Design and Implementation of Human-Computer Interfaces
Dr. Samit Bhattacharya
Department of Computer Science and Engineering
Indian Institute of Technology – Guwahati

Lecture – 35
White–Box Testing Case Study

Hello and welcome to NPTEL MOOCS course on design and implementation of human-computer interfaces. We will continue our previous lecture that is lecture number 29. So, this is a continuation of lecture number 29. In the previous lecture, what we have learned is about white-box testing. So, if you may recall it so this is one of the several code testing methods. Broadly, there are two categories of testing, review based and execution based.

We have seen the review-based testing earlier. We have gone through one case study for review-based code testing. We have also seen that there are two types of execution-based code testing, one is black-box testing and the other one is white-box testing. Earlier we have gone through in details about black-box testing and also we have gone through a case study on black-box testing.

In the previous lecture, that is lecture number 29, we have learned about white-box testing in details including the different ways to perform white-box testing. In this lecture, we are going to continue that discussion and we are going to go through a case study to better understand white-box testing approaches.

(Refer Slide Time: 02:01)

Like in the previous case study discussions, here also we will show you the documentation for white-box testing, how to create the document, how it looks like and also what should be there inside the document. So, like before we will have a cover page which will mention the purpose of that document like here it is mentioned that it is a testing document for business management software.

Also, there is this version information, version 1.1, date of creation of this document like before we have seen, the creators, the place of creation and some additional information like who was the supervisor, who supervise the document creation that is the cover page like we have seen in the earlier case studies.

(Refer Slide Time: 03:02)

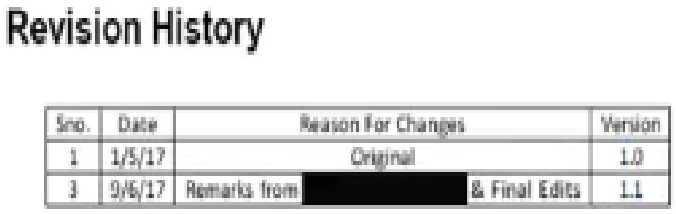
In the next page as usual, we will have the table of contents. In this particular testing document, the document contains some revision history, then an introductory section which includes purpose, conventions scope, all this information that we have seen already in the earlier documents. Then one section on mapping between DFD and function. So, this testing document actually makes it easier for the reader to understand the functions that have been used in this system.

And how those functions were obtained from the system design document that is the DFT. So according to this document, this whole system contains 5 modules module 1, 2, 3, 4, 5 and each module contains several functions so that mapping is shown in this section, mapping between DFD and functions. Then it has one section on choosing three important functions. So, this section of course is optional.

And this is only used to demonstrate the idea because showing all the details of all the functions and test report for all the functions would have lengthened the document without adding any extra pedagogical value. So here instead, what we have chosen to do is basically focus on three important functions and so how those three functions were used to perform white-box testing and also black-box testing in this test report.

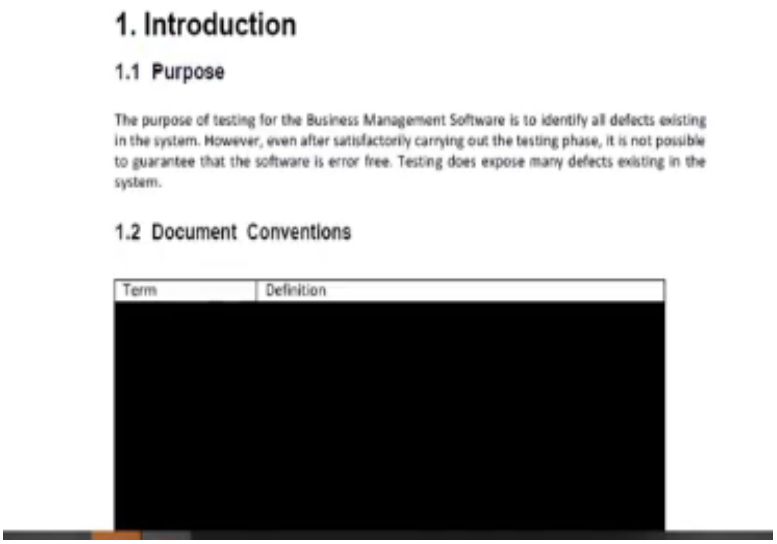
So next section contains a report on black-box testing for those three functions and the subsequent section contains a report on white-box testing for those three functions. Earlier of course we have seen black-box testing, so we will not spend much time on this black-box testing report part and we will focus on the white-box testing report. The document ends with a conclusion section followed by a set of references.

(Refer Slide Time: 05:10)



The next page contains some revision history as before. So, it records the historical evolution of the document, first person when it was created, then when it was revised and second person was created and so on. In this particular case, there were only two versions, so those two were recorded here. But in a typical situation, there may be many more such versions, so this history will contain all details about all those versions.

(Refer Slide Time: 05:41)



Now, that is followed by the introduction section. Since we have already seen in the earlier documents how the introduction section should be created, so we will skip the details here, only we will mention what are the subsections namely purpose, purpose of this document, then conventions that are used in the document, scope of the project all these details should be mentioned here in this introduction section as we have seen in the earlier case study reports.

