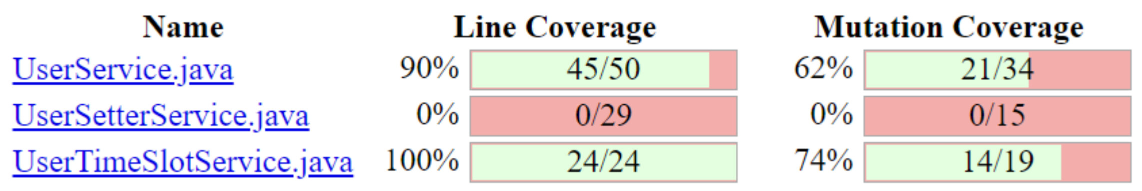
Assignment 3 - Report

1. AUTOMATED MUTATIONS

USER SERVICE

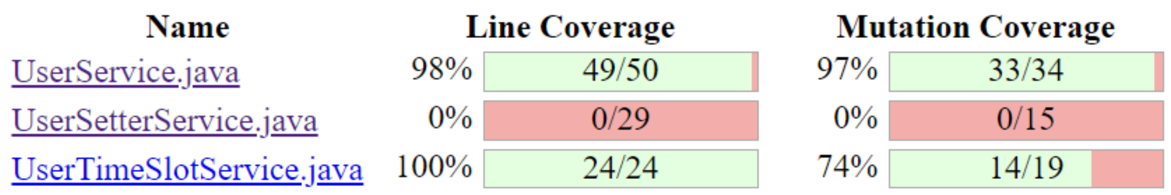
Due to the limited time for delivering the final product, the *UserService* has only been tested using its controller. This is why some of the branches of the methods inside this class were not covered and the results delivered by the analysis of *PiTest* were low.



Name	Line Coverage		Mutation Coverage	
UserService.java	90%	45/50	62%	21/34
UserSetterService.java	0%	0/29	0%	0/15
UserTimeSlotService.java	100%	24/24	74%	14/19

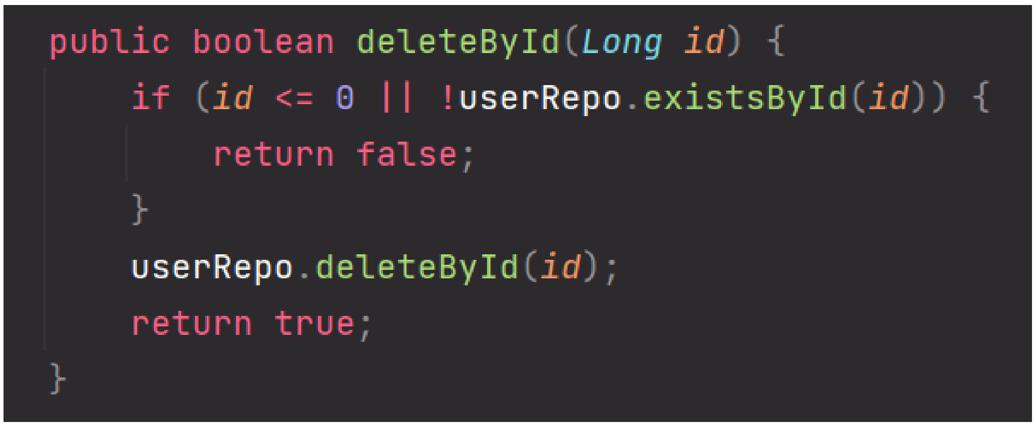
Before starting to create a separate test class for *UserService*, the mutation score was 62%, which was only created using the tests from the *UserController*. In order to increase the coverage, we have looked at each method and taken into consideration all potential paths a variable can take and all edge cases.

To make sure that every used method is thoroughly tested, we have not only created tests for the mutants that were not "destroyed" (commit link), but took every method inside this class and written tests for them so that the coverage does not depend on the controller coupled to the service. We have used unit testing, mocking the *UserRepository* and the *UserService* to see their behavior when simulating various scenarios (commit link 1, commit link 2).



Name	Line Coverage		Mutation Coverage	
UserService.java	98%	49/50	97%	33/34
UserSetterService.java	0%	0/29	0%	0/15
UserTimeSlotService.java	100%	24/24	74%	14/19

In the end, we have achieved 97% mutation coverage and 98% line coverage for this service. The reason that we could not reach 100% for both categories was one condition inside the method *deleteById*, illustrated below:



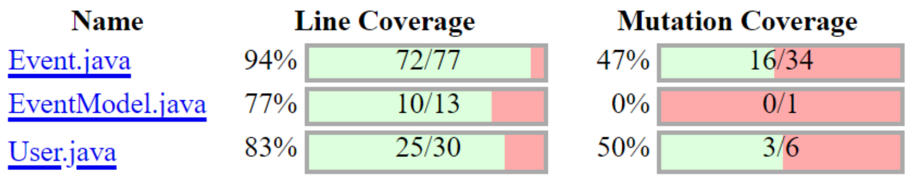
```java
public boolean deleteById(Long id) {
    if (id <= 0 || !userRepo.existsById(id)) {
        return false;
    }
    userRepo.deleteById(id);
    return true;
}
```

As you can see, the first step in this method's check is to ensure that an ID is legitimate. Our implementation ensures that whenever a new user is created, its id will never be less than 1. *PiTest* has the potential to modify the conditional boundary, which in some circumstances could become *id >= 0*. The test that checks the border of id 0 will still pass even though it does not include any ids greater than 0, indicating that the mutant is still "alive".

EVENT

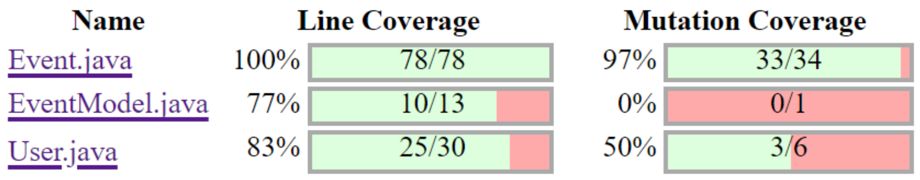
The Mutation Coverage for the *Event* class was only 47%. We were able to raise the Line Coverage to 100% and the Mutation Coverage to 97%. The only line we were not able to cover was the return value of adding to an ArrayList (in the *enqueue* method). Replacing the return value with "true" always survives. Unless an exception is thrown, the *add(E)* method for ArrayLists should always return true, and our method creates a safe-to-add object within the method.

