sciendo RIGA TECHNICAL UNIVERSITY

Applied Computer Systems

ISSN 2255-8691 (online)
ISSN 2255-8683 (print)
December 2018, vol. 23, no. 2, pp. 109–117
doi: 10.2478/acss-2018-0014
https://content.sciendo.com

An Analysis on Java Programming Language Decompiler Capabilities

Konstantins Gusarovs*
Riga Technical University, Riga, Latvia

***Abstract* – Along with new artifact development, software engineering also includes other tasks. One of these tasks is the reverse engineering of binary artifacts. This task can be performed by using special "decompiler" software. In the present paper, the author performs a comparison of four different Java programming language decompilers that have been chosen based on both personal experience and results of a software developer survey.**

***Keywords* – Decompilation, Java, reverse engineering.**

I. Introduction

While software development is usually about producing new artifacts, i.e., turning the code written in some programming language to a binary distribution, sometimes it is necessary to perform reverse operation, which is reverse engineering [1]. Reverse engineering is the process of extracting the knowledge or design blueprints from anything man-made. In relation to software engineering, this can be described as extraction of the source code from the binary (compiled) files. While at first sight such a process might seem conflicting with the copyright, sometimes it is necessary to perform such an operation. An example might be a necessity to fix defects in program or library that was developed by a company some time ago, but any source code for it is missing. Other aspect of reverse engineering in software might be a necessity to obtain some information from the given libraries/programs that have no source code available – for example, cryptographic keys etc. Basically, such cases that are related to reverse engineering of own products are valid and legal use cases. Another example of a valid reverse engineering application is the study of computer viruses [1] by the authors of anti-virus software, which is a necessity for understanding of how malicious software works and how to act against it.

Based on a TIOBE index [2], one of the most popular programming languages used in an enterprise development is Java programming language [3], which is an object-oriented programming language that uses bytecode instructions executed by a stack-based virtual machine. The fact that Java is built around bytecode instead of an assembly language offers several advantages – for example, code written once can be executed on different platforms, given a virtual machine implementation exists for the aforementioned platforms. From the reverse engineering point of view, it means that it is necessary to only process bytecode, which in comparison to the assembly languages, contains fewer instructions. Thus, in order to turn Java bytecode to the source code, it is necessary to be able to transform around 200 different instructions [4] for the latest Java version (10) at the moment, which in comparison, for example, to Intel processor assembly instruction set [5] containing around 2000 different instructions seems an easier task.

Such a task can be performed by software called "decompiler" [1], which translates binary artifact into the source code with a certain amount of precision. Several decompilers exist for the Java programming language, and the goal of the paper is to compare these decompilers in order to provide recommendations for the software developers.

This paper is structured as follows. In Section II, the chosen list of Java programming language decompiler software is given. Section III presents a short introduction to the Java programming language binary file format and bytecode instructions. Section IV gives several examples on Java bytecode decompilation techniques that are used by the decompilation software. In Section V, a test case developed by the author of the paper is described. Section VI shows the results of test case decompilation along with a short analysis of the obtained results. Section VII describes additional test results as well as comparison of decompilers using additional criteria defined by the author of the paper. Finally, in the last section conclusions are made and recommendations about Java decompiler software are given.

II. Java Decompiler Software

Several decompiler programs exist for the Java programming language. In order to choose one to use, it would be necessary to perform the comparison of these programs. In this section, the author provides a list of such software. The list of the decompiler software is built using both the author's personal experience on using such software and results of the survey performed by the author at his current workplace in order to determine what other programmers would recommend using in order to solve such a task:

- JD Project [6] is a modular decompiler that can be run as a standalone application or be integrated into development environments, such as Eclipse [7] or Intellij IDEA [8].
- CFR [9] is distributed in a form of library that contains a command line interface (CLI) and can also be used as part of other software.

* Corresponding author's e-mail: konstantins.gusarovs@gmail.com

- Procyon [10] is a framework that can be integrated into other applications and contains CLI. Several graphical user interface (GUI) implementations exist for it.
- Fernflower [11] is a Java decompiler used in the Intellij IDEA [8] development environment. It is distributed in a form of a library that also has CLI interface.

