Assignment 3 Group 26

Marcus Jakobsson
Toto Roomi
Rémi Grasset
Kerem Robin Yurt
Anton Bölenius

February 2024

1 Introduction

- Name: jabref
- URL: jabref
- Fork: jabref_26
- Jabref is an opensource reference manager. It is java-based.

2 Grade aim

- Marcus: P+
- Robin: P+
- Remi: P+
- Anton: P
- Toto: P

For the students aiming for P+, each of us met the requirements of the first 3 criteria (4 tests, issue tracker, refactoring, extraordinary).

3 Onboarding experience

Did it build and run as documented?
We had a variety of machines with different Operating systems and java versions. The windows and mac machines were successful in building the project, however, at first, they did not run all the tests successfully. The linux machine

failed on java 17 but everything ran flawlessly on version 21 and after following their guide. One windows machine attempted to build using WSL. Things worked out towards the end, some machines still skipped and failed a few tests, but it wasn't significant enough to hinder the work.

4 Complexity

Anton & Rémi
CCN: 16, Counted 16;
43 16 338 0 60 ArgumentProcessor::importAndOpenFiles@528-587@./org/jabref/cli/ArgumentProcessor.java
Marcus & Robin
CCN: 20, Counted 22
EditAction::execute@52-99@./org/jabref/gui/edit/EditAction.java
Toto & Rémi
CCN: 16, Counted: 16 24 16 159 2 25 GroupTreeView::ContextAction::ContextAction@629-653@./org/jabref/gui/groups/GroupTreeView.java
Anton & Robin
CCN: 17, Counted: 17 20 17 173 0 20 SpecialFieldValueViewModel::getToolTipText@33-52@./org/jabref/gui/specialfields/SpecialFieldValueViewModel.java
Toto & Marcus
CCN: 16, Counted 16;
ModsImporter::handleAuthorsInNamePart@551-589@./org/jabref/logic/importer/fileformat/ModsImporter.java

1. **What are your results for five complex functions?**

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

     Most of the methods got the same result, however, there was one function "execute" in the class "EditAction" where lizard got a CCN of 20 and where we manually got 21. It's hard to say why we differed, one possibility could be switch statements that can sometimes be interpreted differently by tools and manual calculations. Some tools might count each case as a separate path, while others might not.

   - Are the results clear?

     The results are clear and mostly agree with our findings.

2. **Are the functions just complex, or also long?**

   These functions are not excessively long, but they contain numerous decision points which lead to a high cyclomatic complexity compared to their sizes:

$$\begin{aligned}
\text{ArgumentProcessor::importAndOpenFiles} &: LOC \simeq 43; CC \simeq 16\\
\text{EditAction::execute} &: LOC \simeq 44; CC \simeq 20\\
\text{ContextAction::ContextAction} &: LOC \simeq 24; CC \simeq 16\\
\text{SpecialFieldValueViewModel::getToolTipText} &: LOC \simeq 20; CC \simeq 17\\
\text{ModsImporter::handleAuthorsInNamePart} &: LOC \simeq 31; CC \simeq 16
\end{aligned}$$

(the previous metrics are the one provided by 'lizard')

3. **Are exceptions taken into account in the given measurements?**

   Lizard takes into account the *try...catch* blocks.

4. **Is the documentation clear w.r.t. all the possible outcomes?**

   Most of the functions we worked on were not well documented. Some included comments to describe the different branches, but it was only partial.

5 Refactoring

importAndOpenFiles:

Regarding the cyclomatic complexity This method has three if statements and one for loop in the later part of the method that could be refactored into separate methods to reduce the high cyclomatic complexity. This change would reduce the cyclomatic complexity by 7. The refactoring could be done in different ways however. For example, each if statement in the end could be refactored into its own function, or they could all be refactored into one function. I think making them into one function makes sense since they are about deciding which type of import command to use. The new function could then be called something like "runConditionalImports". This function could then be refactored further, making each if statement in its own function, which would maintain low CC.

execute:

This method has two switch statements that could be refactored into two methods to decrease the high cyclomatic complexity. This refactoring would decrease the current cyclomatic complexity by 14.

ContextAction:

This method is dependent of a switch statement which is the reason to a high cyclomatic complexity of 16. It would be unnecessary to refactor this method since the switch statement is the reason for the complexity.

getToolTipText:

This method has it's high complexity due to the switch statement. It is not neccessary to do a refactoring for this method.

handleAuthorsInNamePart:

This methods cyclomatic complexity could be decreased by 4, by moving the last if else statement to a separate method.
Plan for refactoring complex code:

Estimated impact of refactoring (lower CC, but other drawbacks?).

