ASSIGNMENT 3

SEM GROUP 01a

Angelika Mentzelopoulou
Marina Serafeimidi
Jarno Berger
Daan Panis
Nicolae Filat
Toma Volentir

INTRODUCTION

For the final assignment of the Software Engineering Methods lab, we were tasked with evaluating and enhancing the quality of our project's test suits through the use of mutation testing. Mutation testing is a technique used to determine the effectiveness of the test suite by introducing small, deliberate changes, called mutants, into the source code. The goal is to ensure that the test suite is able to detect and eliminate these mutants, thus providing a higher level of confidence in the overall test coverage.

TASK 1 - AUTOMATED MUTATION TESTING

The first task of the assignment focused on using a mutation tool to isolate classes vulnerable to mutants and improve their coverage to account for that. We decided on Pitest as our mutation testing tool of choice, as was recommended by the course staff. Using the tool, we were able to identify which classes on our project had a mutation score lower than 70%, ending up with the following selection of classes:

Classes for which we increased mutation coverage, along with their initial coverage:

- Authentication
    - AppUser (40%, 2/5)
    - HashedPassword (0%, 0/1)
    - PasswordHashingService (0%, 0/1)
    - PasswordWasChangedEvent (0%, 0/1)
    - RegistrationService (50%, 2/4)
    - HasEvents (0%, 0/2)
    - AuthenticationController (33%, 1/3)
    - WebSecurityConfig (67%, 2/3)
- Cart
    - PizzaService (50%, 8/16)
- Checkout
    - OrderService (31%, 8/26)
    - PickupTimeConfig (0%, 0/1)
    - OrderBuilder (92%, 12/13)
- Customer
    - CustomerService (95%, 19/20)
    - CustomerController (85%, 11/13)
- Store
    - StoreDataLoader (0%, 0/1)
    - EmailNotificationService (32%, 7/22)
    - StoreRestController (78%, 14/18)

Having identified the classes we would work on, we were now able to improve our test suite to ensure our tests would catch as many mutants as possible. We provide an analysis per microservice and class, of the calculated PIT scores before and after our improvements, as well as a description of the changes we made in the code.

> AUTHENTICATION MICROSERVICE

Commit SHA:

PIT scores before:

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
16	99%	117/118	61%	20/33

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
nl.tudelft.sem.template.authentication.authentication	1	100%	9/9	100%	2/2
nl.tudelft.sem.template.authentication.config	1	100%	11/11	67%	2/3
nl.tudelft.sem.template.authentication.controllers	1	100%	11/11	33%	1/3
nl.tudelft.sem.template.authentication.dataloaders	2	100%	20/20	100%	4/4
nl.tudelft.sem.template.authentication.domain	2	100%	10/10	50%	2/4
nl.tudelft.sem.template.authentication.domain.user	9	98%	56/57	53%	9/17

Package Summary

nl.tudelft.sem.template.authentication.domain

Number of Classes	Line Coverage		Mutation Coverage	
2	100%	10/10	50%	2/4

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
AuthenticationExceptionHandler.java	100%	3/3	100%	2/2
HasEvents.java	100%	7/7	0%	0/2

Package Summary

nl.tudelft.sem.template.authentication.controllers

Number of Classes	Line Coverage		Mutation Coverage	
1	100%	11/11	33%	1/3

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
AuthenticationController.java	100%	11/11	33%	1/3

Package Summary

nl.tudelft.sem.template.authentication.domain.user

Number of Classes	Line Coverage		Mutation Coverage	
9	98%	56/57	53%	9/17

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
AppUser.java	100%	20/20	40%	2/5
AuthenticationService.java	100%	4/4	100%	1/1
HashedPassword.java	100%	5/5	0%	0/1
HashedPasswordAttributeConverter.java	100%	3/3	100%	2/2
Password.java	100%	5/5	100%	1/1
PasswordHashingService.java	100%	4/4	0%	0/1
PasswordWasChangedEvent.java	75%	3/4	0%	0/1
RegistrationService.java	100%	8/8	50%	2/4
UserWasCreatedEvent.java	100%	4/4	100%	1/1

The changes that were made per class are as follows:

AppUser

When running the tool, 5 mutants were injected here. Only 2 were already killed by our tests, so to increase our mutation coverage, we further investigated the class. By adding a thorough test that included mocking, we managed to kill the most important surviving mutant. Specifically, in the changePassword() method, we assured that when a user decided to change his password for authentication, the event was being recorded, by verifying the protected method call. To test also that the whole event generation system from spring works we created an integration test that would assert that the events are published when a user is created or when he updates the password.