The aforementioned survey on Java decompiler software conducted by the author is based on two questions:

- Which Java decompilers are you familiar with?
- Which Java decompiler would you recommend to use?

There were a total of 247 people that answered these questions. Results of the survey are shown in Tables I and II.

TABLE I

WHICH JAVA DECOMPILERS ARE YOU FAMILIAR WITH

Decompiler	**Total answers**
Fernflower	200
JD Project	158
Procyon	141
CFR	50
JAD	2

TABLE II

WHICH JAVA DECOMPILER WOULD YOU RECOMMEND TO USE

Decompiler	**Total answers**
Fernflower	121
JD Project	74
Procyon	44
CFR	8

Results of the survey show that most of the people are familiar with the Fernflower decompiler software and recommend to use it, which can be explained by the fact that it is built into the development environment used by the company, which is Intellij IDEA [8].

Several other implementations of Java programming language decompiler exist; however, in most cases these implementations are outdated and unsupported. Thus, in the paper, four decompilers are compared.

III. AN INTRODUCTION TO THE JAVA VIRTUAL MACHINE BINARY FILE FORMAT

Java virtual machine (JVM) uses binary .class files that contain result of the source code compilation [12]. These files contain all the necessary information about the compile unit (which is basically a Java class or interface), including:

- Version of the compiler that produced given .class file. This allows the JVM to detect if it should be able to load and execute given file.
- Constant pool containing various string literals, class and interface names, field and method names as well as other constants that are used in the given compilation unit.
- Access flags that determine both visibility and type of a given compilation unit. This information defines if and how the actual class contained in the file can be instantiated and subclassed.
- Information on a base class and interfaces given compilation unit inherits from or implements.
- Field and method list along with their access flags and other modifiers.
- Attributes of class, its fields and methods that are used to determine additional information on the aforementioned components of a compilation units. One of the attributes is the actual code of a method, others represent different information that can be used in a runtime, e.g., annotations, which are syntactic metadata attached to the given member, or list of exceptions that might be thrown during the execution of a selected method.

The code attribute contains the actual bytecode instruction listing that will be used during appropriate method invocation. As it has already been mentioned, JVM bytecode contains approx. 200 instructions that can be divided into the following groups:

- Mathematical operations – these instructions are meant for the actual mathematic operation representation (for example, `DADD` instruction sums up 2 `double` type variables), as well as constant loading onto the top of the stack (for example, `ICONST_0…ICONST_5` instructions allow loading integers from 0 to 5).
- Stack operations – JVM is a stack-based virtual machine, which means that it does not use registers of any kind. Instead, all the local variables are loaded onto stack and can be processed on it. These instructions allow both writing and reading information contained on the top of the stack (for example, `ALOAD` allows pushing the object to the stack, while `ASTORE` fetches it and stores in local variable), as well as creating new objects on the top of the stack (`NEW` allows creating a new object, while `NEWARRAY` creates a new array of a given type). Several instructions in this group are also meant to copy objects (`DUP` creates a copy of a variable on the top of the stack and pushes it to the top) or remove them from stack without storing into any of the local variables (`POP`).
- Type conversion instructions – for example, `D2I` converts double type variable on the top of the stack to the int type and pushes the result on the top of the stack.
- Type checking instructions that allow both checking the type of variable on the top of the stack and replacing it with the check result (`INSTANCEOF`) or throwing a runtime exception (`CHECKCAST`).
- Numerical type comparison instructions – for example, `LCMP` that compares two `long` type variables on the top of the stack.
- Synchronization instructions `MONITORENTER` and `MONITOREXIT` that are used to obtain mutually exclusive access to the given resource.

