Assignment 3: 14a

Task 2: Manual Mutation Testing

The resource pool microservice is a core domain of our program. It takes care of all of the functionality for resource pools as well as faculties. Furthermore, it handles the scheduling of requests per day and faculty. Since the program is supposed to be a scheduler, it's safe to say that this microservice is a core domain.
For all classes, we created a duplicate method that contains a mutant. Those methods are called methodXMutated. In each test class, there is a newly created test that tests both the unmutated version of the method, as well as the mutated one.

RpManagementService: contributeNode

The Resource Pool Management Service is mainly responsible for creating faculties and for allowing nodes to be contributed to and deleted from a given faculty. It is for this reason that the class can be described as critical since, without it, the system would not have any faculties to distribute resources to.
The method chosen to perform mutation testing on was contributeNode. The reason for this is that a key requirement is to be able to expand the system's resources through the use of user-contributed nodes.
In order to strengthen the testing suite for this class, a bug will be injected for the purposes of seeing if the current testing suite is able to detect it. The chosen bug will be a statement deletion bug. More specifically, the line that is responsible for re-saving the faculty with the updated node will be omitted in the mutation. The justification for this choice is that it is a trivial mistake that could be made due to a lack of attention during implementation.
The test case to remove the mutant involved creating a new faculty with empty initial node resources. The contributeNodeMutated method is then called which fails to save the updated node resources to the faculty. This means that when the faculty's node resources are checked again, they are still zero which is to be expected from the mutation. In order for the test to successfully remove the mutant, an assertion that an AssertionFailedError was thrown is required. The commit that contains the discussed changes can be found here.

DailyScheduleService: releaseAllResourcesToFreePool

This class takes care of all of the business logic for the DailySchedule. Since the DailySchedule stores the requests for a day per faculty, it is a critical class for the scheduler. Releasing all the resources to the free pool is core functionality since it takes care of one big requirement out of the scenario: "6 Hours before the day starts requests no longer require approval to be run on the supercomputer and requests should automatically be accepted on a first come first serve basis until 0 resources are available". The "releasingAllResources-ToFreePool" method makes sure that six hours before the start of the day, all unused resources are released to the free pool, so they can be used for automatic approval.
The type of bug we used in this method is a Relational Operator Replacement (ROR). On line 274 in the DailyScheduleService, there is an if statement with a not equals operator. In the mutated test, we changed this to an equals operator. This line checks whether the resource pool to release resources for is not the free pool, because the resources for the free pool can not be released. In order for the test to kill the mutant, it checks whether the ReleaseResourcesException is thrown, with the message that resources can not be released for the free resource pool. The mutated method had to be changed a bit more since in the regular method the exception is only cought and not thrown. In order to test it, we made sure that the method actually throws the exception. Commit can be found here.

Resources: subtract

The Resources class stands at the base of all scheduling operations. Every time a request is submitted, scheduled, or rejected, several Resources instances are created and updated. For example, when a request is approved, the schedule of the faculty needs to reflect that and reduce its available resources - this is done by using the subtract function in the Resources class that takes two Resources A and B, and returns A - B. This might seem trivial at first - take the CPU, GPU and Memory of A and subtract from it the CPU, GPU, and Memory of B. However, due to the extremely similar syntax - resources.getGpu(), resources.getCpu(), the minuses and pluses - it is very easy to make a 1 symbol mistake that can include a bug at the very base level of the application that will propagate to its higher levels. This example is not imaginary, but something we actually experienced during development that caused a serious problem. While that bug is now fixed, a very similar bug would be to instead of subtracting one of the subcategories of Resources, to add them together. This bug would have the same magnitude of effect as the other one and is as easy to accidentally implement since it is also only a 1 symbol change (from a “-” to a “+”) in an already syntax-heavy operation. Due to these reasons, we decided to manually inject it. After running our whole test suite with the mutated method, it passed and did not kill the mutant. To fix this, we implemented a test in ResourcesUnitTests that performs the subtraction operations and checks their validity. After implementing this test, the test suite broke and successfully killed the newly introduced mutant. We put back the original, bug-free subtract() method in its place, and named the mutated one subtractMutated(). It can be found in the Resources class. As for the test that catches the bug, we left it to test the normal subtract() method but also made another instance that calls subtractMutated(). Since it kills the mutant and fails, we changed the assertEquals statement to assertNotEquals, so that the test passes. If one wants to check that it does indeed kill the mutant, all one needs to do is swap the assertNotEquals to assertEquals (as it is in the test of the working bug-free method). The 2 commits made can be found here and here.

DistributionService: saveDistribution

In our system, the SysAdmin can manually update the percentage distribution of the base resources of the supercomputer. The business logic for updating this distribution happens in this class. This makes the class very critical for the application to function because otherwise, all resources would always be in the free pool.
The method saveDistribution is a core part of the system because this method saves temporary percentage resource distribution and converts it to absolute resource numbers. Without this method functioning the whole system of updating the distribution wouldn’t function because the created distribution would never or wrongly be saved to the system. For example, if everything in this class works perfectly but the method to save the distribution doesn’t properly check if all the percentages add up to 100 the system will just lose or gain resources out of nowhere, which is of course ridiculous.
As you can see in the explained example above, this is a big threat to the system. This is why we chose to create a method where we forgot to call the validateInput method. This type of mutant is called a statement deletion bug. The test we used to kill this mutant is called killSaveDistributionMutant. In this test, we add two distributions, but we make the GPU add up to 120 instead of 100. This makes it so that our normal method throws the “ResourceSumNotCorrectException” exception with the message “gpu”. But we also call the mutated method at the bottom and you can see that the method just pretends everything is fine and saves the new distribution to the system, creating more GPU resources out of nowhere. Commit can be found here.