Coding And Code Testing

CS 345-346

Medical Assistance Application

Group 13

Name	Roll Number
Tanay Maheshwari	190101092
Vignesh Ravichandra Rao	190101109
Tattukolla Lokesh	190101094
Shreyansh Meena	190101084

Contents

1 Introduction

1.1 Purpose

This Medical Assistance application is a telemedicine application which assists in the remote delivery of healthcare services like consultations. This Coding and Code Testing document is a part of the assignment under CS 346 - Software Engineering Lab. The intended audience is our course instructor, Dr Samit Bhattacharya, and the teaching assistants. This document provides a mapping between our design document and coding, the code of the important and representative use case functions and its Black Box Testing (design of test suites on the basis of equivalence partitioning and boundary values) and White Box testing reports (Path Coverage Test).

1.2 Product Scope

The COVID-19 has shown the world the importance of telemedicine. More and more people have started to prefer short online consultations over physically visiting clinics. This application allows healthcare providers to evaluate, diagnose and treat patients without the need for an in-person visit. The application aims to connect doctors and patients in an easy-to-use and convenient manner.

1.3 Overview

The remaining part of the Design document contains:

1. Mapping between DFD and Coding
2. Code
3. Black Box Texting
4. White Box Testing

1.4 References

- CS345 Lecture Slides by Dr. Samit Bhattacharya.
- Fundamentals of Software Engineering, Rajib Mall
- Roger Pressman –S/W Engineering: A Practitioner's Approach

2 Technology Used: Flutter

- What is Flutter:
  Flutter is Google's portable UI toolkit for crafting beautiful, natively compiled applications for mobile, web, and desktop from a single codebase. Flutter works with existing code, is used by developers and organizations around the world, and is free and open source.

- Why Use Flutter:

  1. Since it is a cross-platform framework, it allows programmers to write code once and they can use it on multiple platforms. This means that a single version of an application runs on both iOS and Android and even web saving lot of time and effort. Morever, it can cater to a larger number of patients using either Android or MacOS.
  2. It is open source.
  3. Flutter uses Dart as an object-oriented programming language to create apps. The prominent features of Dart include a rich standard library, garbage collection, strong typing, generics, and async-awaits.
  4. Native App like performance
  5. Hot reload and advanced debugging features for development

3 Mapping between DFD and Coding

Dart allows us to write asynchronous code. For that it has Future, async and await keywords. A future (lower case "f") is an instance of the Future (capitalized "F") class. A future represents the result of an asynchronous operation, and can have two states: uncompleted or completed. **Return type of an asynchronous function is 'Future <T> ' where T is the data type returned.**

- 1.2 Login -> Future <void> loginWithEmail(String email, String password)
- 1.1 Registration -> Future <void> signUpWithEmail(String email, String password)
- 1.3 Forgot Password -> Future <void> forgotPassword(String email)
- 2.1.1 Search By Name -> Future<List<Doctor>> searchDoctorByName(String searchName)
- 2.1.2 Search By Speciality -> Future<List<Doctor>> searchDoctorBySpeciality(String searchSpeciality)
- 2.1.3 Search By Symptom -> Future<List<Doctor>> searchDoctorBySymptom(String searchSymptom)
- 3.1.1 Chat consultation -> Future¡void¿ sendChatMessage(String senderID, string receiverID)
- 3.1.2 Video/Audio consultation -> Future<void> startAudioVideoConsultation(String doctorID, string patientID)
- 5.1 Rating -> Future<bool> giveRating(String doctorId, int stars)

- 5.2 Description -> Future<bool> giveReviewDescription(String doctorId, String text)
- 4.1 Set Reminder -> Future<void> setReminder(DateTime time, String information)
- 4.2 Remove Reminder -> Future<void> removeReminder(String notificationID)
- 3.2 Reports -> Future¡void¿ uploadReport(String doctorID, string patientID)
- 3.3 Prescription -> Future¡void¿ givePrescription(String doctorID, string patientID)
- 2.2 Request Appointment -> Future¡void¿ requestAppointment(String doctorId, String patientID)
- 2.3 Confirm Appointment -> Future<bool> confirmAppointment(int hours, int mins)
- 2.4 Payment -¿ Future¡bool¿ payFees();

