to have the smallest possible compiler that exists for a language. The language consists of only eight characters which together are Turing complete and are thus able to solve any computational problem like most other programming languages.

As the name suggests, Whitespace is a programming language where the only characters recognized by the compiler are whitespace characters [3]. Every other character is ignored by the compiler. A program designed in this language only consists of tabs, spaces and line feed (new line) characters. This means that working Whitespace programs to us humans look like empty files while in reality, these files contain all sorts of functionality. Thus if you would like to show others what you want to do, you need to add comments to the file. Whitespace thus shows in a way the importance of commenting. In most projects, we work with large code bases. If there are barely any comments, it can be difficult to grasp how some things work together. Commenting improves this lack of clarity and gives the programmer a better idea of how everything works. This language is also Turing complete and is thus able to solve any computational problems like most other programming languages.

It might seem pointless to refactor Whitespace code because of the lack of real-life applications. However, as stated before refactoring has more benefits next to clean code. The refactored code is safer, better optimized and easier to maintain.

First, we will look at different refactorings and look at which of these refactorings are possible within the scope of Whitespace. After finding out which refactorings are possible on Whitespace, we will create a solution which seeks out these refactoring possibilities and applies them to the code.

Our goal can then be defined as creating a tool which finds and applies refactorings on a given Whitespace program. The following research questions will help achieve our goal:

- RQ 1: Which refactorings are possible on Whitespace code?
- RQ 2: Can we create a validated tool which detects and applies the possible refactorings on Whitespace?

TScIT 38, Februari, 2023, Enschede, The Netherlands

2 RELATED WORK

Because Esoteric languages are not designed to be used in real-life applications, people use these languages to experiment and challenge certain existing ideas to see whether there are better options available in the future [9]. Languages such as Three Star Programmer explore ideas on what is needed to have a Turing complete language [9].

Refactoring has first been coined by Opdyke et al. [8], yet refactoring was by then already a big part of the programming cycle. One big contributor to the popularization of most refactoring methods is the book *Refactoring - Improving the Design of Existing Code* [7]. This book first stresses the importance of refactoring code. Next to an introduction to refactoring, this handbook provides a catalogue of dozens of tips for improving code with the use of refactorings methods. The importance of refactoring can not be understated. Refactoring has multiple benefits, such as making your code easier to test [12]. Next to being able to refactor as the coder, there exist tools which refactor code for programmers instead of the programmer performing the refactorings. One paper presents a tool that is able to refactor large c++ code bases using clangMR [11]. In Another paper, Baqais et. al performed a systematic literature review that suggests, proposes or implements an automated refactoring process [2]. Finally, there is different work on the detection of possible refactorings. Tsantalis et al. argue that the placement of different class methods and class attributes is guided by metrics and conceptual criteria [10]. The paper proposes a methodology which decides how effective different placements of classes and attributes are. The method is semi-automatic since the user still has to decide if the refactoring should be applied.

While refactoring of esoteric languages has little research into it, we believe it is important to see which refactorings are possible on such a minimal language.

3 METHODOLOGY AND APPROACH

First, we will find out which refactorings exist and can be deployed on the language Whitespace. We will then create a Whitespace program which serves as a baseline program where refactorings are going to be applied too. Finally, we will build a tool which finds and tries to apply refactorings to improve the quality of the Whitespace program.

4 POSSIBLE REFACTORINGS

4.1 Refactoring categories

We first looked at what refactoring categories are possible. The following categories were available [1]:

- Composing Methods
- Moving Features between Objects
- Organizing Data
- Simplifying Conditional Expressions
- Simplifying Method Calls
- Dealing with Generalization
- Code Smells

We are ruling out everything that has to do with object programming patterns since Whitespace is an assembler-like language. Such languages generally miss the object programming data structures needed to perform such refactorings. Thus *Moving features between objects*, *organizing data* and *dealing with generalization* are high-level refactorings which we will not go over. Whitespace does have instructions which are called labels. These labels are points in the code you can jump to where a certain piece of code gets executed. This functionality borrows some refactoring ideas from the *simplifying method calls* and *composing methods*. Whitespace also has instructions for conditional jumps. This makes some of the ideas for *simplifying conditional expressions* possible. Next to this, we will look at some *code* smells and perform refactorings. We will thus be looking through the following categories:

- Composing Methods
- Simplifying Conditional Expressions
- Simplifying Method Calls
- Code Smells

4.2 Possible refactoring methods

With these chosen categories, we have created a list of refactorings which can be performed on Whitespace code. The following list is the refactorings we found that are possible on Whitespace code:

- Extract method
- Inline method
- Rename method
- Consolidate conditional expression
- Consolidate duplicate conditional fragments
- Remove dead code
- Remove duplicate methods

4.2.1 Extract method. The *extract method* refactoring is a refactoring where a grouped sequence of instructions gets extracted into its own method so that this new method describes with its method name what the sequence of instructions is supposed to do. This is useful when you have a large method which does multiple sub-tasks to perform its functionality. Making it clear what the function does in these sub-steps is nice for the next reader of the code, so the readers are able to easily deduce what your code does.

