JUnit testing report

Group Guillemot

TEST RATIONALE AND EXPECTED RESULTS

```
@Test
public void whenConstructionThenParentNotPresent() {
    assertEquals("Parent was present at construction.", false, m.getParentPresent());
}
```

The first test ensures that parents aren't granted access while the construction is still taking place, we check this with the result from the getter 'getParentPresent()' which returns a Boolean confirming their status, if the result is different from what we expect, in this case 'true' then the message "Parent was present at construction" is displayed. When run, this test passes and so we know the code is working as desired.

```
@Test
public void whenConstructionThenClosed() {
    assertEquals("Room was open at Construction.", false, m.isOpen());
}
```

Most of our tests work in a similar manner so that successful and failure results are easily distinguishable and recognizable. In the test shown above we check that the meeting room is closed while construction is ongoing. Once again, we compare the Boolean result of the operation against the expected result and write a clear message so that in case the test fails the programmer will know exactly which part is not working.

```
@Test
public void whenCloseThenClosed() {
    m.open();
    m.close();
    assertEquals("Room was open after closing.", false, m.isOpen());
}
```

In this use case we are checking that whenever the 'close()' function is called, the system is actually closed. In order to do this we call the operation 'isOpen()' and compare its Boolean result to our expected one in order to know if the function was successful

```
@Test
public void testSetSchedule() {
    // Tests with negatives:
    //Negative meeting length should not occur in normal use, but when used should lead to each

    m1.setSchedule(st, 1, -1);

    assertEquals(true, m1.getSchedule().size() == 1
            && m1.getSchedule().get(0).getStartTime().compareTo(new DateAndTime(4, 7)) == 0);

    m1.setSchedule(st, 2, -1);

    assertEquals(true, m1.getSchedule().size() == 2
            && m1.getSchedule().get(1).getStartTime().compareTo(new DateAndTime(4, 6)) == 0);
    // A negative number of meeting gets treated as scheduling no meetings.
    m1.setSchedule(st, -1, 1);

    assertEquals(true, m1.getSchedule().size() == 0);

    // test with zeroes:
    //When scheduling 0 meetings, the size of a schedule should be 0
    m1.setSchedule(st, 0, 1);

    assertEquals(true, m1.getSchedule().size() == 0);
    //When Scheduling meetings of length 0, they should all start at the same start time
    m1.setSchedule(st, 2, 0);
```

The selected comment states: Negative meeting length should not occur in normal use, but when used should lead to each meeting in the schedule starting earlier than the last

```
    assertEquals(true, m1.getSchedule().size() == 2
            && m1.getSchedule().get(1).getStartTime().compareTo(new DateAndTime(4, 7)) == 0);

    // test with positives (control for normal result)
    // This is the only case that should occur during normal use
    m1.setSchedule(st, 2, 5);
    assertEquals(true, m1.getSchedule().size() == 2
            && m1.getSchedule().get(1).getStartTime().compareTo(new DateAndTime(4, 12)) == 0);
}
```

In this test we now check that the ‘setSchedule()’ method is only taking valid inputs for meeting number and the duration of the meeting.

```
@Test
public void givenOpenAndAppointmentNowWhenJoinThenEnter() {
    m.setSchedule(st, 1, 5);
    m.makeAppointment(p, 0);
    m.open();
    m.join(p);
    assertEquals("Parent was not able to join open meeting room for their appointment.", true,
            m.getParentPresent());
}
```

In this test we ensure that the parent is allowed to join a meeting provided that the meeting room is open, they have an appointment and that the current time is the time for that meeting. To do this, we set the schedule for this parent and make the appointment for them, then open the system and attempt to join the meeting. After all this we check if the parent is able to join the meeting by comparing the result of ‘getParentPresent()’ to the expected result of the parent being present in the meeting room(true).

```
@Test
public void givenClosedWhenJoinThenRefused() {
    m.setSchedule(st, 1, 5);
    m.makeAppointment(p, 0);
    m.open();
    m.close();
    m.join(p);
    assertEquals("Parent was able to join closed meeting room .", false, m.getParentsWaitingInLobby().contains(p));
}
```