- Method invocation instructions, such as `INVOKEVIRTUAL`, that allow invoking methods in a different way.
- Instructions that allow the invoked method to return its invocation result to other running code – for example, `ARETURN` allows using an object as an invocation result, while `RETURN` means that the method has not returned any result at all.
- Branching instructions, which are used to change the program flow during code execution. JVM has both conditional (for example, `IFACMP_EQ` that changes the flow if two objects are identical by reference) and unconditional branching instructions (`GOTO`). Special instructions also exist for switch language construct processing, for example, `TABLESWITCH`.
- Debugger instruction `BREAKPOINT` that is not included in the compiled Java code. Instead, debuggers are injecting this instruction dynamically.
- `ATHROW` – instruction meant to throw an exception.
- `ARRAYLENGTH` – instruction that allows getting the length of the array on the top of the stack.
- `NOP` – an empty instruction.

Each method in .class file also has information on local variables being used. It might or might not contain information on variable names – it depends on the way the Java code was compiled. If local variable names are omitted during the compilation, then there is only information on local variable logical numbers (indices) and types.

IV. JVM Bytecode Decompiling Techniques

As it can be seen from the previous section, decompilation of Java bytecode would require extracting from the .class file information on its members such as fields and methods, transforming appropriate method bytecode into the source code and adding additional information on access flags for all the parts of the class file.

In the present paper, the author focuses on the bytecode transformation possibilities, since other decompilation tasks can be handled in a straightforward way by extracting the necessary information and using simple transformation techniques.

As for the bytecode, it is necessary to understand that most JVM bytecode instructions can also be handled in a pretty simple and straightforward way. In this case, the author is talking about all the groups of the bytecode instructions, except the branching. The decompiler would also require keeping the track on the state of the JVM stack during method invocation, so it is possible to determine, which objects are being loaded on the stack and read from it. It is also necessary to analyse the local variable table to determine, which of the local variables are used during method invocation. The author would like to provide several examples on how the decompilation techniques for these instruction groups might work.

The first example is shown in Fig. 1. In this case, the code itself consists of three mathematical instructions – two load integer constants on the top of the stack, third sums given integers up. The last instruction in the given bytecode example tells JVM to use the variable on the top of the stack as the return value of the method.

```
ICONST_1
ICONST_2
IADD
IRETURN
```

Fig. 1. JVM bytecode using mathematical operations.

To restore the source code from the given byte code fragment, decompiler must keep track on what actually is happening during this bytecode execution. It is possible to see that because of the first instruction invocation integer constant 1 is being put on the top of the stack, which results in the following stack state: `[1]`. Second instruction puts integer constant 2 on the top of the stack, so the stack becomes: `[2,1]`. Then next instruction removes two top stack members, sums them up and puts the result of this mathematical operation on the top of the stack, so the stack is being turned to `[1+2]`. Finally, last bytecode instruction tells JVM to use the variable on the top of the stack as a method return value. By following this information, it is possible to see that given bytecode fragment corresponds to the Java source code shown in Fig. 2.

```
return 1 + 2;
```

Fig. 2. First bytecode fragment decompilation result.

Figure 3 shows a similar bytecode fragment with the only exception that instead of integer constants, local variables are used.

```
ILOAD 1
ILOAD 2
IADD
IRETURN
```

Fig. 3. JVM bytecode using mathematical operations and local variables.

Decompilation logic for this bytecode fragment is the same as in the previous example: decompiler should keep the track on what the state of the stack should become because of each instruction execution and use this information to rebuild the source code. When local variables are used in the bytecode instructions, decompiler should refer to appropriate methods' local variable table to determine actual variable names if they are present. Figure 4 shows two possible outcomes of this bytecode decompilation result – the first one assumes that local variable names are present, while the second one assumes this information was removed during the compilation process, and decompiler had to generate variable names based on their indices in the local variable table.

```
return a + b;
return var1 + var2;
```

Fig. 4. Second bytecode fragment decompilation result.

It is possible to see that without considering branching instructions, decompiler's task would be to rebuild what would happen during appropriate bytecode instruction invocation and emit the appropriate syntactic constructions because of its work.

