Group 28 : DD2480
Assignment 3: Code complexity, coverage

3.1 Part 0: Project choice
Repo: https://github.com/TEAMMATES/teammates - This is the project website for the TEAMMATES feedback management tool for education.

Our repo (forked): https://github.com/vladdobre/teammates

3.3 Onboarding
How good is the "onboarding" documentation? 1. How easily can you build the project? Briefly describe if everything worked as documented or not:

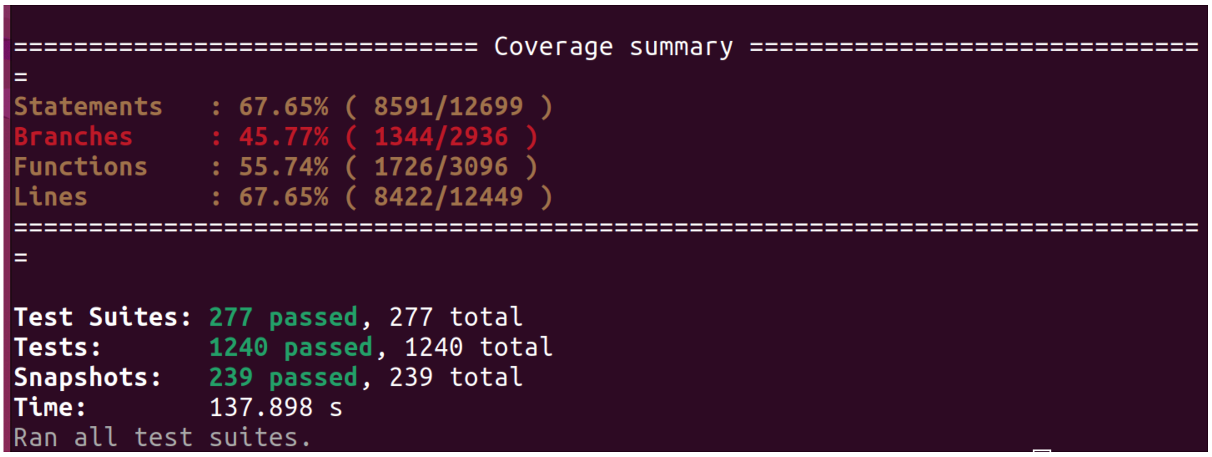
(a) Did you have to install a lot of additional tools to build the software?
Yes, we need to install : Node.js (minimum version 18) / Angular CLI version 16 and use Java JDK 11 (Alternatively, it is possible to use JDK 17, as long as newer language features are not used).

(b) Were those tools well documented?
Yes, the project is very well documented:
https://teammates.github.io/teammates/setting-up.html

(c) Were other components installed automatically by the build script?
There were prerequisites that had to be installed before building, but nothing was installed during the build.

(d) Did the build conclude automatically without errors?
Yes, by following the instructions in the onboarding documentation we were able to run both the front-end and back-end of the project without any errors.

(e) How well do examples and tests run on your system(s)?
We can run either the tests for the front-end or backend. The instruction are in the link below : https://teammates.github.io/teammates/development.html
For the front-end we got :

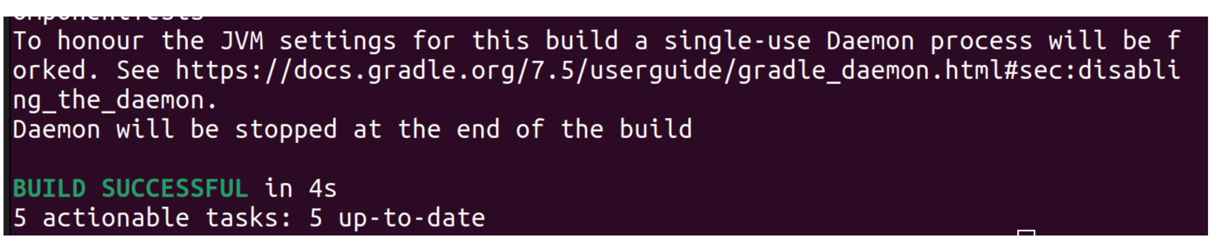


```
=============================== Coverage summary ===============================
=
Statements   : 67.65% ( 8591/12699 )
Branches     : 45.77% ( 1344/2936 )
Functions    : 55.74% ( 1726/3096 )
Lines        : 67.65% ( 8422/12449 )
================================================================================
=

Test Suites: 277 passed, 277 total
Tests:       1240 passed, 1240 total
Snapshots:   239 passed, 239 total
Time:        137.898 s
Ran all test suites.
```

