Fundamentals of Software Engineering

Assignment #3

Johan Norlin Erik Vinblad Douglas Fischer
Wenqi Cao Robin Claesson

2024-02-21

1 Project Choice

`fabric-loader`: a Minecraft mod loader written in Java. The project provides mod-loading facilities and abstractions for other mods to use.

Original repository: `https://github.com/FabricMC/fabric-loader`

Our fork: `https://github.com/DD2480-Group-4/lab3-fabric-loader`

1.1 Onboarding Experience

The project is built using `Gradle`, with everything needed to build the software included within the repository. The build file is rather complex, and there is a lack of documentation throughout the project. This makes for a barrier of entry, however, the build concludes with no errors and all included tests pass.

Initially, there was a discussion about whether the group should proceed with `fabric-loader`, or continue with another project. The group chose to stick with this repository due to one group member having extensive experience working with the project's ecosystem.

2 `ModSolver::deriveVersion`

2.1 Complexity

A function that is 66 lines long. The purpose of `deriveVersion` is to perform checks on the provided version range to find an appropriate value. The function is used to determine which version of a dependency is recommended to the user when a dependency is missing. Many of these operations could be refactored into separate functions. Some functions theoretically could throw exceptions, but they would only throw exceptions on invalid inputs, which are checked before running them. However, if these conditions were to not be checked sufficiently at any point, it would be the responsibility of the caller to catch the exception. This function contains little to no documentation, only a few inline comments.

- Manual CCN count: 24 (22 if `&&` statements are ignored).
- Cross-reference by a second group member: 30 (or 28).
- Third group member: 26.
- Fourth group member: 26.
- `Lizard`: 24.

2.2 Testing

Five tests were added to a new test class. This test class contains a helper function which calls the deriveVersion function via reflection since deriveVersion is declared as private within a private class. The test cases are as follows:

1. `minExclusiveReturnsNextVersion`: Tests that the function increments the minimum version to a pre-release of the next version when the minimum version is a full version and the range is non-inclusive.
2. `maxExclusiveReturnsLargePreRelease`: Tests that the function decrements the maximum version to a pre-release for the previous version when the maximum version is a full version and the range is non-inclusive.
3. `maxExclusivePreReleaseSmallADecrementsToCapitalZ`: Tests that the pre-release with the value 'a' decrements to the pre-release with the value 'Z'.
4. `maxExclusiveLastPreReleaseDecrementsToPreviousVersion`: Tests that decrementing empty pre-release decrements the version number to the previous version. Also, test that version numbers with less than 3 numbers will be padded to include 3 numbers.
5. `unboundedReturns1`: Tests that an empty version range returns version 1.

2.3 Refactoring Plan

In terms of refactoring, this function does multiple things. For semantic versions, it performs an increment on the lower bound or a decrement on the upper bound to get a version within the range. These operations will do different things depending on many parameters, and should therefore be their own functions. One increment function, and one decrement function. The decrement operation is also quite complex as it is, mostly due to the complex decrement operation of the pre-release string. This should perhaps also be its own function. In theory, this way you should be able to get the cyclomatic complexity down to less than 10.

2.4 Carried out Refactoring (P+)

To simplify `deriveVersion`, I split out the increment and decrement of the version into separate functions. Since the decrement of the version was still quite complex, I also split the pre-release part of it out into its own function as well. When counting the cyclomatic complexity with Lizard after performing the refactor, I get the following results:

- `ModSolver::deriveVersion`: 7
- `ModSolver::incrementVersion`: 4
- `ModSolver::decrementVersion`: 6
- `ModSolver::decrementPreRelease`: 10

As we can see, even the worst offender, `ModSolver::decrementPreRelease`, is about 58% less complex than the original function (based on the original metric measured by Lizard for `ModSolver::deriveVersion`, which was 24).

> *Command to fetch changes*:
> `git diff master johan-p+-refactor-modsolverderiveversion`

3 `StringUtil::wrapLines`

3.1 Complexity

A rather short function with 40 lines of code, with the responsibility to wrap strings that exceed a certain limit. This function does not serve a specific purpose for one part of the program, but is rather a utility function. This function is not documented but has a somewhat clear purpose.