4 Code

We have chosen 3 important functions of our Application

- searchDoctor
- confirmAppointment
- giveRating

The reasons for choosing these are

- serachDoctor: Filters the list of doctors based on their name, this helps users find the required based on the doctor name, this function is very important for the patients to find the appropriate doctor for their diagnosis to book the appointment.

```
Future<List<Doctor>> searchDoctorByName(String searchName) async {
  List<Doctor> result = []; // Variable to store final result
  // Check if search String is of valid length and type
  // Else raise an Exception
  if (searchName.length < 5 || searchName.length > 20 || !isAlpha(searchName)) {
    throw Exception("Search string is not valid"); // Exit by throwing exception
  }
  // Get all doctors from Doctor Data store
  List<Doctor> allDoctorsAvailable = await getAllDoctors;
  // Filter out all doctors based on those who contain same name as search string
  for (int index = 0; index < allDoctorsAvailable.length; index++) {
    // Check if search string is a substring of any doctor's name
    // Convert both doctor name and search string to lowercase for case insensitivity
    if (allDoctorsAvailable[index]
        .fullName
        .toLowerCase()
        .contains(searchName.toLowerCase())) {
      // If match found, add to result list
      result.add(allDoctorsAvailable[index]);
    }
   }
  return result
}
```

- confirmAppointement: This is another function which helps doctors to confirm or reject appointment requests from patients according to their schedules and other commitments.

```
Future<bool> confirmAppointment(int hours, int mins) async {
    // Test if entered time is invalid
1    if (hours<0 || hours>23 || mins < 0 || mins>59) {
2      throw Exception("Invalid hours/mins"); // Exit function by raising exception
     }
     // Fetch all appointments confirmed by doctor for today
3    List<AppointmentModel> allConfirmedAppointments = await getDoctorAppointments();
     // Check if entered time conflicts with any of the already scheduled appointments
4    if (scheduleConflict(allConfirmedAppointments,hours,mins)) {
5      return false;
     }
6    return true;
   }
```

- giveRating: Using this Patients can give feedback by rating to the doctors (whom they consulted) on a scale of 5 stars, which helps both the new users (to choose the doctors) and doctors (if they can improve or amend their style)

```
Future<bool> giveRating(String doctorId, int stars) async {
    // Check if input is invalid
1    if (stars < 1 || stars > 5 || doctorId.length != 20) {
2      throw Exception("Invalid number of stars");
     }
     // Get all past appointments of patient
3    List<AppointmentModel> allAppointments = await getAllPatientAppointments();
4    bool patientConsultedDoctor = false;
     // Check if patient has ever consulted the doctor
     // If yes, allow patient to give rating else do not
5    for (int index = 0; index < allAppointments.length; index++) {
6      if (allAppointments[index].doctorID == doctorId) {
7        patientConsultedDoctor = true;
         // Write rating into datastore
8        writeRating(doctorId, stars);
       }
9    }
     // Returns true if review is written else false
10    return patientConsultedDoctor;
   }
```

5 Black Box Texting

We have designed test cases from examination of the input/output values only for the 3 important functions named earlier in both approaches of Equivalence Class Partitioning and

Boundary Value Analysis.
In equivalence Class Partitioning we tried to divide the domain of input values to the unit under test is partitioned into a set of equivalence classes. The partitioning is done such that for every input data belonging to the same equivalence class, the program behaves similarly. Coming to Boundary Value Analysis we examined if the equivalence classes to check if any of the equivalence classes contains a range of values and for those classes that is a range of values, the boundary values also need to be included in the test suite.

5.1 searchDoctor

The input is a string of alphabets (doctor name) of length between 5 and 20, the output returns the list of matching doctors and empty list if no matching doctor is found

5.1.1 Equivalence Class & Partitioning

The input value class can be divided into three broad equivalence classes: invalid class of strings with length less than 5, valid class of strings with length between 5 and 20, invalid class of strings with length greater than 20. The valid class can be partitioned into two classes: strings with non-alphabet characters and strings with alphabets.