For the back-end we got :

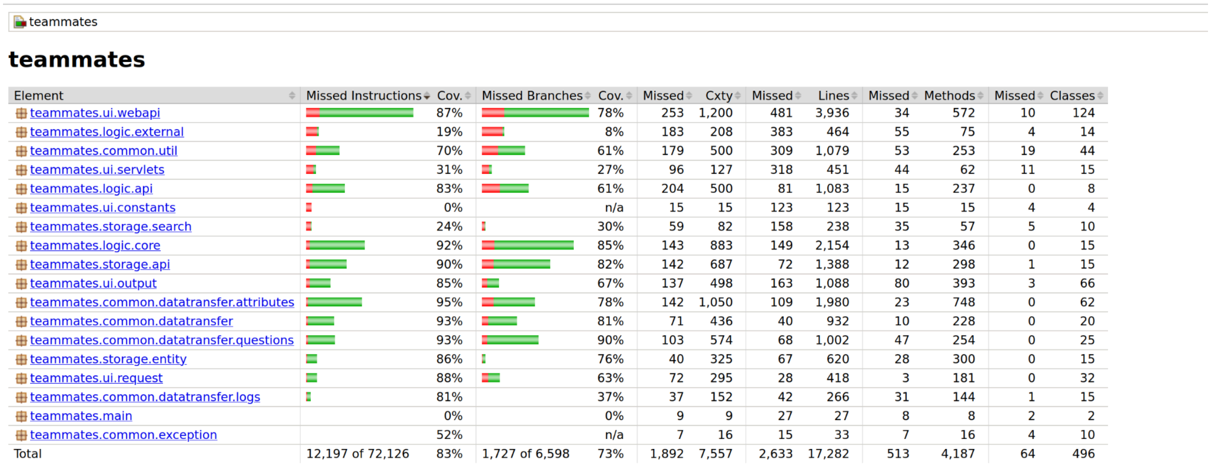
```
./gradlew componentTests jacocoReport
```



```
To honour the JVM settings for this build a single-use Daemon process will be f
orked. See https://docs.gradle.org/7.5/userguide/gradle_daemon.html#sec:disabli
ng_the_daemon.
Daemon will be stopped at the end of the build

BUILD SUCCESSFUL in 4s
5 actionable tasks: 5 up-to-date
```

```
Go to the:
~/teammates/build/reports/jacoco/jacocoReport/html/jacoco-resources
We get the index.html (that contains the coverage test):
```



teammates

teammates

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
teammates.ui.webapi		87%		78%	253	1,200	481	3,936	34	572	10	124
teammates.logic.external		19%		8%	183	208	383	464	55	75	4	14
teammates.common.util		70%		61%	179	500	309	1,079	53	253	19	44
teammates.ui.servlets		31%		27%	96	127	318	451	44	62	11	15
teammates.logic.api		83%		61%	204	500	81	1,083	15	237	0	8
teammates.ui.constants		0%		n/a	15	15	123	123	15	15	4	4
teammates.storage.search		24%		30%	59	82	158	238	35	57	5	10
teammates.logic.core		92%		85%	143	883	149	2,154	13	346	0	15
teammates.storage.api		90%		82%	142	687	72	1,388	12	298	1	15
teammates.ui.output		85%		67%	137	498	163	1,088	80	393	3	66
teammates.common.datatransfer.attributes		95%		78%	142	1,050	109	1,980	23	748	0	62
teammates.common.datatransfer		93%		81%	71	436	40	932	10	228	0	20
teammates.common.datatransfer.questions		93%		90%	103	574	68	1,002	47	254	0	25
teammates.storage.entity		86%		76%	40	325	67	620	28	300	0	15
teammates.ui.request		88%		63%	72	295	28	418	3	181	0	32
teammates.common.datatransfer.logs		81%		37%	37	152	42	266	31	144	1	15
teammates.main		0%		0%	9	9	27	27	8	8	2	2
teammates.common.exception		52%		n/a	7	16	15	33	7	16	4	10
Total	12,197 of 72,126	83%	1,727 of 6,598	73%	1,892	7,557	2,633	17,282	513	4,187	64	496

Do you plan to continue or choose another project?

We decided to continue with this project. No time to switch to another project and the documentation for this project was good.

3.4 Part 1: Complexity measurement
Run a complexity measurement tool such as lizard on the code base. If you cannot find a tool for your platform, run "wc -l" to measure the lines of code per file, and count the lines of code for some large functions using a text editor.

