Report for assignment 3

Group 5
Julia Potrus
Erik Betzholtz
Rasmuss Per Brodin
Vincent Lagerros

February 21, 2025

1 Introduction

1.1 Project

Name: json-iterator java

URL: https://github.com/json-iterator/java

1.2 One or two sentences describing it

A json (JavaScript Object Notation) serializer and deserializer for Java.

1.3 Onboarding experience

1. Did you have to install a lot of additional tools to build the software?

   No, the only requirements were Maven and Java. All group members had these tools prior to starting this assignment

2. Were those tools well documented?

   No, since the project does not contain a well-documented README. Thus, there is no information about the tools themselves or versions required to compile the project.

3. Were other components installed automatically by the build script?

   There were no external tools installed during the build script other than the Java dependencies of the code.

4. Did the build conclude automatically without errors?

No, the Java version and some packages are outdated, so while the project might have been in a valid state at the last commit, it does not build out of the box with modern tools.

5. How well do examples and tests run on your system(s)?

   A lot of the tests failed, almost 1/3, but the project examples looked to be working.

6. Do you plan to continue or choose another project?

   Yes, we will continue with this project.

1.4 Complexity

Complexity according to lizard:
writeLongBits: 8
readAny: 8
writeLong: 9
readNumber: 20

1. What are your results for four complex functions?

   - Did all methods (tools vs. manual count) get the same result?
   - Are the results clear?

   readAny(): The result was a cyclomatic complexity of 8. This was both the manual calculation and the one given by Lizard. The results are clear when using the formula ($Complexity = DecisionPoints + 1$), but not with the formulas given on the Wikipedia page.

   writeLongBits(): The result was a cyclomatic complexity of 8. This was both the manual calculation and the one given by Lizard. The calculations were done using the formula $Complexity = Edges - Nodes + 2P$ for the complete graph and calculations see figure 1.

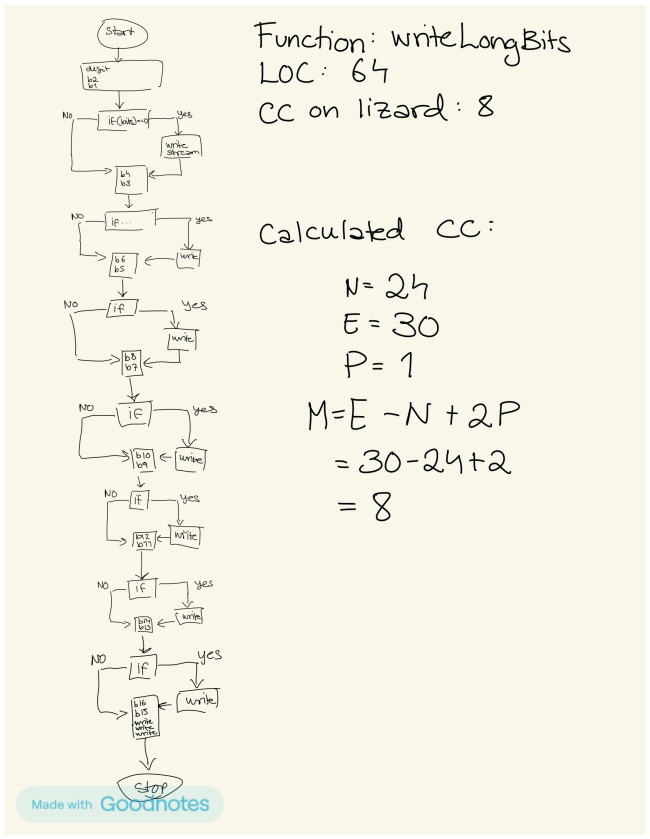

Figure 1: The CC calculations of the writeLongBits function

writeLong(): The result was a cyclomatic complexity of 9. This was both the manual calculation and the one given by Lizard. The calculations are very similar to those for writeLongBits seen in figure 1 due to the structure of the two functions being similar. In comparison to writeLongBits, writeLong contains one additional nestled if-statement contributing to two additional nodes and three additional edges in the graph of the function and thus increasing the complexity by 1.

readNumber(): The result was a cyclomatic complexity of 20. This was done by $Complexity = Edges - Nodes + 2P$, and verified with lizard. What is interesting here is that the function does not look like it might have a cyclomatic complexity of 20 when looking at the code, as it is relatively simple. However the majority of the complexity comes from a switch case, that adds 13 complexity compared to 3 if statements. This is also shown in the graph where two edges represent more than one edge. 2.

