Improving Nothingness. Refactoring Whitespace. (Extended Abstract)

Rutger Witmans, Vadim Zaytsev

University of Twente. Enschede, The Netherlands

Abstract

In this paper we explore the possibilities of refactoring code in Whitespace, a programming language which only recognises whitespace characters as code, and tolerates textual comments that describe the program. The paper presents a list of possible refactorings applicable to Whitespace, and describes a tool that automates some of these refactorings. The functionality of the tool is demonstrated with concrete examples.

Refactoring is a important systematic process of improving code without creating new functionality, improving long-term properties of the code such as readability, maintainability, changeability, testability, extendability and safety. We argue that, despite the lack of real-life applications for Whitespace specifically, it is beneficial to apply refactoring methodology to it, since lessons learnt from esoteric languages can be ported elsewhere — in this case, to assemblers and similarly restrictive software languages.

Keywords

Whitespace, program refactoring, esoteric languages, second generation languages

1. Motivation

There are many ways to classify software languages [10]. One of them is a spectrum from the most mainstream and widespread languages, to most exotic and esoteric ones. The mainstream side can be represented by the TIOBE index [8], the current top ten being Python, C, Java, C++, C#, Visual Basic, JavaScript, SQL, PHP and Go. Esoteric[1] languages, on the other side, are designed for one specific purpose of local interest: to have the smallest compiler, as it was the case with BRAINF*CK [5], to use statements that are as far from all other languages as possible, as it was with INTERCAL [16], or to provide a feasibly tiny playground for implementing legacy languages, as it happened with BABYCOBOL [18]. One of such languages is WHITESPACE [2], and it was designed from a driving principle that whitespace — the part of the source code which is traditionally ignored by the compiler as insignificant — is precisely the only part of the code which is significant, and the rest of the code such as visible punctuation, letters and numbers, are insignificant and skipped by the compiler. The language was designed by Edwin Brady around 2003 [2], and has enjoyed some attention in the meantime, leading to the existence of many implementations and programs to try software evolution techniques and tools on.

Refactoring [11] can be used as a standalone technique, often applied manually by developers (with automation support from the IDE) with the original intent — to improve the design of existing code [4]. However, it is also very useful as a part of composite techniques. For instance, one can apply it as a program transformation on elements of a test suite, possibly augmenting it with more test cases with known execution outcomes. In the past, this is exactly what the second author has tried to do [6] to augment the labour-intensive process of testing the Raincode Assembler Compiler [1, 17] with mutative fuzzing. The endeavour was ultimately unsuccessful: fuzzing only worked on the level of macros (where it did contribute somewhat, and found at least one off-by-one bug in the compiler), but the original goal of testing the instruction implementations failed. The main reason was the difficulty to define any kind of refactoring transformations that make sense: changing even one bit of the test program had potentially numerous and hardly predictable effects.

Several years later, we try a different approach: instead of codeveloping all the elements of the fuzzing infrastructure, we focus only on refactorings; and instead of facing a gigantic language requiring 1500+ pages of documentation just to cover the byte-level basics, we focus on one tiny esoteric language with similar properties — namely, difficulty of defining what constitutes a refactoring within it. If by any chance our results will happen to help some Whitespace developer to improve readability of their code, that could only make the world a better place to live in.

SATToSE'23: 15th Seminar on Advanced Techniques and Tools for Software Evolution, June 12–14, 2023, Salerno, Italy

r.c.h.witmans@student.utwente.nl (R. Witmans); vadim@grammarware.net (V. Zaytsev)
http://grammarware.net (V. Zaytsev)
0000-0001-7764-4224 (V. Zaytsev)

[1] "**Esoteric** — intended for or likely to be understood by only a small number of people with a specialised knowledge or interest." Oxford Languages Dictionary

2. Possible Refactorings

Since the world has moved way past the list of refactorings proposed by Opdyke [11] and Fowler et al [4] in the 1990s, we have mostly relied on developer-created grey sources like *Refactoring Guru* [13]. We first looked at what refactoring categories are possible. The following categories are available:

- **[CM]** Composing Methods
- **[MF]** Moving Features between Objects
- **[OD]** Organising Data
- **[SE]** Simplifying Conditional Expressions
- **[SM]** Simplifying Method Calls
- **[DG]** Dealing with Generalisation
- **[CS]** 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 **[MF]**, **[OD]** and **[DG]** are high-level refactorings which we will not translate that well. 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 **[SM]** and **[CM]**. Whitespace also has instructions for conditional jumps. This makes some of the ideas for **[SE]** possible. Next to this, the list of code smells look promising enough to deliver at least some refactorings for us to perform, so we will be looking into **[CS]** as well. We will thus be looking through the following categories:

- **[CM]** Composing ~~Methods~~ Labels
- **[SE]** Simplifying Conditional Expressions
- **[SM]** Simplifying ~~Method Calls~~ Label Jumps
- **[CS]** Code Smells

With these chosen categories, we have created a list of refactorings which can be performed on Whitespace code. The following list is the refactorings is our result so far:

- **[EM]** Extract method
- **[IM]** Inline method
- **[RM]** Rename method
- **[CC]** Consolidate conditional expression
- **[CD]** Consolidate duplicate conditional fragments
- **[RD]** Remove dead code
- **[RC]** Remove clone/duplicate methods

2.1. [EM] 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.

2.2. [IM] 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.

2.3. [RM] 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.

See subsection 3.1 for implementation details of this refactoring.

2.4. [CC] 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.

2.5. [CD] 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.

2.6. [RD] 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" [9]. 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.

See subsection 3.2 for implementation details of this refactoring.

2.7. [RC] Remove clone/duplicate methods

Last up, we will be removing duplicate methods. Duplicate methods are two methods which have the exact same functionality. For Whitespace, this will mean to us that there are two different labels which are followed by the same code and are not (co)recursive. Our tool is going to remove these methods since duplicate methods only cause confusion and do not have any benefits to a programmer.

See subsection 3.3 for implementation details of this refactoring.

3. Implementation Details

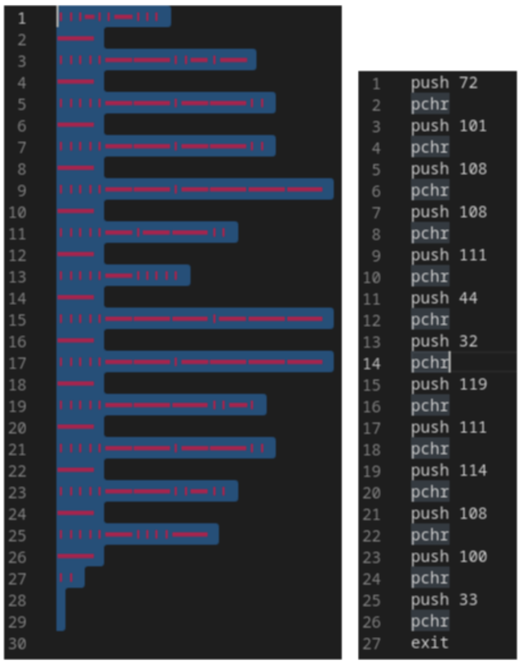
Listing 1: "The IR of Whitespace"

```
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,
```

For this project, we wanted to rely on 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 “`whitespace-rs`” [3]. 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.

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 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 Listing 1 you will see the IR



```
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
```

Figure 1: *Hello, World* in Whitespace and Whitespace IR

of the Whitespace library we have decided to use. All the 24 commands have their own unique and human-understandable name. Using this IR, we are able to create our refactorings in an easier-to-understand language.

In Figure 1, on the left 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 Figure 1, on the right you can 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 analyse test programs.

3.1. [RM] Rename method

Given the small size of the space of possible label names in Whitespace, the obvious automated refactoring would be one that minimises label names. Serendipitously, this functionality was already included in `whitespace-rs`, so we could simply rely on their implementation to achieve our first working refactoring [3].

3.2. [RD] Remove 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.

A combination of [RD] with [RM] can be seen on Figure 2 (in pure Whitespace) or on Figure 3 (in Whitespace IR).