Assumption: It is assumed that already doctor with names Sachin Rahul, Sourav Laxman, Roger Nadal, Hemang Chandra Reddy, Doofenshmirtz registered and [ ] denotes empty list So based on the Equivalence Class Partitioning we have the following test suite:
T_1:= {

(“S2”, [ ]),

//string is having non-alphabet characters with length less than 5

(“Si”, [ ]),

// string is having only alphabet characters with length less than 5

(“S@ch!n”, [ ]),

// string is having non-alphabet characters with length btw 5 and 20

(“Sachin”, [“Sachin Rahul”]),

// string is having alphabet characters with length btw 5-20

(“L@*m@naabcdvegdfhfgddde”, [ ]),

// string is having non-alphabet characters with length > 20

(“Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch”, [ ])

// string is having alphabet characters with length > 20

}

5.1.2 Boundary Value Analysis

Here we are having an equivalence class of strings of length 5 to 20, so here we need to check boundary cases separately, we need to include strings of length 4, 5, 20, 21 also in the test suite.
T_2:= {

(“Meen”, [ ]),

// string of length 4

(“Nadal”, [“Roger Nadal”]),

// string of length 5

(“Hemang Chandra Reddy”, [“Hemang Chandra Reddy”]),

// string of length 20

(“Miltoni Chandra Reddy”, [ ])

// string of length 21

}

5.1.3 Final Test Suite

Now we need to include test cases of both the Equivalence class partitioning and BVA in our suite.

$T_1 \cup T_2$:= {
(“S2”, [ ]),
(“Si”, [ ]),
(“S@ch!n”, [ ]),
(“Sachin”, [“Sachin Rahul”]),
(“L@*m@naabcdvegdfhfgddde”, [ ]),
(“Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch”, [ ])
(“Meen”, [ ]),
(“Nadal”, [“Roger Nadal”]),
(“Hemang Chandra Reddy”, [“Hemang Chandra Reddy”]),
(“Miltoni Chandra Reddy”, [ ])
}

5.2 confirmAppointment

The inputs to the doctor will be time (int, int)(hours and minutes) of the current day and the output will be true if the doctor agrees for the appointment and false if not. (The input will also have the doctor name which user searched in the searchDoctor function)

5.2.1 Equivalence Class & Partitioning

The input class can divided into three broad classes, when the first integer (hour) is less than 0 (invalid class), the first integer is between 0 and 23 (valid class), and when it's greater than 23 (invalid class). Now the valid class can be further partitioned into three classes, when the second integer (minute) is less than 0, between 0 and 59 and greater than 59.

[Doctor Sachin Rahul will be available between 9:00 and 18:00 today. Doctor Doofenshmirtz is available between 20:00 to 23:59] So based on the Equivalence Class Partitioning we have the following test suite:
T_1:= {

((-2, -2), false),

// hour is entered less than 0 including minute

((-2, 30), false),

// hour is entered less than 0 while minute entered is between 0 and 59

((-2, 70), false),

// hour is entered less than 0 while minute entered is > 59

((10, -2), false),

// hour entered is btw 0 and 23, while minute entered is less than 0

((10, 30), true),

// hour entered is btw 0 and 23, while minute entered is btw 0 and 59

((10, 80), false),

// hour entered is btw 0 and 23, while minute entered is greater than 59

((27, -2), false),

// hour entered is > 23, while minute entered is less than 0

((27, 30), false),

// hour entered is > 23, while minute entered is btw 0 and 59

((27, 80), false)

// hour entered is > 23, while minute entered is greater than 59

}

5.2.2 Boundary Value Analysis

We have a range of values so we need to do BVA for both hour and minute. For hour: -1, 0, 23, 24 and for minute: -1, 0, 59, 60. For these we need the following test cases:
T_2:= {

((-1, 23), false),

//hour entered is -1

((0, 30), false),

//hour entered is 0

((23, 30), true),

// hour entered is 23

((24, 30), false),

// hour entered is 24

((23, -1), false),

// minute entered is -1

((23, 0), true),

// minute entered is 0

((10, 59), true),

// minute entered is 59

((23, 60), false)

// minute entered is 60

}

5.2.3 Final Test Suite

Now we need to include test cases of both the Equivalence class partitioning and BVA in our suite.

