

Assignment 3 - Mutation Testing

Quick note: All our commits concerning this assignment were merged into a development branch unique to this assignment. This means that all the commits can be viewed by finding the merge request marked with the label “hey TA look here”.

Part 1

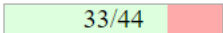
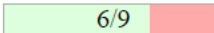
UserController (In user microservice)

I chose UserController which is a class in user microservice for the mutation test improvement in task 1. Before the improvement, the mutation coverage of this class is 67%. (as shown in the picture below)

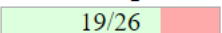
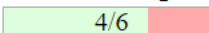
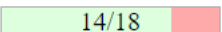
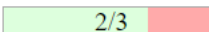
Pit Test Coverage Report

Package Summary

nl.tudelft.sem.template.user.controllers

Number of Classes	Line Coverage	Mutation Coverage
2	75% 	67% 

Breakdown by Class

Name	Line Coverage	Mutation Coverage
NotificationController.java	73% 	67% 
UserController.java	78% 	67% 

Report generated by [PIT](#) 1.5.1

I improved this by adding a test that asserts the exception thrown, to make sure this will kill the mutation. After this improvement, the new mutation coverage becomes 100% and the line coverage becomes 89%. (as shown in the picture below)

Pit Test Coverage Report

Package Summary

nl.tudelft.sem.template.user.controllers

Number of Classes	Line Coverage	Mutation Coverage
2	80% <div><div>35/44</div></div>	78% <div><div>7/9</div></div>

Breakdown by Class

Name	Line Coverage	Mutation Coverage
NotificationController.java	73% <div><div>19/26</div></div>	67% <div><div>4/6</div></div>
UserController.java	89% <div><div>16/18</div></div>	100% <div><div>3/3</div></div>

TrainingControllerEdit (In activity microservice)

I wanted to add tests for the TrainingControllerEdit class, in order to improve its mutation coverage for task 1. Before the improvement, the mutation coverage was 33% only and the line coverage was 50%:

Pit Test Coverage Report

Package Summary

nl.tudelft.sem.template.activity.controllers

Number of Classes	Line Coverage	Mutation Coverage
4	76% <div><div>50/66</div></div>	74% <div><div>17/23</div></div>

Breakdown by Class

Name	Line Coverage	Mutation Coverage
CompetitionController.java	81% <div><div>25/31</div></div>	83% <div><div>10/12</div></div>
TrainingControllerCreate.java	88% <div><div>14/16</div></div>	83% <div><div>5/6</div></div>
TrainingControllerEdit.java	50% <div><div>6/12</div></div>	33% <div><div>1/3</div></div>
TrainingControllerUser.java	71% <div><div>5/7</div></div>	50% <div><div>1/2</div></div>

Report generated by [PIT](#) 1.5.1

In order to improve this, I decided to add a new test class to test this specific class. The TrainingControllerEdit class has 2 methods, joinTraining (which doesn't throw an exception, as it handles it inside the method) and getTrainings (which actually throws an exception). In order to

maximize the score for both mutation and line coverages, we need to check whether the methods handle the exceptions properly. For this, I used `thenThrow`, in order to be able to throw exceptions. The new test class (`TrainingControllerEditTest`) has 3 methods which test the functionality of the original class (2 of which throw exceptions), which maximizes the coverage for both mutations and lines:

Pit Test Coverage Report

Package Summary

nl.tudelft.sem.template.activity.controllers

Number of Classes	Line Coverage	Mutation Coverage
4	85% <div><div>56/66</div></div>	83% <div><div>19/23</div></div>

Breakdown by Class

Name	Line Coverage	Mutation Coverage
CompetitionController.java	81% <div><div>25/31</div></div>	83% <div><div>10/12</div></div>
TrainingControllerCreate.java	88% <div><div>14/16</div></div>	83% <div><div>5/6</div></div>
TrainingControllerEdit.java	100% <div><div>12/12</div></div>	100% <div><div>3/3</div></div>
TrainingControllerUser.java	71% <div><div>5/7</div></div>	50% <div><div>1/2</div></div>