4.2.2 Inline method. The *inline method* refactoring is the opposite of this. If some functionality of a method is small, there is the possibility of performing that function on the spot. Refactoring code with this method gets rid of code which clutters the program without bringing new functionality. We see that these first two methods of refactoring have opposite ideas in mind, yet both methods are able to be utilized exclusively from each other. For some methods, you might have made use of too many methods. This makes it unclear how the method works. On the other hand, using too few methods overwhelms the reader and makes the reader get lost in certain details which are not important. Because of this balance, it will be tricky to automate this process. While it is possible to automate this based on self-defined predicates, we will not be doing this in our paper because this is beyond the scope of this research.

4.2.3 Consolidate conditional expression. The *consolidate conditional expression* refactoring is a refactoring method where one looks at all the different branches and then checks what branches lead to the same instructions. We then group these branches into a singular conditional statement that performs these actions. Grouping these conditionals gives clarity to code, especially if you name this expression. While this can be done in whitespace using labels and performing the conditional logic under one of these labels, we would like to argue that this refactoring is still too subjective. We cannot easily decide whether a conditional statement is complex and needs changing. We have thus decided not to implement this refactoring into the tool.

4.2.4 Consolidate duplicate conditional fragments. The *consolidate duplicate conditional fragments* refactoring checks whether all branches execute the same piece of code and then extracts this piece out of the branches. This refactoring makes clear what piece of code always needs to be executed no matter what conditional branch you might have taken. This clears up confusion about what the if-statement tries to separate resulting in cleaner code. We have chosen not to implement this method.

4.3 Chosen refactorings for automation

After eliminating those refactorings, we have come to the following three refactorings we will automate:

- Rename method
- Remove dead code
- Remove duplicate methods

4.3.1 Rename method. The *rename method* refactoring is quite self-explanatory. The purpose of this refactoring normally is to rename the method in order to make it more clear what the method does. In the case of Whitespace, this is impossible. Labels do not have ordinary names. Instead, they are made up of a combination of tabs and spaces. Because of this, the naming of labels is purely there to keep uniqueness. However, since the naming does not matter, we instead rename the labels to keep them as small as possible. Not only will this increase the number of labels we will have available to us, but it will also allow us to keep the Whitespace code as small as possible.

4.3.2 *Remove dead code.*

To explain *removing dead code*, we will first explain what dead code is. "Dead or inactive code is any code that has no effect on the application's behaviour" [5]. With this definition, we see that we want to remove code that has no effect on the application we are writing. While this is trivial to do as a human, as a robot it is quite hard to notice when code is unused. Because of this, we will eliminate unused methods instead, to keep complications lower.

4.3.3 *Remove duplicate methods.*

Last up, we will be *removing duplicate methods*. Duplicate methods are two methods which have the exact same functionality. Our tool is going to remove these methods since duplicate methods only cause confusion and do not have any benefits to a programmer.

5 AUTOMATIC REFACTORING

For an automatic refactoring tool to work, we need to be able to perform the following steps:

- Reading a Whitespace file and constructing the Whitespace code into an intermediate representation which is easier to work with.
- Performing checks on the intermediate representation and then performing refactorings based on those checks
- Transforming the results from the refactorings back into Whitespace code.

5.1 Whitespace library

For our research, we need a tool which parses Whitespace code and turns it into an intermediate representation and after the refactorings turn the newly refactored intermediate representation back into Whitespace code. We have decided to use the Rust library "whitespacers" [4]. This tool has all the features necessary for testing, creating and transforming Whitespace code. The library has created its own intermediate representation, thus saving us the hassle of coming up with such a representation. The library is able to run our Whitespace programs, giving us the ability to test for changes in behaviour.
One more feature this library has which was unexpected was the ability to minimize label names. Since we already decided that we wanted to perform this action in our tool, we have decided to use the implementation of the tool for this, thus achieving our first refactoring.