$T_1 \cup T_2$:= {
((-2, -2), false),
((-2, 30), false),
((-2, 70), false),
((10, -2), false),
((10, 30), true),
((10, 80), false),
((27, -2), false),
((27, 30), false),
((27, 80), false)
((-1, 23), false),
((0, 30), false),

((23, 30), true),
((24, 30), false),
((23, -1), false),
((23, 0), true),
((10, 59), true),
((23, 60), false)
}

5.3 giveRating

This function takes input of the rating stars (in the form of integer) from the user and the doctor-id against whom the rating is being given. If the given inputs are valid it will return true giving the confirmation of success or false incase of failure.

5.3.1 Equivalence Class & Partitioning

The input class can be divided into broadly three classes: integer less than 1 (invalid), integer between 1 and 5 (valid), integer greater than 5 (invalid), and further these can partitioned based on whether the entered doctor-id's length is less 20 (invalid), equal to 20 (valid), ¿ 20 (invalid) So based on the Equivalence Class Partitoning we have the following test suite:
-»»(The reason for including the test case is written right of each test case)
T_1:={

((“abc123adjh”, -2), false),
// entered integer is less than 1 with doctor-id length $<$ 20
((“abc123adjh”, 2), false),
// entered integer is btw 1 and 5 with doctor-id length $<$ 20
((“abc123adjh”, 7), false),
// entered integer is greater than 5 with doctor-id length $<$ 20
((“abc123adjhabc123adjh”, -2), false),
// entered integer is less than 1 with doctor-id length = 20
((“abc123adjhabc123adjh”, 2), true),
// entered integer is btw 1 and 5 with doctor-id length = 20
((“abc123adjhabc123adjh”, 8), false),
// entered integer is greater than 5 with doctor-id length $>$ 20
((“abc123adjhabc123adjh23”, -2), false),
// entered integer is less than 1 with doctor-id length $>$ 20
((“abc123adjhabc123adjh23”, 4), false),
// entered integer is btw 1 and 5 with doctor-id length $>$ 20
((“abc123adjhabc123adjh23”, 9), false),
// entered integer is greater than 5 with doctor-id length $>$ 20
}

5.3.2 Boundary Value Analysis

Here as the doctor-id length is not a range of values but instead a discrete value no need to define boundary test cases (according to Rajib Mall tb section 10.6.2, 2nd para). But for the integer (no.of stars) we need to define boundary cases: 0,1,5,6 For these we need the following test cases:
T_2:={

((“abc123adjhabc123adjh”, 0), false),

// entered integer is 0

((“abc123adjhabc123adjh”, 1), true),

// entered integer is 1

((“abc123adjhabc123adjh”, 4), true),

// entered integer is 5

((“abc123adjhabc123adjh”, 6), false),

// entered integer is 6

}

5.3.3 Final Test Suite

Now we need to include test cases of both the Equivalence class partitioning and BVA in our suite.