HashedPassword

For this class, no methods were being tested, so we proceeded to not only kill the mutants injected by the tool, but also create a complete test suite. By adding tests for both the toString() and the equals() method, mutation coverage was raised to 100%.

PasswordHashingService

For this class the mutation was to return null in the hash() function of the service.Even though there was an integration test for the /register endpoint it was mocking the password hashing service and the mutant survived. By adding a simple unit test that asserted whether the hash function is not null the mutant was successfully killed.

PasswordWasChangedEvent

In this class the function getUser() was not tested. By adding tests for the getter we managed to get 100% mutant coverage for this class too.

RegistrationService

In this class there were two mutants:

- one that returned null in registerUser() function instead of the user that registered
- one that flipped the condition of checking if the NetIdIsUnique

After checking intellij tools we found out that we do not actually need the user back after we register, thus making the method return nothing did not change the functionality of the code. Even though the last mutant was killed by some integration tests, PITest did not run those. I created another test that PITest found and achieved 100% mutation coverage on this class.

HasEvents

The mutants for this class were related to the annotations from spring for publishing events. To properly test this feature I created an integration test for checking that the events are published. The annotation @AfterDomainEventPublication in spring is responsible for generating an event only once. Thus, we also tested that saving twice to the database the same user results in a single event generation.

AuthenticationController

In this class there were two mutants:

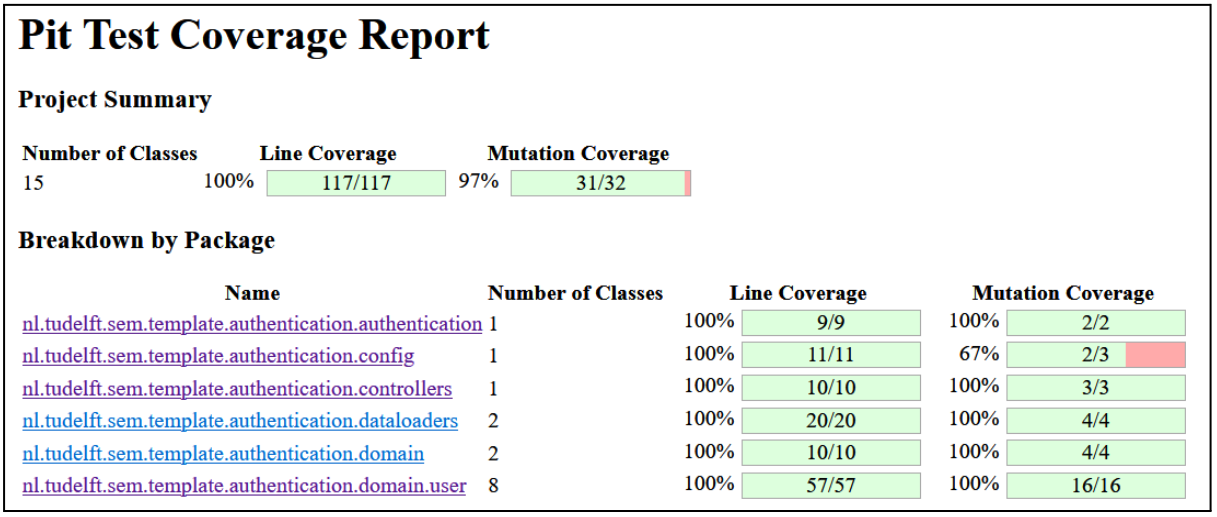
- one removed the stacktrace printing of a potential error
- another one returned null in the register endpoint

Since the error printing was for debugging I removed that line from the code. To fix the null return in the response I changed the function to return an OK status with the message “Registration successful” and assert the received message in an integration test.
This achieved 100% mutation coverage.

WebSecurityConfig

After the above changes, the mutation coverage for the Authentication Microservice was improved to almost 100% for all packages of the microservice. In regards to the config package, the WebSecurityConfig class is the one bringing down the coverage. Despite having extensive tests for it and 100% line coverage, Pitest does not recognize the improvements. Since the tests that we wrote to cover the webConfig is an integration test pitest does not know that the integration test covers that line and it does not run it against the mutant. PITest was not designed for integration tests, but mostly for unit tests.