5.2 Intermediate Representation

To show what our Intermediate Representation (IR) looks like, we first have to explain how Whitespace works. Whitespace has five different types of commands. These types all have a different Instruction Modification Parameter (IMP). The IMP is a unique sequence of whitespace characters that selects one of these instruction types. After choosing an instruction type, you now enter the corresponding combination of whitespace characters to select the instruction you want. Some of these instructions have parameters, which are a

```
    Push {value: Integer},
    PushBig {value: BigInteger},
    Duplicate,
    Copy {index: usize},
    Swap,
    Discard,
    Slide {amount: usize},
    Add,
    Subtract,
    Multiply,
    Divide,
    Modulo,
    Set,
    Get,
    Label,
    Call {index: usize},
    Jump {index: usize},
    JumpIfZero {index: usize},
    JumpIfNegative {index: usize},
    EndSubroutine,
    EndProgram,
    PrintChar,
    PrintNum,
    InputChar,
    InputNum
```

Fig. 1. The IR of Whitespace

sequence of tabs and spaces, terminated with a Line Feed character. All Whitespace programs end with three line feed characters, indicating that there is no more code to parse. Combining all these instructions gives us a total of 24 instructions in the Whitespace language.
With this in mind, in figure 1 you will see the IR of the Whitespace library we have decided to use. The 24 commands all have their own unique and human-understandable name. Using this IR, we are able to create our refactorings in an easier-to-understand language.

5.3 Example program with whitespace

In 2, you will see a complete working program in Whitespace, specifically, a "Hello, world!" program. In the example, you see a combination of spaces (the vertical stripes), tabs (the horizontal stripes) and line-end characters at the end of each line. For most people, it is not clear how this program should behave. For example, some lines contain more than one instruction. That is why we would like to use the IR. In 3, we find the IR version of the same Whitespace program. Here it becomes clear that every letter first gets pushed onto the stack by their ASCII code and then printed. When it finally finished printing the last character, the program exits. Using the IR to create Whitespace programs was convenient for us since it sped up the time it took to create and analyze test programs.

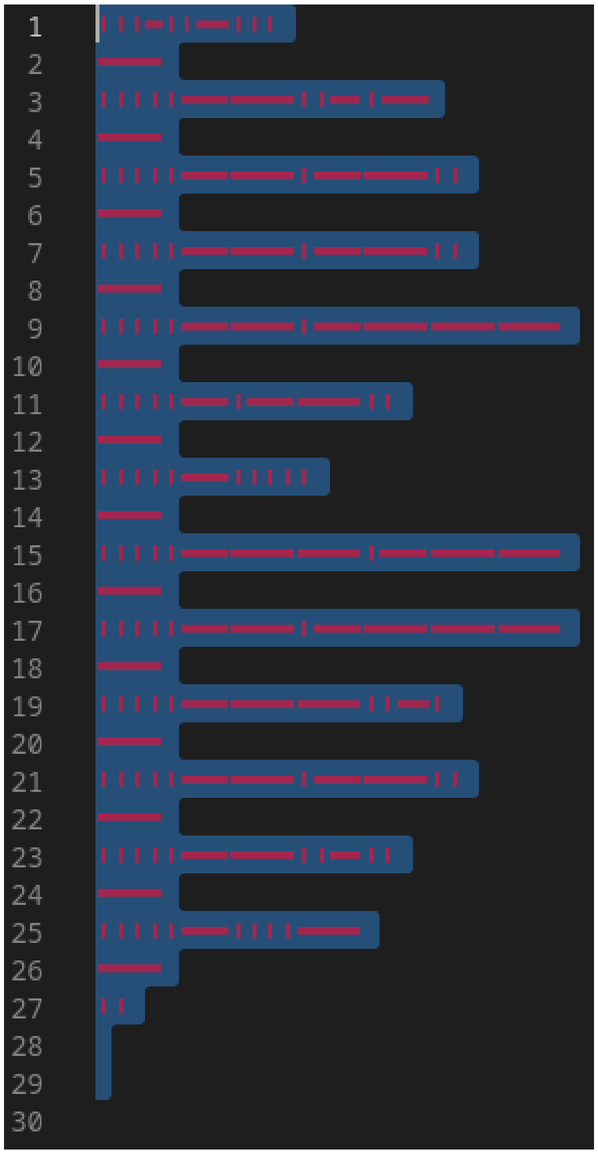
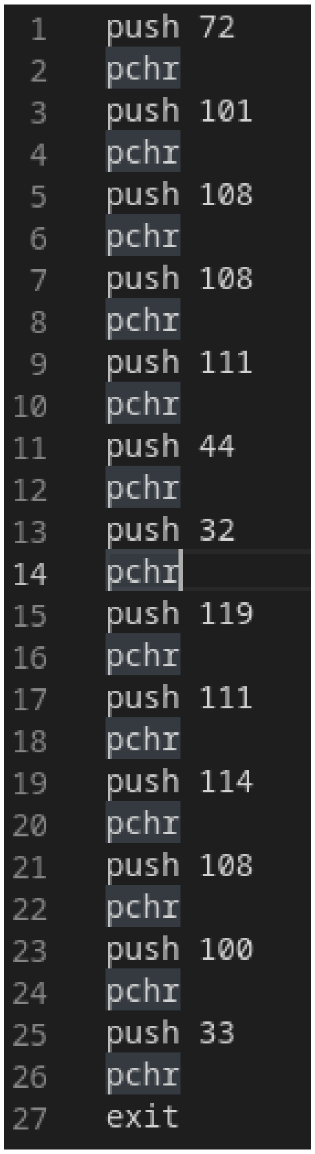