- Manual CCN count: 12 (14 if including || operator and including the function itself).
- Cross-reference by a second group member: 14.
- `Lizard`: 14.

3.2 Testing

Two tests were added, namely:

1. `longStringPerformsWrapping`: Test that verifies that a long string that exceeds a given limit will in fact wrap over to the next line.
2. `shortStringNoWrapping`: Test that verifies that a short string (empty string) given a generous limit should not wrap, and should in fact return as yet another empty string.

3.3 Refactoring Plan

`wrapLines` follows the structure of a parser. The current implementation consists of nested `if-else` statements wrapped in a large loop. This has the purpose of finding whitespace within the string to perform wrapping (i.e. to append a newline character to the current string using a `StringBuilder`). The functionality of this method does not have to be split up into several functions, but could rather benefit from incorporating an outer `foreach` loop containing a `switch` statement instead. This makes for a cleaner code snippet, and enhances readability. The cyclomatic complexity would not necessarily change, but would make the code more scalable.

3.4 Carried out Refactoring (P+)

3.4.1 Added tests

Since I previously planned to go for P, only two unit tests had been incorporated. Before carrying out this refactoring, two more were added, namely:

- `testWrapLines_escapeSequences`: A test that verifies that a string containing escape sequences is wrapped correctly.

- `testWrapLines_withQuotations`: A test that verifies that strings containing quotations are handled correctly.

After this addition, 100% branch coverage was achieved. This ensured that the refactored `wrapLines` adhered to the same functionality.

3.4.2 Refactoring

My original plan to simplify the method involved data structures that enhance readability and scalability but at the expense of CCN. Incorporating a `foreach` loop lacks peeking into the future and a `switch` statement comes with an extra branch due to the `default` case. Instead, I opted to rewrite the line-wrapping algorithm to perform for the following steps:

1. Ignore carriage returns.
2. Record the previously encountered blank space that might potentially be used for a split.
3. If the current line exceeds the limit, either split on the next character if that is a blank space or revert to the last encountered space and split there.

This removes any redundancy, including the prior implementation concerning splits inside quotations which would never occur. Additionally, there was previously a check to terminate the `for` loop at the last iteration.

The carried out refactoring lowered the cyclomatic complexity from 13 to 7, thus a reduction of 46%. The following result was recorded by `Lizard` (which includes the function itself):

```
 lizard StringUtil.java
================================================
  NLOC    CCN   token  PARAM  length  location
------------------------------------------------
      18      6    160      1      24 StringUtil::capitalize@22-45@StringUtil.java
       8      2     67      2       9 StringUtil::splitNamespaced@47-55@StringUtil.java
      41      8    260      2      49 StringUtil::wrapLines@57-105@StringUtil.java
1 file analyzed.
```

To summarize, this refactoring ensured 100% branch coverage, and an overall complexity reduction of 46%. The changes can be fetched using the following command:

```
git diff 16-combine-all-tests erik-p+-refactor-wraplines
```

4 `ResultAnalyzer::formatVersionRequirements`

4.1 Complexity

A function used to format a collection of mod version requirements into a string representation. It is 52 lines long (46 non-comment lines). The amount of nested branch statements makes the code quite complex and hard to follow. The function is not considered long, but it should be mentioned that the code could be refactored. This function lacks documentation, apart from a few inline comments.

- Manual CCN count: 18.
- Cross-reference by a second group member: 21.
- Third group member: 19.
- `Lizard`: 22.

4.2 Testing

Five test cases have been added to this function. These tests cover 12 out of the 17 manually identified branches.

1. `testEmptyCollection` tests that the function returns that the range is unsatisfiable when provided with an empty list.
2. `testUndefinedVersionInterval` tests that providing the function with unspecified intervals returns that any version is required.
3. `testLowerRequiredVersion` tests that providing the function with a min (inclusive) required version and no max required version returns that any version from and including the min version is required.
4. `testInclusiveExclusiveVersionIntervalWithEmptyListItems` tests that providing the function with a min (inclusive) required version and a max (exclusive) required version returns that any version from and including the min version to and excluding the max version is required. Also tests that the function skips empty intervals.
5. `testSpecificVersionOnInclusiveAndExclusive` tests that provide the function with required versions min=max returns a specific required version if the interval is min- and max-inclusive. If the interval is not both min- and max-inclusive it skips the interval as it is considered empty/invalid.