PIT scores after:



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
15	100%	117/117	97%	31/32

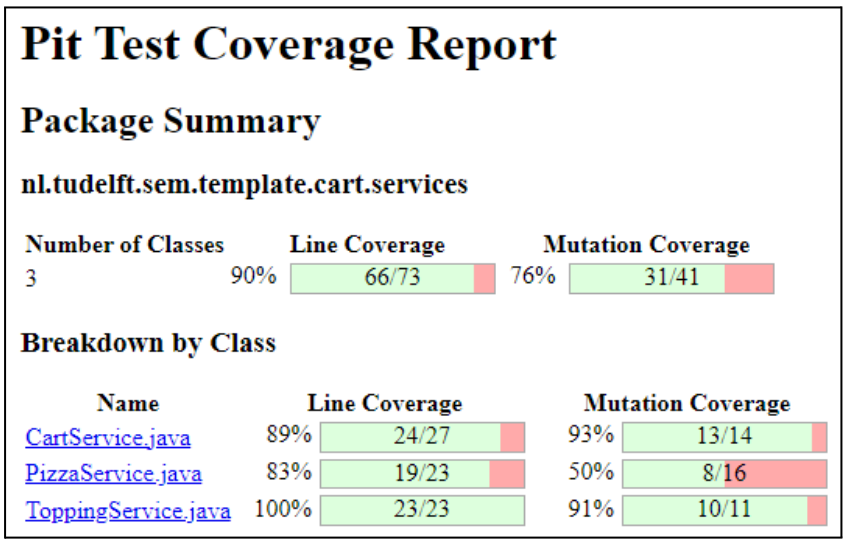
Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
nl.tudelft.sem.template.authentication.authentication	1	100%	9/9	100%	2/2
nl.tudelft.sem.template.authentication.config	1	100%	11/11	67%	2/3
nl.tudelft.sem.template.authentication.controllers	1	100%	10/10	100%	3/3
nl.tudelft.sem.template.authentication.dataloaders	2	100%	20/20	100%	4/4
nl.tudelft.sem.template.authentication.domain	2	100%	10/10	100%	4/4
nl.tudelft.sem.template.authentication.domain.user	8	100%	57/57	100%	16/16

> CART MICROSERVICE

Commit SHA: [7f87e4e3a94cfd7f18f433ad68649de0b65db26b]

PIT scores before:



Pit Test Coverage Report

Package Summary

nl.tudelft.sem.template.cart.services

Number of Classes	Line Coverage		Mutation Coverage	
3	90%	66/73	76%	31/41

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
CartService.java	89%	24/27	93%	13/14
PizzaService.java	83%	19/23	50%	8/16
ToppingService.java	100%	23/23	91%	10/11

The changes that were made per class are as follows:

PizzaService

It was clear from the generated coverage statistics that PizzaService was the weakest class in regards to mutation coverage, which meant it was quite vulnerable. To remedy this, 4 new tests were added to PizzaServiceTest class to kill the mutants. The filter method in pizzaService was added quite close to the deadline of the project, therefore we did not have time to test it. So in this assignment we made sure to test it extensively. As a result, almost all the mutants were killed, getting close to perfect coverage. A final test verifying that a pizza was actually being edited was added to the suite, finishing the job at 100% mutation coverage.

PIT scores after:

Pit Test Coverage Report

Package Summary

nl.tudelft.sem.template.cart.services

Number of Classes	Line Coverage		Mutation Coverage	
3	96%	70/73	95%	39/41

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
CartService.java	89%	24/27	93%	13/14
PizzaService.java	100%	23/23	100%	16/16
ToppingService.java	100%	23/23	91%	10/11

> CHECKOUT MICROSERVICE

Commit SHA: [aa920d61654f7cc8a945397ea01352105245d65e]

PIT scores before:

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
6	87%	128/147	73%	53/73

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
nl.tudelft.sem.template.checkout.controllers	1	100%	65/65	100%	26/26
nl.tudelft.sem.template.checkout.domain	5	77%	63/82	57%	27/47

Package Summary

nl.tudelft.sem.template.checkout.domain

Number of Classes	Line Coverage		Mutation Coverage	
5	77%	63/82	57%	27/47

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
LocalDateTimeConverter.java	100%	4/4	100%	2/2
Order.java	100%	17/17	100%	5/5
OrderBuilder.java	100%	21/21	92%	12/13
OrderService.java	44%	15/34	31%	8/26
PickupTimeConfig.java	100%	6/6	0%	0/1

The changes that were made per class are as follows:

OrderService

After assignment 2, 4 new methods have been added to the OrderService class (relocated from OrderController because of refactoring). By adding 17 new tests to properly test everything and alter one pre-existing test, we were able to kill all mutants.

