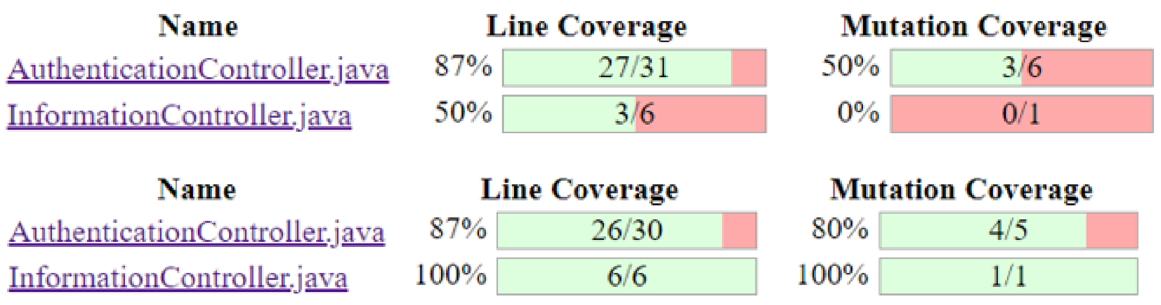
Assignment 3 Group 23a

After exploring different tools, PiTest was chosen as our mutation testing tool, because it provides detailed feedback on the possible mutants. Having generated the report from PiTest, a number of classes with less than 70% mutation coverage have been identified and additional test cases have been created in order to kill the mutants that were able to survive the previous test suite.

Automated Mutation Testing

One of the first classes with a low mutation score was the `AuthenticationController` inside the `User` microservice. The issue with it was that it wasn't tested whether the result was not just `null` and that it actually returned something. A method confirming that the result was a `ResponseEntity` was added to the test suite.

The second class whose mutation score was improved was the `AuthenticationController` also in the `User` microservice. One of its mutants was due to a print statement inside the code which was used during production for debugging purposes. The other mutant was again a possible `null` being returned by its register method. This was again handled quickly by mocking its dependencies and registering a user. In the photos below the changes in mutation coverage for these two classes can be observed.
The commit for both classes.
https://gitlab.ewi.tudelft.nl/cse2115/2022-2023/SEM23a/-/merge_requests/52/commits

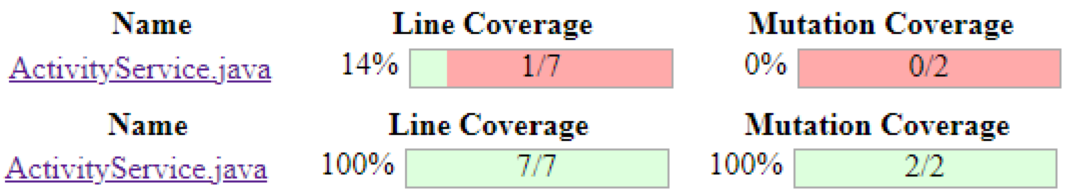


Name	Line Coverage		Mutation Coverage	
AuthenticationController.java	87%	27/31	50%	3/6
InformationController.java	50%	3/6	0%	0/1

Name	Line Coverage		Mutation Coverage	
AuthenticationController.java	87%	26/30	80%	4/5
InformationController.java	100%	6/6	100%	1/1

`ActivityService` in `HOA` microservice
The service had problems with `getActivities(...)` which didn't have tests. There were 2 mutants that weren't covered: the return value can be replaced with null and the authentication header setting might be removed, which is a crucial part of the http request. Both of these mutants were then covered, ensuring that the correct request will be sent.