Report generated by [PIT](#) 1.5.1

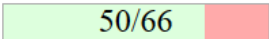
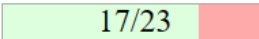
TrainingControllerUser (In activity microservice)

Similar to the `TrainingControllerEdit` class, this could also be improved, as it only has 50% mutation testing coverage and 71% line coverage:

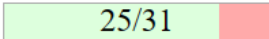
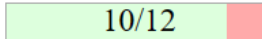
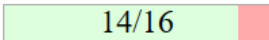
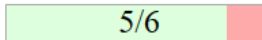
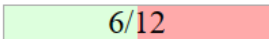
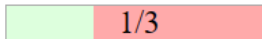
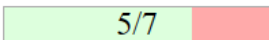
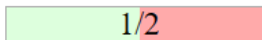
Pit Test Coverage Report

Package Summary

nl.tudelft.sem.template.activity.controllers

Number of Classes	Line Coverage	Mutation Coverage
4	76% 	74% 

Breakdown by Class

Name	Line Coverage	Mutation Coverage
CompetitionController.java	81% 	83% 
TrainingControllerCreate.java	88% 	83% 
TrainingControllerEdit.java	50% 	33% 
TrainingControllerUser.java	71% 	50% 

Report generated by [PIT](#) 1.5.1

Again, I found out that an exception is thrown but not covered in the tests. Firstly, I created a new class, whose aim was to test this class (TrainingControllerUserTest). I created a method to properly the method (informUser), without any exceptions being thrown. When I tried to throw an exception though, Pitest was giving an error regarding the fact that it cannot throw an exception. In order to find out why, I looked into the informUser methods inside the TrainingServiceUserSide and ActivityService classes. An exception couldn't be thrown because there was no exception thrown inside these methods in the first place. In the TrainingServiceUserSide class there is clearly no need for an exception (as the method has only one line of code). Inside the ActivityService, rather than throwing an exception, the following string is returned: "Could not find activity" (line 46). This design choice, of course, was to avoid exceptions. Thus, instead of throwing an exception in the informUser in the TrainingControllerUser class, I decided to simply remove that code, as there wasn't the need for any exception handling in the first place (and leave the test class as I initially implemented it). After the modifications, these are the improvements (note that the screenshot was taken after the modifications in TrainingControllerEdit as well):

Pit Test Coverage Report

Package Summary

nl.tudelft.sem.template.activity.controllers

Number of Classes	Line Coverage	Mutation Coverage
4	88% 56/64	86% 19/22

Breakdown by Class

Name	Line Coverage	Mutation Coverage
CompetitionController.java	81% 25/31	83% 10/12
TrainingControllerCreate.java	88% 14/16	83% 5/6
TrainingControllerEdit.java	100% 12/12	100% 3/3
TrainingControllerUser.java	100% 5/5	100% 1/1

Report generated by [PIT](#) 1.5.1

The coverages for the TrainingControllerUser class are maximized.

BoatRestService (In activity microservice)

The BoatRestService class started at only 39% line coverage and 0% mutation coverage, being one of the classes with the lowest test scores in the entire project:

Pit Test Coverage Report

Package Summary

nl.tudelft.sem.template.activity.domain.services

Number of Classes	Line Coverage	Mutation Coverage
9	74% 197/268	68% 96/142

Breakdown by Class

Name	Line Coverage	Mutation Coverage
ActivityService.java	100% 21/21	92% 11/12
BoatRestService.java	39% 7/18	0% 0/6
CompetitionServiceServerSide.java	96% 45/47	85% 17/20
CompetitionServiceUserSide.java	63% 32/51	59% 26/44
RestService.java	0% 0/11	0% 0/2
RestServiceFacade.java	67% 4/6	0% 0/2
TrainingServiceServerSide.java	94% 34/36	88% 15/17
TrainingServiceUserSide.java	76% 47/62	79% 27/34
UserRestService.java	44% 7/16	0% 0/5

