

SEM Assignment 3 - Testing

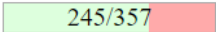
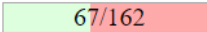
Group 18b

January 25, 2023

1 Automated Mutation Testing

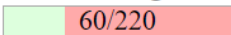
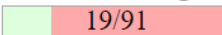
The main domain chosen to perform additional testing on was the HOA in the HOA microservice. Additionally, there was a class we determined needed testing in the Voting microservice/domain too. A tool used for aiding this process was the PITest library. It was run on all of the modules in the application and it was determined that the microservices that would benefit the most from being examined for mutants would be the HOA and Voting microservices. This attributes itself to the large amount of business logic contained in them. The HOA microservice started with an initial 69% line coverage and a 41% mutant coverage. After thorough testing, the result was brought to (insert here). The Voting microservice's initial results were 27% line coverage and 21% mutation coverage and are pictured below. Despite the seemingly larger amount of uncovered lines and lower mutation percentage in Voting, the team still decided to focus on the HOA microservice. This is due to the sensitive voting and the importance of votes being correctly counted. The team wanted thorough inspection of it so it was left mainly to the manual mutation testing section of the assignment.

Project Summary

Number of Classes	Line Coverage	Mutation Coverage
19	69% 	41% 

Initial Score By PITest on the Entire HOA Microservice

Project Summary

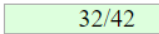
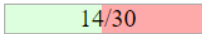
Number of Classes	Line Coverage	Mutation Coverage
12	27% 	21% 

Initial Score By PITest on the Entire Voting Microservice

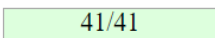
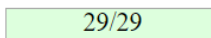
1.1 HOA Microservice - UserService

The class holds the responsibilities for verifying HOA memberships and privileges. As a result, it is quite an important class with functionalities relating to different homeowner associations and board memberships. Running PITest on it, it was discovered that the class not only had incomplete line coverage of 76%, but also a worrying 53% of mutants surviving. Taking a closer look

at the tests, it was seen that some tests were not asserting correct behavior and were implementing incorrect mocking principles (such as mocking the class it is testing). They failed to catch certain behaviors of the methods tested, despite passing. After those faults in the testing were found, correct behavior was asserted through instead mocking the dependencies (different services) of the different methods. Additionally, the testing helped in identifying faults in the codebase, such as in the `findByDisplayName(...)` method, which was not explicitly accounting for the case of a user not existing. The team is pleased to say that the rest of the code in the class was clean, just insufficiently tested at times. The report from PITest is visible below. The commit responsible for this class' tests is this.

[UserService.java](#) 76%  47% 

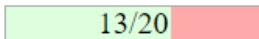
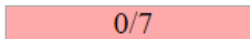
Initial Score By PITest - Line Coverage (Left), Mutant Coverage(Right)

[UserService.java](#) 100%  100% 

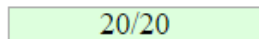
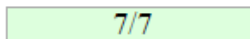
Score By PITest After Mutation Testing - Line Coverage (Left), Mutant Coverage(Right)

1.2 HOA Microservice - User

This is the entity for the database responsible for storing the users in the database. The mutation coverage prior to this assignment was 0%. The test classes responsible for any prior test coverage are `ActivityTest`, `UserServiceTest`, `PnbControllerTest`. Improving the coverage was done using unit tests, and with their help, a mutation coverage of 100% was achieved. A majority of the mutations involved replacing what is returned with special values (like `null`), but there is also some logic relating to a user's membership in a HOA and logic related to the creation of data transfer objects. The implemented tests ensured that all those mutants were killed. Link to the commit with the introduced tests

[User.java](#) 65%  0% 

Score By PITest Before Mutation Testing - Line Coverage (Left), Mutant Coverage(Right)

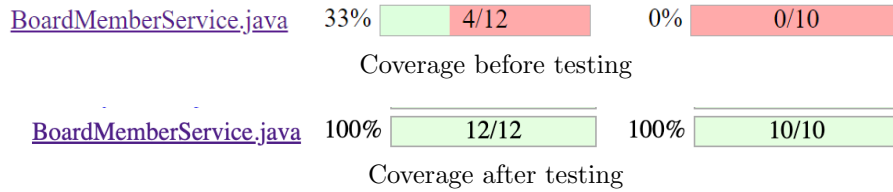
[User.java](#) 100%  100% 

Score By PITest After Mutation Testing - Line Coverage (Left), Mutant Coverage(Right)

1.3 HOA Microservice - BoardMemberService

The `BoardMemberService` was a largely untested class prior to this assignment, sitting with a mere 0% mutation coverage, and only 33% line coverage in general (which was also useless as only the constructor was "tested". It was probably ignored due to it being mostly a data access class, not containing any meaningful business logic. Looking at the results of PITest confirmed that we were wrong about assuming that there could be no possible issues in this code.