4.3 Refactoring Plan

The function can be refactored by splitting the main logic into smaller methods. The main goal would be to get rid of the nested branch statements so the first 3 "else if" statements would be a priority. The total CC including all new methods would most likely not decrease as the branching is necessary. However, the readability would increase, the code would become more modular and the testing capability would also be improved.

4.4 Carried out Refactoring (P+)

To refactor `formatVersionRequirements`, I created 4 new helper functions. The first helper function `formatInterval` was created in order to lift out all version interval handling logic from the main function `formatVersionRequirements`. This has improved the readability quite a bit as the loop previously contained 35 lines of nested if statements. The other 3 helper functions are called from the `formatInterval` function and their purpose is to handle all nested if statements in order to improve the readability of the `formatInterval` function as to not just shift the problem from one function to another. Prior to refactoring, `formatVersionRequirements` had a CC of 18. 35% of which is 7 rounded up from the actual value 6,3. Nor the refactored function or its helper functions individually have a CC higher than 8 so the Cyclomatic Complexity has definitely been improved by at least 35%. Worth noting is that in order to refactor the function, one extra if-statement has been added in order to handle null returns, thus increasing the amount of possible branches in the code. All tests that worked before also work now.

The following results in CCN were obtained after the refactoring:

- `ResultAnalyzer::formatInterval`: 9
- `ResultAnalyzer::formatOnMinNullInterval`: 3
- `ResultAnalyzer::formatOnMaxNullInterval`: 2
- `ResultAnalyzer::formatOnEqualMinMaxInterval`: 3
- `ResultAnalyzer::formatVersionRequirements`: 4

Command to fetch changes:
`git diff 16-combine-all-tests douglas-p+-refactor-formatversionrequirements`

5 SemanticVersionImpl::SemanticVersionImpl

5.1 Complexity

The function is both complex, and long (83 lines). It consists of multiple decision points (`if`-statements and loops), as well as a `try-catch` statement. This function acts as a constructor for a class that represents a semantic version. Semantic versioning is a versioning scheme that uses a three-part version number: `major.minor.patch`. The build and pre-release components are optional and are separated from the version number by a '`+`' and a '`-`', respectively. The '`x`' values are placeholders that can be used to represent any value. Only part of the exceptions are taken into consideration. There is no documentation present, and thus seems very unclear.

- Manual CCN count: 18 (26 if `&&` and || operators are included, as well as the function itself).
- Cross-reference by a second group member: 18 (25 if including other operators and excluding the function itself).
- `Lizard`: 25.

5.2 Testing

Four test cases have been added to this function:

1. `Test case 1:` Test prerelease string format. It tests the scenario where the input version string has an invalid prerelease string format like the 'gamma?' in '2.2.7-gamma?.7'. The input string has an invalid prerelease string format because it contains a question mark (?), which is not allowed in a prerelease string according to the Semantic Versioning specification. The expected result is false, indicating that the input string does not conform to the Semantic Versioning format.
2. `Test case 2:` Test `version.startsWith(".")`. It checks if the method `tryParseSemantic` returns false when the input string is ".2.7-gamma.7". It is not allowed in a Semantic Versioning version string to have the input string start with a dot (.). The expected result is false, indicating that the input string does not conform to the Semantic Versioning format.
3. `Test case 3:` Test the `version component[i]` is negative. It tests if the method `tryParseSemantic` returns false when the input string is "2.-2.0" and the second parameter is false. The input string contains a + character, which is not allowed in a Semantic Versioning version string. The expected result is false, indicating that the input string does not conform to the Semantic Versioning format.
4. `Test case 4:` Test `componentStrings.length` is 0. It checks if the method `tryParseSemantic` returns false when the input string is an empty

string "" and the second parameter is true. The input string is empty, which is not allowed in a Semantic Versioning version string. The expected result is false, indicating that the input string does not conform to the Semantic Versioning format.

5.3 Refactoring Plan

The SemanticVersionImpl function can be refactored into smaller functions that handle specific tasks, such as parsing the version string, handling the build and pre-release components, and validating the version number components. This can make the code easier to understand and maintain as the constructor function. Moreover, there's some part of the code that will never be reached and thus can be deleted in the refactoring process.