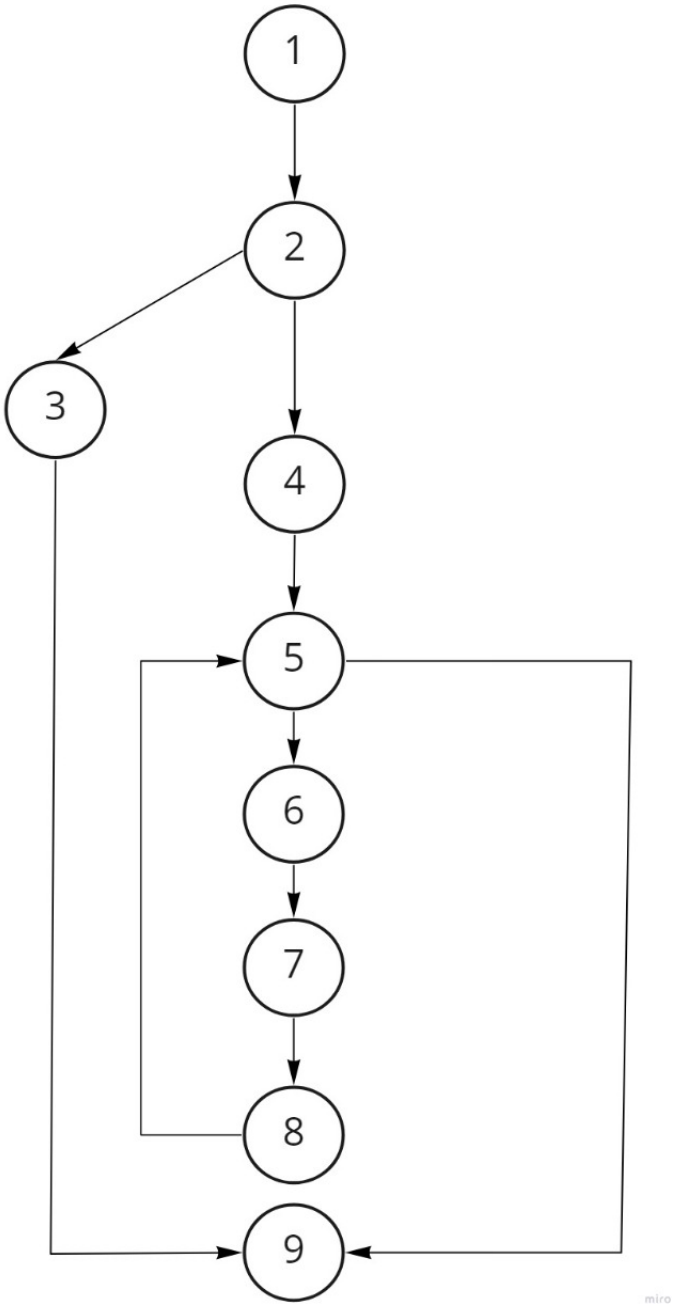
$T_1 \cup T_2$:= {
((“abc123adjh”, -2), false),
((“abc123adjh”, 2), false),
((“abc123adjh”, 7), false),
((“abc123adjhabc123adjh”, -2), false),
((“abc123adjhabc123adjh”, 2), true),
((“abc123adjhabc123adjh”, 8), false),
((“abc123adjhabc123adjh23”, -2), false),
((“abc123adjhabc123adjh23”, 4), false),
((“abc123adjhabc123adjh23”, 9), false),
((“abc123adjhabc123adjh”, 0), false),
((“abc123adjhabc123adjh”, 1), true),
((“abc123adjhabc123adjh”, 4), true),
((“abc123adjhabc123adjh”, 6), false),
}

6 White Box Testing

In this test cases are designed using knowledge of internal structure of software. A test suite achieves path coverage if it executes each linearly independent paths (or basis paths) at least once. A linearly independent path can be defined in terms of the control flow graph (CFG) of a program. Now we will see the CFG and test suites of the three functions we defined earlier.

6.1 searchDoctor

6.1.1 Control Flow Graph



6.1.2 Cyclomatic Complexity

The cyclomatic complexity of the CFG is (Edges - Vertices + 2)
=> 10 - 9 + 2 = 3.
So now we got the upper bound of the number of test cases required.

