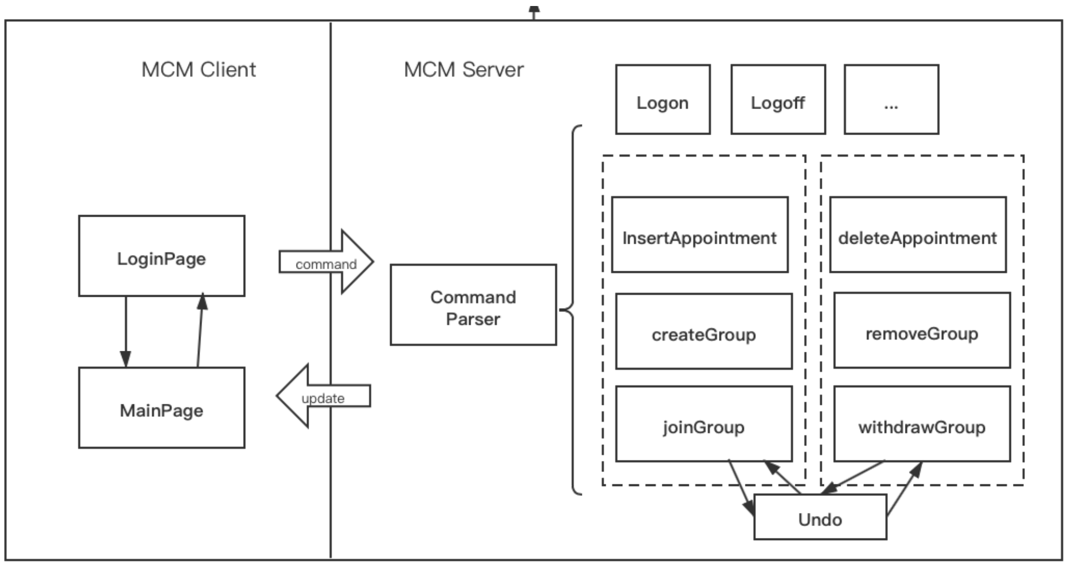
CSIT 5100 Assignment 1 Report

LIN Jialiang 20656855

1. Introduction

This report states the testing information of a Multi-User Calendar Manager (MCM) program. Firstly, the report would briefly illustrate the MCM system, which includes the server program and client program. Then the report would show the construction of testing cases as well as testing result (code coverage of MCM client program). Finally, the report would show some analysis of the testing.

2. Basic Structure of MCM



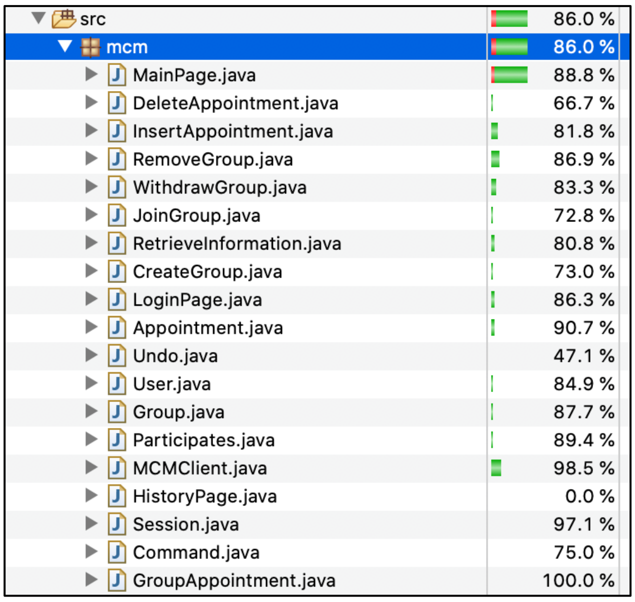
As the structure picture shows, MCM has two parts: MCM Client and MCM Server. MCM Client is responsible for associating with users, and users can operate on MCM Client graphical user interface. After receiving users' interactions, MCM Client send corresponding commands to MCM Server, which is responsible for handling these commands, such as inserting or deleting appointments, creating or joining groups, registering users, etc. Therefore, MCM Client is like the front-end of the whole program and MCM Server is the back-end.