5.4 Carried out Refactoring (P+)

To refactor the function, the buildPrereleaseHandler, wildcardComparer, and wildcardStripper methods are extracted from the SemanticVersionImpl constructor to improve readability and maintainability. These methods handle specific tasks related to parsing the version string, checking for wildcards, and stripping extra wildcards. The constructor then calls these methods to perform these tasks, making the code more modular and easier to understand. Moreover, I delete the redundant and useless code that will never be reached (e.g., 'if' statement is always True or False) in the old implementation to reduce Cyclomatic Complexity (CC). The CCN of the old version is 25 while the refactored version is only 15. Thereby, the Cyclomatic Complexity has been improved by 40%.

```
PS D:\My Document\My Graduate Study\All_Courses\DD2480 Software Engineering Fundamental\Assignments\lab3-fabric-loader>
                                                                                                                  lizard "D:\
My Document\My Graduate Study\All_Courses\DD2480 Software Engineering Fundamental\Assignments\lab3-fabric-loader\src\main\java\net\
fabricmc\loader\impl\util\version\SemanticVersionImpl.java"
================================================
  NLOC    CCN   token  PARAM  length  location
------------------------------------------------
      40     15    308      2      57 SemanticVersionImpl::SemanticVersionImpl@47-103@D:\My Document\My Graduate Study\All_Courses\
DD2480 Software Engineering Fundamental\Assignments\lab3-fabric-loader\src\main\java\net\fabricmc\loader\impl\util\version\Semantic
VersionImpl.java
      21      5    156      1      26 SemanticVersionImpl::BuildPrereleaseHandler@105-130@D:\My Document\My Graduate Study\All_Cour
ses\DD2480 Software Engineering Fundamental\Assignments\lab3-fabric-loader\src\main\java\net\fabricmc\loader\impl\util\version\Sema
nticVersionImpl.java
       3      3     31      1       3 SemanticVersionImpl::WildcardComparer@132-134@D:\My Document\My Graduate Study\All_Courses\DD
2480 Software Engineering Fundamental\Assignments\lab3-fabric-loader\src\main\java\net\fabricmc\loader\impl\util\version\SemanticVe
rsionImpl.java
       5      3     46      2       5 SemanticVersionImpl::WildcardStripper@136-140@D:\My Document\My Graduate Study\All_Courses\DD
2480 Software Engineering Fundamental\Assignments\lab3-fabric-loader\src\main\java\net\fabricmc\loader\impl\util\version\SemanticVe
rsionImpl.java
```

Command to fetch changes:
`git diff master wenqi-p+-refactor-SemanticVersionImplSemanticVersionImpl`

6 `ResultAnalyzer::addErrorToList`

6.1 Complexity

A function that is around 50 lines long (if blank lines are included). It is not considered overly complex, nor long. Its purpose is to print an error to a `PrintWriter`. This is not directly related to high CC, but rather due to the number of different formatting rules. This function can cause expectations, and there is no documentation available.

- Manual CCN count: 12.
- Cross-reference by a second group member: 12.
- `Lizard`: 12.

6.2 Testing

Two tests were added:

1. Ran the function with an empty matches list and the `presentForOtherEnv` flag set to false. This should result in the function printing out an error to the given `PrintWriter` that the version range is missing.
2. Ran the function with an empty matches list and the `presentForOtherEnv` flag set to true. This should result in the function printing out an error to the given `PrintWriter` that the version range is disabled for this environment.

6.3 Refactoring Plan

There is a large combination of loops and if statements in the middle of the function to find a string named reason that is later used as a key. This logic could be separated into a new function that returns this string, which would lower the cyclomatic complexity of this function by over 50%.

7 Coverage Measurement & Improvement

7.1 DIY Coverage Tool

This tool utilizes a boolean array to measure what branches have been reached. The user itself places flags (`arr[x] = true;`) for each branch. Therefore, this tool could theoretically work for every construct.

> *Link to branch*: `https://github.com/DD2480-Group-4/lab3-fabric-loader/tree/issue1-implement-diy-coverage-tool`
>
> *File containing solution*: `https://github.com/DD2480-Group-4/lab3-fabric-loader/blob/issue1-implement-diy-coverage-tool/src/main/java/net/fabricmc/TestCoverage.java`
>
> *Command to fetch the changes*: `git diff d69cb72 551dd97`

The coverage has been cross-referenced with `Jacoco`, a Java code coverage library that provides detailed reports.