If the branching instructions also appear in the bytecode, it is necessary to analyse, to where the control flows are being redirected by a branching instruction. Figure 5 provides the first example of a bytecode with branching instructions.

```
   ILOAD 1
   ILOAD 2
   IF_ICMPLT L1
   ICONST_1
   IRETURN
L1:
   ICONST_2
   IRETURN
```

Fig. 5. First branching example.

Here, two local integer variables are compared, and if the second one is less than the first one, control flow redirect appears. It targets the label after branching instruction, so decompiler should be able to determine that such branching corresponds to the `if` language construction. Instructions that can be found just after branching instruction, in turn, correspond to the `else` part of `if` statement. Thus, the decompilation result should be as shown in Fig. 6.

```
if (a < b) {
    return 1;
} else {
    return 2;
}
```

Fig. 6. First branching example decompilation result.

Another branching example is when the label, which is a target of a control flow redirect, is located before the branching instruction. Such cases correspond to the loops in the code (`while`, `for`, `do…while`). An example of such a bytecode is given in Fig. 7.

```
L1:
    …
    ILOAD 1
    ILOAD 2
    IF_ICMPLT L1
    RETURN
```

Fig. 7. Second branching example.

Such bytecode should be decompiled to the result that is shown in Fig. 8.

```
while (a < b) {
    …
}
```

Fig. 8. Second branching example decompilation result.

It is worth mentioning that all the examples that are given in this section are primitive enough, and real decompiler should also be able to handle nested branching instructions and more complex control flow redirections. However, these seem to be enough to demonstrate, how Java decompiler should work and what information it should be using during the reverse engineering process. In the next section, the author would like to discuss a test case that he has developed for decompiler capability testing on a more complicated branching code.

V. Developed Test Case

To test how Java decompilers can handle complex branching cases, it is necessary to develop an example that would contain such kinds of branching. To develop such a bytecode, it is necessary to understand what a complex branching is in JVM bytecode.

While nested branches might result in a complex decompiled code, such cases are not a problem to decompile – decompiler software should keep track on where control flow redirection labels are located in the bytecode, and should be able to detect correct syntactic constructions.

To perform a test of decompiler capabilities, the author has developed a test case for this kind of software, which is based on a single loop. Loop itself contains several instructions that are computing random integer number between 0 and 2 and then checks, which number was generated in order to decide if the loop should be terminated. This kind of logic in bytecode corresponds to the `while (true)` loop construction that has some `break` instructions inside. However, it is also possible to enter the loop itself in its various states, which means that when first entering the loop, it is equally possible to start with any possible `break` branch. Such a bytecode means that decompiler software should be able to come up with a solution on how such a loop can be entered.

As a result, an example class with two methods has been developed using Java bytecode. First method of this class is meant to calculate random number in 0…2 interval. Bytecode source of this method is presented in Fig. 9.

```
NEW java/util/Random
DUP
INVOKESPECIAL java/util/Random.<init> ()V
INVOKEVIRTUAL java/util/Random.nextInt ()I
ICONST_3
IREM
IRETURN
```

Fig. 9. Random number generation method.

This method creates a new instance of java.util.Random class and invokes nextInt() method on it. Then it calculates the remainder of division operation of the generated random integer and 3 and returns it as a result. Java source code corresponding to this method is presented in Fig. 10.

```
return new Random().nextInt() % 3;
```

Fig. 10. Random number generation method – Java code.

```
    ALOAD 0
    INVOKESPECIAL Test.rem()I
    ISTORE 1

    ILOAD 1
    ICONST_0
    IF_ICMPEQ LOOP_PART1

    ILOAD 1
    ICONST_1
    IF_ICMPEQ LOOP_PART2

    ILOAD 1
    ICONST_2
    IF_ICMPEQ LOOP_PART3

LOOP_START:

    LOOP_PART1:
        INVOKESTATIC java/lang/System.gc()V
        ALOAD 0
        INVOKESPECIAL Test.rem()I
        ICONST_0

        IF_ICMPEQ LOOP_END

    LOOP_PART2:
        INVOKESTATIC java/lang/System.gc()V
        ALOAD 0
        INVOKESPECIAL Test.rem()I
        ICONST_0

        IF_ICMPEQ LOOP_END

    LOOP_PART3:
        INVOKESTATIC java/lang/System.gc()V
        ALOAD 0
        INVOKESPECIAL Test.rem()I
        ICONST_0

        IF_ICMPEQ LOOP_END

        GOTO LOOP_START

LOOP_END:
    RETURN
```

Fig. 11. The main test method.