After running the pitest tool on this class, the results showed that all of its methods, including the constructor had mutants that the current test suite could not kill. This is when it became clear that the test suite did not include a test class specific for it, so to begin with it was necessary to create such a test class, which was named BoatRestServiceTest. The convoluted nature of the class made it a hassle to create reliable tests to kill the mutants that could be created for the deserialize() and performBoatRequest() methods in it. However, the entire constructor for this class had 0 line coverage and a lot of mutants that could be tackled fairly easily. We employed the use of tests that took into account boolean mutants, as well as a bit of refactoring of the constructor so that the possibilities of breaking the class became fairly limited. The end results showed 60% line coverage and 43% mutation coverage. Both of these coverages could be extended even more, but not without a fair share of effort being put into the refactoring of the class. The final scores can be seen in the picture below:

Pit Test Coverage Report

Package Summary

nl.tudelft.sem.template.activity.domain.services

Number of Classes	Line Coverage	Mutation Coverage
9	77% 207/270	71% 101/143

Breakdown by Class

Name	Line Coverage	Mutation Coverage
ActivityService.java	100% 21/21	92% 11/12
BoatRestService.java	60% 12/20	43% 3/7
CompetitionServiceServerSide.java	96% 45/47	90% 18/20
CompetitionServiceUserSide.java	63% 32/51	59% 26/44
RestService.java	45% 5/11	0% 0/2
RestServiceFacade.java	67% 4/6	0% 0/2
TrainingServiceServerSide.java	94% 34/36	88% 15/17
TrainingServiceUserSide.java	76% 47/62	82% 28/34
UserRestService.java	44% 7/16	0% 0/5

UserRestService (in activity microservice)

Pit Test Coverage Report

Package Summary

nl.tudelft.sem.template.activity.domain.services

Number of Classes	Line Coverage	Mutation Coverage
9	76% 205/270	69% 99/143

Breakdown by Class

Name	Line Coverage	Mutation Coverage
ActivityService.java	100% 21/21	100% 12/12
BoatRestService.java	50% 10/20	14% 1/7
CompetitionServiceServerSide.java	96% 45/47	90% 18/20
CompetitionServiceUserSide.java	63% 32/51	57% 25/44
RestService.java	45% 5/11	0% 0/2
RestServiceFacade.java	67% 4/6	0% 0/2
TrainingServiceServerSide.java	94% 34/36	88% 15/17
TrainingServiceUserSide.java	76% 47/62	82% 28/34
UserRestService.java	44% 7/16	0% 0/5

The Pit Test report for the UserRestService class indicated that all of its methods had mutants that the test suite could not kill. A wider issue was the fact that there was no test suite specifically for this class.

Hence, we decided to create the UserRestServiceTest class in order to properly test all the methods contained within this class. The task of creating mutant-killing tests seemed to be quite complex, due to the nature of the deserialize and performUserRequest methods. We employed the use of tests that took into account boolean mutants, as well as a bit of refactoring of the constructor so that the possibilities of breaking the class became fairly limited. The end results showed 76% line coverage and 71% mutation coverage. This could be extended to cover all the mutants, however it would necessitate a significant amount of refactoring for the class in general. The Pit Test Report scores after the mutation testing can be seen in the following screenshot.

Pit Test Coverage Report

Package Summary

nl.tudelft.sem.template.activity.domain.services

Number of Classes	Line Coverage	Mutation Coverage
9	77% 208/269	71% 102/144

Breakdown by Class

Name	Line Coverage	Mutation Coverage
ActivityService.java	100% 21/21	92% 11/12
BoatRestService.java	39% 7/18	0% 0/6
CompetitionServiceServerSide.java	96% 45/47	90% 18/20
CompetitionServiceUserSide.java	63% 32/51	59% 26/44
RestService.java	45% 5/11	0% 0/2
RestServiceFacade.java	67% 4/6	0% 0/2
TrainingServiceServerSide.java	94% 34/36	88% 15/17
TrainingServiceUserSide.java	76% 47/62	79% 27/34
UserRestService.java	76% 13/17	71% 5/7

Part 2

CompetitionServiceUserSide(In activity microservice)

I chose the CompetitionServiceUserSide class because this class is one of the most important classes, and it is responsible for getting suitable competitions for users and letting users join any competition that he or she wants to. Inside this class, getSuitable() is essential because this one will allow the user to know all the competitions he or she is suitable for. I checked this method, and found the logical part which checks whether each competition is suitable for such a user or not has not been tested. To facilitate testing, I bring the logical part into another function suitableBoatIds. I injected 4 bugs. The first one is that starting time of the competition is less than 1 day the time the rower send a request to join that competition, the second one is instead of returning the checkGender() we always return true or false, and also the same as checkAmateur(), and checkOrganization(). These bugs guarantee that the method works properly and returns the desired results; even if we manually change the conditions, the test suite will catch all the bugs.