(Refer Slide Time: 06:13)

To better understand the system, let us first quickly have a look at the second section that is mapping between DFD and functions. It will give us some idea of what this system is all about and what are the functions that are part of the system. In other words, it will give us some idea on the overall complexity of the system. So, as we have seen in the table of contents part there are 5 modules for this system.

Module 1 contains 3 submodules as shown here. Module 1.1 is a function create new account. Module 1.2 is another function login user account and module 1.3 has sub submodules 1.3.1, 1.3.2, 1.3.3. Now 1.3.1 is a function change password, 3.2 is another function forgot password and 3.3 is a third function set new password. As you can see, each module and submodule is mapped to a function. So, that is about module 1 primarily related to login and setting, resetting of passwords.

Then module 2 contains 5 submodules. Module 2.1 is the view stock data, module 2.2 is search stock items, module 2.3 contains 3 sub modules; 2.3.1 add new item, 2.3.2 update item and 2.3.3 void delete item. Module 2.4 is purchase list and module 2.5 is update purchase list. As you can see from this module, this overall system is related to management of a business that means whatever items are bought, sold, entered into stock.

All these things can be done with the software essentially this is a kind of ERP software which allows someone to run business smoothly. So, the first module is related to logging into the software, second module is related to the details about the stock items. Module 3 contains 4 submodules, 3.1 is add sold items, 3.2 is view sales record, module 3.3 is add purchased item, 3.4 is view purchase record.

(Refer Slide Time: 08:49)

Then Module 4 contains 5 submodules, 4.1 is add record, 4.2 is update record, 4.3 is delete record, 4.4 is search record and 4.5 is show debt record. And finally module 5 contains 4 submodules; 5.1 is create new employee, 5.2 is delete employee, 5.3 is mark employee

attendance, 5.4 view attendance record. So essentially, this software allows us to log in, to maintain details about the employee including their attendance.

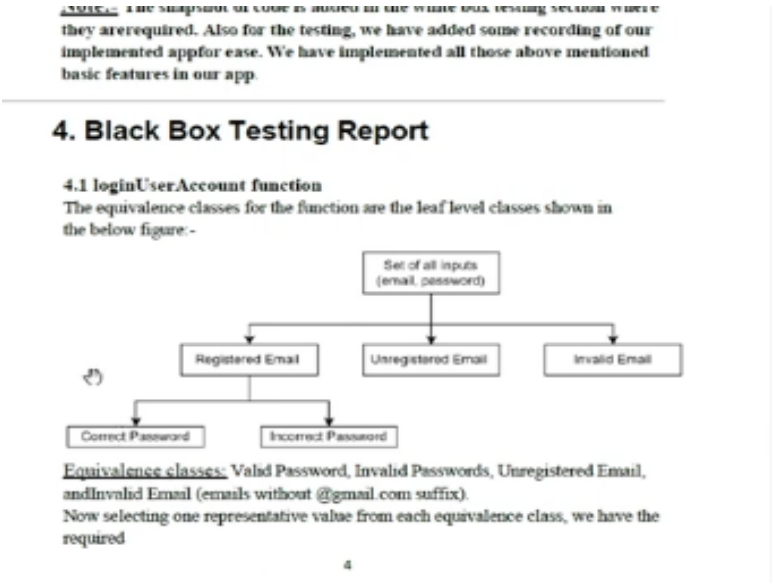
To keep track of the items that are sold, that are purchased, in short everything that a business requires to keep track of. So, these are the modules and each module contains several submodules. Some of the submodules contain further submodules as we have seen and each of these submodule or sub submodule or a module is mapped to a function as listed in this section. Now, you can see that there are a large number of such functions.

The purpose of this lecture is just to demonstrate how to create a test report. So, there to keep things simple what we have done is we have chosen three of the important functions and for those three functions, we have created the testing report. So, what are those three functions? Login user account, search stock item and add sold item detail. Also, there is some justification given for why these three functions are considered to be important and emphasized for creating the report but that is irrelevant for our purpose.

So, these three functions as you can see are highlighted here. In module 1 we have this login user account function. In module 2 we have search stock item function and in module 3 we have this adds sold item detail function. So, login user account is basically a function which we get when we map module 1.2. Search stock item is a function which we get when we map module 2.2. And module 3.1 is mapped to add sold item details.

And also in this function as you can see the function login user account does not return anything and it takes as input user email id and password. Search stock item take as input item name and returns the stock details or the records for that particular item. Add sold item detail function takes as input item name and the quantity and returns nothing. So, these are the three functions that we have chosen for further elaboration of the case study.

(Refer Slide Time: 12:02)



they arerequired. Also for the testing, we have added some recording of our implemented appfor ease. We have implemented all those above mentioned basic features in our app.