3.3. [RC] Remove clone/duplicate methods

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

- Analyse 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

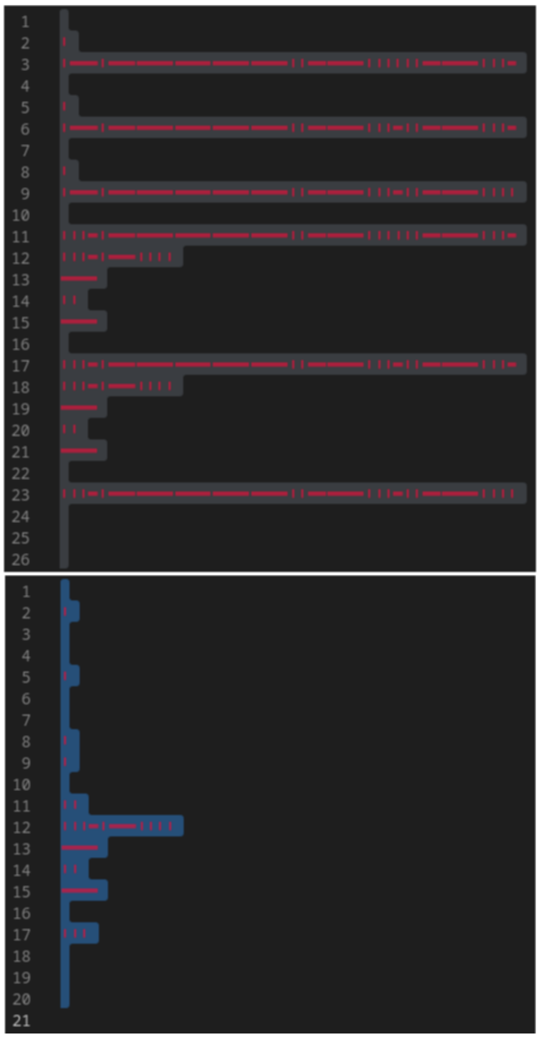


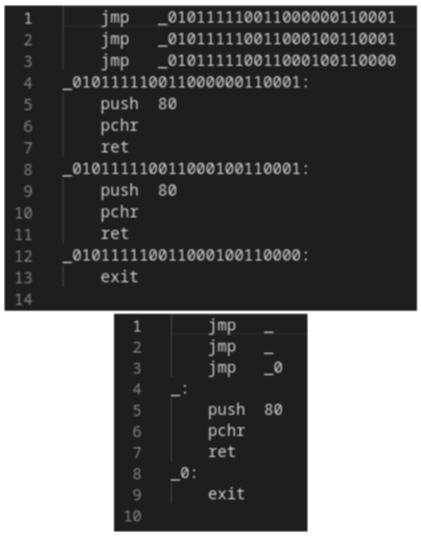
Figure 2: Duplicate method removal: before and after

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.

A combination of [RC] with [RM] can be seen on Figure 4 (in pure Whitespace) or on Figure 5 (in Whitespace IR).

4. Evaluation

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



```
    jmp   _010111110011000000110001
    jmp   _010111110011000100110001
    jmp   _010111110011000100110000
_010111110011000000110001:
    push  80
    pchr
    ret
_010111110011000100110001:
    push  80
    pchr
    ret
_010111110011000100110000:
    exit
```

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

Figure 3: Duplicate method removal: before and after, in IR

human-friendly. However, we solved this problem by writing in the format of the library their IR. The library was then able to recognise this format and transform the IR into a whitespace-encoded file, solving the issue of writing Whitespace code.

4.1. Testing [RD]

In software languages that permit low-level branching constructs, there are many ways to use a “method”, and thus dead code detection must run very advanced code analysis algorithms and apply domain-specific heuristics. For example, HLASM has an EX(ECUTE) instruction which can modify the target address of a branching instruction at runtime. COBOL has a statement that can ALTER a target of an existing GO TO statement. Older versions of FORTRAN had computable GOTOs. Luckily, Whitespace is a bit more straightforward, and making a call graph of all Call and Jump locations to all the Labels, is sufficient, if we take fall-throughs into account.

4.2. Testing [RC]

Code clone management has been a topic of research for many years [12]. Researchers and practitioners identify many different clone types and clone detection method families [7]. For this research, we opted for type-1 equivalence (precise character-level equality) of labelled sections after all possible [RM] and [RD] refactorings have been applied. Our tool marks clone pairs and in the second pass removes one of them and replaces all calls to it with the calls to the remaining one.