PickupTimeConfig

A test was added which checks whether the Jackson Object Mapper that we use to parse JSON into DateTime classes is an actual non null object. This killed the mutant which replaced all the code by “return null;” in the method from this class.

OrderBuilder

Additionally, the **OrderBuilder** had an untested method, namely the setPrice(). A new test was added for this in the **OrderTest** class. Through this, we achieved 100% mutation coverage, instead of the original 92% (12/13). Also, in order to make pitest work, the database was flushed before each run of the method from the integration test package for order controller.

PIT scores after:

Number of Classes	Line Coverage		Mutation Coverage	
6	100%	147/147	100%	73/73

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
nl.tudelft.sem.template.checkout.controllers	1	100%	65/65	100%	26/26
nl.tudelft.sem.template.checkout.domain	5	100%	82/82	100%	47/47

Package Summary

nl.tudelft.sem.template.checkout.domain

Number of Classes	Line Coverage		Mutation Coverage	
5	100%	82/82	100%	47/47

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
LocalDateTimeConverter.java	100%	4/4	100%	2/2
Order.java	100%	17/17	100%	5/5
OrderBuilder.java	100%	21/21	100%	13/13
OrderService.java	100%	34/34	100%	26/26
PickupTimeConfig.java	100%	6/6	100%	1/1

> CUSTOMER MICROSERVICE

Commit SHA: [ecc975e9604b1b19a36a189bd6011eb689092676]

PIT scores before:

Pit Test Coverage Report

Package Summary

nl.tudelft.sem.template.customer.domain

Number of Classes	Line Coverage		Mutation Coverage	
2	96%	45/47	95%	20/21

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
CustomerExceptionHandler.java	100%	2/2	100%	1/1
CustomerService.java	96%	43/45	95%	19/20

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
3	95%	74/78	91%	31/34

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
nl.tudelft.sem.template.customer.controllers	1	94%	29/31	85%	11/13
nl.tudelft.sem.template.customer.domain	2	96%	45/47	95%	20/21

Pit Test Coverage Report

Package Summary

nl.tudelft.sem.template.customer.controllers

Number of Classes	Line Coverage		Mutation Coverage	
1	94%	29/31	85%	11/13

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
CustomerController.java	94%	29/31	85%	11/13

The changes that were made per class are as follows:

CustomerService

In the CustomerService there were two mutations that were not caught in our current test suite.

The first mutation that wasn't caught was in the **checkUsedCoupons** method. This method is supposed to filter out coupons in a list of coupons based on whether they were already used by a user. One check we do, is that if the list of used coupons of the user is null or empty, that we return the entire list of coupons (since none have been used). The mutant changes the return statement to return an empty list instead of the entire list of coupons. To fix this we added another test that checks this case.

The second mutant that wasn't caught was in the **removeFromUsedCoupon** method. In this method we have a check whether the **getCustomerByNetId** method returns a null object. In this case it will immediately return. This check is actually unnecessary since the **getCustomerByNetId** method will never return null, since it will throw an exception if the customer doesn't exist. This also makes it virtually impossible to test for this case, so we decided to remove this null check.

CustomerController

The CustomerController had two simple mutants that weren't being caught. The mutants replaced the return value of the **removeFromUsedCoupons** and **updateAllergens** methods to null. To catch these mutants we updated two tests to assert that the request actually returns the expected response.

PIT scores after:

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
3	97%	74/76	100%	33/33	100%	33/33

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
nl.tudelft.sem.template.customer.controllers	1	94%	29/31	100%	13/13
nl.tudelft.sem.template.customer.domain	2	100%	45/45	100%	20/20

> STORE MICROSERVICE

Commit SHA: f8d6f0d4cb7e13a7722b8075130e4888b866c0d7

PIT scores before:

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
3	100%	71/71	51%	21/41

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
nl.tudelft.sem.template.store	1	100%	11/11	0%	0/1
nl.tudelft.sem.template.store.controller	1	100%	32/32	78%	14/18
nl.tudelft.sem.template.store.services	1	100%	28/28	32%	7/22