Since there were no tests, I have generated 4 unit tests, each test is for each bug. As a result the mutations described are now caught by our test suite.

TrainingServiceServerSide (In activity microservice)

The TrainingServiceServerSide class is an absolutely essential class to our application, located in our activity microservice, which is part of our core domain. This class is essential because it handles the maintenance of the activities. If this class were to fail, we could not properly store activities and the entire purpose of our application would not function.

When performing manual mutation testing on this class I found 3 glaring issues. Twice I've stumbled upon the issue of the return value (string) that could be modified without any test failing. The other issue was that a conditional negation did not make the test fail.

In the deleteTraining method and the editTraining method, I found the issue of the return value not being checked. These methods are critical to our application, because without them the user would simply have no way to interact with their activities we are storing.

The editTraining method additionally also had the issue of a conditional negation being untested when checking for the ownership of an activity the user wants to edit.

This conditional negation issue is especially dangerous, since if this bug goes unchecked, everyone will be able to edit everyone's training! Ownership would never be checked by our application.

To solve these 3 issues I made 2 unit tests, 1 for the deleteTraining and 1 for the editTraining, the unit test for the editTraining tests both of the aforementioned issues.

As a result the mutations described are now caught by our test suite.

TrainingServiceUserSide (In activity microservice)

I chose the TrainingServiceUserSide class, as it is long, complex and important for the application, being the service that handles all the training (alongside the TrainingServiceServerSide class). This is inside the activity microservice which is, of course, a core domain. The getSuitableTrainings method is long and essential to the application, as it retrieves the user all the trainings which are available to him (the ones which have boats with a certain position available and the ones which start in at least half an hour after the user makes the request of joining). Firstly, if we manually change the if branch on line 198 (which is a null checker), the tests will still pass (changing the `userData == null` to `userData != null`). This is clearly a problem that must be solved. For this, we will create a test in a new class (TrainingServiceUserSideTest) that checks this exception handling. The name of the method which handles this is `getSuitableTrainingsTestExceptions`. Another mutant that is not killed is created if we switch the operator `>` with `<` or `==` on line 204. For this, we must create another method which doesn't throw any exceptions, namely `getSuitableTrainingsTestNormal`. Once we create a test method which doesn't handle any exception but rather runs the lines 202-211 (by

creating a simple list of 3 trainings, each with a unique boat and each at a different time period) we can see that the mutant is indeed killed.

CompetitionServiceServerSide (in the activity microservice)

Similar to the TrainingServiceServerSide that we tackled earlier, the CompetitionServiceServerSide class is an absolutely essential class in our application that handles the maintenance of competitions. Failure in the functioning of this class would yield the application useless since we won't be able to store competitions.

On performing manual mutation testing on this class, we found some critical issues. On 2 occasions, the return value string could be modified with no tests failing. Another thing was that a negation on a conditional statement caused the tests to pass regardless.

In the deleteCompetition and editCompetition methods, we found the issue of the return String not being checked. These methods are critical to our application, because without them the user would simply have no way to interact with their activities we are storing.

The editCompetition method additionally also had the issue of a conditional negation being untested when checking for the ownership of a competition the user wants to edit.

The conditional negation was a massive issue, since this means that any user can edit another user's competition. Our application never appropriately checked for the ownership of the application.

We created a new test class for CompetitionServiceServerSide which additionally tests the methods of this class rigorously, and added some necessary tests to this class

To solve these 3 issues in particular, in our new test class, I made 2 unit tests, 1 for the deleteCompetition and 1 for the editCompetition, the unit test for the editCompetition tests both of the aforementioned issues.

As a result the mutations described are now caught by our test suite.