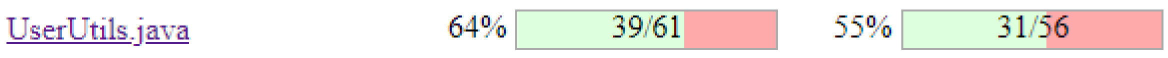
# Task 2: Manual Mutation Testing

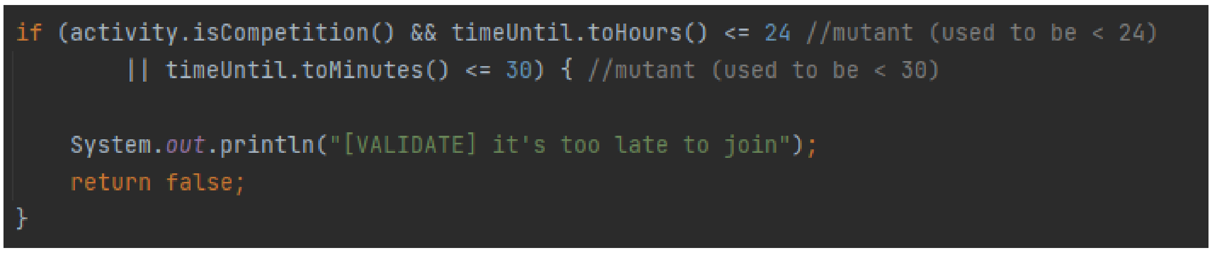
The planning microservice is chosen to do the manual mutation testing. The purpose of the application is matching rowers to a suitable activity. Planning microservice is designed to handle all matching functionalities. Therefore, it has a lot of the critical classes coping with the activity filtering.

## I. UserUtils



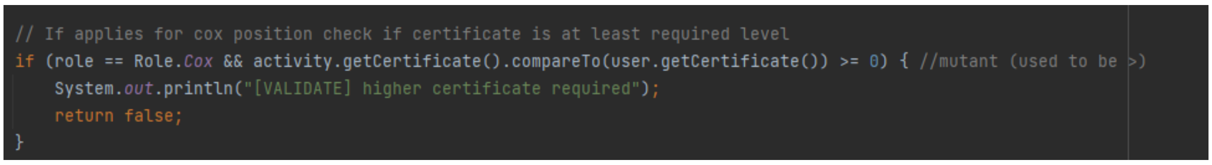
| | | | | |
|---|---|---|---|---|
| UserUtils.java | 64% | 39/61 | 55% | 31/56 |

UserUtils class handles the logic which decides whether a user has required criteria for the activity. It has CC of 19 so a lot of conditions that may have broken boundaries, that's why we introduced mutants for certain boundaries.

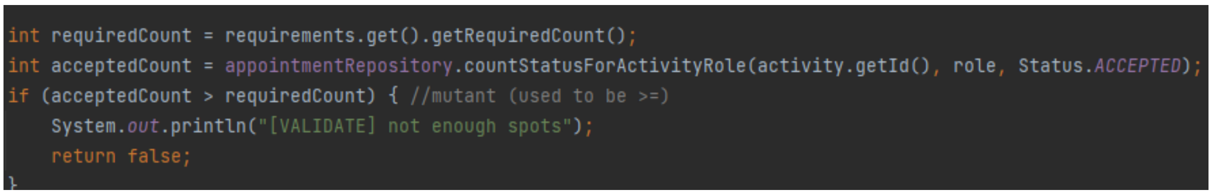


```java
if (activity.isCompetition() && timeUntil.toHours() <= 24 //mutant (used to be < 24)
        || timeUntil.toMinutes() <= 30) { //mutant (used to be < 30)

    System.out.println("[VALIDATE] it's too late to join");
    return false;
}
```



```java
// If applies for cox position check if certificate is at least required level
if (role == Role.Cox && activity.getCertificate().compareTo(user.getCertificate()) >= 0) { //mutant (used to be >)
    System.out.println("[VALIDATE] higher certificate required");
    return false;
}
```