Count the cyclomatic complexity of five large functions in the code (not necessarily the largest ones, in case you cannot sort them automatically by size). Exclude third-party code and generated code in this. Next, count the cyclomatic complexity of these five complex functions by hand.

***Note:* This document assumes your group has five members; if your group is smaller, total numbers scale with the group size: 8 functions with four active group members, or six with three active members.**

***Note:* Use at least two group members to count the complexity separately, to get reliable results. Use a third member if the two counts differ**.

The five complex functions we chose:

1) CCN 43
FeedbackQuestionsLogic::getRecipientsOfQuestion@291-461@.\src\main\java\teammates\logic\core\FeedbackQuestionsLogic.java
2) CCN 30
FeedbackMsqQuestionDetails::validateQuestionDetails@114-198@.\src\main\java\teammates\common\datatransfer\questions\FeedbackMsqQuestionDetails.java
3) CCN 29
FeedbackQuestionAttributes::equals@236-324@.\src\main\java\teammates\common\datatransfer\attributes\FeedbackQuestionAttributes.java
4) CCN 27
LocalLoggingService::isRequestFilterSatisfied@149-202@.\src\main\java\teammates\logic\external\LocalLoggingService.java
5) CCN 26
FeedbackResponsesLogic::isResponseVisibleForUser@652-701@.\src\main\java\teammates\logic\core\FeedbackResponsesLogic.java

1. What are your results? Did everyone get the same result? Is there something that is unclear? If you have a tool, is its result the same as yours?

Function1	**Function2**	**Function3**	**Function4**	**Function5**
40	19	10	7	8
41	22	27	12	12
Tool: 43	Tool: 30	Tool: 29	Tool: 27	Tool: 26

2. Are the functions/methods with high CC also very long in terms of LOC?
Function1 - 170 LOC with CC 40/41 with tool 43
Function2 - 84 LOC with CC 19/22 with tool 30
Function3 - 88 LOC with CC 10/27 with tool 29
Function4 - 54 LOC with CC 7/12 with tool 27
Function5 - 50 LOC with CC 8/12 with tool 26

3. What is the purpose of these functions? Is it related to the high CC?

Function1 - The purpose of the getRecipientsOfQuestion method is to get a map of recipients for a given feedback question. The recipients are determined based on the type of the question and the giver where the giver is either a student or an instructor. The high cyclomatic complexity comes from the various number of recipient cases. The high CC can be managed by refactoring the getRecipientsOfQuestion into smaller, more manageable methods.

Function2 - The purpose of the validateQuestionDetails is to validate the details of a MSQ which in this context stands for multiple select questions. It checks for various through various conditions and if any of these conditions are not met, we add a corresponding error message to a list of errors. The high cyclomatic complexity comes from the various amounts of conditions that need to be checked. These checks could probably be sorted deeper, meaning that some conditions are very similar while others are not, those that share some similarity could be checked in another method.

Function3 - The purpose of equals is to determine whether two FeedbackQuestionAttributes objects are equal based on their parameters. The high cyclomatic complexity of the function comes as a trade-off for an overall more efficient solution. The complexity comes from the checking of individual parameters for equality between the two objects. These checks could be done in a single if statement, but that would require checking all parameters before returning a result. This issue could be sidestepped by using the short-circuit property of the AND operator.

Function4 - The purpose of the isRequestFilterSatisfied function is to determine whether a given log entry satisfies various filtering criteria based on different attributes such as action class, latency, status, registration key, email, and Google ID.

Function5 - The purpose of the isResponseVisibleForUser function is to determine whether a feedback response is visible to a particular user based on their role (instructor or student), the question's visibility settings, and the user's association with the response's recipient or giver. Additionally, it considers instructor permissions to determine if the response should be visible to them.

4. If your programming language uses exceptions: Are they taken into account by the tool? If you think of an exception as another possible branch (to the catch block or the end of the function), how is the CC affected?
Yes, we added an exception to function4 without changing any functionality and the CC changed from 27 to 28 with the tool. We used Codalyze as the tool.

5. Is the documentation of the function clear about the different possible outcomes induced by different branches taken?
Function1 - The documentation consisted of a very brief description of the method and a clear description of the parameters and what the method returns.

Function2 - The documentation did not contain an abstract summary of the function but the majority of the decision based statements were well documented.

Function3 - There was no documentation, but the outcome of the function was clear due to the way the code was written.

Function4 - No, there was no documentation for the function.