Fig. 2. Hello, world! program in whitespace



```
push 72
pchr
push 101
pchr
push 108
pchr
push 108
pchr
push 111
pchr
push 44
pchr
push 32
pchr
push 119
pchr
push 111
pchr
push 114
pchr
push 108
pchr
push 100
pchr
push 33
pchr
exit
```

Fig. 3. Hello, world! program in whitespace

5.4 Removing dead code

For our dead code removal, we have created a plan to detect unused methods and then remove these methods. Our plan is as follows:

- Look at all our jump instructions and store to what label they jump to.
- If there is a label which is not jumped towards, we will eliminate this label with the code corresponding with this label.

Using this approach we are easily able to detect if methods are not called. There are some downsides to this method which we will now point out.
If there are two methods which will reference each other that are not called through the main method, they will both still be seen as used code. This can be fixed by storing the label in which the method is called, and seeing whether this name space is reached via the main method. If it is, then this piece of code is not dead, otherwise, you can mark it as dead code.
That would not fix the second issue, however. If the code mentions a jump to a certain label, but it would never take this jump, then this called method would still be seen as a used piece of code. However, This cannot be true since this part of the code is never reached. One would have to guarantee that this piece cannot be reached using more complicated techniques.
Finally, we are just looking at dead methods and not dead code in general. If code is specifically told to stop the execution and there are calls to other methods after stopping execution, these called

methods should be seen as dead. However, since we have not put in checks to detect this behaviour, these methods are not removed.

5.5 Removing duplicate methods

For removing duplicate methods, we have created a plan to detect these instances. Our plan is as follows:

- Analyze the code of all the methods.
- Group methods that have duplicate code.
- Remove grouped methods until there is one left.
- Change all jumps from the removed labels to the grouped method that is left.

With this, we have created a way to remove duplicate code without changing behaviour. There is one issue left with this implementation. If two instructions are swapped which are interchangeable, this plan would not be comprehensive to detect all method duplication. The way to fix this interchangeable code problem is to find all patterns where code can be interchanged without changing behaviour and detect duplicate code using these patterns.

5.6 Writing test programs

With all of the refactorings finished, we needed some test programs to test whether the refactorings are applied correctly and kept their behaviour. This turned out to be a problem, since writing valid Whitespace code is not human-friendly. However, we solved this problem by writing in the format of the library their IR. The library was then able to recognize this format and transform the IR into a whitespace-encoded file, solving the issue of writing Whitespace code.

6 TESTING

For testing, we will be highlighting the two different refactoring methods. We will first show an equivalent problem in a python program, and then show equivalent code in the Whitespace IR before and after the refactoring. Finally, we will show you the difference in the Whitespace code. In both tests, there are initially long labels. In both tests, these labels get renamed to the smallest possible unique label.

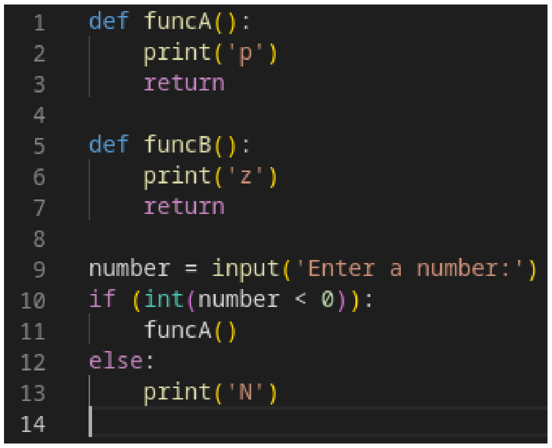
6.1 Testing of removal dead code

In 4 we see a python program which contains two methods. FuncA is in use and funcB is never used in the program. Since funcB does not do anything, we would like to recognize this method as being dead and eliminate it.

In 9 we see the IR of whitespace code before and after refactoring. Before the refactoring, we see big label names with a method which is never called in the lifetime of the program. Since this method is not used, it is marked as dead code by the tool. After refactoring, we subsequently see that this method has been removed. We also see that the labels of the methods have been shortened to the smallest possible unique names. In 6 we see the difference in the whitespace that has been generated.