> *Results*: `https://drive.google.com/drive/u/0/folders/1gRfzlGb3SG19DYvnMVUi__y95cuZ0hW0`

7.2 Coverage Improvement

The measurement tool provides the user with information about how many branches were reached during testing out of all existing branches. However, the maximum number of branches is determined when the user adds the setting of the flags to the function. It is also possible to get information about exactly which branches have been taken if the user chooses to print the boolean array.

7.2.1 Limitations

The tool is specifically designed for this project, and not particularly generic. It can only check the branch coverage of branches that are explicitly added. If a branch is missed by a user, it will not appear in the coverage. Our DIY tool does not support ternary expressions, nor inline if-statements (they have to be manually broken up by the user). The DIY tools also ignores '`&&`' and '||' operators.

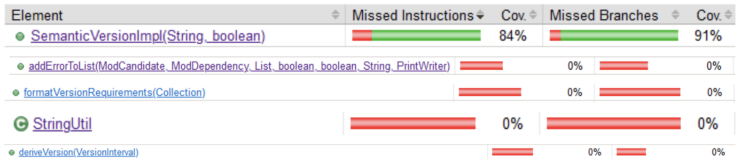
7.2.2 Consistency

This tool is inconsistent with automatic tools such as `Jacoco` since they do not ignore operators like '`&&`' and '||'. `Jacoco` treats the '`&&`' operator as creating two paths in the code: one where the first condition is true, the second condition is true too, and the other where the first condition is false (so the second condition is not evaluated). Each path through the code increases the complexity by 1. However, our DIY tool fulfilled all the task requirements but omitted this part for convenience.

7.3 Results of Improvement

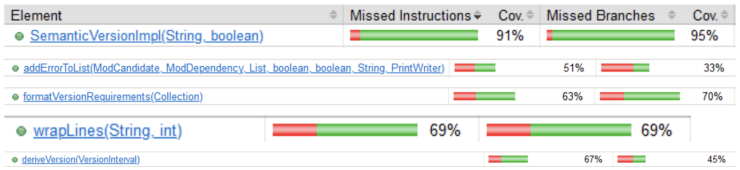
The following coverage was obtained before any improvements:

```
StringUtils_wrapLines: 0/12
ModSolver_deriveVersion: 0/22
ResultAnalyzer_formatVersionRequirements: 0/16
ResultAnalyzer_addErrorToList: 0/11
SemanticVersionImpl_SemanticVersionImpl: 15/19
```



After the improvements:

```
StringUtils_wrapLines: 8/12
ModSolver_deriveVersion: 15/22
ResultAnalyzer_formatVersionRequirements: 12/17
ResultAnalyzer_addErrorToList: 2/11
SemanticVersionImpl_SemanticVersionImpl: 17/19
```



Method	DIY	Jacoco (%)
`ModSolver::deriveVersion`	15/22	45
`StringUtils::wrapLines`	8/12	69
`ResultAnalyzer::formatVersionRequirements`	12/17	70
`SemanticVersionImpl::SemanticVersionImpl`	17/19	95
`ResultAnalyzer::addErrorToList`	2/11	33

7.4 Self-Assessment

The group feel more confident with writing tests and navigating GitHub, as well as generally using `Git` as a version control system. There is potential for improvement, namely with the addition of a new group member the team

dynamic shifted slightly. Due to the short deadline, we did not have time to properly examine the strengths and weaknesses of all group members. Due to the nature of the assignment, synchronising the work was not as important, thus our previous workflow of implementer, tester and reviewer was not utilized fully.

7.5 Overall Experience

This is an open-source project without a strict refactoring plan. This makes for some redundancy and irregularities in the code, such as a lack of documentation and testing.

Private methods and classes can make writing tests unnecessarily complicated. This can be worked around with reflection, but that will increase the cyclomatic complexity of the test.

Additionally, when doing this assignment, it is a good idea to check what types the inputs to a function are before choosing it, as they can be so complex that it is almost impossible to complete the assignment within the allotted time.