BEFORE



Name	Line Coverage		Mutation Coverage	
Event.java	94%	72/77	47%	16/34
EventModel.java	77%	10/13	0%	0/1
User.java	83%	25/30	50%	3/6

AFTER

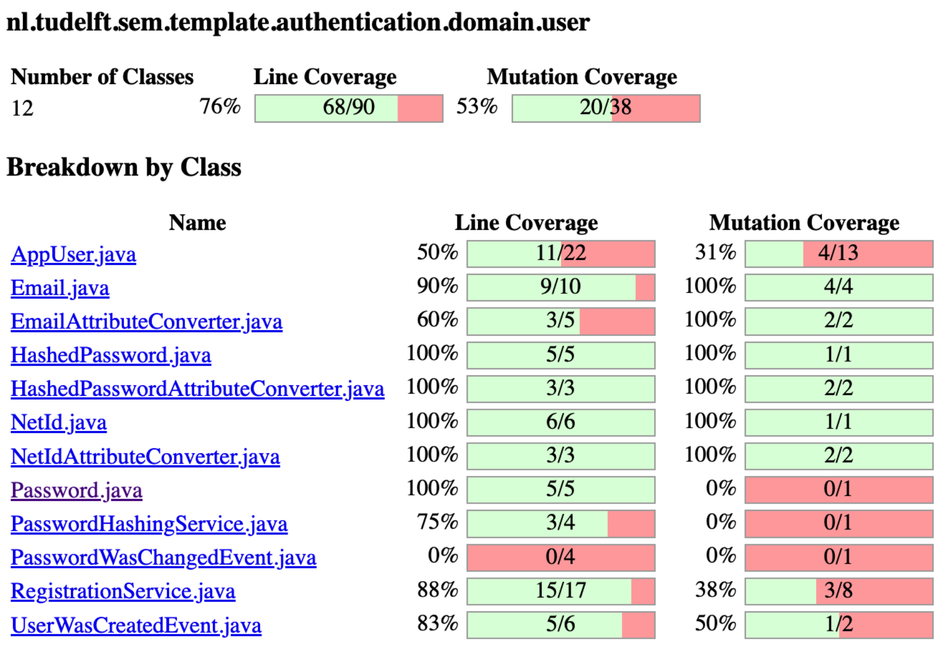


Name	Line Coverage		Mutation Coverage	
Event.java	100%	78/78	97%	33/34
EventModel.java	77%	10/13	0%	0/1
User.java	83%	25/30	50%	3/6

APPUSER and smaller classes in nl.tudelft.sem.template.authentication.domain.user

Before applying any refactoring to improve the mutation testing the coverage was 53% for the entire folder and 31% for the class. This was mainly caused by the fact that methods inside the *AppUser* were not tested. The only method that could not be tested was a private method of the parent of a user class and when running a mock test it would not be able to access it to assert a specific action we decided not to change this method to have public access rights as it was a bad design choice. After all, the coverage has been greatly increased from 53% to 95% and for the AppUser class from 31% to 92%.

BEFORE



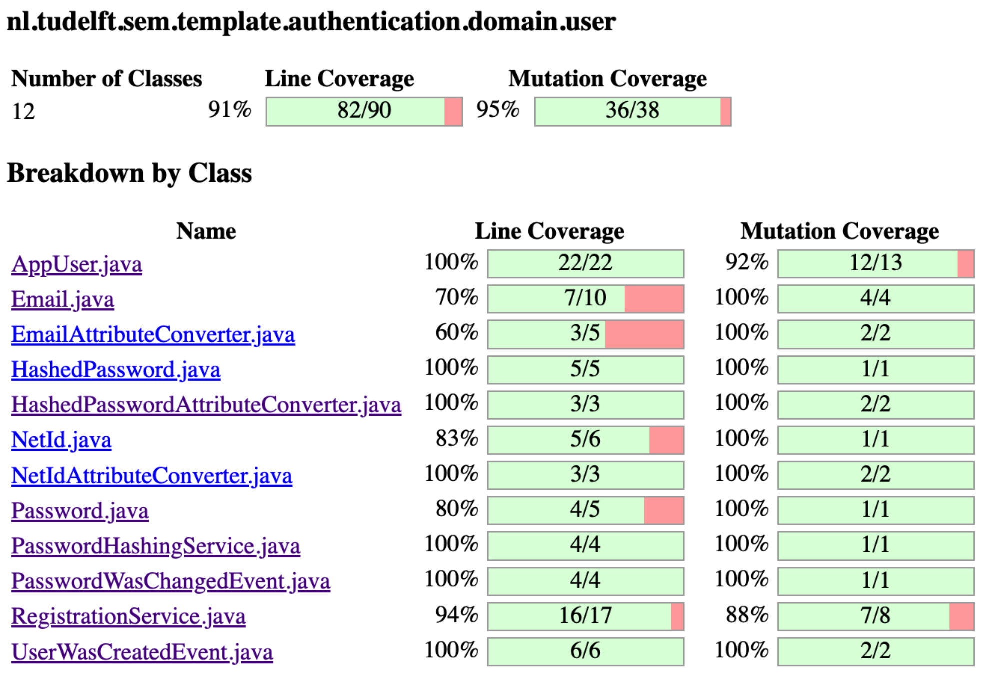
nl.tudelft.sem.template.authentication.domain.user

Number of Classes	Line Coverage		Mutation Coverage	
12	76%	68/90	53%	20/38

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
AppUser.java	50%	11/22	31%	4/13
Email.java	90%	9/10	100%	4/4
EmailAttributeConverter.java	60%	3/5	100%	2/2
HashedPassword.java	100%	5/5	100%	1/1
HashedPasswordAttributeConverter.java	100%	3/3	100%	2/2
NetId.java	100%	6/6	100%	1/1
NetIdAttributeConverter.java	100%	3/3	100%	2/2
Password.java	100%	5/5	0%	0/1
PasswordHashingService.java	75%	3/4	0%	0/1
PasswordWasChangedEvent.java	0%	0/4	0%	0/1
RegistrationService.java	88%	15/17	38%	3/8
UserWasCreatedEvent.java	83%	5/6	50%	1/2

