DD2480 Assignment 3

Daglas Aitsen
David Hübinette
Benjamin Widman
Pierre Segerström

February 2026

1 Project

Name: TEAMMATES

URL: https://github.com/TEAMMATES/teammates

From their GitHub: "TEAMMATES is a free online tool for managing peer evaluations and other feedback paths of your students. It is provided as a cloud-based service for educators/students and is currently used by hundreds of universities across the world."

2 Onboarding experience

Following the instructions on getting started with development was straightforward and for us no new tools needed to be installed to build the software. We focus on the backend since it is written in Java and uses Gradle like we are used to. Gradle automatically got all dependencies and built without issue. All tests passed.

3 Complexity

1. ***What are your results for five complex functions? Did all methods (tools vs. manual count) get the same result? Are the results clear?***

   We look at functions inside `src/main/java/teammates/`. Cyclomatic complexity of some functions, reported by `lizard`:

   (a) `logic/api/EmailGenerator::generateFeedbackSessionSummaryOfCourse`: 23

   (b) `logic/core/FeedbackQuestionsLogic::getRecipientsOfQuestion`: 43

   (c) `logic/core/FeedbackQuestionsLogic::populateFieldsToGenerateInQuestion`: 25

   (d) `logic/external/LocalLoggingService::isRequestFilterSatisfied`: 27

   Manual counts largely matched these numbers, but were slightly lower in all cases, probably because we counted slightly differently than `lizard`. It is worth noting that there is ambiguity around what should count towards the cyclomatic complexity, and in which cases. Additionally, our slightly mismatching results could also be due to the difficulty of keeping track of such a large number of branches.

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

   They are mainly long with a lot of branches after each other, but in some places there is greater complexity due to the deep nesting.

3. ***What is the purpose of the functions?***

   (a) `logic/api/EmailGenerator::generateFeedbackSessionSummaryOfCourse`: Generate an email being sent to a user, containing a summary of a feedback session in a course.

   (b) `logic/core/FeedbackQuestionsLogic::getRecipientsOfQuestion`: Getting all users who should be receiving a certain feedback question.

   (c) `logic/core/FeedbackQuestionsLogic::populateFieldsToGenerateInQuestion`: Filling in fields in a feedback question dynamically, not by user.

   (d) `logic/external/LocalLoggingService::isRequestFilterSatisfied`: Checks if logs satisfy the filter input by user or not.

The functions are performing quite complex things, which might be the reason behind the high CC.

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

   Yes, exceptions are taken into account by `lizard` since each try-catch block increases the CC. However, one could also count each point where an exception can be thrown as another branch so if a lot of functions throw exceptions the code complexity increases by a lot.

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

   No, the documentation only describes the general expected output. With this high code complexity it would already require a lot of documentation for the same function just to describe the outcome of different branches. The last function did not even have any documentation, probably due to being non-public.

4 Refactoring

Here, we describe our plan for refactoring each function to reduce the cyclomatic complexity. This will result in more functions, but also in more readable and more testable code. To see the patch of performed refactorings:

```
git diff master origin/refactoring
```

4.1 Function 1

`EmailGenerator::generateFeedbackSessionSummaryOfCourse`

The complexity of this function is not necessary. I feel like it handles many responsibilities in one place, thus reducing the readability a lot. The method performs several distinct steps: validating the email type, determining whether the recipient is a student or instructor, building the join section of the email, building the links section of the email, and finally assembling the full email. These could be easily divided into smaller functions, and I feel as if it makes sense to do so as well.

One method could focus on loading the recipient and gathering data. Another could generate the join section, while another builds the link section. The final step would simply combine all prepared fragments into the email template and return it. This would also be easier to test, since you could create atomic tests for each section of the email.

4.2 Function 2

`FeedbackQuestionsLogic::getRecipientsOfQuestion`

The high complexity is not necessary for this function. The function contains a massive switch case with many nested if statements. For readability and to write better tests the separate cases could be refactored into private getter functions such as getStudentRecipients() and getInstructorRecipients() that are called from the main function. This would allow for simpler testing of the function and a reduction in cyclic complexity from around 45 to around 10. Another change could be to write a private function for the giver information which is now an if statement that is not necessary for this function.