The bytecode of the main test method is shown in Fig. 11. In order to make decompiler's task more complicated when processing it, a dummy method invocation of `java.lang.System.gc()` has been added after each loop entry point. This method has been chosen due to the fact that it has `void` return type meaning and there is no necessity to clean up the stack after it was invoked. Without this invocation, some decompilers are able to reconstruct the code in the way shown in Fig. 12. While this result is correct from the reverse engineering point of view, it is kind of a shortcut taken by the decompiler and does not reflect the results wanted from this test case.

```
public void test() {
    int i = rem();
    while (
           ((i == 0) || ((i == 1) || (i != 2)))
                  && ((rem() != 0) &&
                  (rem() != 0) && (rem() !=
0))
         ) {
    }
}
```

Fig. 12. Decompiled method without System.gc() calls.

VI. Results Of Test Case Decompilation

```
public void test() {
    int i = this.rem();
    if (i != 0) {
        if (i != 1) {
            if (i == 2) {
                System.gc();
                if (this.rem() == 0) {
                    return;
                }
            }

            System.gc();
            if (this.rem() == 0) {
                return;
            }
        }
    } else {
        System.gc();
        if (this.rem() == 0) {
            return;
        }
    }

    do {
        System.gc();
        if (this.rem() == 0) {
            break;
        }

        System.gc();
        if (this.rem() == 0) {
            break;
        }

        System.gc();
    } while (this.rem() != 0);

}
```

Fig. 13. Fernflower decompilation result.

In this section, the author demonstrates the results of test example decompilation using different decompilers as well as analyses these results.

Figure 13 shows the decompilation result obtained from Fernflower decompiler. It is possible to see that decompiler has unrolled the first possible iteration of the main loop – it is able to detect that a loop entering code may be invoked up to a single time, after which the loop should continue in a normal way. As a result, this solution has led to a code duplication; however, the result is readable and can be used.

```
public void test() {
    int i = rem();
    if (i != 0) {
        if (i == 1) {
            break label31;
        }
        if (i == 2) {
            break label42;
        }
    }
    label31:
    label42:
    do {
        System.gc();
        if (rem() == 0) {
            break;
        }
        System.gc();
        if (rem() == 0) {
            break;
        }
        System.gc();
    } while (rem() != 0);
}
```

Fig. 14. JD Project decompilation result.

Figure 14 shows decompilation result obtained by using JD Project decompiler. In this case, decompiler did not unroll the first loop iteration and utilised Java ability to use `break` instruction to redirect control flow to any label. This technique allowed avoiding code duplication in the unrolled loop iteration. However, the result obtained from the JD Project is not compilable, since Java does not support using `break` instruction with labels that are defined after it. It is also worth mentioning that JD Project decompiler did not use `this` keyword, since it was able to identify that `rem()` method belonged to the same class.

Figure 15 shows results that were obtained from the CFR decompiler. It was unable to produce the compilable code at all, also it did not attempt to perform the unrolling of the first loop iteration.

Finally, Fig. 16 shows the result of using the Procyon decompiler tool. In this case, decompiler software has not performed loop unroll. Instead, the result of initial method invocation before a loop is stored in final local variable that cannot be changed and is rechecked on every iteration in order to decide, if certain actions have to be performed. It is possible to say that Procyon decompiler moved all the loop entry point selection inside the loop and ensured that appropriate variable responsible for this logic could not be changed. The decompilation result can be compiled and executed.

```
public void test() {
    block1:
    {
        i = this.rem();
        if (i == 0) break block1;
        if (i == 1) **GOTO lbl9
        if (i == 2) **GOTO lbl11
    }
    do {
        System.gc();
        if (this.rem() == 0) return;
        lbl9:
        // 2 sources:
        System.gc();
        if (this.rem() == 0) return;
        lbl11:
        // 2 sources:
        System.gc();
        if (this.rem() == 0) return;
    } while (true);
}
```