3. Testing Result and Construction

Testing Result

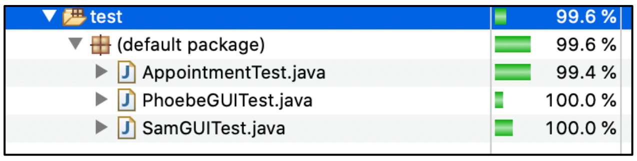
In MCM program, we construct test cases on MCM client part to test its functionalities. The test cases can achieve 89.5% statement coverage, 86.8% line coverage and 74.5% branch coverage as well. In terms of statement coverage, the test cases specifically achieve 86% coverage in MCMClient source code and 99.6% coverage in test codes.

The details of code coverage on each package are as followed:
For src folder:



src	86.0 %
mcm	86.0 %
MainPage.java	88.8 %
DeleteAppointment.java	66.7 %
InsertAppointment.java	81.8 %
RemoveGroup.java	86.9 %
WithdrawGroup.java	83.3 %
JoinGroup.java	72.8 %
RetrieveInformation.java	80.8 %
CreateGroup.java	73.0 %
LoginPage.java	86.3 %
Appointment.java	90.7 %
Undo.java	47.1 %
User.java	84.9 %
Group.java	87.7 %
Participates.java	89.4 %
MCMClient.java	98.5 %
HistoryPage.java	0.0 %
Session.java	97.1 %
Command.java	75.0 %
GroupAppointment.java	100.0 %

For test folder:



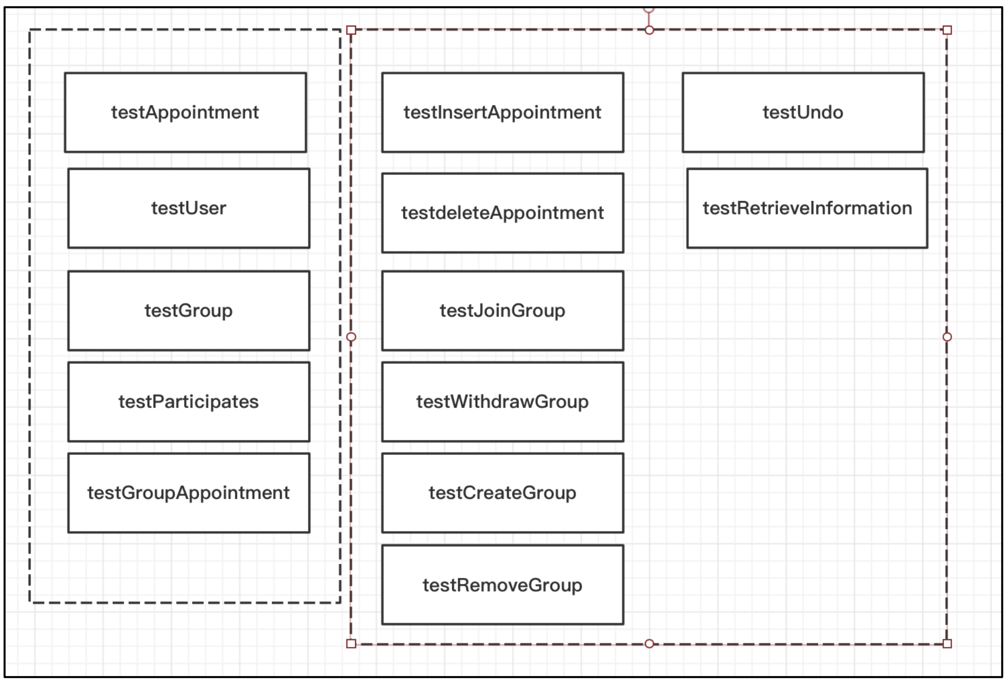
test	99.6 %
(default package)	99.6 %
AppointmentTest.java	99.4 %
PhoebeGUITest.java	100.0 %
SamGUITest.java	100.0 %