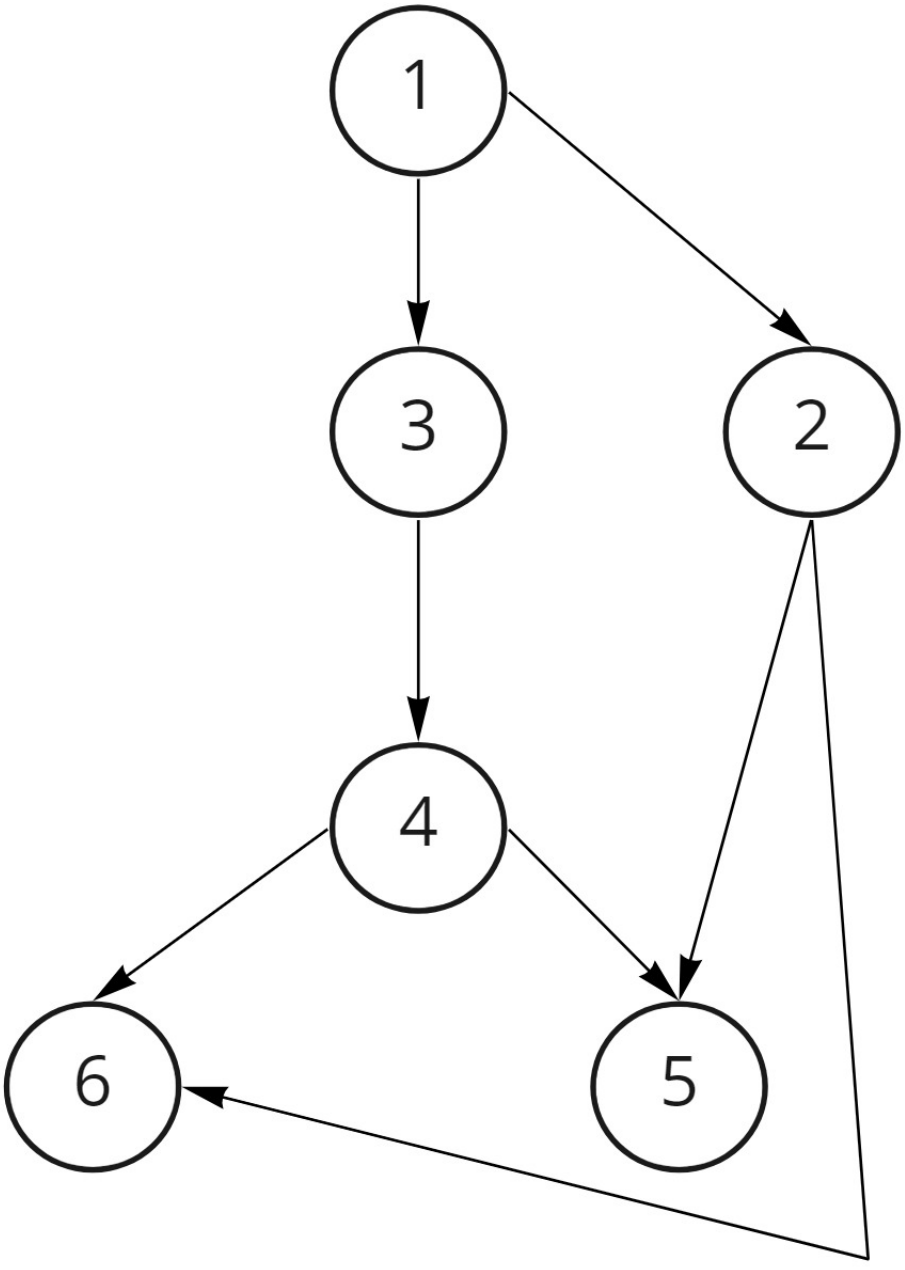
6.1.3 Linearly Independent Set of Paths

The Linearly Independent Paths and their corresponding test cases are:
Path 1: 1-2-9
Test Case: If the length of string is less than 5 or greater than 20 or is not of correct type

Input: “ab2”
Path 2: 1-2–4-5-6-7-8-5-9
Test Case: Length of entered string between 5 and 20 with proper type and the doctor name is registered in the database
Input: “Sachin”
Path 3: 1-2-4-5-9
Test Case: Length of entered string between 5 and 20 with proper type and the doctor name is registered in the database
Input: “Xaveria”