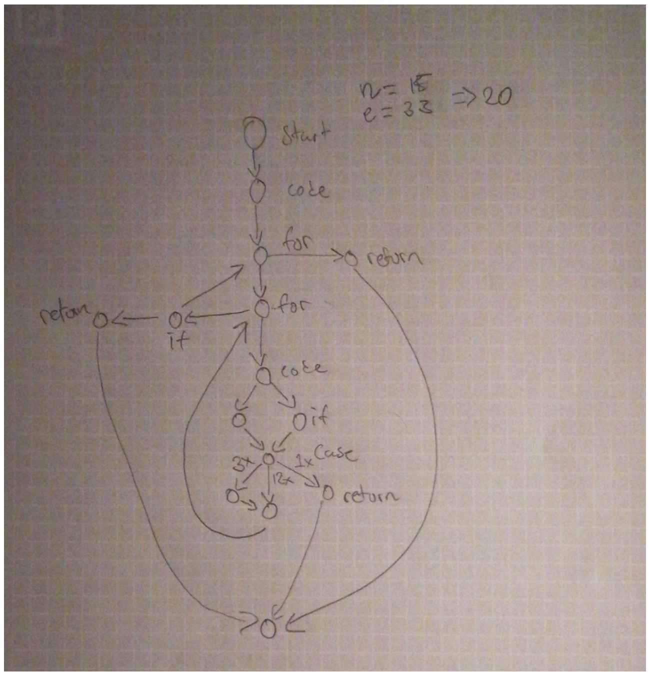

Figure 2: The CC calculations of the readNumber function

2. Are the functions just complex, or also long?
   readAny(): The function is not very complex, it consists mainly of a large switch case.

   writeLongBits(): The function is not very complex, it consists mainly of flat if statements.

   writeLong(): The function is not very complex, it consists mainly of flat if statements.

   readNumber(): This function is very complex according to the tool, but has actually a medium complexity. The complexity does not come from its length, but the switch case.

3. What is the purpose of the functions?

   readAny(): If the type of a json field or value is not known at compile-time, then this function can detect the type of the next JSON value and returns it as an Any object. This any object can then be type checked, and used differently based on the content.

writeLongBits(): The function encodes a long value into a compact, custom string representation using a lookup table called DIGITS and writes it to a JsonStream. It processes the long value byte by byte, and stops early if the remaining bits are zero.

writeLong(): The function encodes a long value into a compact, custom string representation using a lookup table called DIGITS and writes it to a JsonStream. It processes the long value tree digits at a time, stopping early if the remaining digits are zero.

readNumber(): This function is for reading in numeral values that can then be parsed as floats, integer or even big integer values based on the content. As such it is primally a helper function for all number parsing functions.

4. Are exceptions taken into account in the given measurements?
   readAny(): No.
   writeLongBits(): No.
   writeLong(): No.
   readNumber(): No.

   In theory, almost every line of java code may throw an error due OOM, but also more commonly array access or null pointer exception. As Java has a verbose error handling, the cyclomatic complexity with exceptions can give unfair answers to very simple functions. Due to this, we did not choose to include the exceptions in the complexity calculation.

5. Is the documentation clear w.r.t. all the possible outcomes?
   readAny(): The function does not contain any documentation.
   writeLongBits(): The function does not contain any documentation.
   writeLong(): The function does not contain any documentation.
   readNumber(): The function does not contain any documentation.

1.5 Refactoring

readNumber(): To reduce the cyclomatic complexity of the readNumber() method there are modifications that can be made.