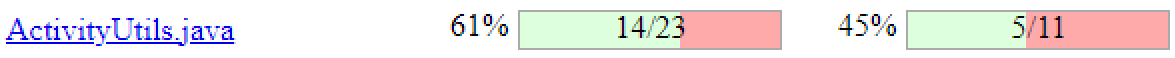


```java
int requiredCount = requirements.get().getRequiredCount();
int acceptedCount = appointmentRepository.countStatusForActivityRole(activity.getId(), role, Status.ACCEPTED);
if (acceptedCount > requiredCount) { //mutant (used to be >=)
    System.out.println("[VALIDATE] not enough spots");
    return false;
}
```

It's added in two commits:

- https://gitlab.ewi.tudelft.nl/cse2115/2022-2023/SEM30a/-/merge_requests/54/diffs?commit_id=b71c52531bbd9275979b05f311d025cf65979f64
- https://gitlab.ewi.tudelft.nl/cse2115/2022-2023/SEM30a/-/merge_requests/54/diffs?commit_id=137ce574f677ef6605a5eac6cffbb09e22a976bb

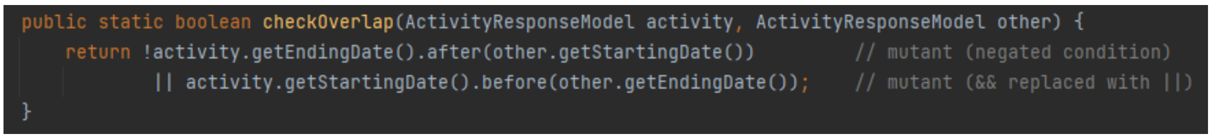
## II. ActivityUtils



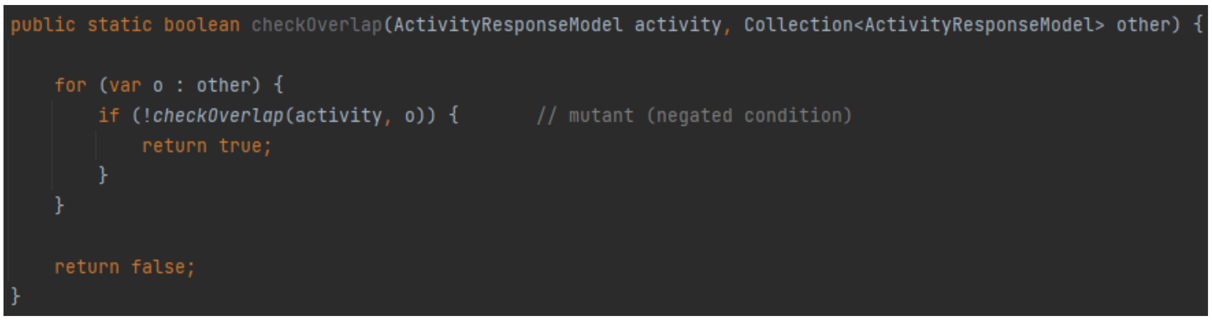
| | | | | |
|---|---|---|---|---|
| ActivityUtils.java | 61% | 14/23 | 45% | 5/11 |

ActivityUtils class handles checking various important conditions related to activities: checking if activities overlap, checking if the user owns the activity and retrieving activities with certain status.

The following manual mutations were created:



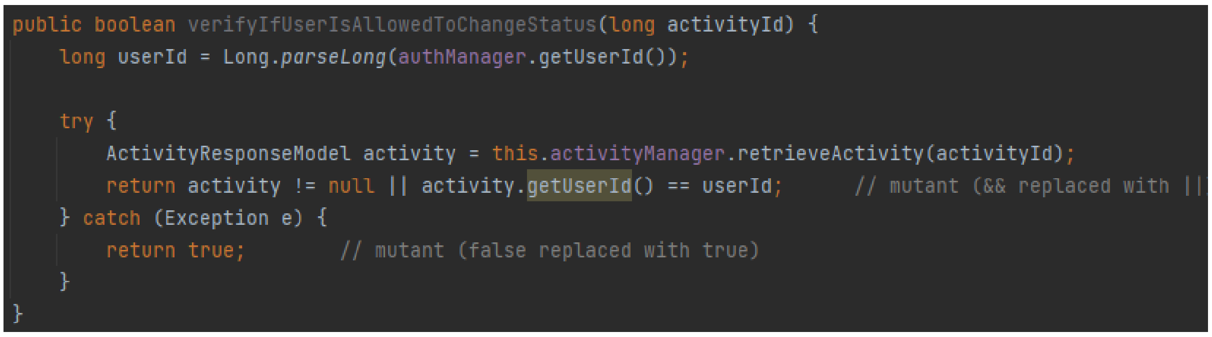
```java
public static boolean checkOverlap(ActivityResponseModel activity, ActivityResponseModel other) {
    return !activity.getEndingDate().after(other.getStartingDate())         // mutant (negated condition)
            || activity.getStartingDate().before(other.getEndingDate());    // mutant (&& replaced with ||)
}
```



```java
public static boolean checkOverlap(ActivityResponseModel activity, Collection<ActivityResponseModel> other) {

    for (var o : other) {
        if (!checkOverlap(activity, o)) {       // mutant (negated condition)
            return true;
        }
    }

    return false;
}
```



```java
public boolean verifyIfUserIsAllowedToChangeStatus(long activityId) {
    long userId = Long.parseLong(authManager.getUserId());

    try {
        ActivityResponseModel activity = this.activityManager.retrieveActivity(activityId);
        return activity != null || activity.getUserId() == userId;      // mutant (&& replaced with ||
    } catch (Exception e) {
        return true;         // mutant (false replaced with true)
    }
}
```

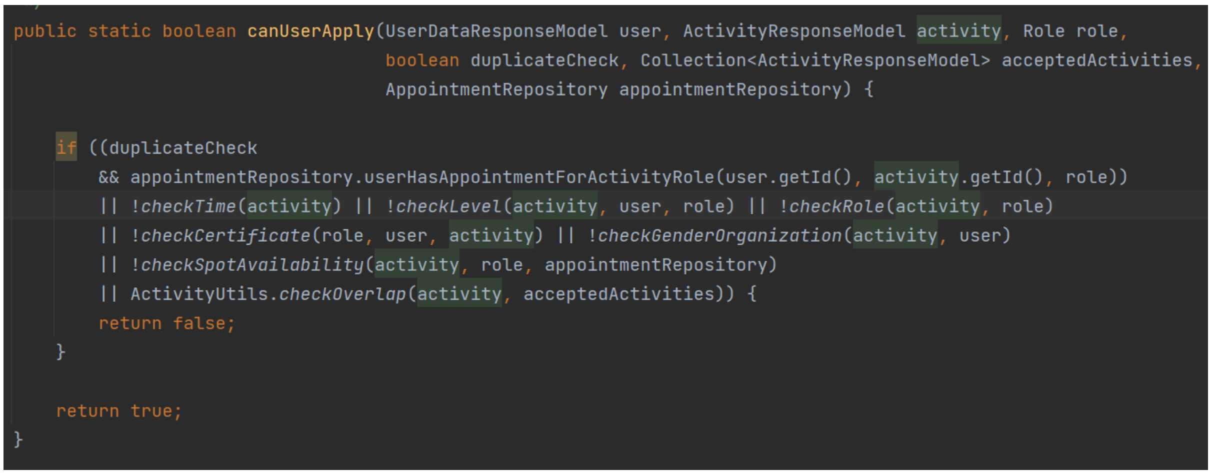
The mutations were tested separately (not all at the same time) in order to catch each one individually.

Appropriate tests were added in the following commit in order to catch these mutations:
https://gitlab.ewi.tudelft.nl/cse2115/2022-2023/SEM30a/-/commit/487e6eea1665a0ec3846f8a5d81c691cdfda2086

## III. UserEligibilityForActivity

| Name | Line Coverage | | Mutation Coverage | |
|---|---|---|---|---|
| UserEligibilityForActivity.java | 0% | 0/48 | 0% | 0/57 |

UserEligibilityForActivity contains the core logic of checking if a user can apply to an activity. When a user applies for an activity, the method *canUserApply* in the class is called to check the user's eligibility. It is an essential part of the application as it ensures that users are matched to a suitable activity.