Fig. 15. CFR decompilation result.

```
public void test() {
    final int i = this.rem();
    while (true) {
        Label_0042: {
            Label_0031: {
                if (i != 0) {
                    if (i == 1) {
                        break Label_0031;
                    }
                    if (i == 2) {
                        break Label_0042;
                    }
                }
                System.gc();
                if (this.rem() == 0) {
                    return;
                }
            }
            System.gc();
            if (this.rem() == 0) {
                return;
            }
        }
        System.gc();
        if (this.rem() != 0) {
            continue;
        }
        break;
    }
}
```

Fig. 16. Procyon decompilation result.

The results obtained from testing decompilers against the developed test case show that a test case itself was complex enough, since two out of four decompilers could not produce a code that could be compiled again. In both cases, decompilers did not attempt to somehow modify the loop entry point selection logic, while Fernflower performed the unrolling of the first loop iteration but Procyon moved this logic inside the loop.

VII. Additional Decompiler Comparison

Both Java 7 and Java 8 versions introduced new capabilities to the Java programming language. For the Java 7 it is possible to identify two features that could affect the decompilation results:

- Try-with-resources syntax [13].
- Ability to use string literals in `switch` statement [14].

Java 8, in turn, introduced functional programming capabilities in a form of lambda expressions [15] and method references [16]. Modern decompilers should be able to support these features.

```
public class Test {
    void java7StringSwitch(final String s) {
        switch (s) {
            case "Hello":
                System.out.println("World");
                break;
            default:
                System.out.println("Hi!");
        }
    }

    void java7TryWithResources() {
        try (
            final InputStream is =
            new FileInputStream("1.txt")
        ) {
            System.out.println(
                    is.available());
        } catch (final IOException e) {
            e.printStackTrace();
        } finally {
            System.out.println("Bye!");
        }
    }

    void java8Lambdas() {
        new Thread(() -> {
            System.out.println("Hello");
            System.out.println("World");
        }).start();
    }

    void java8MethodReferences() {
        new Thread(
          this::java7TryWithResources)
                .start();
    }
}
```

Fig. 17. Java 7/8 feature test case.

The author of the present paper does not consider new syntax construction introduced in newer Java versions – 9 and 10, since both these versions based on the author's experience are not yet widely adopted.

In order to test decompiler capabilities, the author has developed another test case in a form of Java code that was compiled and later decompiled in order to compare the decompilation results with an original source code. This example is shown in Fig. 17. The results of this test case decompilation are shown in Table III.

TABLE III

Java 7/8 Feature Decompilation Results

Feature	Fernflower	Procyon	JD Project	CFR
Try-with-resources	See Fig. 18	OK	See Fig. 18	See Fig. 18
String literal in `switch`	See Fig. 19	OK	OK	Error
Lambda expressions	OK	OK	Error	OK
Method references	OK	OK	Error	OK

Here, only Procyon decompiler was able to restore the code to the state, which was accurately representing the source code. Both CFR and JD Project were unable to handle some of the features with JD Project not supporting Java 8 features and CFR failing on string literal in `switch` statement. Figure 18 represents results obtained in try-with-resources statement decompilation by three out of four decompilers. With minor differences it was the same among all.

```
void java7TryWithResources() {
    try {
        InputStream is = new
          FileInputStream("1.txt");
        Throwable var2 = null;

        try {
            System.out
              .println(is.available());
        } catch (Throwable var20) {
            var2 = var20;
            throw var20;
        } finally {
            if (is != null) {
                if (var2 != null) {
                    try {
                        is.close();
                    } catch (Throwable var1) {
                        var2
                          .addSuppressed(var1);
                    }
                } else {
                    is.close();
                }
            }

        }
    } catch (IOException var22) {
        var22.printStackTrace();
    } finally {
        System.out.println("Bye!");
    }
}
```

Fig. 18. Try-with-resources decompiled.