6.2 Testing of duplicate methods removal

In 5 we see a python program which has two methods and both methods are being called in the main function. When we look closer,



```python
def funcA():
    print('p')
    return

def funcB():
    print('z')
    return

number = input('Enter a number:')
if (int(number < 0)):
    funcA()
else:
    print('N')

```

Fig. 4. Example of unused methods in Python

we see that both methods achieve the same thing, printing one letter and returning back to the calling place. Ideally, the tool should mark that these methods achieve the same functionality. It should then remove one of the two methods and replace all calls to the method with the left-over method.

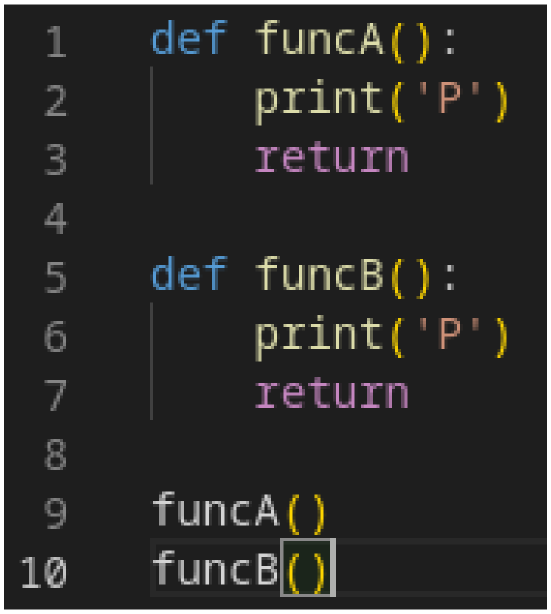
In 7 we see the IR of whitespace code before and after refactoring. Before the refactoring, we see big label names and two methods which achieve the same functionality. In the main method, both methods are called and then the program exits. Our tool will mark both methods, and removes one of the two methods. The tool then replaces all of the method calls to the remaining method. We see that after refactoring, this is indeed what has happened to the file. We see only one function with a smaller label, and both jumps are to this remaining method. In 6 we see the difference in the whitespace that has been generated.

7 CONCLUSION

In this paper, we have presented a tool that checks and performs refactorings on Whitespace code. To answer research question one, we first looked at different refactoring categories. From there we found seven refactorings that are possible on the Whitespace language. To answer research question 2, we first chose three refactorings we wanted to implement into the tool. We then implemented a tool which reads Whitespace code, performs refactorings on this code using the generated IR, and transforms the IR back into Whitespace code. This shows that it is possible to create a tool which detects and applies possible refactorings on Whitespace. This work shows that even with minimal circumstances, it is always possible to refactor code. Furthermore, refactoring code is always useful, be that code clarity or a minimal code footprint. We conclude that refactoring Whitespace code is possible and that refactoring Whitespace code improves the readability and usability of such code.

8 FUTURE WORK

There are several areas for future research that can build upon the work presented here. In this section, we will discuss potential extensions to our approach.


```python
def funcA():
    print('P')
    return

def funcB():
    print('P')
    return

funcA()
funcB()
```

Fig. 5. Example of duplicate methods in Python

The refactorings proposed in this paper are a work in progress and further research and development are needed to fully realize the functionality. Next to this, more refactorings can be implemented, such as the different conditional refactorings mentioned in section 4.2 "Possible refactoring methods".
Furthermore, while some testing has been performed, there could be more tests added. Generating tests to show results that accurately depict the tool is something worth considering to be done. Finally, doing some tests on the speed of the tool and looking where it is the slowest and why will also benefit the usability of the tool and future additions. For anyone who would like to look at the tool or work on it further, you can find the tool over at https://github.com/rwitmans/whiteref/tree/master/whiteref.

REFERENCES

[1] 2022. Refactoring: clean your code. https://refactoring.guru/refactoring [Online; accessed 22. Nov. 2022].

[2] Abdulrahman Ahmed Bobakr Baqais and Mohammad Alshayeb. 2020. Automatic software refactoring: a systematic literature review. *Software Qual. J.* 28, 2 (June 2020), 459–502. https://doi.org/10.1007/s11219-019-09477-y

[3] Edwin Brady. 2022. Whitespace. https://web.archive.org/web/20150623025348/http://compsoc.dur.ac.uk/whitespace [Online; accessed 23. Nov. 2022].

[4] CensoredUsername. 2023. whitespace-rs. https://github.com/CensoredUsername/whitespace-rs [Online; accessed 19. Jan. 2023].