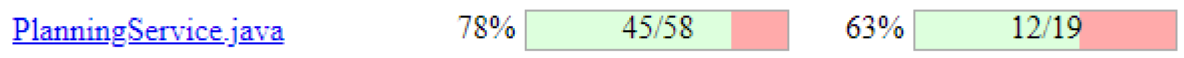
```java
public static boolean canUserApply(UserDataResponseModel user, ActivityResponseModel activity, Role role,
                                   boolean duplicateCheck, Collection<ActivityResponseModel> acceptedActivities,
                                   AppointmentRepository appointmentRepository) {

    if ((duplicateCheck
        && appointmentRepository.userHasAppointmentForActivityRole(user.getId(), activity.getId(), role))
        || !checkTime(activity) || !checkLevel(activity, user, role) || !checkRole(activity, role)
        || !checkCertificate(role, user, activity) || !checkGenderOrganization(activity, user)
        || !checkSpotAvailability(activity, role, appointmentRepository)
        || ActivityUtils.checkOverlap(activity, acceptedActivities)) {
        return false;
    }

    return true;
}
```

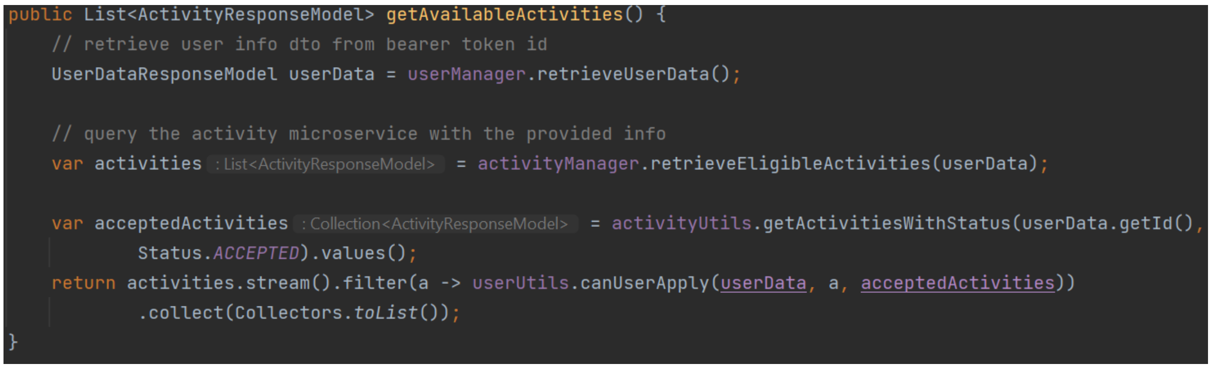
The mutant is created in the *canUserApply* method. As seen in the above image, It collects all the methods that individually check each criteria of the activity into conditional operators and returns a boolean. The mutant created is conditional operator replacement. The negation operator of each method call is eliminated individually. For instance, the negation of the *checkTime* is removed while all others are kept. The tests catching these mutants are added in the following commit: https://gitlab.ewi.tudelft.nl/cse2115/2022-2023/SEM30a/-/commit/9b7ca5ac4e6913499fae2e9cca4c450df6dc1c44.

## IV. PlanningService



| | | | | |
|---|---|---|---|---|
| PlanningService.java | 78% | 45/58 | 63% | 12/19 |

The *PlanningService* class manages appointments by using all the utility classes of the planning microservice. Users are matched to activities via appointment objects which keeps all the information of the user and activity corresponding to the match. Thus, it is one of the most important classes of the planning microservice.



```
public List<ActivityResponseModel> getAvailableActivities() {
    // retrieve user info dto from bearer token id
    UserDataResponseModel userData = userManager.retrieveUserData();

    // query the activity microservice with the provided info
    var activities : List<ActivityResponseModel> = activityManager.retrieveEligibleActivities(userData);

    var acceptedActivities : Collection<ActivityResponseModel> = activityUtils.getActivitiesWithStatus(userData.getId(),
            Status.ACCEPTED).values();
    return activities.stream().filter(a -> userUtils.canUserApply(userData, a, acceptedActivities))
            .collect(Collectors.toList());
}
```

Within the class, the *getAvailableActivities* function returns all eligible activities to the users. Rowers select activities from the list given by this method, so it is critical in providing the suitable activities. The mutant created is negating the filtering condition in the return statement and tests are added in the following commit: https://gitlab.ewi.tudelft.nl/cse2115/2022-2023/SEM30a/-/commit/40510d8731c5ec8961de0cbdfdef7390c3c9a883.