AFTER



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

Number of Classes	Line Coverage		Mutation Coverage	
12	91%	82/90	95%	36/38

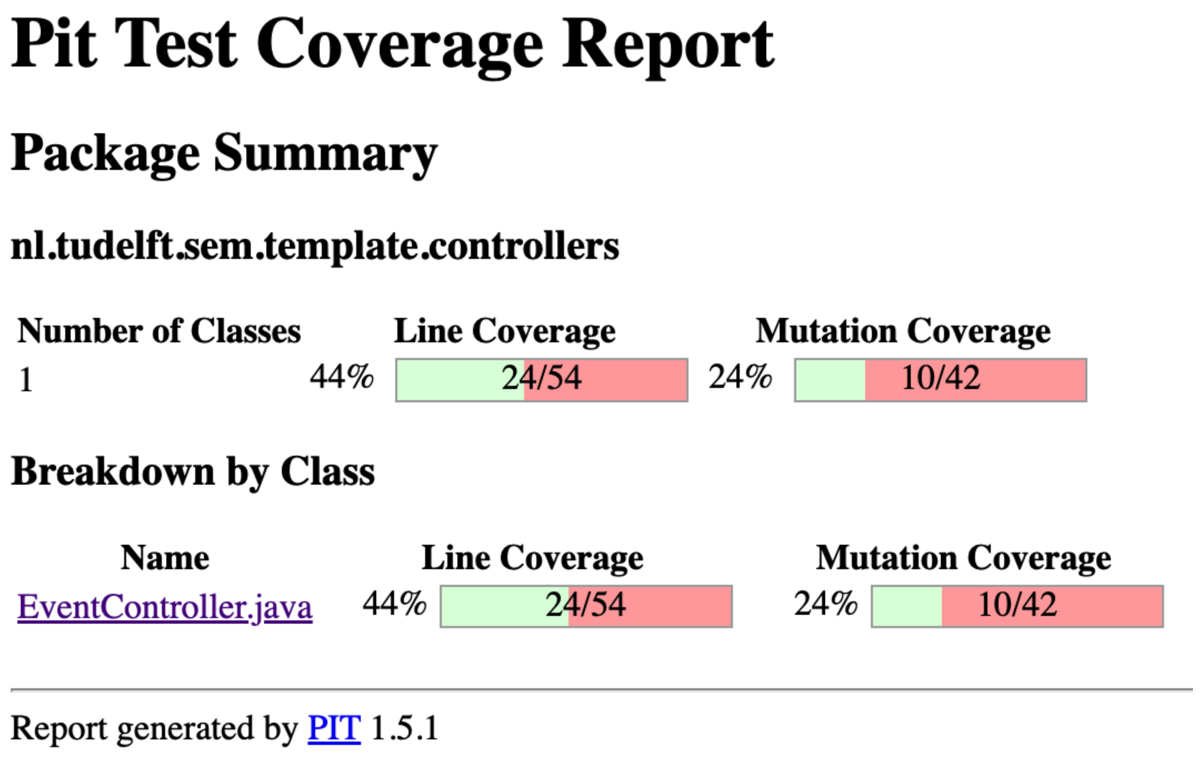
Breakdown by Class

Name	Line Coverage		Mutation Coverage	
AppUser.java	100%	22/22	92%	12/13
Email.java	70%	7/10	100%	4/4
EmailAttributeConverter.java	60%	3/5	100%	2/2
HashedPassword.java	100%	5/5	100%	1/1
HashedPasswordAttributeConverter.java	100%	3/3	100%	2/2
NetId.java	83%	5/6	100%	1/1
NetIdAttributeConverter.java	100%	3/3	100%	2/2
Password.java	80%	4/5	100%	1/1
PasswordHashingService.java	100%	4/4	100%	1/1
PasswordWasChangedEvent.java	100%	4/4	100%	1/1
RegistrationService.java	94%	16/17	88%	7/8
UserWasCreatedEvent.java	100%	6/6	100%	2/2

EventController class

In this class, because of some changes made in refactoring, a lot of tests became obsolete, and some of the already existing ones did not cover some branches created in the process. Before applying changes and adding new tests the line coverage was at 44% and mutation coverage at 24%. That is why we decided to improve tests for this class. For the new tests, we decided to use mockMvc which we think is a better way of testing the controllers than the one we used in other tests as it mocks the request to the controller as well, not only verifies the function. In the testing process, we removed some System.out.print lines that were obsolete and caused some mutants to be added. We added new tests that covered and killed all mutants in the newly added branches. The only problem we encountered was the use of *Timestamp* with the current system time in the accept method. We decided to extract the lines to another function in the EventService class and mock the response to test everything properly. Also, we removed one else statement in the same method that we deemed unnecessary because the logic in there would never be triggered. By adding all the changes we managed to increase line coverage and mutation coverage to 100% for EventController class.

BEFORE



Pit Test Coverage Report

Package Summary

nl.tudelft.sem.template.controllers

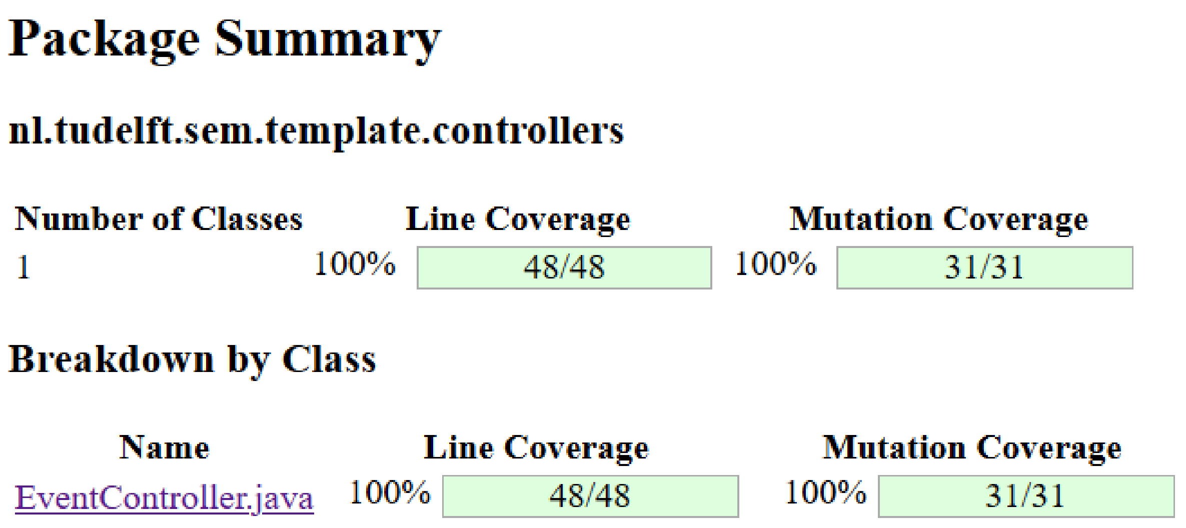
Number of Classes	Line Coverage		Mutation Coverage	
1	44%	24/54	24%	10/42