- Reduce the complexity of the switch case (the main source of complexity in the method) to the cases valid char and invalid char, this can be done by adding all the valid chars to a set and checking for membership in this set instead of each character in an individual case.
- Choosing to use StringBuilder instead for buffer expansion. This removes the decision point `if (j == iter.reusableChars.length)` since

the buffer re-sizes dynamically.

This would reduce the cyclomatic complexity of the method from 20 to 6. With a majority from the removed switch case. Drawbacks from this refactoring include removing the switch-cases decreasing the readability of the code.

readStringSlowPath() in IterImpl.java: One possible refactoring of this function could be to separate these overarching if/elseif-statements into separate functions. This is basically the most straight-forward way of reducing the complexity of each of the functions (since most of the complexity is just nestled if-statements). One could, for example, make one function which handles cases with escaped characters and call that function in the first part of the if/elseif-statement and then make one function which handles multi-byte unicode issues and call that function in the second part of the if/elseif-statement. That would basically split the complexity in two while having a rather minimal effect on performance. This code is however supposed to be well performing so the performance loss might not be worth it.

readStringSlowPath() in IterImplForStreaming.java: One possible refactoring of this function is to put the switch case for when we encounter a backslash in a helper function so that the helper function is called only when we actually encounter a backslash. One drawback of this is that the code might be more difficult to understand since you need to jump between different functions (which also makes the code file longer).

decodeFast(): The main cause of complexity in this function is handling invalid input data, or special conditions. As such it might be possible to refactor it into separate functions like isValidBase64, decodeContent, decodeTail. The function also abuses the ternary operator, where you could do basic assignment. This would reduce the complexity from 18 to $\tilde{3}$, but might cause confusion if a reader have to jump around to read, and understand a simple function.

2 Coverage

2.1 Tools

The tool we used for coverage was the built into our IDEs, and it was as streamlined as just pressing an additional button when testing. We did not run into any problems using it, and it highlighted in easy-to-understand colors and numbers what branches were taken. The only problem was figuring out where the coverage was coming from, as the coverage only showed what was covered. It was both easy to use and integrate into our development experience.

2.2 Your own coverage tool

readNumbers()

readStringSlowPath() in IterImplForStreaming readStringSlowPath() in IterImpl

decodeFast()

Our tool is quite simple, and only consists of a set where the developer has to insert a value into the set at each branch. After all the tests are run, we simply check how big the set is, and what values are present in the set to generate our coverage report. It is also important that our solution does not detail every single line, but instead sections. This means that the coverage may be 50% while an IDE might report 1% if we have an early return that is much shorter than the rest of the function. If an exception happens mid section, it might give invalid results. Additionally we do not track the amount of times each branch is taken, only if a branch is taken, meaning that it is less detailed than the IDE.

2.3 Evaluation

1. How detailed is your coverage measurement?
   readNumber(): Every decision point is logged, allowing to track branch coverage of the tests.

   readStringSlowPath(): Every decision point is logged with one data collection point for each outcome. This means each part of the code will be flagged if it is reached at least once.

   readStringSlowPath() in IterImplForStreaming.java: Every decision point is logged, allowing to track the branch coverage for the entire function.

   decodeFast(): Every if and for loop is logged, however some ternary operator are not.

2. What are the limitations of your own tool?
   readNumber(): Other forms of coverage such as Path Coverage or tracking how many times conditions execute are not tracked.

   readStringSlowPath() in IterImpl: Path coverage is not tracked and would be tedious to implement since almost the entire function is included in a loop, containing many if statements. How many times a certain branch is reached is not tracked either.

   readStringSlowPath() in IterImplForStreaming.java: This tool does not track Path Coverage or the amount of times different conditions are executed.

   decodeFast(): The main limitation is that ternary operator are not logged, and it would require changing the code a lot to include. This limits how many branches that are logged.