6.2 confirmAppointment

6.2.1 Control Flow Graph



6.2.2 Cyclomatic Complexity

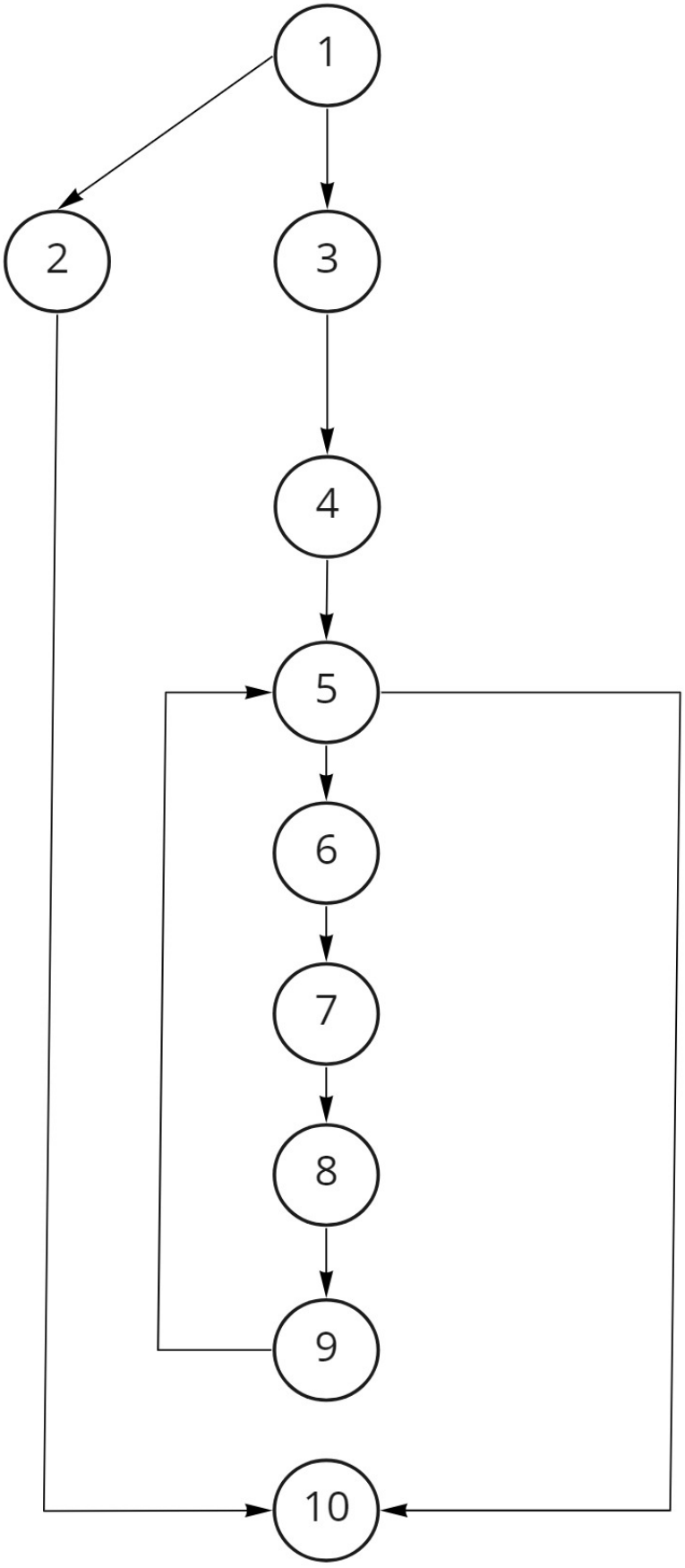
The cyclomatic complexity of the CFG is (Edges - Vertices + 2)
=> 7 - 6 + 2 = 3.
So now we got the upper bound of the number of test cases required.

6.2.3 Linearly Independent Set of Paths

The Linearly Independent Paths and their corresponding test cases are:
Path 1: 1-2-5
Test case: If the entered hours and minutes are not proper range
Input: (27, 60)
Path 2: 1-3-4-5
Test Case: If the entered time is in range and there is conflict with schedule or doctor is not available at that time
Input: (21, 20)
Path 3: 1-3-4-6
Test Case: If the entered time is in range and there is no conflict with schedule available
Input: (10, 30)
PS: Here test cases are based on the previous test cases of balck box testing.

6.3 giveRating

6.3.1 Control Flow Graph



6.3.2 Cyclomatic Complexity

The cyclomatic complexity of the CFG is (Edges - Vertices + 2)
=> 11 - 10 + 2 = 3.
So now we got the upper bound of the number of test cases required.

6.3.3 Linearly Independent Set of Paths

The Linearly Independent Paths and their corresponding test cases are:
Path 1: 1-2-10
Test Case: Here the stars given are either less than 1 or greater than 5 or the doctor id is not valid.
Input = ("abc123adjh", 6)
Path2: 1-3-4-5-6-7-8-9-5-10
Test case: Here the entered stars and doctor id isvalid(that is patient consulted the doctor previosuly)
Input = ("abc123adjhabc123adjh", 4)
Path 3: 1-3-4-5-10
Test case: The entered stars are in range but the doctor id is not found, that is patient did consult the doctor previously but tried to rate.
Input = ("abc123adjhabctggggff", 1)

Github link of the repo is https://github.com/Itachi8374/MedicalAssistanceSWE