4. Black Box Testing Report

4.1 loginUserAccount function

The equivalence classes for the function are the leaf level classes shown in the below figure:-

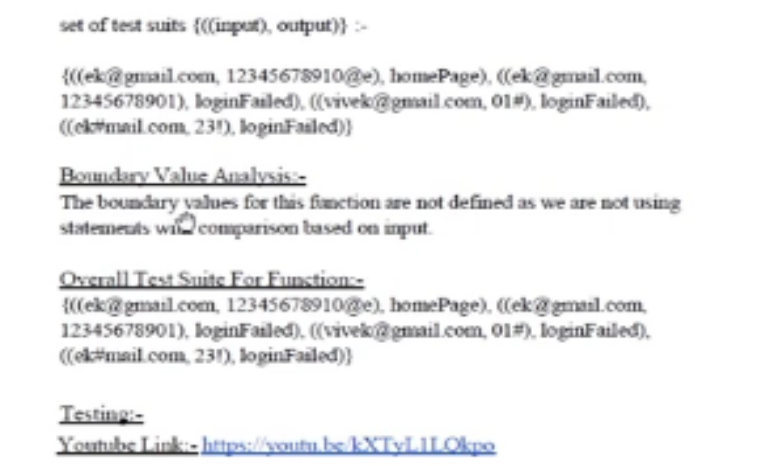
Equivalence classes: Valid Password, Invalid Passwords, Unregistered Email, andInvalid Email (emails without @gmail.com suffix).
Now selecting one representative value from each equivalence class, we have the required

4

So, there is black box testing report which is part of this testing report document. Of course, we have seen a case study on black box testing earlier, so we will not spend time on this black box testing here, only just to recollect in black box testing what we require is identification of equivalence classes which have been done here. And then based on the equivalence classes, so test suit needs to be created.

So for each of the functions, equivalence classes were identified like for login user account function these are the equivalence classes, correct password, incorrect password, unregistered email, invalid email.

(Refer Slide Time: 12:42)



set of test suits {((input), output)} :-

{((ek@gmail.com, 12345678910@e), homePage), ((ek@gmail.com, 12345678901), loginFailed), ((vivek@gmail.com, 01#), loginFailed), ((ek#mail.com, 23!), loginFailed)}

Boundary Value Analysis:-
The boundary values for this function are not defined as we are not using statements with comparison based on input.

Overall Test Suite For Function:-
{((ek@gmail.com, 12345678910@e), homePage), ((ek@gmail.com, 12345678901), loginFailed), ((vivek@gmail.com, 01#), loginFailed), ((ek#mail.com, 23!), loginFailed)}

Testing:-
Youtube Link:- https://youtu.be/kXTyL1LOkpo

And correspondingly some test cases were created to form the test suit.

(Refer Slide Time: 12:48)

The boundary values for this function are not defined as we are not using statements with comparison based on input.

Overall Test Suite For Function:-
{((ek@gmail.com, 12345678910@e), homePage), ((ek@gmail.com, 12345678901), loginFailed), ((vivek@gmail.com, 01#), loginFailed), ((ek#mail.com, 23!), loginFailed)}

Testing:-
Youtube Link:- https://youtu.be/kXTyL1LQkpo

Test suit 1:-
Input: ek@gmail.com, 12345678910@e
Expected Output: homePage
System Output: homepage

Test suit 2:-
Input: ek@gmail.com, 12345678901
Expected Output: Login Failed Message
System Output: Login Failed Message

Test suit 3:-
Input: vivek@gmail.com, 01#
Expected Output: Login Failed Message
System Output: Login Failed Message

mailto:vivek@gmail.com 5

In this particular function case, no boundary cases have been considered.

(Refer Slide Time: 12:55)

Test suit 4:-
Input: ek#mail.com, 23!
Expected Output: Login Failed Message
System Output: Login Failed Message

4.2 searchStockItem function
The equivalence classes are the leaf level classes shown in the below figure: -

Equivalence classes: Valid Product Names (Product already created by the user) andInvalid Product Name (Product not created by the user).
Selecting one representative value from each equivalence class, we have the required testsuite {(input, output)}:
{(pen, StockData), (chair, Error)}

Note: Consider pen is stored in the database as a product and chair is not.

Boundary Value Analysis:-The boundary values for this function are not defined as we are not using statements with comparison based on input.

Overall Test Suite For Function:-
{(pen, StockData), (chair, No Record Message)}

Testing:-
Youtube Link:- https://youtube.com/shorts/hsXIcNrP0i4?feature=share
Test suit 1:-
Input: pen
Expected Output:
StockDataSystem
Output: StockData

However, four search stock item function equivalence classes were mentioned, valid product names and invalid product names as well as boundary value analysis was done.

(Refer Slide Time: 13:06)

Test suit 2:-
Input: chair
Expected Output: No Record MessageSystem
Output: No Record Message

4.3 addSoldItemDetail function

The equivalence classes are the leaf level classes shown in the below figure -

Equivalence classes: CurrentStock >= Quantity, CurrentStock < Quantity, Quantity <= 0and Invalid Product Names.
Selecting one representative value from each equivalence class, we have the required test suite{((input), output)}:-
{((toothpaste, -1), fail message), ((toothpaste, 2), success message), ((toothpaste, 60), fail message), ((toothpaste, 0), fail message), ((table, 2), fail message)}

Note: toothpaste is a valid product name with available quantity 30 and table is

And same is true for the third function that is add sold item detail. However, we will not spend time on this as we have already covered it in detail in earlier lectures.