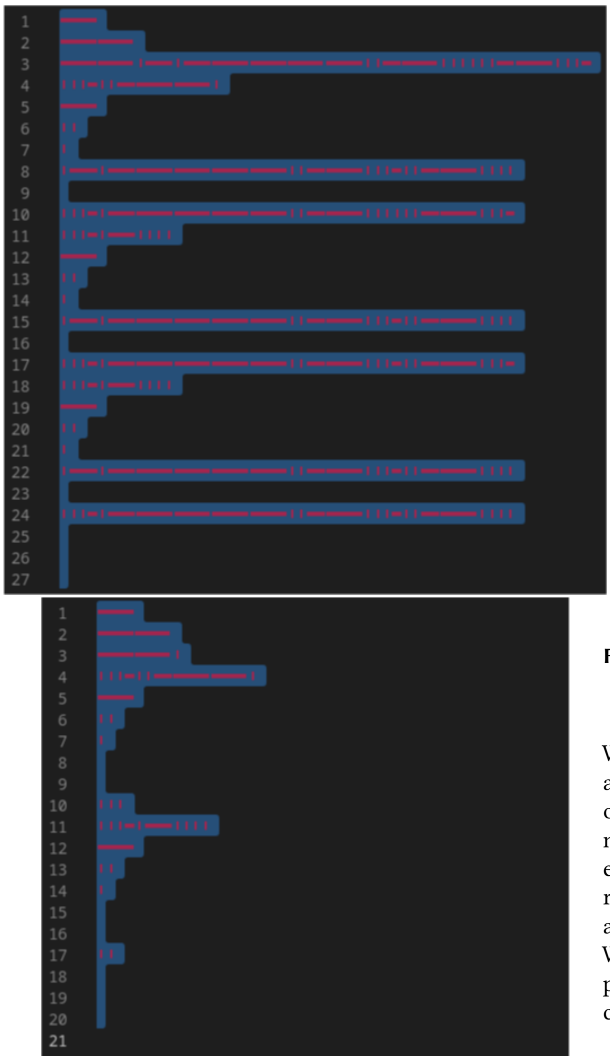
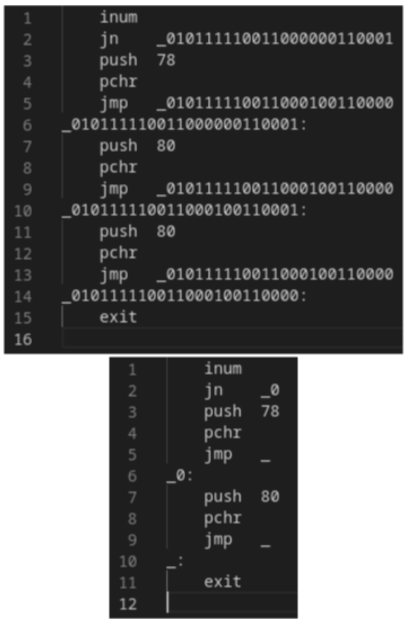


Figure 4: Unused method removal: before and after



```
    inum
    jn    _0101111100110000000110001
    push  78
    pchr
    jmp   _0101111100110001001100000
_0101111100110000000110001:
    push  80
    pchr
    jmp   _0101111100110001001100000
_0101111100110001001100001:
    push  80
    pchr
    jmp   _0101111100110001001100000
_0101111100110001001100000:
    exit
```

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

Figure 5: Unused method removal: before and after, in IR

5. Concluding Remarks

In this extended abstract, based on recent graduation project [14], we have described our approach to refactoring in Whitespace. The tool is also available for public use under GPL-3 license [15]. To answer our main research question about what refactorings would make sense in Whitespace, we first looked at different refactoring categories. From there we identified seven refactorings that are possible on Whitespace code. To address the question of implementability, we chose three refactorings and implemented them into a tool. Our tool 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 even in minimal assembler-like languages. 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.

The refactorings proposed in this extended abstract, are a work in progress. Further research and development are needed to fully realise the envisioned functionality. Next to this, more refactorings can be implemented, such as the different conditional refactorings mentioned in section 2.