Most of the mutants were changing return statements to either literal values, like changing a Boolean return value to either true or false, making the method completely useless, and potentially breaking any code we had elsewhere. Another common mutant was returning empty lists, resulting in similar issues to the boolean one. Writing tests raised both the line and the mutation coverage to 100%



1.4 HOA Microservice - Results

As a result from the HOA microservice's extensive testing, it achieved a substantial improvement in the PITest score. The classes that were not tested for mutations were classes whose edge cases were handled by Spring making them not as high-priority for testing. Results are pictured below:

Project Summary

Number of Classes	Line Coverage	Mutation Coverage
19	74% <div><div style="width: 74%;"></div></div> 262/356	61% <div><div style="width: 61%;"></div></div> 99/161

Final Mutation Results HOA

1.5 Voting Microservice - ConcreteTimeKeeper

The class is crucial to the voting microservice due to the fact that it allows for the timely counting of the votes and permits the windowing of an election. It is an important implementation that a low of classes in the microservice depend on and its testing was insufficient. The commit that was responsible for its testing was this.

Name	Line Coverage	Mutation Coverage
ConcreteTimeKeeper.java	70% <div><div style="width: 70%;"></div></div> 7/10	43% <div><div style="width: 43%;"></div></div> 3/7
Score By PITest Before Mutation Testing		
Name	Line Coverage	Mutation Coverage
ConcreteTimeKeeper.java	100% <div><div style="width: 100%;"></div></div> 10/10	86% <div><div style="width: 86%;"></div></div> 6/7
Score By PITest After Mutation Testing		

Addendum: There is no collated results section for the Voting microservice mutation testing since further refactorings (without the aid of PITest) have been discussed later in the second part of the assignment.

2 Manual Mutation Testing Domain

For this assignment, we had to choose a core domain and four critical classes. We went with the "Voting" code domain. Our choice is motivated by the importance of the functionality of the microservice, so the team wanted to test it manually. The chosen classes, methods, and their justifications are the following:

2.1 `VoteController - initializeVoting(VotingModel votingModel)`

This class contains all of the API endpoints related to the voting functionality. Without it, other microservices would have no way to access the functionality this microservice offers. The `initializeVoting` method is called whenever a user wants to initialize a new voting procedure. Of course, without doing this, no vote could be cast at all.

2.2 `VotingService - getOptions(int hoaId)`

The `VotingService` class is responsible for all of the general business logic in the microservice. That is, once a request is initially processed by the `VoteController`, fulfilling the request is usually orchestrated by this class. Even though the naming of the `getOptions` method suggests that it is a simple getter, it actually has logic that is a bit more complex. It looks up the appropriate voting procedure based on the HOA ID provided, and returns the options that a user can choose from when voting.

2.3 `RequirementVoteBuilder - build()`

This class is responsible for building a voting procedure about requirements in an HOA, with the given properties. As the name suggests, it uses the builder design pattern to achieve this. The `build` method returns a new voting procedure with the parameters given previously.

2.4 `RequirementResultsCollator - collateResults(...)`

Once everyone has voted, the votes need to be counted and a decision has to be made. This is the task of the `RequirementResultsCollator` class and specifically the `collateResults` method. The method returns an object with the decision.

3 The Manual Mutations

3.1 `initializeVoting - negate if (!electionOngoing)`

One check done in the code for the method is whether or not an election is already ongoing for the specified HOA. If so, a chunk of code is skipped and instead a response with status `SEE OTHER` is returned. When ignoring this if condition, multiple board elections could theoretically take place at the same time which is not desired behaviour. To kill this mutant, we made sure to implement a test that attempted to start the same election twice and verified if the proper error status code would be returned.

3.2 `getOptions - negate if (requestedVote != null)`

The `getOptions` method retrieves the candidates or options of an election or vote. If the method is called for an election that does not exist, a null check ensures that a `NullPointerException` is avoided. When this check is ignored, a null value could potentially be returned which would crash the entire application. To kill this mutant, we wrote a test that attempted to retrieve the options of a vote which had no defined options and asserted that the method would return an empty list instead of a null value.

3.3 build - negate if (options == null)

In case not all necessary fields are set in the builder, it is a useful feature to automatically set these fields to something that is sensible by default. However, getting the boolean comparison may result in some fields - ones that *have* been set - to be "forgotten". The test we wrote checks whether the options field provided to the builder actually ends up in the voting instance we built.

3.4 collateResults - flip ">" when evaluating passed

When evaluating vote results, it is quite crucial to get the final verdict right. In the current implementation of this, the line in which this is determined, is quite difficult to read:

```
boolean passed = aggregatedResults.get(1) > aggregatedResults.get(0);
```

Simply flipping the symbol in the middle (which is an easy mistake to make) would yield a wrong result most of the time. Testing this was easy, we just cast two "for" votes and one "against", then asserted that the vote passed.