Testing Construction

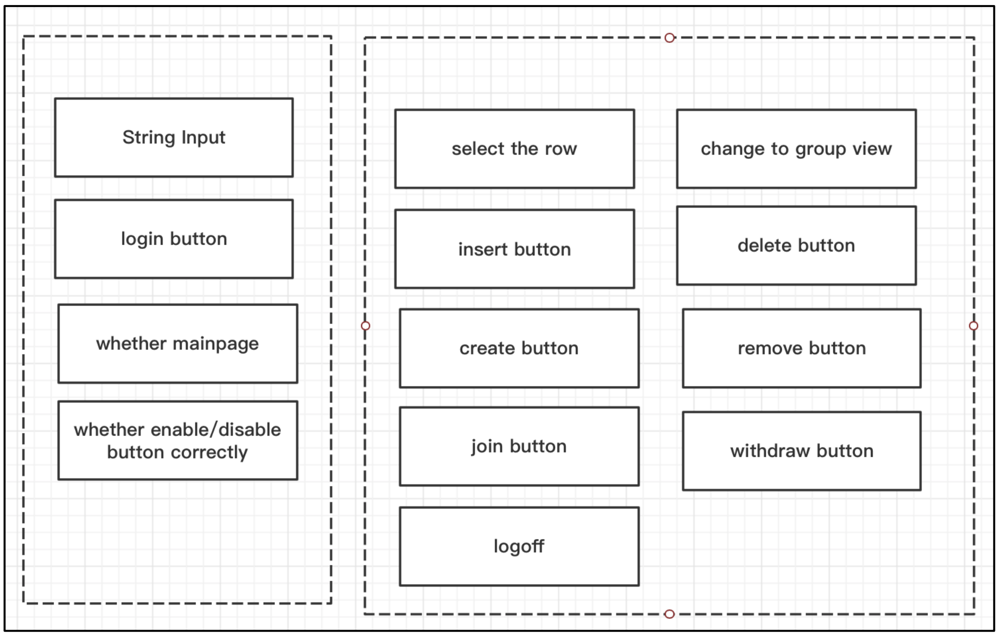
To test MCM Client, test cases should be written. We use Junit 5 to build up our test cases under the /test folder and execute them in Eclipse with Eclemma to evaluate the corresponding code coverage. The test cases that I constructed can be divided into two parts, one of which test the functionality of MCM program, other of which test the GUI component.

Firstly, I test the functionality of MCM, for example, insert/delete appointments, create/remove/join/withdraw groups, undo, as well as the corresponding class such as Appointment, Group, User, etc.

As the following picture shows, I generate several test methods to test different class based on the idea of testing the functionality of MCM. I test the basic meta class like Appointment, User, Group, etc. at first to make sure its corresponding methods are correct. Then I test the main functionality of MCM, like insert/delete appointments, etc.. In this process, testing become harder, because I need to come up more situations to achieve a higher coverage.



Secondly, I test whether the GUI components function well in MCM Client. This key of this part is to simulate the procedure of a user, to check whether the system give the correct response. The process can be treated as two parts as well, one of which is to check the GUI functions in loginpage and other of which is in mainpage. The testing is new to me since I need to mock the actual clicking or selecting or typing in, etc. But once I master the simulation procedure, the testing is not new to me, just to check whether the actual output is the same as our expected output. During this process, I also have found an interesting thing, which is concurrency I illustrated in *Analysis* part later.



4. Analysis from the testing

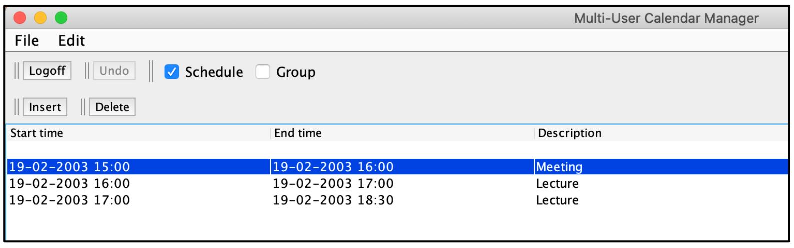
High Coverage

To achieve high coverage, we should not only test the GUI components of MCM Client, but also come up with test cases to test the functionality of MCM, which actually be called in MCM Server. Therefore, I build up test cases to simulate the GUI operations in terms of GUI testing. And I also construct test cases to test the functionality of MCM, which would not be called in MCM Client, thus in need to be tested separately. In order to cover more statements, I take more scenarios into consideration. There are illustrated in *Testing Construction* part and here I skip it.