4.3 Function 3

`FeedbackQuestionsLogic::populateFieldsToGenerateInQuestion`

The high complexity is not necessary for this function. To refactor this function, the first evident repetition is the type-checking at the start and end by creating a helper abstraction (such as an interface), which unifies the retrieval and saving of option lists for both `MCQ` and `MSQ` types. After that, what remains in this function is the massive switch statement. To make this more readable and testable, the logic for each "case" can be extracted into separate private methods, for example: `generateStudentOptions` and `generateTeamOptions`. These functions should focus solely on retrieving and filtering data, removing that responsibility from the main flow of the function. Consequently, the main function `populateFieldsToGenerateInQuestion` would then only serve as a higher-level orchestrator, which delegates tasks to smaller functions.

4.4 Function 4

`LocalLoggingService::isRequestFilterSatisfied`

The high complexity is not necessary for this function. Some of the complexity comes from pretty simple if statements doing early returns, but one of the biggest if statements contributes considerably to the complexity since it has two additional if statements inside and a big switch statement, which is where the check on the input latency filter is done. One refactor one could do is to break out the entire inside of the outer if statement into it's own method. However, that does not decrease the complexity enough for it to be a 35% reduction of the method. Instead, one could also include the checks on the action class filter and the status filter as well to then have a method essentially checking all of the filters related to the log details.

This refactoring was also carried out and took the checks on `actionsClassFilter`, `statusFilter` and `latencyFilter` and split them into a new method called `isLogDetailFilterSatisfied`. This brough the cyclomatic complexity of `isRequestFilterSatisfied` from 27 down to 17, which is a 37% reduction.

5 Coverage

5.1 Tools

We decided to use Jacoco as the coverage tool. The documentation was clear and easy to understand, although we did not have to go through it in depth. Due to the fact that TEAMMATES had already integrated Jacoco into the project, we simply had to run ./gradlew jacocoReport to generate a full report that was easy to search for or specific functions.

5.2 Your own coverage tool

The git command that is used to obtain the patch:

```
git diff master origin/development
```

Our DIY tool consists of:

- Adding a static boolean array to the class being measured
- Assigning each index in the array to represent a specific branch.
- Setting the corresponding index to true whenever execution reached that branch.
- Printing the coverage results after tests finished using an @AfterClass method in the test class.

Since we are manually instrumenting the code, our DIY tool can handle if/else statements, exceptions, and loop entry points. It can handle ternary operators but they have to be manually instrumented into if/else statements.

5.3 Evaluation

1. ***How detailed is your coverage measurement?***

   The level of detail of the coverage measurement depends fully on the instrumentation, since each instrumented branch is assigned a unique ID and the tool measures if the individual branches were executed during testing.

2. ***What are the limitations of your own tool?***

   The obvious limitation is that we have to manually apply each "flag", which is error-prone and possibly inconsistent depending on the developer who placed them. In addition, the solution is extremely non-flexible, since changes to the code means that one needs to place new flags, but also re-index all existing flags to maintain the ordering.

3. ***Are the results of your tool consistent with existing coverage tools?***

   We found that our tool covered the same branches as the tool we used for reporting the CC (Jacoco), i.e. our "uncovered" flags were also the only ones that were "red" in the Jacoco report. However, when you look at the branch coverage ratios, they were slightly higher in the Jacoco report. This means that Jacoco's internal calculation of the coverage ratio differs slightly from ours, even though our DIY solution reports the same "misses". Jacoco has a second nuance to the branch coverage, where they report "partially covered" branches, which most likely has an effect on their numbers being slightly higher.

6 Coverage improvement

Show the comments that describe the requirements for the coverage.

`git diff master origin/testing`

6.1 Function 1

`EmailGenerator::generateFeedbackSessionSummaryOfCourse`

Report of old coverage: **93.75%**
Report of new coverage: **97.92%**

Test cases added:

- testGenerateFeedbackSessionSummaryOfCourse_sessionNotOpenedYet
- testGenerateFeedbackSessionSummaryOfCourse_invalidEmailType

6.2 Function 2

`FeedbackQuestionsLogic::getRecipientsOfQuestion`

Report of old coverage: **72%**
Report of new coverage: **74%**

Test cases added:

- testGetRecipientsOfQuestion_instructorWithoutPrivilege_studentsCase
- testGetRecipientsOfQuestion_instructorWithoutPrivilege_teamsCase

6.3 Function 3

`FeedbackQuestionsLogic::populateFieldsToGenerateInQuestion`

Report of old coverage: **90%**
Report of new coverage: **96%**

Test cases added (for P):

- testPopulateFieldsToGenerateInQuestion_otherQuestionType_doesNothing
- testPopulateFieldsToGenerateInQuestion_invalidGenerateOptionsFor_throwsAssertionError

Pierre/pise@kth.se – Two more test cases added/enhanced (for P+):