Function5 - No, there was no documentation for the function.

3.5 Part 2: Coverage measurement & improvement

Identify one or two code coverage tools that work for your platform and use them on your chosen project.

We used JaCoCo.

JaCoco before:
Function1 coverage - Missed branches: Cov. 84% | Missed Instructions: Cov. 86%
Function2 coverage - Missed branches: Cov. 86% | Missed Instructions: Cov. 97%
Function3 coverage - Missed branches: Cov. 44% | Missed Instructions: Cov. 65%
Function4 coverage - Missed branches: Cov. 0% | Missed Instructions: Cov. 0%
Function5 coverage - Missed branches: Cov. 100% | Missed Instructions: Cov. 100%

3.5.1 Task 1: DIY

Implement branch coverage by manual instrumentation of the source code for five functions with high cyclomatic complexity. Use a separate development branch or repo for this, as this code is not permanent. The simplest method for this is as follows:

1. Identify all branches in the given functions; assign a unique number (ID) to each one.
2. Before the program starts (before the first instruction in “main” or as the first step in the unit test harness), create data structures that hold coverage information about specific branches.
3. Before the first statement of each branch outcome (including to-be-added “else” clauses if none exist), add a line that sets a flag if the branch is reached.
4. At the end of the program (as the last instruction in “main” or at the end of all unit tests), write all information about the taken branches taken to a file or console.

Note: **It is perfectly fine if your coverage tool is inefficient (memory-wise or time-wise) and its output may be hard to read. For example, you may allocate 100 coverage measurement points to each function, so you can easily divide the entries into a fixed-size array.**

1. What is the quality of your own coverage measurement? Does it take into account ternary operators (condition ? yes : no) and exceptions, if available in your language?

No, but it could be adapted to do that.

2. What are the limitations of your tool? How would the instrumentation change if you modify the program?

The main limitation is that we have to manually add code to update the map.

3. If you have an automated tool, are your results consistent with the ones produced by existing tool(s)?

Our tool is not automated.

3.5.2 Task 2: Coverage improvement
In the five functions with high cyclomatic complexity, what is the current branch coverage? Is branch coverage higher or lower than in the rest of the code (if you have automated coverage)?
Keep a record (copy) of your coverage before you start working on new tests.

JaCoCo before new tests:
Function1 coverage - Missed branches: Cov. 84% | Missed Instructions: Cov. 86%
Function2 coverage - Missed branches: Cov. 86% | Missed Instructions: Cov. 97%
Function3 coverage - Missed branches: Cov. 44% | Missed Instructions: Cov. 65%
Function4 coverage - Missed branches: Cov. 0% | Missed Instructions: Cov. 0%
Function5 coverage - Missed branches: Cov. 100% | Missed Instructions: Cov. 100%

Our coverage tool before new tests:
Function1 coverage - 64.44%
Function2 coverage - 93.75%
Function3 coverage - 13.80%
Function4 coverage - 0%
Function5 coverage - 100%
##
JaCoCo after new tests:
Function1 coverage - Missed branches: Cov. 94% | Missed Instructions: Cov. 89%
Function2 coverage - Missed branches: Cov. 99% | Missed Instructions: Cov. 97%
Function3 coverage - Missed branches: Cov. 75% | Missed Instructions: Cov. 65%
Function4 coverage - Missed branches: Cov. 58% | Missed Instructions: Cov. 40%
Function5 coverage - Missed branches: Cov. 100% | Missed Instructions: Cov. 100%

Our coverage tool after new tests:
Function1 coverage - 77.77%
Function2 coverage - 100%
Function3 coverage - ~38%
Function4 coverage - ~71.4%
Function5 coverage - 100%

Furthermore, **make sure you add the new tests on a different branch than the one you used for coverage instrumentation**. Having identified “weak spots” in coverage, try to **improve coverage with additional test cases.**

1. Identify the requirements that are tested or untested by the given test suite.

Function1: The function contains some tests based on the questions in the database, but it covers only 64%. Additionally, some important cases were not tested. Therefore, I have added four additional tests to cover the maximum possible scenarios. For example, if a team can send a question to itself, we should test this case and see how the function deals with it. Also added another type of Student that can send questions (There are 3 types, and only one was tested).

Function2: There were a lot of existing tests that checked almost everything but every relevant test did not include any documentation. The existing tests cover 93.75% of branches.

Function3: This function should check the equality between all the parameters of two FeedbackQuestionAttributes objects. The existing tests check only for the equality of between courseId and feedbackSessionName.