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, profiling the tool itself could help identifying useful optimisations to improve its usability, especially regarding possible future extensions For anyone who would like to look at the tool or work on it further, you can find the tool over at GitHub [15].

References

[1] V. Blagodarov, Y. Jaradin, V. Zaytsev, Tool Demo: Raincode Assembler Compiler, in: T. van der Storm, E. Balland, D. Varró (Eds.), Proceedings of the Ninth International Conference on Software Language Engineering (SLE), 2016, pp. 221–225. doi:10.1145/2997364.2997387.

[2] E. Brady, Whitespace, https://web.archive.org/web/20150623025348/http://compsoc.dur.ac.uk/whitespace, 2003.

[3] CensoredUsername, whitespace-rs, https://github.com/CensoredUsername/whitespace-rs, 2016.

[4] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, Refactoring: Improving the Design or Existing Code, Addison-Wesley Professional, 1999.

[5] S. Graue, R. Berge, C. Pressey, et al., brainfuck — esolang, https://esolangs.org/wiki/Brainfuck, 2005.

[6] A. Gül, V. Zaytsev, Mutative Fuzzing for an Assembler Compiler, in: D. Di Nucci, C. De Roover (Eds.), Post-proceedings of the 18th Belgium-Netherlands Software Evolution Workshop (BENEVOL), volume 2605 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2020, pp. 18–24. URL: http://ceur-ws.org/Vol-2605/18.pdf.

[7] A. Hamid, V. Zaytsev, Detecting Refactorable Clones by Slicing Program Dependence Graphs, in: D. Di Ruscio, V. Zaytsev (Eds.), Post-proceedings of the Seventh Seminar in Series on Advanced Techniques and Tools for Software Evolution (SATToSE 2014), volume 1354 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2015, pp. 37–48. URL: http://ceur-ws.org/Vol-1354/paper-04.pdf.

[8] P. Jansen, et al., TIOBE Index for April 2023, https://www.tiobe.com/tiobe-index/, 2023.

[9] C. de Kruif, Using δ-NFGs to Identify and Eliminate Dead Code in C# Programs, Bachelor's thesis, Universiteit Twente, 2022. URL: http://purl.utwente.nl/essays/91890.

[10] R. Lämmel, D. Mosen, A. Varanovich, Method and Tool Support for Classifying Software Languages with Wikipedia, in: M. Erwig, R. F. Paige, E. Van Wyk (Eds.), Proceedings of the Sixth International Conference on Software Language Engineering, volume 8225 of *LNCS*, Springer, 2013, pp. 249–259. doi:10.1007/978-3-319-02654-1_14.

[11] W. F. Opdyke, Refactoring Object-Oriented Frameworks, Ph.D. thesis, University of Illinois at Urbana-Champaign, 1992.

[12] C. K. Roy, J. R. Cordy, Benchmarks for Software Clone Detection: A Ten-Year Retrospective, in: Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering, IEEE Computer Society, 2018, pp. 26–37. doi:10.1109/SANER.2018.8330194.

[13] A. Shvets, Refactoring: Clean Your Code, https://refactoring.guru/refactoring, 2022.

[14] R. Witmans, Improving Nothingness: Refactorings on Whitespace, Bachelor's thesis, Universiteit Twente, 2023. URL: http://purl.utwente.nl/essays/94374.

[15] R. Witmans, whiteref, https://github.com/rwitmans/whiteref/, 2023.

[16] D. R. Woods, J. M. Lyon, The INTERCAL Programming Language Reference Manual, https://www.muppetlabs.com/~breadbox/intercal-man/, 1973.

[17] V. Zaytsev, Modelling of Language Syntax and Semantics: The Case of the Assembler Compiler, Proceedings of the 16th European Conference on Modelling Foundations and Applications in the Journal of Object Technology (ECMFA@JOT) 19 (2020). doi:10.5381/jot.2020.19.2.a5.

[18] V. Zaytsev, Software Language Engineers' Worst Nightmare, in: R. Lämmel, L. Tratt, J. De Lara (Eds.), Proceedings of the 13th International Conference on Software Language Engineering (SLE), ACM, 2020, pp. 72–85. doi:10.1145/3426425.3426933.