3. Are the results of your tool consistent with existing coverage tools?
   readNumber(): Yes it shows that the same branches are untested as my

DIY tool. After the new tests were implemented the tool was consistent and showed better coverage.

readNumber(): The automated tool does not seem to take into consideration unwritten else-statements which the DIY tool does. This likely results in some slight differences in code coverage statistics since the function contains many if-statements.

readStringSlowPath() in IterImplForStreaming.java: Yes, when running the DIY tool, I see that the coverage has hit on the branches I have tested for, and when using an existing coverage tool I can see that those parts of the code are now covered (marked with green and uncovered marked with red)

readStringSlowPath() in IterImpl: The DIY tool for readStringSlowPath seems to take into consideration more types branches (mainly unwritten else statements) than the automated tool. This likely leads to a disparity in the increase in coverage of the automated tool and the DIY tool after writing tests.

decodeFast(): Yes, the results correlate largely an IDE, but may be less detailed and fine grained as detailed in the sections above.

2.4 Coverage improvement

2.4.1 readStringSlowPath() in IterImplForStreaming.java

1. When a simple string is encountered, the function should append all characters into iter.reusableChars and stop at the closing quote. Making branch 0 and 1 hit.
2. If the reusableChars array is full before appending a character, the function should expand the buffer making branch 32 hit.

Report of old coverage:

```
Branch Coverage Report for readStringSlowPath():
    2_Backslash: Not Hit
    26_MultiByte_4Byte: Not Hit
    22_MultiByte_2Byte: Not Hit
    25_Else_MultiByte_3Byte: Not Hit
    31_ExpandBuffer: Not Hit
    15_LowSurrogateCheck: Not Hit
    28_UnicodeCheck: Not Hit
    14_ValidHighSurrogate: Not Hit
    29_InvalidUnicodeCheck: Not Hit
    11_Unicode: Not Hit
    3_Case_b: Not Hit
    1_ExitLoop: Not Hit
    20_DefaultCase: Not Hit
    27_InvalidMultiByte: Not Hit
    17_InvalidSurrogate: Not Hit
    9_Case_Slash: Not Hit
    32_ExpandBuffer_Final: Not Hit
    16_ValidLowSurrogate: Not Hit
    24_MultiByte_3Byte: Not Hit
    23_Else_MultiByte_2Byte: Not Hit
    13_InvalidSurrogate: Not Hit
    5_Case_n: Not Hit
    10_Case_Backslash: Not Hit
    21_MultiByteCheck: Not Hit
    0_EnterOuterLoop: Not Hit
    6_Case_f: Not Hit
    12_HighSurrogateCheck: Not Hit
    8_Case_Quote: Not Hit
    7_Case_r: Not Hit
    18_ElseBranch: Not Hit
    19_InvalidSurrogate: Not Hit
    30_ExpandBuffer: Not Hit
    4_Case_t: Not Hit

Branch Coverage Report for readStringSlowPath():
    2_Backslash: Not Hit
    26_MultiByte_4Byte: Not Hit
    22_MultiByte_2Byte: Not Hit
    25_Else_MultiByte_3Byte: Not Hit
    31_ExpandBuffer: Not Hit
    15_LowSurrogateCheck: Not Hit
    28_UnicodeCheck: Not Hit
    14_ValidHighSurrogate: Not Hit
    29_InvalidUnicodeCheck: Not Hit
    11_Unicode: Not Hit
    3_Case_b: Not Hit
    1_ExitLoop: Hit
    20_DefaultCase: Not Hit
    27_InvalidMultiByte: Not Hit
    17_InvalidSurrogate: Not Hit
    9_Case_Slash: Not Hit
    32_ExpandBuffer_Final: Hit
    16_ValidLowSurrogate: Not Hit
    24_MultiByte_3Byte: Not Hit
    23_Else_MultiByte_2Byte: Not Hit
    13_InvalidSurrogate: Not Hit
    5_Case_n: Not Hit
    10_Case_Backslash: Not Hit
    21_MultiByteCheck: Not Hit
```

```
0_EnterOuterLoop: Hit
6_Case_f: Not Hit
12_HighSurrogateCheck: Not Hit
8_Case_Quote: Not Hit
7_Case_r: Not Hit
18_ElseBranch: Not Hit
19_InvalidSurrogate: Not Hit
30_ExpandBuffer: Not Hit
4_Case_t: Not Hit
```

Test cases added: testReadStringSlowPathWithSimpleString() and testReadStringSlowPathExpandsTheBufferWhenFull().