Concurrency

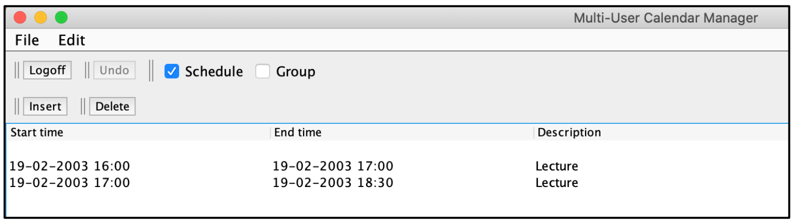
From my observation on MCM server and MCM client and test cases, they run concurrently so that we cannot know the running order of statement for sure. Take the following case as an example:

Here is the scenario: phoebe has 3 appointments and he should have 2 appointments after deleting the first one. Therefore, by following this scenario, we would get 3 elements in appointmentSet and 2 elements after mocking on clicking.

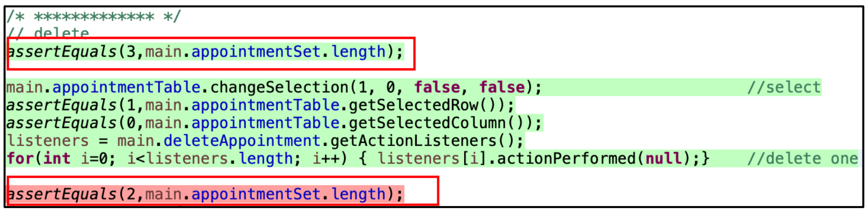


```
Start time          End time            Description
19-02-2003 15:00    19-02-2003 16:00    Meeting
19-02-2003 16:00    19-02-2003 17:00    Lecture
19-02-2003 17:00    19-02-2003 18:30    Lecture
```

We did have 3 items before mocking on clicking delete button as the above picture shows and we did have 2 items after doing so. But the problem is that the first assertion of 3 items succeed but the second assertion of 2 items failed as the following two pictures. I was confused at first because items should be updated correctly.

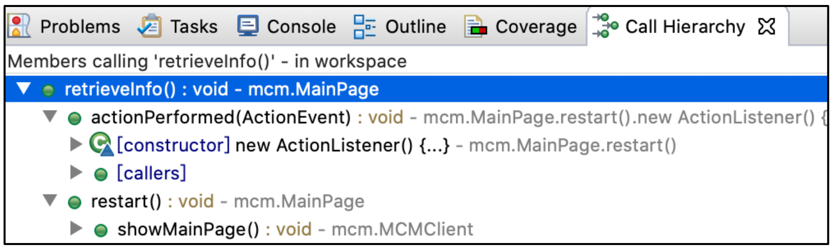


```
Start time          End time            Description
19-02-2003 16:00    19-02-2003 17:00    Lecture
19-02-2003 17:00    19-02-2003 18:30    Lecture
```



```
/* ************* */
// delete
assertEquals(3,main.appointmentSet.length);

main.appointmentTable.changeSelection(1, 0, false, false);                //select
assertEquals(1,main.appointmentTable.getSelectedRow());
assertEquals(0,main.appointmentTable.getSelectedColumn());
listeners = main.deleteAppointment.getActionListeners();
for(int i=0; i<listeners.length; i++) { listeners[i].actionPerformed(null);}    //delete one

assertEquals(2,main.appointmentSet.length);
```

The function retriveInfo() in Mainpage.java is responsible for the updating job.