[5] Cato de Kruif. 2022. Using d-NFGs to identify and eliminate dead code in C programs. http://essay.utwente.nl/91890/

[6] Brandee Easter. 2020. Fully Human, Fully Machine: Rhetorics of Digital Disembodiment in Programming. *Rhetoric Review* 39, 2 (April 2020), 202–215. https://doi.org/10.1080/07350198.2020.1727096

[7] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA.

[8] William F. Opdyke and Ralph E. Johnson. 1993. Creating Abstract Superclasses by Refactoring. *ResearchGate* (Jan. 1993), 66–73. https://doi.org/10.1145/170791.170804

[9] Daniel Temkin. 2017. Language without code: intentionally unusable, uncomputable, or conceptual programming languages. *1.* 9, 3 (Sept. 2017), 83–91. https://doi.org/10.7559/citarj.v9i3.432

[10] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2009. Identification of Move Method Refactoring Opportunities. *IEEE Trans. Software Eng.* 35, 3 (Jan. 2009), 347–367. https://doi.org/10.1109/TSE.2009.1

[11] Hyrum K. Wright, Daniel Jasper, Manuel Klimek, Chandler Carruth, and Zhanyong Wan. 2013. Large-Scale Automated Refactoring Using ClangMR. In *2013 IEEE International Conference on Software Maintenance.* IEEE, 548–551. https://doi.org/10.1109/ICSM.2013.93

[12] Morteza Zakeri-Nasrabadi and Saeed Parsa. 2022. An ensemble meta-estimator to predict source code testability■. *Appl. Soft Comput.* 129, C (Nov. 2022). https://doi.org/10.1016/j.asoc.2022.109562