Figure 19, in turn, shows results for string literal in switch statement decompilation by Fernflower. It is possible to see that decompiler could not identify the correct type of the `switch` instruction, but instead used the information in the bytecode to decompile it in a straightforward way.

```
void java7StringSwitch(String s) {
    byte var3 = -1;
    switch(s.hashCode()) {
        case 69609650:
            if (s.equals("Hello")) {
                var3 = 0;
            }
        default:
            switch(var3) {
                case 0:
                    System.out
                      .println("World");
                    break;
                default:
                    System.out.println("Hi!");
            }

    }
}
```

Fig. 19. String literal in switch statement decompiled by Fernflower.

In addition to Java 7/8 feature support, it is possible to define other requirements for the Java programming language decompiler software. One of these requirements would be an ability to modify the decompiled code during decompilation process. This, for example, could be achieved, if decompiler rebuilds abstract syntax tree (AST) [17] for the generated code and allows processing it using the visitor pattern [18]. Such a feature should be part of decompiler's API and should not require modifying its source code.

Another requirement is the license under which decompiler is developed and distributed – it can enforce limitations on an ability to use the decompiler in commercial products or modify its source code, if it is available.

When looking at the extension possibility for the decompiled code modification, it is as follows:

- Fernflower is an open source decompiler that has no built-in API to support these kinds of modification, so one would have to modify the source code of the compiler itself to allow for decompiled code processing.
- Procyon has a built-in API that supports the visitor pattern for a decompiled AST processing. Thus, it is not necessary to do any modifications to the decompiler's code – it is possible to add one's own extensions to it.
- JD Project has no documented API that would allow modifying the decompiled Java source code and adding such a functionality would require decompiler's source code modification.
- CFR is a closed source project and it has no documented code modification feature. Author of the CFR himself offers to decompile the CFR to modify it.

Licensing for the four compared Java decompilers is the following:

- Fernflower is an open-source project that uses Apache 2.0 license [19] making it available for the commercial projects and modification of the source code.
- Procyon also is an open-source project that is licensed under Apache 2.0 license [19].
- JD Project is an open-source project licensed under GNU GPLv3 license [20], which means that it is free for a non-commercial use.
- Finally, CFR is a closed-source project. It is licensed under MIT license [21] that allows using it in commercial projects as well as modifying it.

VIII. Conclusion

In the present paper, the author has performed a study of the Java decompilers and its capabilities by analysing performance in different test cases, the licensing model and abilities to extend the decompilation results.

To choose Java decompilers to be compared, the author has used both his personal experience and a survey of Java developers. As a result, four different decompilers have been chosen with Fernflower and JD Project both being mostly renown and recommended to use by other people. Surprisingly, JD project has shown very poor performance in the author's developed test case and Java 7/8 feature decompilation. It is also licensed under the GNU GPLv3 license that might limit its usage in commercial projects. Fernflower, in turn, has performed well enough in the main test case, but it has not been precise when decompiling Java 7 features; however, Java 8 feature support seems mature enough.

Two other compilers, which are not popular among the surveyed developers, have shown the following results. CFR is unable to decompile the main test case as well as one of Java 7 test cases. It has performed very well in Java 8 test cases and has a commercial use friendly license, but it is a closed-source project, which means that its modification might be a challenging task in case it is required. Procyon, in turn, is able to handle both the main test case and Java 7/8 test cases. It is only decompiler that is able to cover all the selected Java 7/8 features producing results close enough to the original source code. It is only decompiler that has an API allowing for the modification of the decompiled code. Procyon represents the decompiled code in a form of an AST tree and allows for visitor pattern usage to process this tree. Procyon is licensed under Apache 2.0 license, which makes it available for commercial products and allows for the source code modification.

It may not be easy to provide the recommendations on which decompiler for the Java language to choose based only on the results of the present research; however, the author himself sees Procyon decompiler behaving the best amongst the ones compared here. It has performed well enough in all the defined test cases, as well as has support for additional features and a friendly licensing model. Interestingly, this decompiler seems to be not very well known and has not received many recommendations during the survey, which might be explained by the fact that developers are not familiar enough with it.