(Refer Slide Time: 13:21)

5.White Box Testing:-

5.1: loginUserAccount function

```
// Take email id and password from user
// and if it matches the already present id,password pair
// redirect the user to home page
  private void loginUserAccount()
  {
    // Take input from user into the edit text fields
    // and store them into string variables
1   String email = emailTextView.getText().toString();
2   String password = passwordTextView.getText().toString();

    // Check whether the email field is non-empty
    // if empty show a message to enter email
3   if (TextUtils.isEmpty(email)) {
4       Toast.makeText(getApplicationContext(),
                "Please enter email!!",
                Toast.LENGTH_LONG)
                .show();
5       return;
    }
```

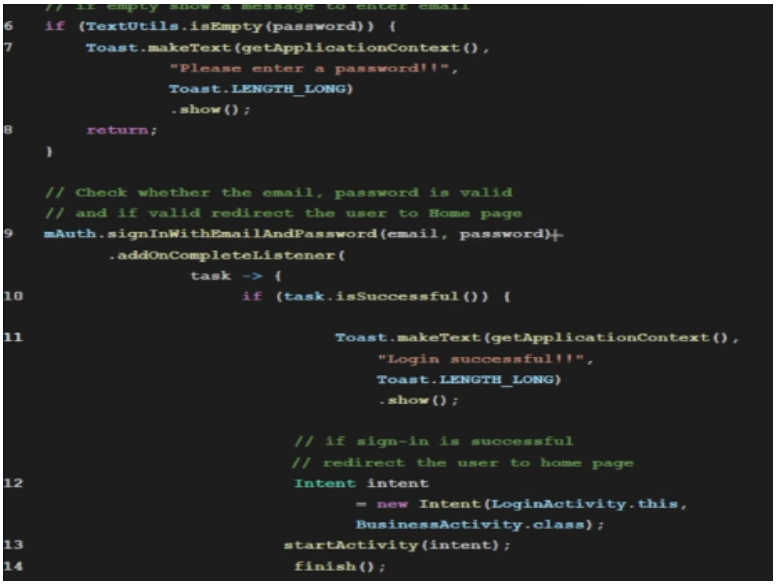
So, we will directly go to the next section that is white box testing that is section number 5 in the test document. So, in white box testing, what we are supposed to do? We are supposed to first go through the structure of the code, so it is a structural analysis. So, we need to know the internal structure, then that structure has to be converted to a CFG for analysis of the code and identification of test cases.

So, let us first see the code for the first function that is login user account function. This is the function, as you can see each line of code is marked with some number like first declaration statement like 1 is a line of code, 2 is another line of code, then you have 3 another line of

code. So, the comment parts are not given any line number as you can see here. Also the insignificant parts were not given line number such as the private void login user account, the function name is not given any line number.

The opening this is not given any line number. So only the significant parts of the code are generally assigned line numbers for further analysis. So, you should not give line numbers to function name, base, comments these types of things. So line number 1 is one line of code, 2 is another line of code, then 3 is if statement, 4 is the body inside the if statement, 5 is a return statement. Note here again that this base, the closing base for the if statement is not given any line number.

(Refer Slide Time: 15:12)



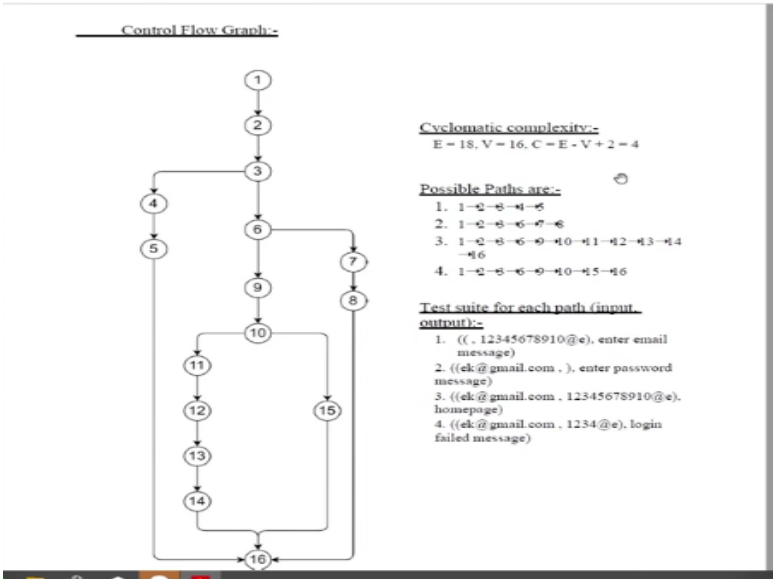
The 6 is another if statement, 7 is the body inside the if statement, 8 is return statement, then 9 another piece of code, 10 is if statement, 11 is the body inside if statement, 12 is another statement inside the body, 13 is again another statement inside the body.