Appendices

A BEFORE AND AFTER RESULTS OF REFACTORING WHITESPACE CODE

A.1 The refactoring of duplicate method

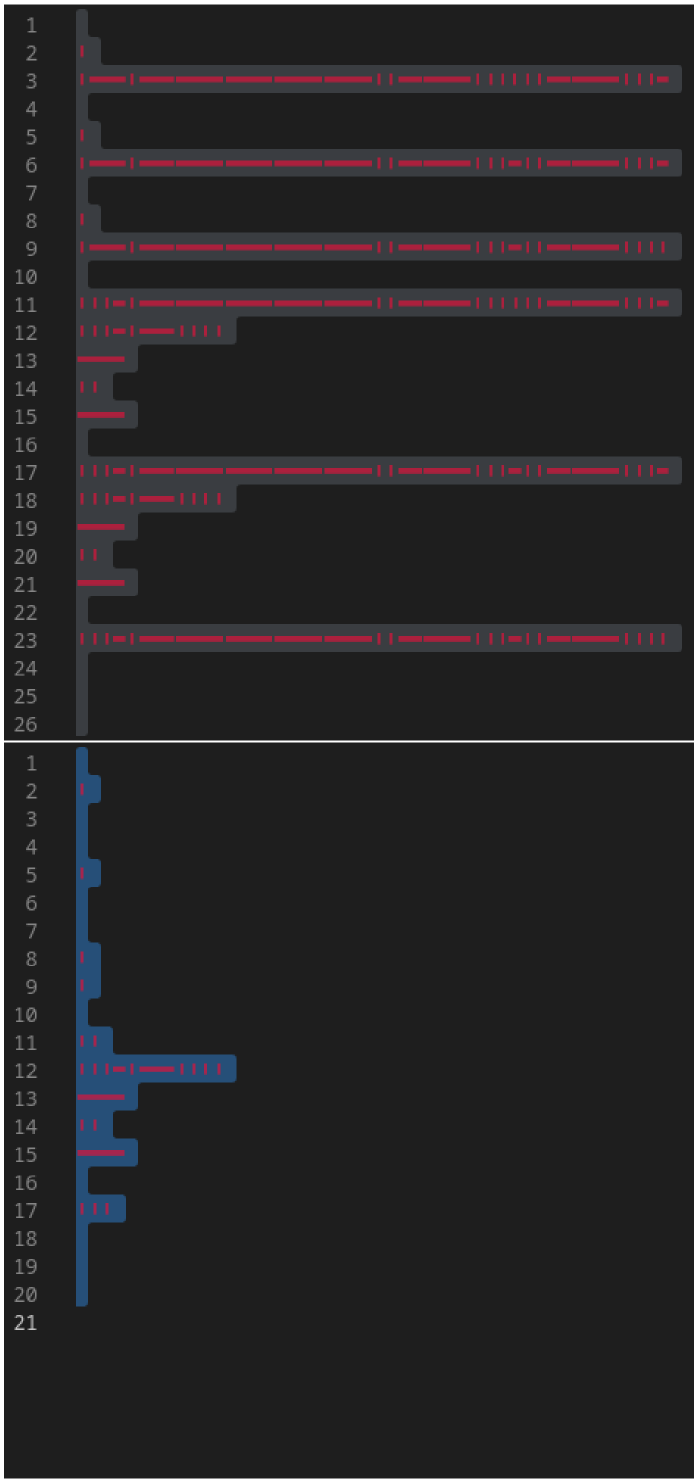
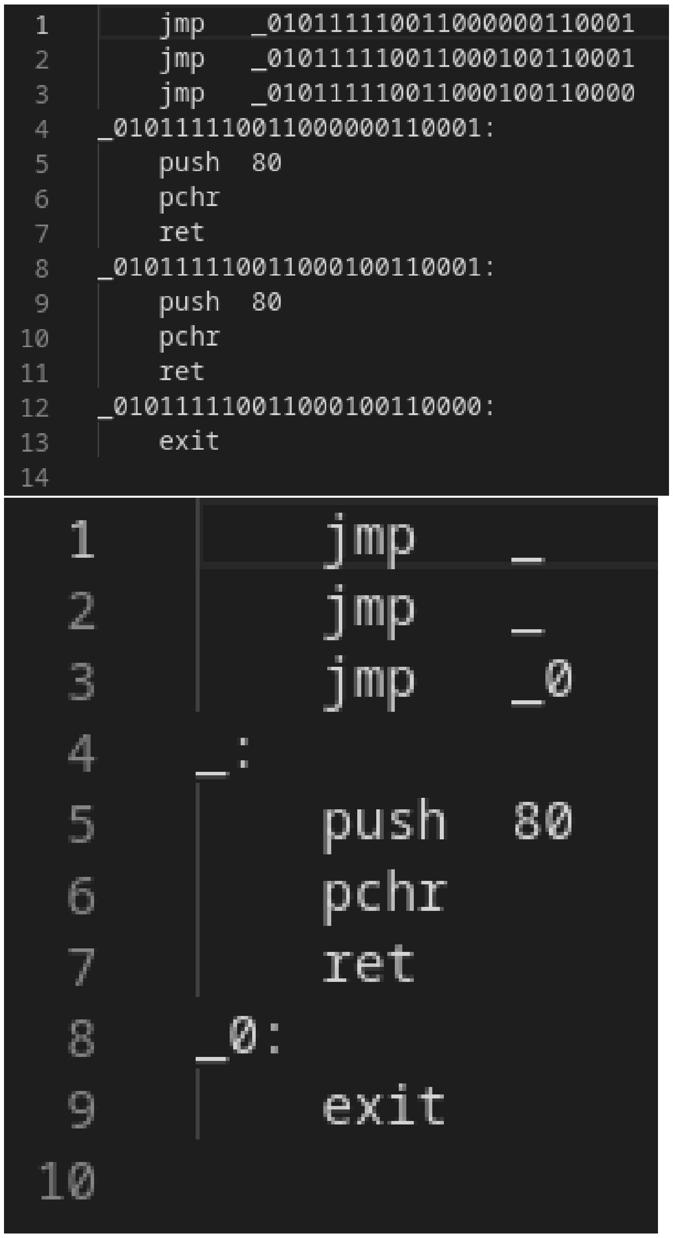


Fig. 6. Before and after refactoring the test file for duplicate method



```
    jmp   _0101111100110000000110001
    jmp   _0101111100110001001100001
    jmp   _0101111100110001001100000
_0101111100110000000110001:
    push  80
    pchr
    ret
_0101111100110001001100001:
    push  80
    pchr
    ret
_0101111100110001001100000:
    exit

```

```
    jmp   _
    jmp   _
    jmp   _0
_:
    push  80
    pchr
    ret
_0:
    exit

```

Fig. 7. Before and after immediate representation of the duplicate method refactoring

