CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Exercise Session 3: Unit Testing

Gregory Gay
DIT636/DAT560 - February 11, 2026

Enter… The Planning System

- Everybody likes meetings.
  - Not true - but we need to book them.
- We don't want to double-book rooms or employees for meetings.
- System to manage schedules and meetings.

The Planning System

Offers the following high-level features:

1. Booking a meeting
2. Booking vacation time
3. Checking availability for a room
4. Checking availability for a person
5. Printing the agenda for a room
6. Printing the agenda for a person

Develop a Test Plan

In groups, come up with an informal “test plan”.

- Given the features and the code documentation, plan unit tests to ensure that these features can be performed without error.
- Omit scheduling, etc. - just come up with a list of test cases to create.

Food for Thought

- Try running the code!
  - Perform exploratory testing to test it at the system level.
- Think about normal and erroneous inputs/actions.
  - How many things can go wrong?
  - You will probably be able to add a normal meeting, but can you add a meeting for February 35th?
  - Try it out - you have the code.

Develop Unit Tests

- If a test is supposed to cause an exception to be thrown, make sure you check for that exception.
- Make sure that expected output is detailed enough to ensure that - if something is supposed to fail - that it fails for the correct reasons.
  - Use proper assertions.

Example - Adding Midsommar

Setup

Test Steps

```
@Test
public void testAddMeeting_holiday() {
    Calendar calendar = new Calendar();
    try {
        Meeting midsommar = new Meeting(6, 26, "Midsommar"); // Create holiday
        calendar.addMeeting(midsommar); // Add to calendar object.
        Boolean added = calendar.isBusy(6, 26, 0, 23); // Verify that it was added.
        assertTrue(added,"Midsommar should be marked as busy on the calendar");
    } catch(TimeConflictException e) {
        fail("Should not throw exception: " + e.getMessage());
    }
}
```

Input

Oracle

Oracle

Can you expose the faults?

Can you expose the faults?

1: getMeeting and removeMeeting perform no error checking on dates.

```
public Meeting getMeeting(int month, int day, int index){
    return occupied.get(month).get(day).get(index);
}

public void removeMeeting(int month, int day, int index){
    occupied.get(month).get(day).remove(index);
}
```

Can you expose the faults?

2: Calendar has a 13th month.

```
public Calendar(){
         occupied = new ArrayList<ArrayList<ArrayList<Meeting>>>();

         for(int i=0;i<=13;i++){
              // Initialize month
              occupied.add(new ArrayList<ArrayList<Meeting>>());
              for(int j=0;j<32;j++){
                   // Initialize days
                   occupied.get(i).add(new ArrayList<Meeting>());
              }
         }
}
```

Can you expose the faults?

3: November has 30 days.

Oh - and we just added a meeting to a day with a date that does not match that date.

```
occupied.get(11).get(30).add(new Meeting(11,31,"Day does not
exist"));
```

Can you expose the faults?

4: Used a >= in checking for illegal times. December no longer exists.

```
if(mMonth < 1 || mMonth >= 12){
          throw new TimeConflictException("Month does not
exist.");
}
```

Can you expose the faults?

5: We should be able to start and end a meeting in the same hour.

```
if(mStart >= mEnd){
    throw new TimeConflictException("Meeting starts before it 
ends.");
}
```

What Other Faults Can You Find?