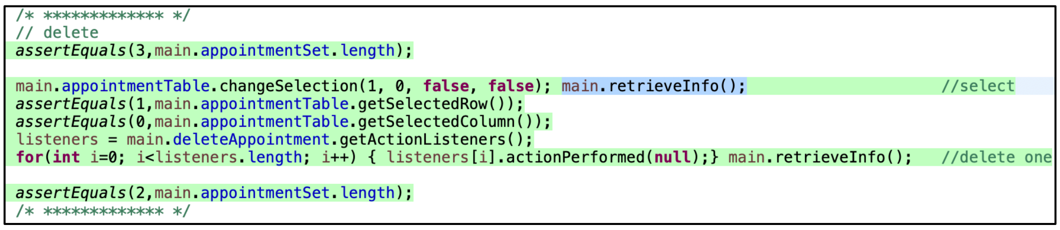
```
Problems   Tasks   Console   Outline   Coverage   Call Hierarchy
Members calling 'retrieveInfo()' - in workspace
▼ retrieveInfo() : void - mcm.MainPage
  ▼ actionPerformed(ActionEvent) : void - mcm.MainPage.restart().new ActionListener() {
    ▶ [constructor] new ActionListener() {...} - mcm.MainPage.restart()
    ▶ [callers]
  ▼ restart() : void - mcm.MainPage
    ▶ showMainPage() : void - mcm.MCMClient
```

It is called when ActionListener capture some actions happening and it is called when restart as well. In other words, if we mock on click some buttons, i.e., delete button, the function retriveInfo() would be called and member variables would update.

In the scenario mentioned above, the retriveInfo() function should be called when clicking on the button. But we still failed in asserting 2 items in appintmentSet, and I found out that the number was 3. It did not change.

So I analyze this scenario and know that it is due to the concurrency. MCM server, client and test cases run concurrently and the retriveInfo() function happening in Client may be later than our assertation. That is why we got the correct result in the Client GUI but failed in asserting.

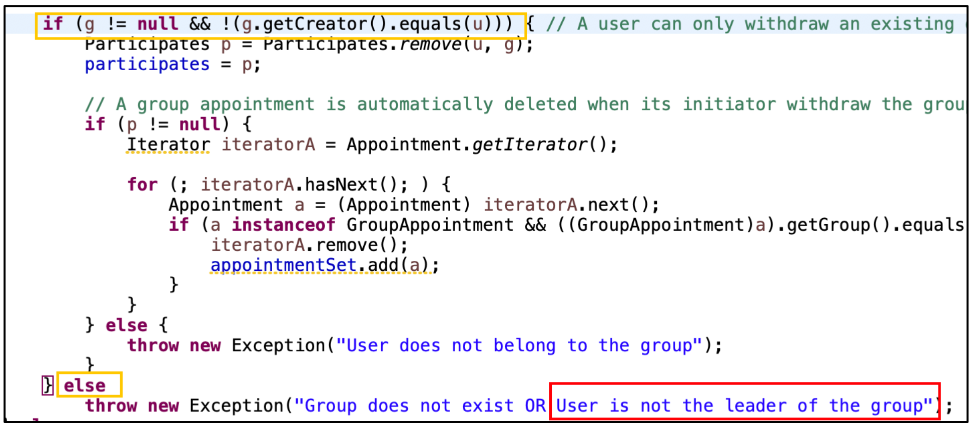
Therefore, under the analysis, we should be able to succeed if we add retrieveInfo() in our cases before assertation. And the experiment shows that the assertation succeed as we expected.



```
/* ************* */
// delete
assertEquals(3,main.appointmentSet.length);

main.appointmentTable.changeSelection(1, 0, false, false); main.retrieveInfo();                     //select
assertEquals(1,main.appointmentTable.getSelectedRow());
assertEquals(0,main.appointmentTable.getSelectedColumn());
listeners = main.deleteAppointment.getActionListeners();
for(int i=0; i<listeners.length; i++) { listeners[i].actionPerformed(null);} main.retrieveInfo();   //delete one

assertEquals(2,main.appointmentSet.length);
/* ************* */
```