Function4: There were no previous tests.

Function5: They have a lot of existing tests that checked pretty much everything we could think of, the coverage tool agrees as they had 100% coverage.

2. Document the requirements (as comments), and use them later as assertions.

3. Create new test cases as needed to improve branch coverage in the given functions. Can you call the function directly? Can you expand on existing tests? Do you have to add additional interfaces to the system (as public methods) to make it possible to set up test data structures?

Function1: I've executed the complete test suite for the function, and I can't expand from the existing test cause each one depends on different data. I had to handle various JSON files representing the database to incorporate more test cases.

Function2: I can call the function directly. I could expand on existing tests since some branches are within a combination of other branches. I did not have to add additional interfaces to the system to make it possible to set up test data structures.

Function3: I can call the function directly. I could expand on existing tests, but would prefer smaller, modular, tests. I don't need to add additional interfaces to the system.

Function4: No, we used something called reflection, as the function is private. There were no existing tests, so we created 4 tests for different branches within the function. One with several assertions.

Function5: (We created more tests for function4 instead - 100% branch coverage with tool).

4. If you have 100 % branch coverage, you can choose other functions or think about path coverage. You may cover all branches in a function, but what does this mean for the combination of branches? Consider the existing tests by hand and check how they cover the branches (in which combinations).

Function1: The function contains only 64% branch coverage, so The test increased this coverage to 77%.

Function2: This function had a very high branch coverage, 93.75% to be exact. Every branch was covered except for one. To reach this branch, there had to be a combination of one other branch. Checking how the existing tests cover the branch needed to reach the one branch not covered helped me write the test. The test I’ve written for this function covers the branch and makes the branch coverage for the method to 100%.

Function3: While the function doesn’t have 100% branch coverage, its coverage increased from 13% to 38%. Since the function usually ends as soon as the condition in most operators is met, there are very few combinations of branches. The existing tests cover basic entry points into the few branches that do appear in the function due to preliminary checks such as making sure that the compared objects are not null.

Function4 and 5: We had 100% branch coverage (function5), so instead we worked on adding more tests to function4, as that function had 0% branch coverage before. While we could have tried to expand on the existing tests (function5), they were already quite extensive and would not give us better branch coverage.

3.5.3 Task 3: Refactoring plan

Is the high complexity you identified really necessary? Is it possible to split up the code (in the five complex functions you have identified) into smaller units to reduce complexity? If so, how would you go about this? **Document your plan.**

Function1 - The function contains numerous switch and if blocks, handling various cases (SELF, STUDENTS, INSTRUCTORS, etc.), which makes the function lengthy and complex. So we can break down the large function into smaller functions, each managing specific logic. Each case in the switch (SELF, STUDENTS, INSTRUCTORS, etc.) is extracted into its own method, such as a handleSelf method for the SELF case… Also, the simplification of conditional logic in isStudentGiver and isInstructorGiver.

Function2 - Most of the complexity is necessary, because the function needs to validate each parameter of the FeedbackMsqQuestionDetails. However, this complexity can be split among smaller functions that focus on related groups of parameters. This will reduce the function's complexity and make the method easier to maintain as changes in how certain parameters are validated no longer require changing the main validation function. An example of parameters that can be checked together are the options of a question whose FeedbackParticipantType is set to None (generateOptionsFor == FeedbackParticipantType.NONE). After refactoring the tool (lizard) reported a CCN of 14.

Function3 - There are different ways one can refactor the equals method in FeedbackQuestionAttributes file. To refactor the method for better readability and making it less complex as in terms of cyclomatic complexity, would be to use a method called Objects.equals(). This method takes in two arguments and checks both arguments for their value. It firstly checks if the arguments are null or not. If both arguments are noll then it returns true, if exactly one argument is null then it returns false. However, if both arguments are non-null values than it uses the equals method to compare the arguments. In terms of readability, this would simplify the massive null and equality checks that the function consist of while also making it less complex since we handle null checks internally and automatically.

Function4 - Most of the complexity is necessary. The isRequestFilterSatisfied function is hard to understand because it has a lot of complicated parts, like checking conditions in different ways and using regular expressions. To make it easier to work with, we split it into smaller pieces. Each piece does one job, like checking if a filter is satisfied or handling regular expressions. This can make the function shorter and simpler to read. In particular, the if statement latencyFilter != null takes up about a third of the function and is quite complex, this could easily be moved to a new function which would reduce complexity and increase readability.