Refactoring by moving some code in new methods will lead to a decrease in cyclomatic complexity since it will be shared by the different methods. However, this might reduce the readability of the code due to non-linear navigation, such as jumps between methods. This issue could be mitigated with good documentation of function interfaces, so that a programmer can easily understand the function interface without checking the implementation. The refactor could also slightly increase the total CC (A reasonable increase should not be an issue since the refactor will still make the unit-testing easier.). Moreover invoking a new method involves managing the stack pointer, which could impact performance. Specifically, if the refactored version includes a loop that calls a new auxiliary function, this could have a negative effect on performance.

5.1 Carried out refactoring (optional, P+)

- Marcus (Link to PR) Reforming RemoveLatexCommandsFormatter::format from CCN 23 to CCN 8
- Robin (Link to PR) Refactored LayoutHelper::parseField@288-398@LayoutHelper.java from CCN 19 to CCN 1, and the method with highest CCN is 10.
- Rémi(Link to PR) Refactored FieldComparator::compare from CCN 18 to CCN 11 (-38.9%) using auxiliary function to do elementary comparisons.

git diff ...

- Marcus: git diff 117c377 ca6152b "*RemoveLatexCommandsFormatter.java"
- Robin: git diff 117c377 b72062b "*LayoutHelper.java"
- Rémi: git diff 117c377 b41c2e2 '*FieldComparator.java'

6 Coverage

6.1 Tools

Document your experience in using a "new"/different coverage tool. How well was the tool documented? Was it possible/easy/difficult to integrate it with your build environment?

We used Jacoco to compute the branch coverage. The tool is well documented and it was simple to integrate it with gradle. We only had to add a line in the build.gradle file, and execute a specific gradle task to generate the report for the whole project.

6.2 Your own coverage tool

Show a patch (or link to a branch) that shows the instrumented code to gather coverage measurements.
Our coverage tool uses a HashMap to store the branches reached. An example of the implementation of the coverage tool can be found here:

git diff 0a6dd4b bdf191a '*MedlinePlainImporter*'

What kinds of constructs does your tool support, and how accurate is its output?
Since the tool is manually implemented, it can cover any code blocks that we decide to (if, &&, ||, try...catch, while, etc). However implementing it exhaustively might require a lot of time, especially on functions with chained if...else statements.

6.3 Evaluation

1. How detailed is your coverage measurement?

   The tool counts the possible branches including AND operators that make if statements count as 2. This is manually set by the programmer.
   The tool can also count the branches created by a try...catch block. It explicitly prints the branches reached, and might even explicitly print the different branches of a if(A && B) statement covered by the tests (depending on how detailed the implementation is).

2. What are the limitations of your own tool?
   The tool is not able to create a graph and and count the edges and nodes. Rather it counts branches that are taken.
   To get an exhaustive report, we might need to artificially create new blocks of code (especially to handle chained if...else if... blocks). Another clear limitation is that the setting of the branches reached has to be done manually by the programmer for each branch rather than inferred by the program such as in jacoco.

3. Are the results of your tool consistent with existing coverage tools?
   Compared to lizard the tool is off by 1 in some tests. This may be because lizard adds 1 for the whole function.
   Lizard considers that if(A && B) is always creating 4 branches. However, under some circumstances, our tool might give a different result. Java implements short-circuit evaluation for operators && and ||, which means that the first condition might be used as a safeguard. For instance let us consider the following piece of code:

   ```
   if((array.length > 1) && !array[1].isEmpty())
   ```

   If the first condition is false, then the second condition cannot be evaluated. Thus, there are not 4 branches but only 3: (True, True), (True, False), (False, *). Thus, our coverage tool might give a different result than lizard's in such situations.

7 Coverage improvement

Show the comments that describe the requirements for the coverage.
Report of coverage before adding new tests:

- Jacoco reports: Reports in HTML
- Handmade coverage tool reports: List of reports in HTML

Report of coverage after adding new tests:

- Jacoco reports: Reports in HTML
- Handmade coverage tool reports: List of reports in HTML

Test cases added:

git diff 117c377 main 'src/test/*'

Number of test cases added: two per team member (P) or at least four (P+).