(Refer Slide Time: 15:37)

```
10                  if (task.isSuccessful()) {

11                      Toast.makeText(getApplicationContext(),
                            "Login successful!!",
                            Toast.LENGTH_LONG)
                            .show();

                        // if sign-in is successful
                        // redirect the user to home page
12                      Intent intent
                              = new Intent(LoginActivity.this,
                              BusinessActivity.class);
13                    startActivity(intent);
14                      finish();
                    }
                    else {

                        // sign-in failed show login fail message
15                      Toast.makeText(getApplicationContext(),
                            "Login failed!!",
                            Toast.LENGTH_LONG)
                            .show();
                    }
                });
16}
```

The 14 is another statement inside the body of if, then 15 is one more statement and the last brace is given a statement line number 16. So, here the rule is of course not followed that should not assign any line number to the braces because this is the end of the code. So, one exception is made here to keep things clear.

(Refer Slide Time: 16:05)



Now, from this code, we have to create a CFG or control flow graph for visualization of the flow of the code and then based on that visualization, we have to identify linearly independent paths for path coverage testing. So, there are 16 lines of codes in the function. So, correspondingly we can make a CFG following the convention as you can see here. So, this is the control flow graph for the function that we have just seen.

So, this graph contains several node, 1 followed by 2, followed by 3. Since 3 is an if statement, so from 3 there are other paths, one goes to 4, one goes to 4 followed by 5 and then comes out at 16 that is the end of the code. Now at 6, there is another if statement, so two more paths 9 followed by 10 or 7, 8 followed by 16. At 10, there is one more if statement so again two more paths followed by 11, 12, 13, 14 or it can go to 15 and eventually they can come to 16 that is the end of the code.

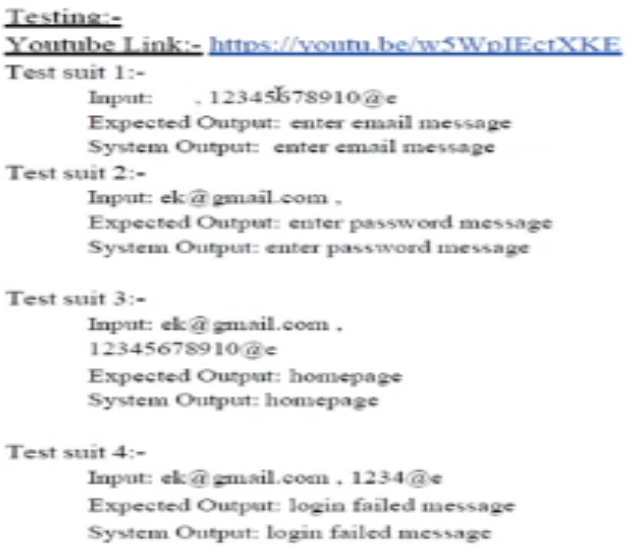
Now, in order to understand how many test cases we require, we need to perform cyclomatic complexity analysis for this code. If you may recollect, so it shows how many linearly independent paths are there. So, the cyclomatic complexity is basically E – V + 2 where E is the number of edges, V is the number of nodes or vertices. In this CFG we have E as 18, V as 16, you can count it from this CFG the number of edges and number of vertices. Then C is E – V + 2 or 4.

Now, 4 indicates that there are 4 linearly independent paths. So, what are those paths? 1, 2, 3, 4, 5 followed by 16 of course, then 1, 2, 3, 6, 7, 8 and 16, at the end it should come 16. Then 1, 2, 3, 6, 9, 10, 11, 12, 13, 14, 16 that is 1, 2, 3, 6, 9, 10, 11, 12, 13, 14, 16. So, this path as shown here is another linearly independent path and then 1, 2, 3, 6, 9, 10, 15, 16 so the other path, So, these are the 4 linearly independent paths that we can have from the CFG.

Corresponding to each path, we have to then design a test case so that each of these paths are tested and then that will give us a path coverage testing. So, we have to have at least 4 test cases to cover the 4 paths. Now, the test cases can be like the ones shown here. In the first two tests, the first path we can have one input as byte string followed by some invalid email ID. Then to test the second path that is 1, 2, 3, 6, 7, 8 and 16 we can have another test case as shown here.

To test the third path 1, 2, 3, 6, 9, 10, 11, 12, 13, 14, 16. we can have this third test case as shown here and to follow the fourth path and perform a testing for the fourth part that is 1, 2, 3, 6, 9, 10, 15, 16 we can have the fourth test case as mentioned here.

(Refer Slide Time: 20:15)



So, in the first case we have input a blank and invalid email id, then expected output is enter email message, system output is enter email message. So, here note that each of these are called a test case, not test suit, so there is some typo here, it should be called test case 1, test case 2, test case 3 and test case 4. Together these 4 test cases will comprise a test suit. And for each of these test cases, we have so the input, expected output and during testing we got some output that is the system output.