Function5 - We can split the checks into different groups. For example, we can handle cases where the user is an instructor separately from cases where they're not. Then, we'll extract each group of checks into its method, making it easier to understand what each part does, one for instructors and one for members for example. Finally, we can move the isRecpientSectionRestricted and isNotAllowedForInstructor to a helper method, since all they do is change the isVisableResponse to false if certain conditions are met.

3.5.4 Task 4: Self-assessment

Assess your way of working (p. 58 in the Essence standard v1.2) by evaluating the checklist on p. 60: In what state are you in? Why? What are obstacles to reach the next state? How have you improved during the course, and where is more improvement possible?

We are in the **Working Well** state. We've improved our teamwork during the previous assignments. We are familiar with the tools we use. We are familiar with our way of working. The practices we've decided, such as using issues, are done almost automatically. We are adaptable to new contexts, as we move from assignment to assignment, and we have an easier time working together than when we started. We could improve our communication, and in the far future, we could look towards a uniform suite of tools. We haven't moved to the **Retired** state as more assignments need to be completed.

Statement of Contributions:

Vlad Dobre:
- Created a plan and refactored function 2
- Created four tests for function 3
- Checked the coverage of function 3 before and after implementing new tests.
- Manually counted the cyclic complexity of functions 2 and 3.
- Assessed the way of working using Essence
- Analyzed function 3
- Aiming for **P+**

Max Israelsson:
- Created a refactoring plan for functions 4 and 5 with Johann.
- Created four tests for function 4 with Johann. One with multiple assertions.
- Checked the coverage of functions 4, and 5 before and after implementing new tests.
- Manually counted the cyclic complexity of functions 4 and 5.
- Created the coverage tool with Johann.

Johann Blörck:
- Created a refactoring plan for functions 4 and 5 with Max.
- Created four tests for function 4 with Max. One with multiple assertions.
- Checked the coverage of functions 4, and 5 before and after implementing new tests.
- Manually counted the cyclic complexity of functions 4 and 5.
- Created the coverage tool with Max.

Kristian Fatohi:
- Created a refactoring plan for functions 3.
- Created one test for function 2 and two tests for function 3.
- Checked the coverage of function 2 before and after implementing new tests.
- Manually counted the cyclic complexity of functions 1 and 2.
- Analyzed function 1 and 2.

Ilias Lachiri:

Created a plan and refactored function 1
- Created four tests for function 1
- Checked the coverage of function 1 before and after implementing new tests.
- Manually counted the cyclic complexity of functions 1 and 3.
- Analyzed function 1 and 3.
- Aiming for **P+**

Note for P+:

The additional functions and exigence are in the following repo :
https://github.com/vladdobre/coverage_tool

4 Grading criteria

4.1 Pass (P)

1. You provide a coherent account of your onboarding experience.
2. You identify five functions/methods with high complexity, and document the purpose of them, and why the complexity should be high (or not). You also manually count the complexity of five functions (on paper or as comments).
3. You clearly describe how to refactor five complex functions into smaller functions.
4. You create a working ad-hoc coverage tool that at least measures coverage of normal branches (if, while) for five functions.
5. Each group member writes at least two new tests that improve coverage, as evidenced by the coverage tool. All tests have at least one meaningful assertion (other than "assert(true)" or equivalent).
6. You provide a coherent account of your self-assessment according to the Essence standard.

4.1.1 Presentation

In the grading session, the group has to

- present some of the complex functions in the grading session,
- show complexity metrics and explain their ramifications, and
- show coverage before and after the new tests (measured by external tools and their own tool).

Furthermore, the coverage measurement implementation and some of the new test cases will be inspected.

4.2 Pass with distinction (P+)

Note: It is possible to achieve a P+ individually in this project, as long as the whole group passes the assignment. In this case, the points below apply individually. Use "Assignment #3, extra coverage" to submit your solution, and leave a comment on Canvas or in the statement of contributions.

4.2.1 3 out of these 5 points:

1. Each group member writes at least four new or enhanced unit tests (twice as many as for P).
2. You use your issue tracker and systematic commit messages to manage your project.
3. You carry out some of Task 3: you refactor at least one of the functions (per group member) with high cyclomatic complexity to reduce it by at least 35 % (for example, by splitting the functions).
4. You get a patch with new tests or a refactoring accepted. Note: the patch must be submitted by the assignment deadline, but it may be accepted later; as long as it is accepted by the end of the course, this point is counted. (Please notify us if this extra point is necessary for a P+.)
5. Something else that is extraordinary, at the discretion of the examiner.