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
EventController.java	44%	24/54	24%	10/42

Report generated by PIT 1.5.1

AFTER



Package Summary

nl.tudelft.sem.template.controllers

Number of Classes	Line Coverage		Mutation Coverage	
1	100%	48/48	100%	31/31

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
EventController.java	100%	48/48	100%	31/31

2. MANUAL MUTATIONS

To simulate what *PiTest* does automatically, we have decided to use the Event core domain and throw in some mutants to a couple of classes. Knowing that all tests were passing before adding mutants, if we run them again afterward, we should see if we tested all possible versions of interactions in the system. The Event core domain is critical for our application. Any bugs in this area can result in a poor user experience, where users may be unable to find the desired event or have to navigate through multiple duplicates to find a different one.

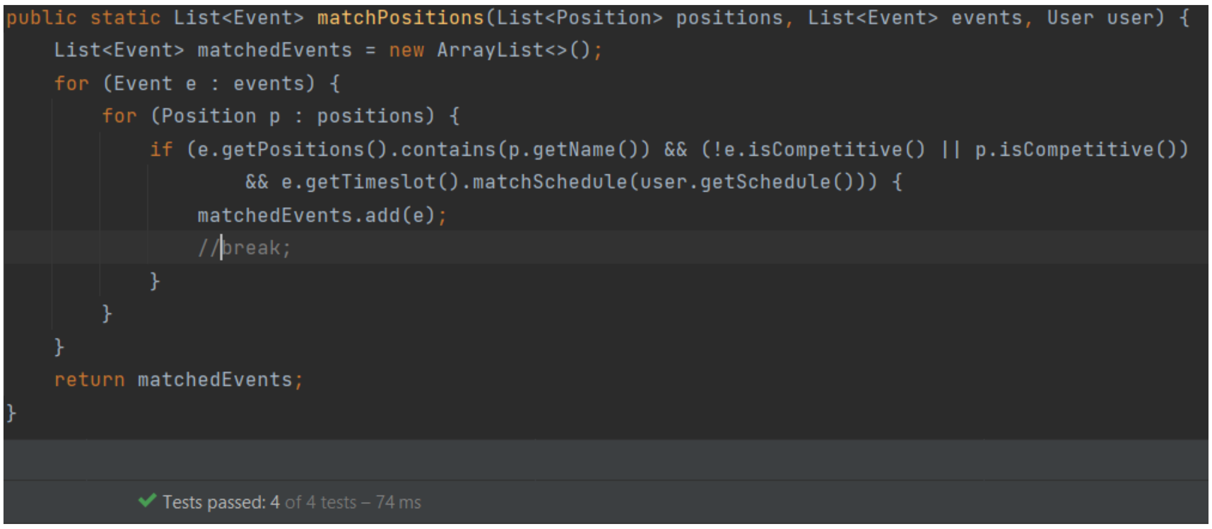
PositionMatcher

Commit with the test can be found at:
https://gitlab.ewi.tudelft.nl/cse2115/2022-2023/SEM29a/-/commit/b8ecad4a422dd1034f82ca55529639f17874821a?merge_request_iid=98

In this class, we look at the *matchPositions* method which is responsible for matching the positions a user can fulfill with events that require a member in that position. Without this method fully functional, a user won't be able to find all or any events that match his position which is a critical part of the user experience.

After trying to introduce multiple mutants to this class, which were all detected by the already present tests, we found that by removing the *"break;"* on line 25, the existing test set wouldn't detect that the same event could be added multiple times. We added a test that can detect this type of mutant.

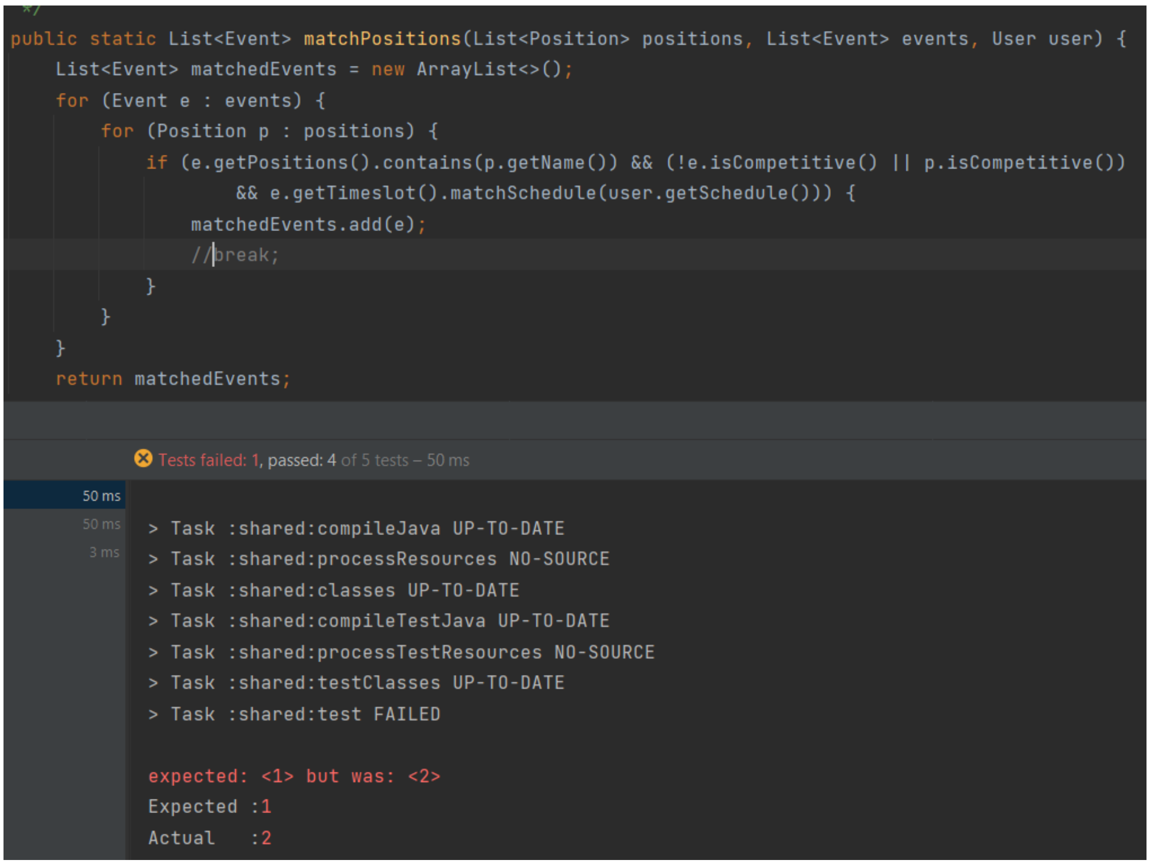
AFTER REMOVING *"break;"*



```
public static List<Event> matchPositions(List<Position> positions, List<Event> events, User user) {
    List<Event> matchedEvents = new ArrayList<>();
    for (Event e : events) {
        for (Position p : positions) {
            if (e.getPositions().contains(p.getName()) && (!e.isCompetitive() || p.isCompetitive())
                    && e.getTimeslot().matchSchedule(user.getSchedule())) {
                matchedEvents.add(e);
                //break;
            }
        }
    }
    return matchedEvents;
}
```

```
✓ Tests passed: 4 of 4 tests – 74 ms
```