Now, whenever there is a match, then so the test was successfully performed, whenever there is a mismatch so there is some problem in the code and we need to refine it. So those points are marked. Those test cases where the system failed to produce the expected output are marked. So, in this document you are supposed to provide the input, the expected output as well as the system generated output as shown in these cases.

(Refer Slide Time: 21:22)

5.2: searchStockItem function

```
// Function to search whether an item is available in the store
// and if available how much quantity is present
private void searchStockItem() {

    // Take the input from user into edit fields and convert into  string
1   String stockName = fieldStockName.getText().toString();

    // Check whether the stock name field is non-empty
    // if empty show a message to enter it.
2   if (TextUtils.isEmpty(stockName)) {
3       Toast.makeText(getApplicationContext(),
                "Please Enter Stock Name!!",
              Toast.LENGTH_LONG)
              .show();
4       return;
    }
    // Find the stock item in the database using the entered name
    // if available show the details
5   DocumentReference docRef = db.collection("stocks").document(stockName);
6    docRef.get().addOnSuccessListener(documentSnapshot -> {
```

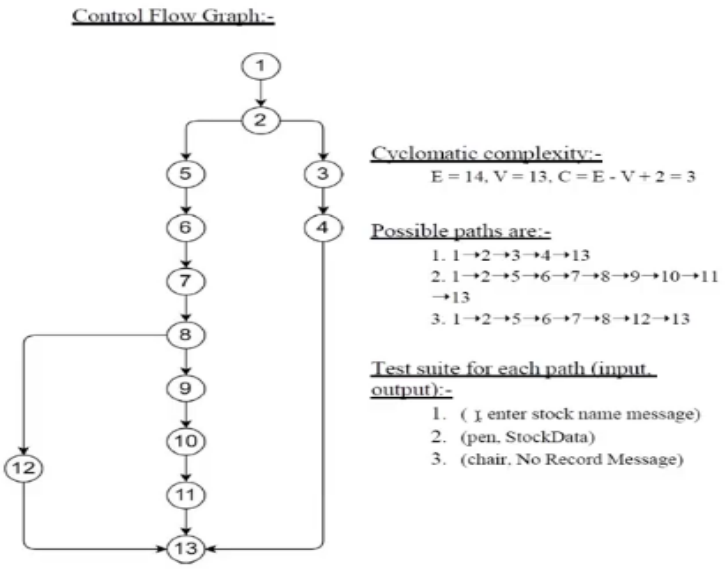
Let us move to the next function that is search stock item function. So, here as you can see like before, we have provided line numbers to each significant line of code. So, there are 1, 2, 3, 4, 5, 6.

(Refer Slide Time: 21:51)

```
7       Stock stocks = documentSnapshot.toObject(Stock.class);

8       if(stocks != null && stocks.getStockName().equals(stockName)){
          // stock found show the details
9           textStockName.setText(String.format("Stock Quantity: %s",
stocks.getStockName()));
10          textStockQuantity.setText(String.format("Stock Quantity: %s",
stocks.getStockQuantity()));
11          textStockPrice.setText(String.format("Stock Price: %s",
stocks.getStockPrice()));
        }
        else{
            // stock not found show not found message
12          Toast.makeText(getApplicationContext(),
                "Sorry No Record Found With Name: "+ stockName +".",
                Toast.LENGTH_LONG).show();
        }       ;
    });
13}
```

7, 8, 9, 10, 11, 12, 13. So there are thirteen 13 lines of codes. That means there will be 13 nodes in the CFG. Let us see the CFG how it looks.

(Refer Slide Time: 22:07)



So, as you can see here, there are 13 nodes in the CFG. The 1 followed by, 2 now at 2 there is an if statement, so there are two paths. One path goes to the node 5, other path goes to the node 3 that is a third line of code, from 3 it goes to 4 and then comes to the end of the code that is 13. The other path goes to line number 5, then line number 6, then line number 7, then line number 8, 8 is another if statement, so two more paths.

One path goes to line number 12 and comes to the end of the program or code and the other path goes to line number 9, 10, 11 and then comes to the last line of the code that is 13. So, here number of edges is 14 and number of vertices or nodes is 13. So, the cyclomatic complexity is E – V + 2 that is 3. So, there are 3 linearly independent paths possible. What are those paths? One is the path 1 followed by 2 followed by 3 followed by 4 followed by 13 that is 1 followed by 2 followed by 3, 4 and 13.

Then second is 1, 2, 5, 6, 7, 8, 9, 10, 11, 13 that is this path 1, 2, 5, 6, 7, 8, 9, 10, 11, 13. And the third one is 1, 2, 5, 6, 7, 8, 12. Since there are 3 linearly independent paths that means at least 3 test cases are required together they will constitute a test suit. So in this case for the second function for white box testing, we require a test suit comprising at least 3 test cases each corresponding to one of the linearly independent paths in the CFG.

The three test cases are shown here. So, for first path 1, 2, 3, 4, 13 the test case is blank, enter stock name message. For second path 1, 2, 5, 6, 7, 8, 9, 10, 11, 13 input is pen, output is stock data. Third path 1, 2, 5, 6, 7, 8, 12, 13. input is chair, output is no record message.