2.4.2 readStringSlowPath() in IterImpl.java

1. Test a complete string ending with "
2. Test a string that contains \\
3. Test string that contains \\b
4. Test string that contains \\t
5. Test string that contains \\n
6. Test string that contains \\f
7. Test string that contains \\r
8. Test string that contains \\/
9. Test string that contains \\\\
10. Test incomplete string that does not end in "

Report of old coverage: The function did not have any associated tests so the coverage was 0 branches.

Report of new coverage: 15 out of 44 branches reached (branch 44-49 do not exist)

Test cases added: testReadSampleString(), testReadEscapedCharString(), testDynamicBufResize() and testIncompleteString().

2.4.3 readNumber()

1. input with non-number characters. Assert that the numbers before the illegal character are returned.
2. Assert that the correct number is returned when `reusableChars.length <= j`

Report of old coverage:

```
Branch Coverage Report for readNumber():
    18_Case9: Hit
    8_Case+: Hit
    7_Case-: Hit
    17_Case8: Hit
    6_CaseE: Hit
    11_Case2: Hit
    19_CaseDefault: Not Hit
    9_Case0: Hit
    16_Case7: Hit
    1_EnterOuterLoop: Hit
    2_EnterInnerLoop: Hit
    12_Case3: Hit
    4_CaseDot: Hit
    5_Casee: Hit
    15_Case6: Hit
    13_Case4: Hit
    20_NotLoadMore: Hit
    10_Case1: Hit
    14_Case5: Hit
    3_Expand: Not Hit
    Total branch coverage is 90.0%
```

Report of new coverage:

```
Branch Coverage Report for readNumber():
    18_Case9: Hit
    8_Case+: Hit
    7_Case-: Hit
    17_Case8: Hit
    6_CaseE: Hit
    11_Case2: Hit
    19_CaseDefault: Hit
    9_Case0: Hit
    16_Case7: Hit
    1_EnterOuterLoop: Hit
    2_EnterInnerLoop: Hit
    12_Case3: Hit
    4_CaseDot: Hit
    5_Casee: Hit
    15_Case6: Hit
    13_Case4: Hit
    20_NotLoadMore: Hit
    10_Case1: Hit
    14_Case5: Hit
    3_Expand: Hit
    Total branch coverage is 100.0%
```

Test cases added: testDefaultCase() and testExpand().

2.4.4 decodeFast()

1. Empty input
2. Input with invalid start or end characters
3. Input with line seprators every 76 characters
4. Large input with trailing ==

Report of old coverage:

```
Branch Coverage Report for decodeFast():
    0: Hit
    1: Not Hit
    2: Not Hit
    3: Not Hit
    4: Hit
    5: Not Hit
    6: Not Hit
    7: Not Hit
    8: Not Hit
    Total branch coverage is 22.22222222222222%
```

Report of new coverage:

```
Branch Coverage Report for decodeFast():
    0: Hit
    1: Hit
    2: Hit
    3: Hit
    4: Hit
    5: Hit
    6: Hit
    7: Hit
    8: Hit
    Total branch coverage is 100.0%
```

Test cases added: testDecodeInvalid(), testDecodeEmpty() testDecodeLineSeperator() and testLastEquals().

2.5 Self-assessment: Way of working

The current state of the mean was evaluated to be in the Collaborating state with some objectives on the Performing state reached. The team is still strong with open communication and members focused on the team's mission. In the Performing state, the team achieved almost all objectives, there was no backtracking/re-work for this assignment, and the team worked efficiently and effectively by dividing work and collaborating when needed. The objective holding the team back from fully achieving the Performing state is the waste continually eliminated, as this is not something we focused on. The self-assessment was unanimous, everyone agreed on both or strong points and improvements.

2.6 Overall experience

We have learned a lot about branch coverage and how it works. It was interesting to have the branch coverage when writing tests since they were helpful in finding meaningful test cases to create!