Related commit:
https://gitlab.ewi.tudelft.nl/cse2115/2022-2023/SEM23a/-/commit/3f82c0e1fbdd52000a34b9cd8b9aa9e20164a2f1

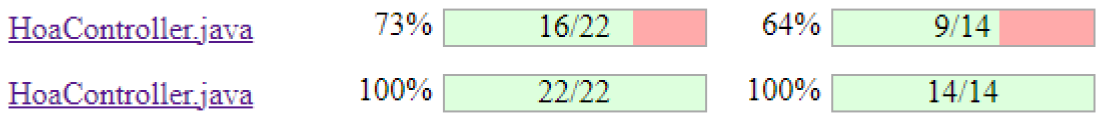


Name	Line Coverage		Mutation Coverage	
ActivityService.java	14%	1/7	0%	0/2

Name	Line Coverage		Mutation Coverage	
ActivityService.java	100%	7/7	100%	2/2

`HoaController` in `HOA` microservice
The main problems of the controller were the `getNoticeBoard(...)` and `getHoaHistory(...)` methods. They were missing tests such as the return is not null, as well as negated conditions not changing anything. Both methods were then properly tested, covering all branches and missing mutation tests.

Related commit:
https://gitlab.ewi.tudelft.nl/cse2115/2022-2023/SEM23a/-/commit/1040e94f5c140fa97b3371807f267020d4241db7



HoaController.java	73%	16/22	64%	9/14
HoaController.java	100%	22/22	100%	14/14

Manual Mutation Testing

When it came down to the manual mutation testing, the focus was on the `activity-and-election` microservice, as it was one of the less-tested packages.

Gathering

This class is responsible for all `Gathering` objects. It extends `Activity` and is one of the four basic activities for our platform and is of great importance when it comes to the users. Each user must be able to host Gatherings and post them on the NoticeBoard. This is one of the most basic functionalities of a homeowners' association and is a core requirement for the project.

The method where the mutant is introduced is `getVotingStrategy()` which is responsible for getting the strategy that is going to be used for counting the reactions people had to the gathering. This is key for the proper functioning of the activities. The mutant was having the method return an empty/null strategy. Strategies can change in the future so we shouldn't assert a specific strategy as this test might have to be changed very soon, rather we need to affirm that there is an actual strategy being returned. The test case handles exactly that, testing if the strategy is not null.

GatheringVotingStrategy

This is one of the classes concerning the strategy design pattern for voting that was not tested and hence lacked mutation testing. This is very important because if there are any mistakes in it, people might not be able to get accurate results for their activities. For example, how many people are going or not going, so the host can prepare accordingly.

The mutant injected was in the `getResult()` method inside the class. This method is crucial to the functionality of the reaction systems, as there would be no point in posting a gathering and reacting to it if one could not retrieve the results. The injected mutant was having the method return a `null` value. The test that kills this mutant is the following: first a gathering is created. Then, this activity is voted upon and a collection of the results is conducted. Finally, the test asserts that the `List<String>` returned has the expected values in the expected order. Without this test when running the test suite of our problem, no issues would have occurred but now we will catch if, during refactoring in the future, something happens to this method.

ActivityService

This class has not really been under close inspection when it came down to mutation testing, this is why now it is going to be tested. Its purpose is retrieving and saving activities from the database. The reason why this is regarded as an important point in the project is that without proper communication with the database, this class will be rendered useless.

The method that will be injected with the mutant is `getActivityById()`, which fetches an activity from the database. This method is of utmost importance since it deals with fetching information about a certain activity. There were two mutants that were injected. The first was inside the if-case, by negating the condition. The second was having the method return an empty/null object. This was fixed in two test cases. The first one asserts the behaviour of the if clause. The second one asserts that the returned object is a non-empty `ActivityModel` and thus, guarantees that the method will never return null.

Election

This class is one of the four main types of activities that can be hosted in an HOA. Without elections, homeowners' associations will not have boards and people who vote for rules which would make the HOA quite pointless. This is why this class is important.

The mutant was injected inside the `getVotingStrategy()` method which was responsible for getting the correct voting strategy. This method is essential since every election in the world needs a strategy on which to count the votes. The mutant was having the method return a null object, hence no strategy. The test case killing this mutant handled exactly that.

Another mutant was injected into this class when testing the method `calculateTimestampShift()`. This method calculated how many days in the future this election is going to be held. Having the end date of an election is necessary because people need to know when they can vote, so as to not miss the election. This method was set to private, but for the purposes of testing, it is now changed to public. The mutant was changing the division into multiplication, as this is an easy mistake to make. The new test case handles this.

https://gitlab.ewi.tudelft.nl/cse2115/2022-2023/SEM23a/-/merge_requests/53/commits