A.2 The refactoring of unused method

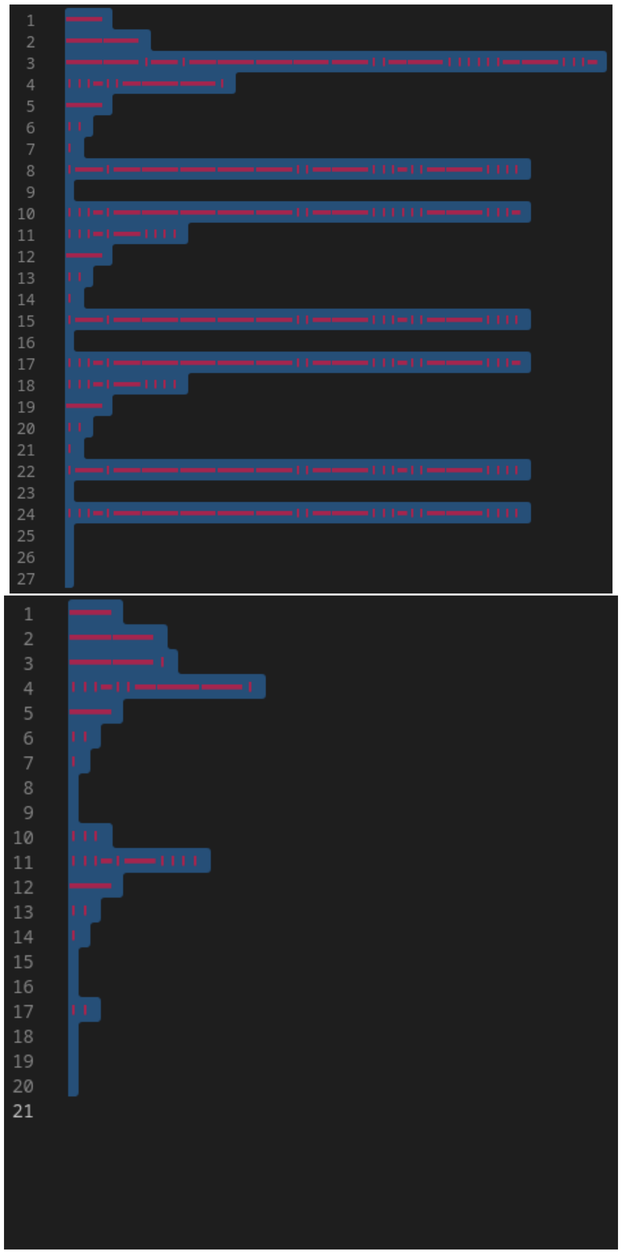
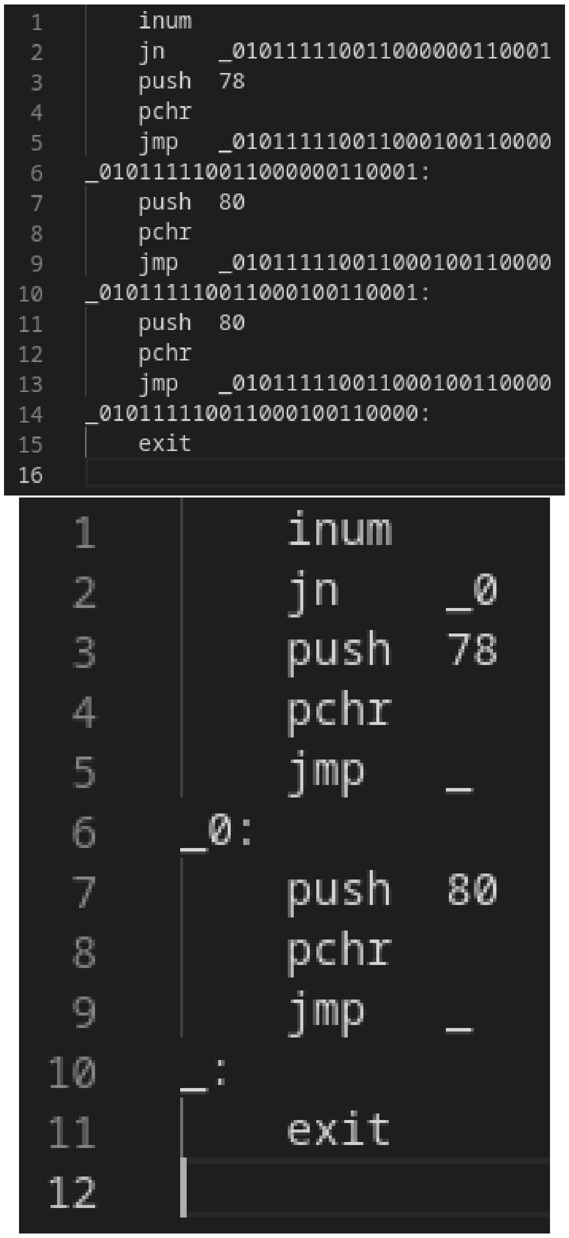


Fig. 8. Before and after refactoring the test file for unused method



```
    inum
    jn    _010111110011000000110001
    push  78
    pchr
    jmp   _010111110011000100110000
_010111110011000000110001:
    push  80
    pchr
    jmp   _010111110011000100110000
_010111110011000100110001:
    push  80
    pchr
    jmp   _010111110011000100110000
_010111110011000100110000:
    exit

```

```
    inum
    jn    _0
    push  78
    pchr
    jmp   _
_0:
    push  80
    pchr
    jmp   _
_:
    exit

```

Fig. 9. Before and after immediate representation of the duplicate method refactoring