References

[1] E. Eilam, *Reversing: Secrets of Reverse Engineering*, 1st edition. USA: Wiley, 2005.

[2] TIOBE Index | TIOBE - The Software Quality Company [Online]. Available: https://www.tiobe.com/tiobe-index/ [Accessed: Sep. 3, 2018].

[3] Java Software | Oracle [Online]. https://www.oracle.com/java/ [Accessed: Sep. 3, 2018].
[4] Chapter 6 The Java Virtual Machine Instruction Set [Online]. https://docs.oracle.com/javase/specs/jvms/se10/html/jvms-6.html [Accessed: Sep. 3, 2018].
[5] Intel® 64 and IA-32 Architectures Software Developer Manuals | Intel® Software [Online]. https://software.intel.com/en-us/articles/intel-sdm [Accessed: Sep. 3, 2018].
[6] Java Decompiler [Online]. http://jd.benow.ca/ [Accessed: Sep. 3, 2018].
[7] Enabling Open Innovation & Collaboration | The Eclipse Foundation [Online]. https://www.eclipse.org/ [Accessed: Sep. 3, 2018].
[8] IntelliJ IDEA: The Java IDE for Professional Developers by JetBrains [Online]. https://www.jetbrains.com/idea/ [Accessed: Sep. 3, 2018].
[9] CFR - yet another java decompiler. [Online]. http://www.benf.org/other/cfr/ [Accessed: Sep. 3, 2018].
[10] mstrobel / Procyon / wiki / Java Decompiler – Bitbucket [Online]. https://bitbucket.org/mstrobel/procyon/wiki/Java%20Decompiler [Accessed: Sep. 3, 2018].
[11] GitHub - fesh0r/fernflower: Unofficial mirror of FernFlower Java decompiler. [Online]. https://github.com/fesh0r/fernflower [Accessed: Sep. 3, 2018].
[12] Chapter 4. The class File Format [Online]. https://docs.oracle.com/javase/specs/jvms/se10/html/jvms-4.html [Accessed: Sep. 3, 2018].
[13] The try-with-resources Statement (The Java™ Tutorials> Essential Classes > Exceptions) [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html [Accessed: Sep. 5, 2018].
[14] Strings in switch Statements [Online]. Available: https://docs.oracle.com/javase/7/docs/technotes/guides/language/strings-switch.html [Accessed: Sep. 5, 2018].
[15] Lambda Expressions (The Java™ Tutorials > Learning the Java Language > Classes and Objects) [Online]. Available: https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html [Accessed: Sep. 5, 2018].
[16] Method References (The Java™ Tutorials > Learning the Java Language > Classes and Objects) [Online]. Available: https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html [Accessed: Sep. 5, 2018].
[17] D. Grune, and C. J. H. Jacobs, *Parsing Techniques – a Practical Guide*, 2nd edition. USA: Springer-Verlag, 2008. https://doi.org/10.1007/978-0-387-68954-8
[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley, 1994.
[19] Apache License, Version 2.0 [Online]. Available: http://www.apache.org/licenses/LICENSE-2.0 [Accessed: Sep. 6, 2018].
[20] The GNU General Public License v3.0 - GNU Project - Free Software Foundation [Online]. Available: https://www.gnu.org/licenses/gpl-3.0.en.html [Accessed: Sep. 6, 2018].
[21] The MIT License | Open Source Initiative [Online]. Available: https://opensource.org/licenses/MIT [Accessed: Sep. 6, 2018].

Konstantins Gusarovs received the Master degree in Computer Systems from Riga Technical University, Latvia, in 2012. He is presently the fourth-year PhD student and Researcher at the Department of Applied Computer Science, Riga Technical University, as well as Java Developer at C.T.Co Ltd. His current research interests include object-oriented software development and automatic obtaining of program code.
E-mail: konstantins.gusarovs@gmail.com