- **Toto**
  src/test/java/org/jabref/logic/bst/util/BstCaseChangersTest.java :
  branchCoverageTestNoneCovered, branchCoverageTestAllCovered
  The function findSpecialChar() checks a certain set of characters in a sries of if statements. The implementation covers all the branches. Before the test 6 branches were covered after its 16.
- **Remi**
  In src/test/java/org/jabref/logic/importer/fileformat/MedlinePlainImporterTest.java:

1 new test to reach branches of MedlinePlainImporter::addDates that expect a creation date with a correct format: MedlinePlainImporterTest::createDateCorrectFormat

3 new tests in src/test/java/org/jabref/logic/bibtex/comparator/FieldComparatorTest for FieldComparator::compare:
This function compares bibentry based on fields, compareYearFieldFirstNan, compareYearFieldSecondNan and bothNonParsable that cover 3 new branches.

- **Marcus**
  - Report of old coverage:
    1. Branch coverage for RemoveLatexCommandsFormatter::format
    2. Branch coverage for RegExpBasedFileFinder::findFile
    3. Branch coverage for RegExpBasedFileFinder::findFile
    4. Branch coverage for LayoutHelper::doBracketedOptionField
  - Report of new coverage:
    1. New Branch coverage for RemoveLatexCommandsFormatter::format
       Our tool
       (Jacoco link)
    2. New Branch coverage for RegExpBasedFileFinder::findFile
       Our tool
       (Jacoco link)
    3. New Branch coverage for RegExpBasedFileFinder::findFile
       Our tool
       (Jacoco link)
    4. New Branch coverage for LayoutHelper::doBracketedOptionField
       Our tool
       (Jacoco link)
  - Test cases added:
    1. specialCommandFollowedByCharacter: Added a test to check when it's removing a latex command with a single character.
    2. findFileNonRecursiveTriggerID_1_3: Added a test to verify that no files are found when searching with a direct path specification.
    3. navigateUpToParentDirectory: Added a test to evaluate the ability to correctly navigate file system paths, specifically testing the behaviour when the search pattern instructs to move up to the parent directory before continuing the search.
    4. testBracketedOptionFieldParsing: Added a test to verify the functionality of creating a layout template that includes a format followed by bracketed option.

- **Robin**
  - Function: LayoutHelper::parseField@288-398@LayoutHelper.java
  - Report of new coverage with the tool JaCocCo can be found here, method *parseField*
  - Report of coverage before with the tool JaCoCo can be found in the github repo documentation. Branch coverage for LayoutHelper::parseField
  - The four test cases that was added is aimed to cover the following branches, and the coverage can be seen here:
    1. begingroup
    2. filename
    3. filepath
    4. endgroup
  - The goal with implementing tests only for `parseField` was due to the reason to cover most branches in one method that was partially covered.
- **Anton**
  - Function: importAndOpenFiles@./src/main/java/org/jabref/cli/ArgumentProcessor.java
  - Test cases added:
    1. noFileImport: Test case for when there are two ".bib" input arguments but only one of the referred files actually exists. In this case the IOException is caught in the catch block and the output file for the existing ".bib" should still be produced.
    2. notBibImport: Test case for when there is a non ".bib" argument left after the existing ".bib", in this case the program should attempt import and then produce the output file for the existing ".bib" file anyway.

8 Self-assessment: Way of working

Current state according to the Essence standard:
The way-of-working is working well for the team. With three assignments finished we have now got a better understanding of how our team is supposed to approach different cases and scenarios. The team has all together evolved and we have now got a better understanding of how each group member prefers to handle issues.

Was the self-assessment unanimous? Any doubts about certain items?
The self-assessment was unanimous, we have developed a good way-of-working throughout the 3 assignments so far and it is getting more and more smooth, one mistake we did on this assignment was that we started quite late with out

first meeting. This prevents team members from working on the problem since choosing of project and delegation of tasks had not been completed.

How have you improved so far? We have improved in the sense that we are better at using github features in a smooth and coordinated fashion. We are also better at communicating when assistance is needed. This way the often time consuming process of learning new tools only has to be done by one member, who then can share the knowledge with the others.

Where is potential for improvement? There is potential for improvement in terms of high level organisation. The team works very well once high level decisions such as task division and delegation is clear and precise. The formation of these decisions may not be optimal, especially when the members have low experience in the subject matter concerned, such as in this assignment.

9 Overall experience

What are your main take-aways from this project? What did you learn?

We all agree that this lab has been challenging uniquely. The biggest challenge was not writing the code, but understanding the overall picture of each function and performing tests where a certain expected response would be given. As there was not much documentation for each function, it was challenging to understand the functionality of individual functions in the overall picture of the project. Eventually, we started to understand each feature and the flow of the current tests, therefore we could create tests to cover the specific branches that were not covered.