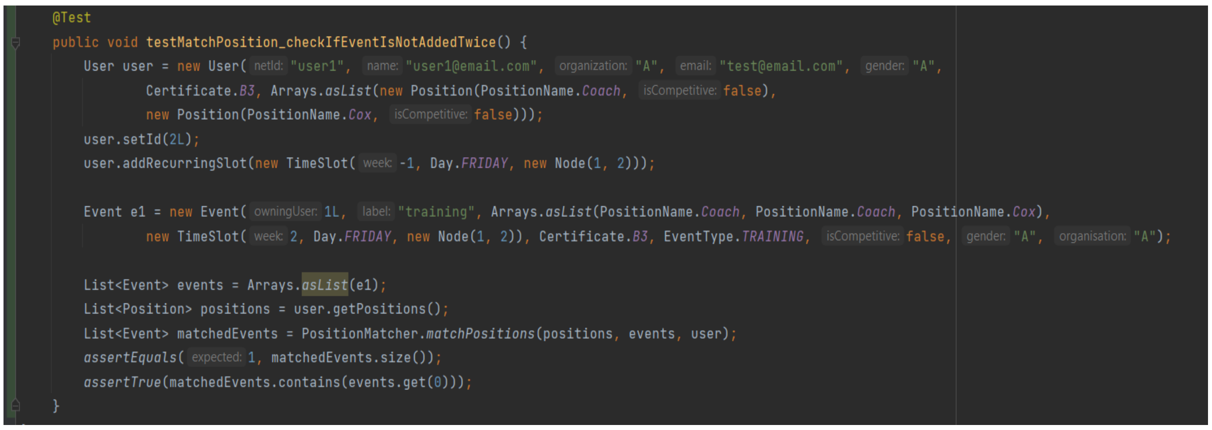
AFTER ADDING THE TEST



```
*/
public static List<Event> matchPositions(List<Position> positions, List<Event> events, User user) {
    List<Event> matchedEvents = new ArrayList<>();
    for (Event e : events) {
        for (Position p : positions) {
            if (e.getPositions().contains(p.getName()) && (!e.isCompetitive() || p.isCompetitive())
                    && e.getTimeslot().matchSchedule(user.getSchedule())) {
                matchedEvents.add(e);
                //break;
            }
        }
    }
    return matchedEvents;
```

```
Tests failed: 1, passed: 4 of 5 tests – 50 ms

50 ms
50 ms
3 ms

> Task :shared:compileJava UP-TO-DATE
> Task :shared:processResources NO-SOURCE
> Task :shared:classes UP-TO-DATE
> Task :shared:compileTestJava UP-TO-DATE
> Task :shared:processTestResources NO-SOURCE
> Task :shared:testClasses UP-TO-DATE
> Task :shared:test FAILED

expected: <1> but was: <2>
Expected :1
Actual   :2
```

TEST



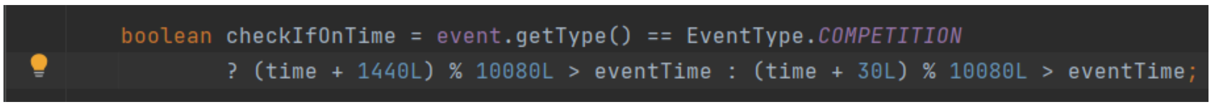
```
@Test
public void testMatchPosition_checkIfEventIsNotAddedTwice() {
    User user = new User( netId: "user1",  name: "user1@email.com",  organization: "A",  email: "test@email.com",  gender: "A",
            Certificate.B3, Arrays.asList(new Position(PositionName.Coach,  isCompetitive: false),
            new Position(PositionName.Cox,  isCompetitive: false)));
    user.setId(2L);
    user.addRecurringSlot(new TimeSlot( week: -1, Day.FRIDAY, new Node(1, 2)));

    Event e1 = new Event( owningUser: 1L,  label: "training", Arrays.asList(PositionName.Coach, PositionName.Coach, PositionName.Cox),
            new TimeSlot( week: 2, Day.FRIDAY, new Node(1, 2)), Certificate.B3, EventType.TRAINING,  isCompetitive: false,  gender: "A",  organisation: "A");

    List<Event> events = Arrays.asList(e1);
    List<Position> positions = user.getPositions();
    List<Event> matchedEvents = PositionMatcher.matchPositions(positions, events, user);
    assertEquals( expected: 1, matchedEvents.size());
    assertTrue(matchedEvents.contains(events.get(0)));
}
```

ValidityChecker

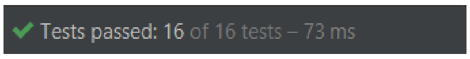
Commit with the tests can be found at:
https://gitlab.ewi.tudelft.nl/cse2115/2022-2023/SEM29a/-/commit/00629c9da0156846c19141ee846f456b234808d5?merge_request_iid=98

In this class, we decided to look at the *canJoin* method. This method checks if the user matches the criteria to join the event and, therefore, is essential for a good user experience. After testing out some mutants on this method, we found that introducing mutants on line 33 really showed some weaknesses in our test suite. Of course, we fixed that by introducing one test for each mutation.