This use case refers to a similar situation except that the meeting room is closed. To do this we follow the same preparation as before we close the operation and then have the parent try to join. So then we access the list of parents waiting in the lobby and then check whether they are among that list. If so, then all is working as it should.

```
@Test
public void givenAppointmentFutureWhenJoinThenParentInLobby() {
    m.setSchedule(st, 2, 5);
    m.makeAppointment(p, 1);
    m.open();
    m.join(p);
    assertEquals("Parent was able to join despite appointment being in the future meeting room .", false,
            m.getParentPresent());
}
```

In this case we make sure that if the parent tries to join a meeting too early, they are instead directed to the lobby. To do this we set a meeting that is further along than the current time using ‘setSchedule()’ and then proceed to attempt to join the meeting. After this we once again compare the presence of the parent to the expected result. And provide a clear message in case the test fails.

```
@Test
public void givenNoAppointmentWhenJoinThenRejected() {
    m.setSchedule(st, 5, 8);
    m.open();
    m.join(p);

    assertEquals("Parent was able to join meeting room without an appointment.", false, m.getParentPresent());
}
```

We are now checking that a parent is not allowed to enter a meeting without an appointment. To do this we still set the schedule but don’t make an appointment for the parent and then attempt to join a meeting anyway. We use the same test as before to verify the parents’ presence at the meeting, which should be false since they do not have an appointment.

```
@Test
public void givenAppointmentPastWhenJoinThenRejected() {
    m.setSchedule(st, 5, 8);
    m.makeAppointment(p, 1); // change current time slot to one after the booked one
    m.open();
    m.join(p);
    assertEquals(
            "Parent was able to join an appointment that was scheduled for before the current slot in the meeting room.",
            false, m.getParentPresent());
}
```

The next use case is one where the parent attempts to join a meeting that has already passed. In order to do this, we set the earliest appointment and then make sure that the current time

slot is after that one. Then we proceed as before attempting to join a meeting and checking if the parent was successful in doing so or not, in this case we expect a negative result.

```
@Test
public void givenInMeetingRoomWhenLeaveThenNoParentPresent() {
    m.setSchedule(st, 2, 5);
    m.makeAppointment(p, 1);
    m.open();
    m.join(p);
    m.leave(p);
    assertEquals("Parent is still present at the appointment in the meeting room.", false, m.getParentPresent());
}
```

In the case where the parent has chosen to leave the meeting before it is over, by using the 'leave()' function while inside a meeting room. To do this we follow the steps to schedule and attend the meeting and then call the 'leave()' function. After this has been done, we once again check the presence of the parent in the meeting and if it is false as expected then the test is successful.

```
@Test
public void givenInLobbyWhenLeaveThenRemoved() {
    m.setSchedule(st, 2, 5);
    m.open();
    m.join(p);
    m.leave(p);
    assertEquals("Parent is still in the lobby.", false, m.getParentsWaitingInLobby().contains(p));
}
```

In this test case we check that when the parent uses the function 'leave()' while they're in the lobby, they are removed from the lobby. In this test we have the parent open the system but not have an appointment so they never leave the lobby, then they use the function 'leave()' which should remove them from there. To check that this is done correctly we access the list of parents waiting and then check if they are still there or not. If the result is negative, then the parent has been successfully removed from the lobby and the test passes.

```
@Test
public void givenInMeetingRoomWhenAllChangeThenRemoved() {
    m.setSchedule(st, 2, 5);
    m.makeAppointment(p, 0);
    m.open();
    m.join(p);
    m.allChange();
    assertEquals("Parent is still in the meeting room.", false, m.getParentPresent());
}
```