Wrong Message

In execute() function of WithdrawGroup.java, there is a conditional branch to determine whether the group exists and whether the current user is the group creator. The logics is that a user can withdraw the group only when the group exists and he is not the creator because the leader cannot withdraw but remove the group. However, as the following picture shows, the exception message is wrong:



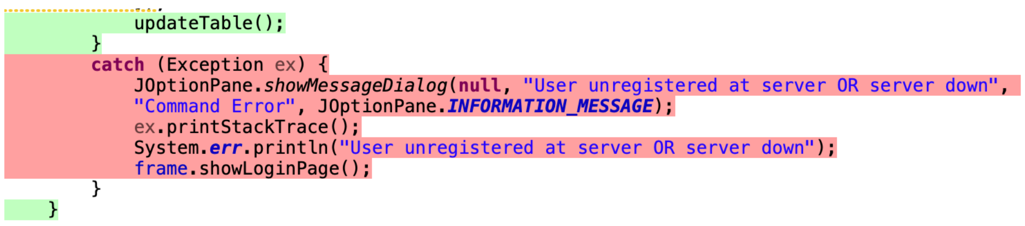
```
if (g != null && !(g.getCreator().equals(u))) { // A user can only withdraw an existing 
    Participates p = Participates.remove(u, g);
    participates = p;

    // A group appointment is automatically deleted when its initiator withdraw the grou
    if (p != null) {
        Iterator iteratorA = Appointment.getIterator();

        for (; iteratorA.hasNext(); ) {
            Appointment a = (Appointment) iteratorA.next();
            if (a instanceof GroupAppointment && ((GroupAppointment)a).getGroup().equals
                iteratorA.remove();
                appointmentSet.add(a);
            }
        }
    } else {
        throw new Exception("User does not belong to the group");
    }
} else
    throw new Exception("Group does not exist OR User is not the leader of the group");
```

The text in the red box should be “User is the leader of the group” indicating that the exception is throwed because leader cannot withdraw his group.

Infeasible Statement

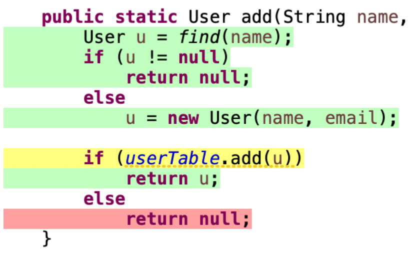
There are some infeasible statements that I cannot trigger them in a programmatic way. Here are two types of infeasible statements that I found in MCM Client.

The first type is the infeasible exception statements like the red ones in the following picture. Because I have to simulate as a registered user so that I can enter the mainpage, current user must be registered. However, the exception would throwed when the user is unregistered or the server is down. Both cases cannot be implemented in a programmatic way, that’ s why the statements are infeasible.



```
            updateTable();
        }
        catch (Exception ex) {
            JOptionPane.showMessageDialog(null, "User unregistered at server OR server down",
            "Command Error", JOptionPane.INFORMATION_MESSAGE);
            ex.printStackTrace();
            System.err.println("User unregistered at server OR server down");
            frame.showLoginPage();
        }
    }
```

The second type is the infeasible branch shown in the following picture. The condition statement in yellow has two cases, whether userTable.add(u) is successful or not. However, the statements above have already tested whether the “name” of user exists or not. If existing, the whole method would return null without entering the yellow branch. Once entering the yellow branch, the “name” has never existed, so userTable.add(u) would definitely be successful. Therefore, the other case would not happen, causing infeasible branch and its corresponding infeasible statements.



```
    public static User add(String name, String email) {
        User u = find(name);
        if (u != null)
            return null;
        else
            u = new User(name, email);

        if (userTable.add(u))
            return u;
        else
            return null;
    }
```

Reference:

- Java API Document:

  https://docs.oracle.com/javase/8/docs/api/

- MCM document:

  https://home.cse.ust.hk/~scc/Password_Only/csit5100/assign/MCM2020/Assignment1/Doc.pdf