BEFORE

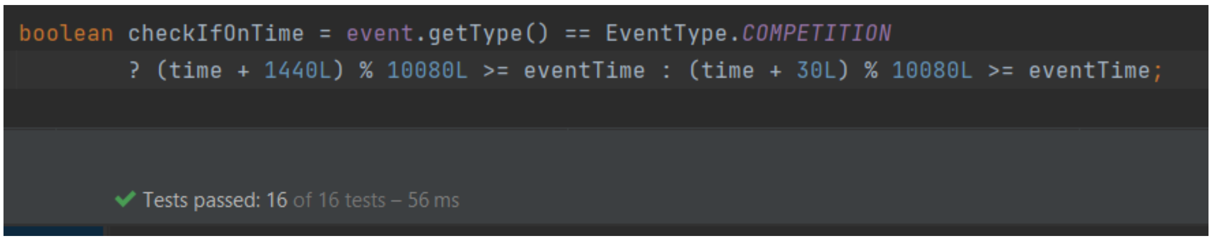


```
boolean checkIfOnTime = event.getType() == EventType.COMPETITION
        ? (time + 1440L) % 10080L > eventTime : (time + 30L) % 10080L > eventTime;
```



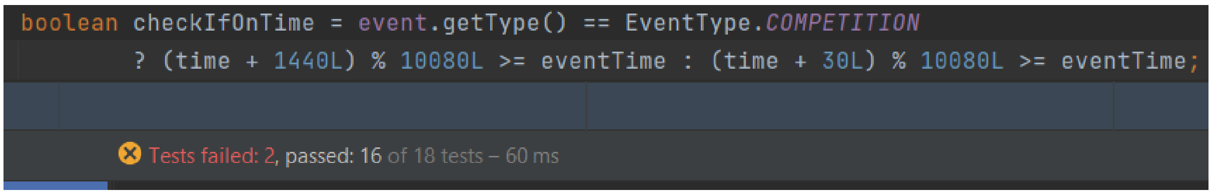
```
✓ Tests passed: 16 of 16 tests – 73 ms
```

INTRODUCING THE MUTANTS



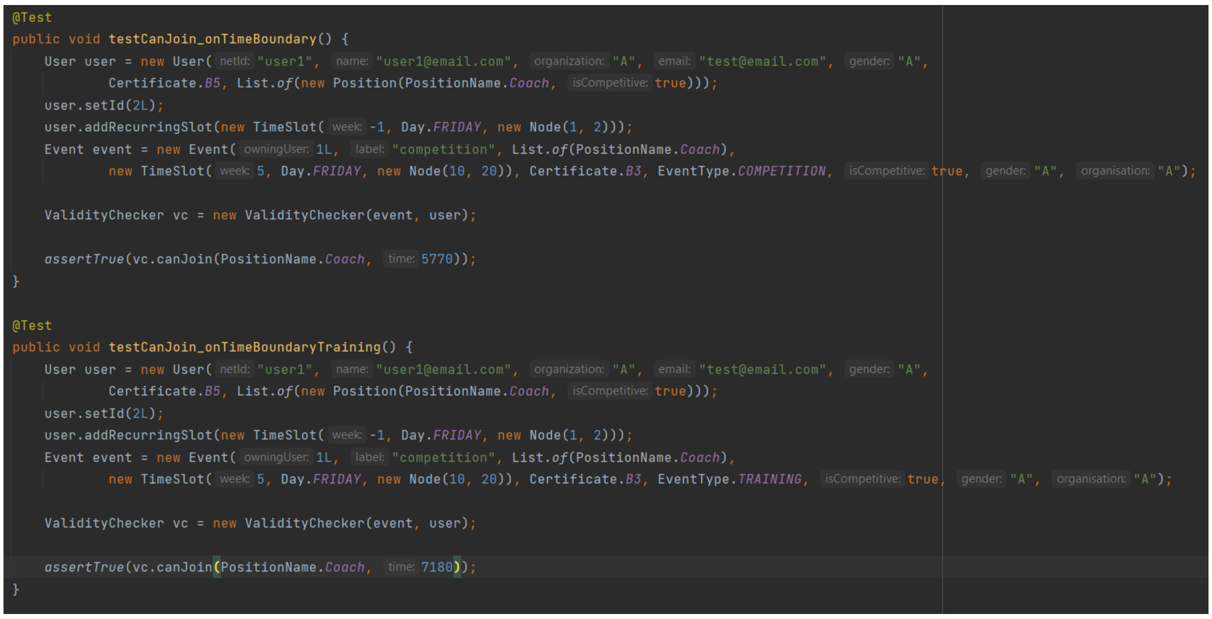
```
boolean checkIfOnTime = event.getType() == EventType.COMPETITION
        ? (time + 1440L) % 10080L >= eventTime : (time + 30L) % 10080L >= eventTime;

✓ Tests passed: 16 of 16 tests – 56 ms
```

AFTER INTRODUCING THE TESTS



```
boolean checkIfOnTime = event.getType() == EventType.COMPETITION
        ? (time + 1440L) % 10080L >= eventTime : (time + 30L) % 10080L >= eventTime;

⊗ Tests failed: 2, passed: 16 of 18 tests – 60 ms
```

THE TESTS



```
@Test
public void testCanJoin_onTimeBoundary() {
    User user = new User( netId: "user1",  name: "user1@email.com",  organization: "A",  email: "test@email.com",  gender: "A",
            Certificate.B5, List.of(new Position(PositionName.Coach,  isCompetitive: true)));
    user.setId(2L);
    user.addRecurringSlot(new TimeSlot( week: -1, Day.FRIDAY, new Node(1, 2)));
    Event event = new Event( owningUser: 1L,  label: "competition", List.of(PositionName.Coach),
            new TimeSlot( week: 5, Day.FRIDAY, new Node(10, 20)), Certificate.B3, EventType.COMPETITION,  isCompetitive: true,  gender: "A",  organisation: "A");

    ValidityChecker vc = new ValidityChecker(event, user);

    assertTrue(vc.canJoin(PositionName.Coach,  time: 5770));
}

@Test
public void testCanJoin_onTimeBoundaryTraining() {
    User user = new User( netId: "user1",  name: "user1@email.com",  organization: "A",  email: "test@email.com",  gender: "A",
            Certificate.B5, List.of(new Position(PositionName.Coach,  isCompetitive: true)));
    user.setId(2L);
    user.addRecurringSlot(new TimeSlot( week: -1, Day.FRIDAY, new Node(1, 2)));
    Event event = new Event( owningUser: 1L,  label: "competition", List.of(PositionName.Coach),
            new TimeSlot( week: 5, Day.FRIDAY, new Node(10, 20)), Certificate.B3, EventType.TRAINING,  isCompetitive: true,  gender: "A",  organisation: "A");

    ValidityChecker vc = new ValidityChecker(event, user);

    assertTrue(vc.canJoin(PositionName.Coach,  time: 7180));
}
```