Package Summary

nl.tudelft.sem.template.store

Number of Classes	Line Coverage		Mutation Coverage	
1	100%	11/11	0%	0/1

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
StoreDataLoader.java	100%	11/11	0%	0/1

Package Summary

nl.tudelft.sem.template.store.services

Number of Classes	Line Coverage		Mutation Coverage	
1	100%	28/28	32%	7/22

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
EmailNotificationService.java	100%	28/28	32%	7/22

Package Summary

nl.tudelft.sem.template.store.controller

Number of Classes	Line Coverage		Mutation Coverage	
1	100%	32/32	78%	14/18

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
StoreRestController.java	100%	32/32	78%	14/18

The changes that were made per class are as follows:

EmailNotificationService

Our tests for the EmailNotificationService were pretty simple mutants that weren't being caught. The mutants introduced removed calls that were setting the sender, recipient and subject of the email notifications that we sent to store owners when orders are placed/canceled (in the methods **notifyOrder** and **notifyOrderRemove** methods). To catch these we updated our existing tests to check whether the sender, recipient and subject were set correctly.
The other mutants were actually quite weird, since it appeared they were being generated on our *"try with resources"* calls that read our HTML templates that we sent as the content of our email notifications. The *"try with resources"* will actually generate a bunch of code that handles closing the resource when your program is compiled to java byte code. Since pitest creates mutants based on your bytecode (not the source code), it was introducing mutants on code that we couldn't actually modify. This is virtually impossible (or at the very least extremely hard) to test but by updating the pitest version to 1.6.3 it correctly ignores *"try with resources"* statements when creating mutants. For more information, see the following GitHub issue: https://github.com/hcoles/pitest/issues/255

StoreRestController

Two of the mutants that weren't caught by our test suite were printStackTrace calls being removed when sending email notifications for placed/canceled orders. These were only used for debugging purposes, so we could safely remove these calls from our source code.
The other two mutants were replacing the response text of the **notifyStore** and **notifyRemoveOrder** methods with an empty string. In order to catch these mutants correctly we updated our existing tests to verify that the request returns a non-empty string as a response.

StoreDataLoader

The StoreDataLoader is responsible for creating some default store locations to the database. There was only mutant that wasn't being caught, in which it removed the call where we log a message to the console to indicate that the stores were added. To catch this mutant we introduced a test which verifies that this message is actually logged to the console.

PIT scores after:

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
3	100%	69/69	100%	29/29	100%	29/29

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
nl.tudelft.sem.template.store	1	100%	11/11	100%	1/1
nl.tudelft.sem.template.store.controller	1	100%	30/30	100%	16/16
nl.tudelft.sem.template.store.services	1	100%	28/28	100%	12/12

TASK 2 - MANUAL MUTATION TESTING

For the second task of the assignment, we were asked to manually create mutants in order to identify weak spots in our test suite. Subsequently, we had to write test cases that would successfully kill those mutants thus improving the mutation coverage, as well as the overall coverage of our project. The core domain of our project we chose to work on and improve is the **Cart** microservice, as it is our largest and most extensive microservice. Within that domain we identified the following four classes to manually mutate, which we deemed critical for our application for the following reasons:

- **`CartController`**
  - Being responsible for adding/removing pizzas to the cart as well as adding extra toppings to pizzas, the CartController is definitely a critical class for our project. Without it, the app's basic functionality would be gone.
- **CartService**
  - The cart service is responsible for providing helper methods to the CartController class. Such helper methods include asserting that a pizza is in the cart or getting the cart from a user's NetId.
- **`PizzaController`**
  - We deemed the PizzaController class as critical to our application because it handles the requests that are responsible for updating the menu by adding/editing/removing pizzas on the database.
- **`ToppingService`**
  - The topping service class is critical for our application because it handles most of the logic with regards to adding, retrieving, modifying and deleting the toppings for the pizzas. This is core functionality of our application and thus it is crucial that this part is functioning correctly.

Having identified these four critical classes, we could now work on manual mutation testing for each one, as described below.