(Refer Slide Time: 24:49)

Testing:-

Youtube Link:- https://youtube.com/shorts/Y4y7Ng42Ugc?feature=share

Test suit 1:-

Input:

ExpectedOutput:sto

ck dataSystem

Output: stock data

Test suit 2:-

Input: pen

Expected Output: No Record Message

System Output: No Record Message

Test suit 3:-

Input: chair

Expected Output: No Record Message

System Output: No Record Message

Like before, so expected output is mentioned. Expected output is stock data, system output is stock data in the first case. In the second case, expected output is no record message, system output is no record message. In the third case, expected output is no record message, system output is no record message. So, all the details are mentioned.

(Refer Slide Time: 25:15)

```
                // update the balance record present in database
26              balance.setAmount(newBalance);
27              db.collection("account").document("balance").set(balance);

28              Toast.makeText(getApplicationContext(),
                        "Sold Record Updated Successfully.",
                        Toast.LENGTH_LONG)
                        .show();

                // return back to previous page
29              startActivity(new Intent(AddSoldRecord.this,
AccountManagement.class));
30              finish();
            });
        }
        else{
            // stock name not found in database show error message
31          Toast.makeText(getApplicationContext(),
                    "Sorry Stock: "+ stockName +" Not Found. Add stock
record first.",
                    Toast.LENGTH_LONG)
                     .show();
        }
    });
32}
```

Similarly, we come to the third function that is add sold item detail. Like before we assign line numbers to each piece of code, each line of code that is of significance. So, just to repeat we do not assign numbers to the function name or the brace or the comments, we assign it to significant line of codes. And one exception is the last base or closing base of the entire function maybe assigned a line of code to indicate the end of the program.

(Video Starts: 25:58) So, here we have 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32. **(Video Ends: 26:22)** Total 32 numbered lines in the code that means in the CFG there should be 32 nodes. Let us see the CFG or the control flow graph. **(Video Starts: 26:35)** So, it is a pretty big one because there are large number of nodes and as you can see we start with node 1, 2, 3.

And at 3 we have if statement, so there are 2 paths, one goes to 6, one goes to 4, 4 followed by 5 and then it comes to the end of the program at 31. Then we have from 3 we can take another path that is 6. At 6 there is one more if statement, so 2 more paths, one goes to 7, 8 followed by the end of the program, other one is 6 followed by 9 followed by 10, 11, 12. At 12, one more if statement, so two more paths, one goes to 13, other goes to 31 and it comes to the end at 32.

So, this last statement should be 32, last node marking. Last ode level should be 32 instead of 31, there is a typo. Then from 12, we can also come to 13, 14, 15, 16, 17, 18, 19, 20, 21, 22. At 22, there is one more if statement, so 2 more paths, one goes to 23 and one comes to the end node that is 32. So, 23 followed by 24, 25, 26, 27, 28, 29, 30 and 32 that is the other path which can be taken from node number 22. **(Video Ends: 28:10)**

So, there are so many branches as you can see in this CFG which indicates that the code is relatively complex compared to the other codes that we have seen.

(Refer Slide Time: 28:22)

Cyclomatic complexity:-

E = 34, V = 31, C = E - V + 2 = 5

Possible Paths are:-

1. 1→2→3→4→5→32
2. 1→2→3→6→7→8→32
3. 1→2→3→6→9→10→11→12→13→14→15→16→17→18→19→20 →21→22→23→24→25→26→27→28→29→30→32
4. 1→2→3→6→9→10→11→12→13→14→15→16→17→18→19→20 →21→22→32
5. 1→2→3→6→9→10→11→12→31→32

Test suite for each path (input, output):-

1. ((, 1), enter stock name message)
2. ((toothpaste,), enter stock quantity message)
3. ((toothpaste, 2), success message)
4. ((toothpaste, 0), fail message)
5. ((table, 2), fail message)

Testing:-

Youtube Link:- https://youtu.be/ciGAOlVSBuk

Test suit 1:-

Input: , 1

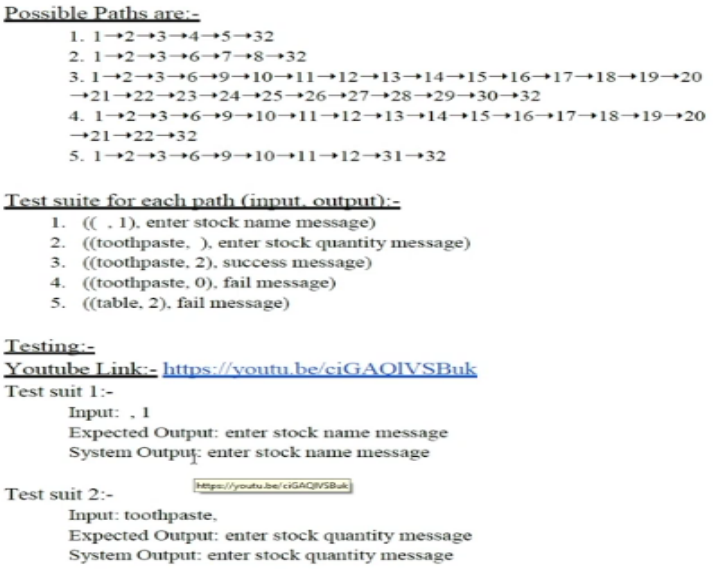
So far this how many linearly independent paths are there. So we will apply the formula, we have edge 34, vertices 32, so we will have 34 – 32. So here we have edge 34, vertices 32 and the complexity it should be 4, so there are some typos, in this case we have edge 35, vertices 32 and cyclomatic complexity will be 35 – 32 + 2 that is 5. So 5 means there are 5 linearly independent paths.