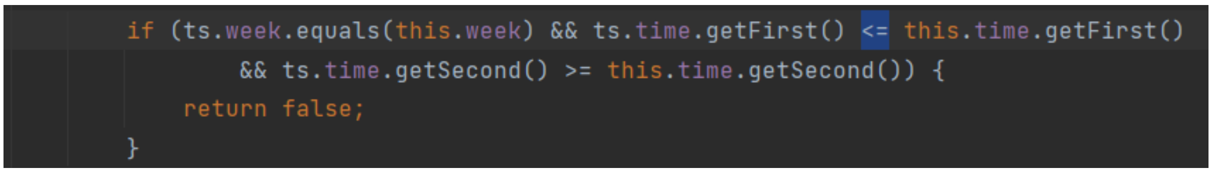
TimeSlot

Commit with the tests can be found at:
https://gitlab.ewi.tudelft.nl/cse2115/2022-2023/SEM29a/-/commit/6292ced54c2b745ed375ee717bdc3e04cea40827?merge_request_iid=97

The purpose of the *TimeSlot* class is to hold time slots for events and users and provide some logic for managing these time slots. In this class, we decided to look at the *matchSchedule* method, which tests whether a given user schedule has a free time slot at the time of the current time slot. This function aids in matching events to users, which makes it a critical part of our application.

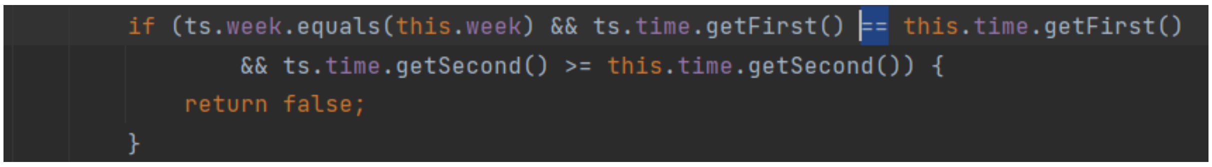
We realized that introducing a mutant in the if statement testing whether a time slot intersecting with the current time slot has been removed from the user schedule does not make our test suite fail, so we introduced a new test for it.

BEFORE



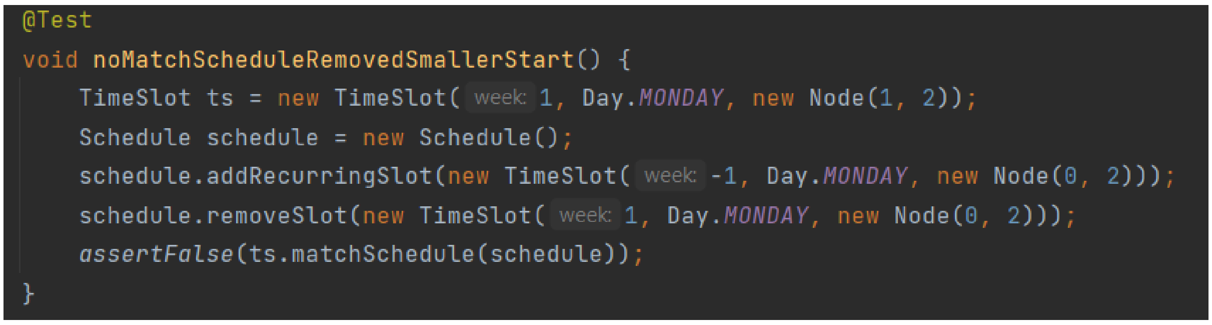
```
if (ts.week.equals(this.week) && ts.time.getFirst() <= this.time.getFirst()
        && ts.time.getSecond() >= this.time.getSecond()) {
    return false;
}
```

INTRODUCING THE MUTANT



```
if (ts.week.equals(this.week) && ts.time.getFirst() == this.time.getFirst()
        && ts.time.getSecond() >= this.time.getSecond()) {
    return false;
}
```

THE TEST



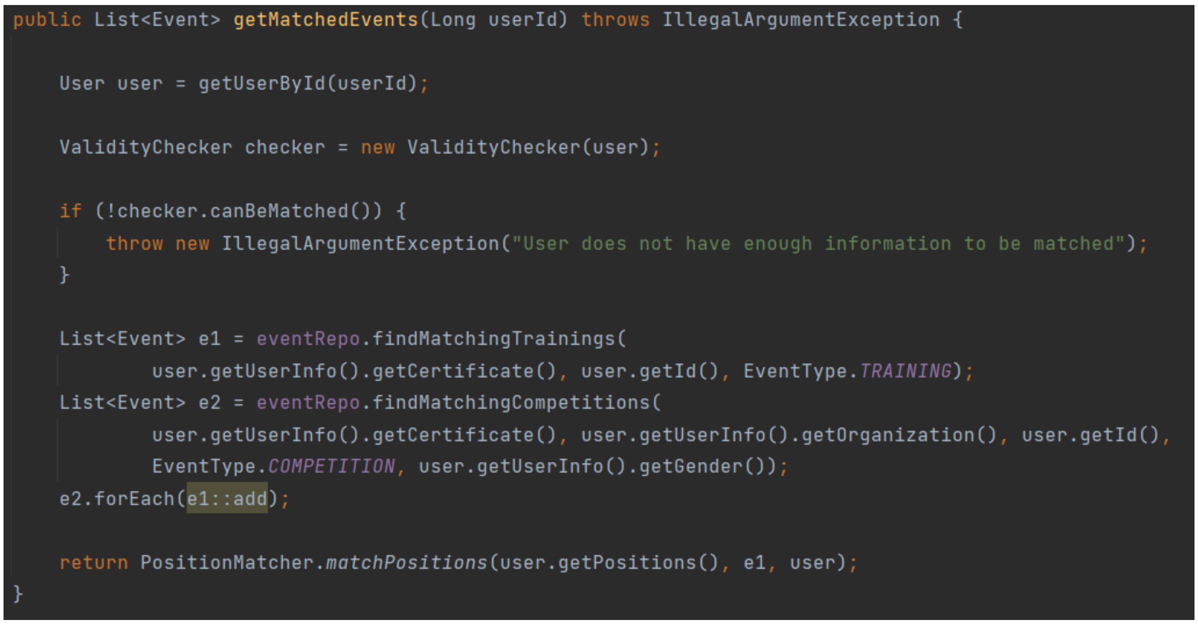
```
@Test
void noMatchScheduleRemovedSmallerStart() {
    TimeSlot ts = new TimeSlot( week: 1, Day.MONDAY, new Node(1, 2));
    Schedule schedule = new Schedule();
    schedule.addRecurringSlot(new TimeSlot( week: -1, Day.MONDAY, new Node(0, 2)));
    schedule.removeSlot(new TimeSlot( week: 1, Day.MONDAY, new Node(0, 2)));
    assertFalse(ts.matchSchedule(schedule));
}
```