CartController

Commit SHA: [01579147b3068ae81f0f29dfebb7cd3f70b52379]

```
@Test
void testAddToCartSingleCart() throws Exception {
    int id1 = defaultPizza1.getId();
    int id2 = defaultPizza2.getId();
    addPizzaRequest(id1).andExpect(status().isOk());
    addPizzaRequest(id2).andExpect(status().isOk());
    var cart :Cart = cartRepository.findAll().stream().filter(x -> x.getNetId().equals(new NetId(TEST_USER))).collect(Collectors.toList()).get(0);
    assertEquals(cart.getPizzasMap().size(), actual: 2);
}
```

For the cart controller we discovered a very interesting mutant for the **addPizzaToCart** endpoint. One of the first things our code does in this method is to retrieve the cart of the user, making use of his netId. After that, we have included a check for whether the current customer does not have a cart yet, in other words if the returned value from the retrieval is null. If this is indeed the case, our code is instructed to create a new cart for them and add that to the database.
Through the process of injecting mutants though, we realized that all our checks were passing even if we removed the check about whether the retrieved cart was nonexistent.

This meant that our tests had omitted verifying that the cart is persistent when ordering multiple pizzas, since removing this check will create a new cart for the user every time they add a new pizza to their cart. We deemed it necessary to add a test suite for this case, since being able to add multiple pizzas to your cart is of vital importance in the implementation. of the whole checkout process, and more specifically the calculation of the final price, possibly having applied coupons as well. Therefore, the solution to this was to add a new test suite testAddToCartSingleCart(), where our assertion assures that the current cart of the user contains the correct amount of pizzas.

PizzaController

Commit SHA: [0b2d72982b34a633fd5d215102775422221f4acc]

In the pizzaController there was an untested method, getPizzasFiltered. Because of this the method could be changed to anything and we would not see any consequences in the tests. For example removing all the code and returning null instead will not make any tests fail since there are not any. That's why we decided that this was a critical issue. We added a test to make sure not only the functionality of the test was working, but also verifying that the correct methods were called and being used. By adding this test the code could not be altered anymore, and therefore the mutants got killed.

ToppingService

Commit SHA: [0b2d72982b34a633fd5d215102775422221f4acc]

The toppingService contained a pretty important issue, the if statement in the picture above was not tested. This is a critical mutant, because the statement prevents an InternalServerError when you try to add a topping that does not exist. To solve this we added a test where we try to edit a topping with a non existent topping. After adding this test if you try to remove the if statement, the test will fail. And with that the mutant is killed.

CartService

Commit SHA:
https://gitlab.ewi.tudelft.nl/cse2115/2022-2023/SEM01a/-/commit/3f27130336685b28d26621f2cdc689cc905e5dc9

```
@Test
void testGetCartDifferentUser() throws Exception {
    // make sure we add another user to the cart database before we proceed
    Cart cartFirstUser = new Cart(new NetId("anotherUser@gmail.com"), Map.of());
    cartRepository.save(cartFirstUser);
    // we have one cart element now
    assertThat(cartRepository.count()).isOne();

    // add two pizzas for the current authenticated user which is TEST_USER
    addPizzaRequest(defaultPizza1.getId());
    addPizzaRequest(defaultPizza2.getId());

    // get the Cart from the TEST_USER
    var cart :Cart = cartRepository.findAll().stream().filter(c -> c.getNetId().equals(new NetId(TEST_USER))).findFirst().get();
    var result :ResultActions = getCartRequest(new NetId(TEST_USER)).andExpect(status().isOk());
    var cartPizzas :List<CartPizza> = Arrays.asList(parseResponseJson(result, CartPizza[].class));
    assertEquals(cart.getPizzasMap().size(), cartPizzas.size());
    cartPizzas.forEach(cp -> {
        var pizzaInMap :Optional<Map<K, V>.Entry<CustomPizza, Integer>> =
            cart.getPizzasMap().entrySet().stream().filter(e -> e.getKey().getId() == cp.getPizza().getId()).findFirst();
        assertTrue(pizzaInMap.isPresent());
        assertEquals(pizzaInMap.get().getValue(), cp.getAmount());
    });
}
```

In the CartService class we discovered another interesting mutant. We introduced the mutant in the function getCartFromSessionNetId() to just get the first cart in the database ignoring entirely the NetId field. Since our tests only worked with one user and not with multiple the tests passed on the mutated program. Thus, getting the cart by netId or just getting the first cart was equivalent with respect to the test suite we had. This was a serious bug not caught by the tests.

In order to fix the mutant, a test was added that would first add another user's cart to the cart database and then proceed to add pizzas to the cart of the authenticated user(TEST USER). This tested that adding pizzas to the authenticated user with the cart 2, still worked as before.