We are now testing the use case where the method 'allChange' is called while the parent is in the meeting room. This should remove the parent from the meeting. In order to check this, we once again follow the procedure to make an appointment and join the meeting, then we call the 'allChange()' method. To make sure this is working properly, we check for the parents' presence and if they are not in the meeting room the test has passed.

```
@Test
public void givenWaitingAndAppointmentNextWhenAllChangeThenEnter() {
    m.setSchedule(st, 2, 5);
    m.makeAppointment(p, 1);
    m.open();
    m.join(p);
    m.allChange();
    assertEquals("Parent is not in the meeting room.", true, m.getParentPresent());
}
```

We are now checking that when a parent is waiting in the lobby, their appointment is next and the 'allChange()' function is called, they are automatically moved into the meeting room. In order to do this we follow the procedure up until the point where the parent is in the lobby and then call the 'allChange()' operation. After this is called, we check that the parent is in the meeting room and if they are the test has passed.

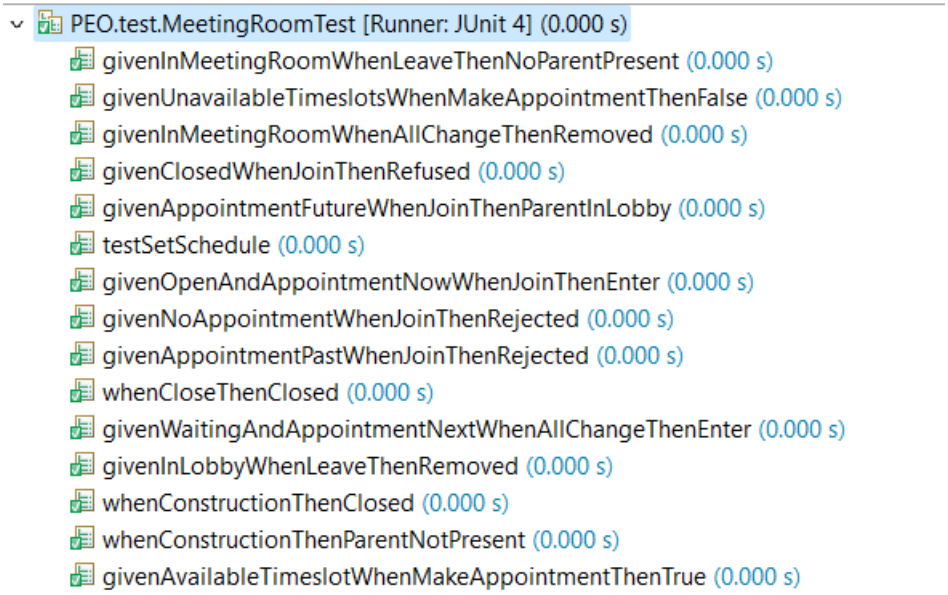
```
@Test
public void givenUnavailableTimeslotsWhenMakeAppointmentThenFalse() {
    m.setSchedule(st, 2, 5);
    m.makeAppointment(p, 1);
    assertEquals("Parent was able to book a time slot that is unavailable", false, m.makeAppointment(std, 1));
    System.out.println(m.getSchedule().get(1).getBookedFor());

}
```

In this use case we test that it is not possible for two parents to book the same time slot. In order to do this we create a schedule and make an appointment for one parent, then we test if a second parent can make that same appointment. If that isn't possible then the code is working as it should.

```
@Test
public void givenAvailableTimeslotWhenMakeAppointmentThenTrue() {
    m.setSchedule(st, 2, 5);
    assertEquals("Parent was not able to book a time slot that was available", true, m.makeAppointment(std, 1));
}
```

In our last test we checked that a parent can successfully book an appointment that is available. To do this we simply set a schedule and test if the 'makeAppointment()' operation is successful.

OBTAINED RESULTS



In conclusion we created this final set of use cases to prioritize and were able to get the desired results in all of them with no test failures. Since we created an appropriate working test for all major actions performed in the meeting room overall the testing was successful.