EventService

Commit with the test can be found at:
https://gitlab.ewi.tudelft.nl/cse2115/2022-2023/SEM29a/-/commit/ed568f155b94c5f402afb43c1ae2b6ace8af85d2?merge_request_iid=98

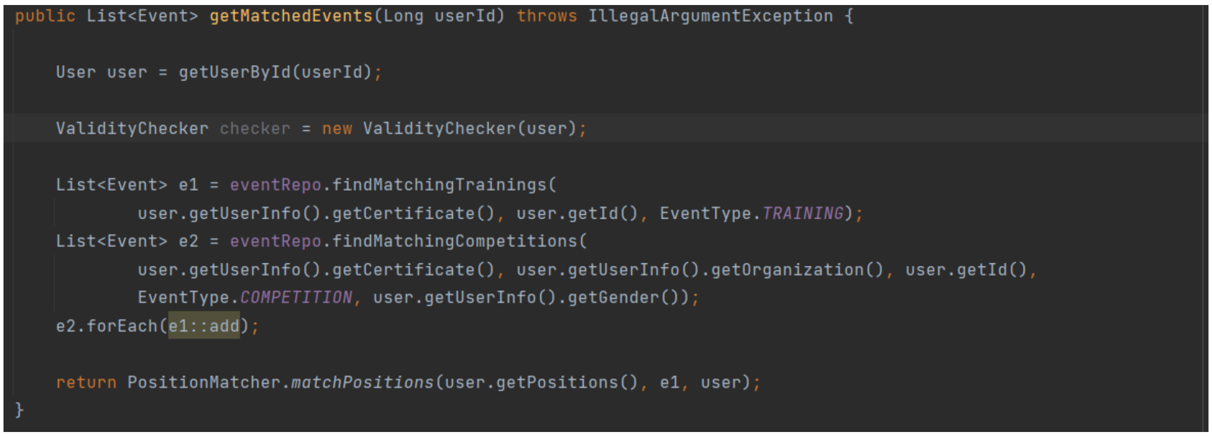
The purpose of the *EventService* class is to provide the underlying functionality of the event microservice. In this class we decided to look at the *getMatchedEvents* method, which returns all the events a user can be matched to, therefore contributing to a core part of the user experience. We realized that removing the check that tests whether a user has a valid profile is not caught by our test suite, so we added a test for it.

BEFORE



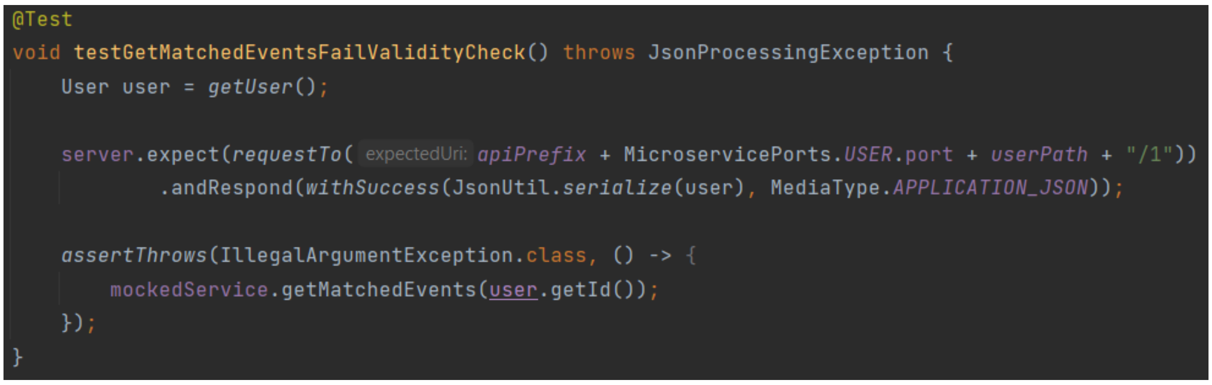
```java
public List<Event> getMatchedEvents(Long userId) throws IllegalArgumentException {

    User user = getUserById(userId);

    ValidityChecker checker = new ValidityChecker(user);

    if (!checker.canBeMatched()) {
        throw new IllegalArgumentException("User does not have enough information to be matched");
    }

    List<Event> e1 = eventRepo.findMatchingTrainings(
            user.getUserInfo().getCertificate(), user.getId(), EventType.TRAINING);
    List<Event> e2 = eventRepo.findMatchingCompetitions(
            user.getUserInfo().getCertificate(), user.getUserInfo().getOrganization(), user.getId(),
            EventType.COMPETITION, user.getUserInfo().getGender());
    e2.forEach(e1::add);

    return PositionMatcher.matchPositions(user.getPositions(), e1, user);
}
```

INTRODUCING THE MUTANT



```java
public List<Event> getMatchedEvents(Long userId) throws IllegalArgumentException {

    User user = getUserById(userId);

    ValidityChecker checker = new ValidityChecker(user);

    List<Event> e1 = eventRepo.findMatchingTrainings(
            user.getUserInfo().getCertificate(), user.getId(), EventType.TRAINING);
    List<Event> e2 = eventRepo.findMatchingCompetitions(
            user.getUserInfo().getCertificate(), user.getUserInfo().getOrganization(), user.getId(),
            EventType.COMPETITION, user.getUserInfo().getGender());
    e2.forEach(e1::add);

    return PositionMatcher.matchPositions(user.getPositions(), e1, user);
}
```

THE TEST



```java
@Test
void testGetMatchedEventsFailValidityCheck() throws JsonProcessingException {
    User user = getUser();

    server.expect(requestTo( expectedUri: apiPrefix + MicroservicePorts.USER.port + userPath + "/1"))
            .andRespond(withSuccess(JsonUtil.serialize(user), MediaType.APPLICATION_JSON));

    assertThrows(IllegalArgumentException.class, () -> {
        mockedService.getMatchedEvents(user.getId());
    });
}
```