- **New**:
  testPopulateFieldsToGenerateInQuestion_courseDisappearedForTeams _throwsAssertionError
- **Enhanced**:
  testPopulateFieldsToGenerateInQuestion_mcqQuestionDifferentGenerateOptions _shouldPopulateCorrectly

6.4 Function 4

`LocalLoggingService::isRequestFilterSatisfied`

Report of old coverage: **0%**
Report of new coverage: **44%**

Test cases added:

- testDefaultLogsContainStatus200
- testDefaultLogsContainQueryLogsAction
- testDefaultLogsContainAboveZeroLatency
- testDefaultLogsDoesNotContainNegativeLatency

7 Self-assessment: Way of working

Current state according to the Essence standard:
"**In place**"

7.1 Checklists

Principles Established:

- ✓ Principles and constraints are committed to by the team.
- ✓ Principles and constraints are agreed to by the stakeholders.
- ✓ The tool needs of the work and its stakeholders are agreed.
- ✓ A recommendation for the approach to be taken is available.
- ✓ The context within which the team will operate is understood.
- ✓ The constraints that apply to the selection, acquisition, and use of practices and tools are known.

Foundation Established:

- ✓ The key practices and tools that form the foundation of the way-of-working are selected.
- ✓ Enough practices for work to start are agreed to by the team.
- ✓ All non-negotiable practices and tools have been identified.
- ✓ The gaps that exist between the practices and tools that are needed and the practices and tools that are available have been analyzed and understood.
- ✓ The capability gaps that exist between what is needed to execute the desired way of working and the capability levels of the team have been analyzed and understood.
- ✓ The selected practices and tools have been integrated to form a usable way-of-working.

In Use:

- ✓ The practices and tools are being used to do real work.
- ✓ The use of the practices and tools selected are regularly inspected.
- ✓ The practices and tools are being adapted to the team's context.
- ✓ The use of the practices and tools is supported by the team.
- ✓ Procedures are in place to handle feedback on the team's way of working.
- ✓ The practices and tools support team communication and collaboration.

In Place:

- ✓ The practices and tools are being used by the whole team to perform their work.
- ✓ All team members have access to the practices and tools required to do their work.
- ✓ The whole team is involved in the inspection and adaptation of the way-of-working.

Working Well:

- ✓ Team members are making progress as planned by using and adapting the way-of-working to suit their current context.
- ✓ The team naturally applies the practices without thinking about them.
- ✓ The tools naturally support the way that the team works.
- ☐ The team continually tunes their use of the practices and tools.

7.2 Reflection

The last time we evaluated our ways-of-working, we assessed them to be "Foundation Established", the second stage. In our initial evaluation, we identified two critical gaps that prevented us from progressing: the lack of regular inspection of our practices and the need for better tool-supported collaboration. To bridge these gaps, we proposed some suggestions. First, we aimed to shift from manual convention-following to automated verification by leveraging automation workflows on GitHub, reducing human error and overhead. Second, we committed to a structured feedback loop in which we would reflect at the end of each assignment. This allowed us to gather actionable insights throughout a work cycle and apply them to the next.

Since that last review, we have successfully integrated those suggestions into our daily work, moving us through the "In Use" and "In Place" stages. Our workflows have become natural to use, utilizing them without thinking. The team is now able to work at different times of the day, but also simultaneously, in order to collaborate towards our common goals. This allows us to stay aligned on common goals regardless of individual schedules. By actively reflecting between assignments, we have transitioned from merely having a set of rules to possessing a functional, team-wide "Way-of-Working" that genuinely supports our productivity.

While we have achieved the majority of the SEMAT criteria, the final hurdle to reaching the "Working Well" state is the requirement: "the team continually tunes their use of the practices and tools". Currently, while we are consistent, we need to become more proactive in critically assessing our established norms. To fulfill this, we need to collectively assess our practices and tools in a critical way, in order to find blind spots and possible improvements to our everyday work. Including this to our ways of working will set us up well for the “Working Well“ stage.

8 Overall experience

Getting up and running with a new project was in this case, and is probably often, not very difficult. The most difficult thing is understanding the new system enough so that you are able to make your contribution. Projects of this size have many different moving parts and when you are making a change or addition somewhere it is likely you need to figure out how code from many different files interact with each other, and that can feel frustrating when you simply want to start writing code. This is probably one of the many reasons some open source projects struggle to get new contributors, due to the high friction to start contributing. On top of understanding the system itself the contributor also needs to understand the practices used, like naming schemes for branches and commits and tools used for linting and formatting. However, we did not submit our patches to the project so we did not need to adhere to these.