So the 5 possible paths are 1, 2, 3, 4, 5, 32; 1, 2, 3, 6, 7, 8, 32; 1, 2, 3, 6, 9, 10, 11, 12 so on up to 32; 1, 2, 3, 6, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 32 and the fifth one is 1, 2, 3, 6, 9, 10, 11, 12, 31, 32. Since there are 5 such paths, so we will have at least 5 test cases comprising our test suite to perform white box testing for this particular function. So, the 5 test cases as shown here.

First test case for the first path that is 1, 2, 3, 4, 5, 32. Then second test case for the second path 1, 2, 3, 6, 7, 8, 32. Third test case for the third path 1, 2, 3, 6, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 8, 9, 30, 32. Fourth one is fourth path that is 1, 2, 3, 6, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 32 and the fifty one is for the fifth path that is 1, 2, 3, 6, 9, 10, 11, 12, 31, 32.

So, these are the minimum 5 test cases that are required one each for each of the linearly independent paths. So, this minimum 5 we require to perform our white box testing. We can have more as we have noted earlier, but at least 5 are required to create our test suit.

(Refer Slide Time: 31:01)



And like before for each of these test cases, again here ignore the typo it should be test case 1, test case 2, each of these test cases we provide the input, provide the expected output and we also provide the system output. So, whatever we achieve after providing this input to the program and the system generated output. If there is a match that means the test succeeds, if there is a mismatch between the expected output and the system output then that test fails that means there is some issue with the code.

(Refer Slide Time: 31:40)

Test suit 3:-
Input: toothpaste, 2
Expected Output: success message
System Output: success message

Test suit 4:-
Input: toothpaste, 0
Expected Output: fail message
System Output: success message

Test suit 5:-
Input: table, 2
Expected Output: fail message
System Output: fail message

Failed Test Case List:-

Failed Test Suite (Input, Output)	Function Name
1. ((toothpaste, -1), fail message)	addSoldItemDetail

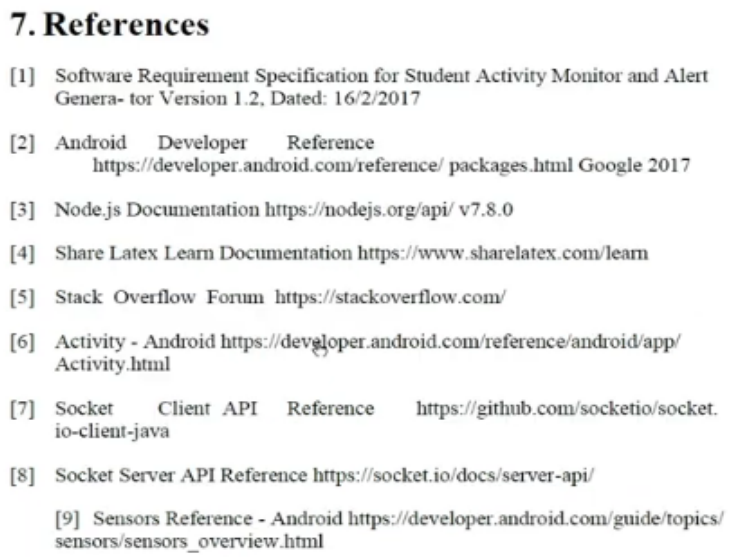
So, in our case we can see here that in case of test case 4 there is a mismatch. So, expected output is a fail message, whereas the system given output is success message. So, there is some issue with this piece of code, we need to take corrective action. So, among the 3 functions, we have seen that when we performed white box testing with the test cases, identify it with the help of CFG and the cyclomatic complexity analysis.

In one case, we encountered a problem that is the expected output and the system generated output did not match. That means there is some issue with the code and we need to address that issue that is the idea behind this testing. So, the failed test case needs to be recorded in the document itself so that later on modifications can take place.

(Refer Slide Time: 32:47)

So, with that we come to the conclusion of this document. Here we mentioned what we have found out and what can be done, like that one test case there was a failure, so that particular function needs to be corrected, to take care of that failure, these typse of things you can write in the conclusion.

(Refer Slide Time: 33:19)



And finally, the document ends with some references that were used to create this particular document. So, that is in summary what should be there in a test report. So, earlier we have seen several such reports, namely review-based testing report, black box testing report and in this lecture, we have seen white box testing reports. The common format we have seen. So, there will be a cover page followed by develop content followed by some sections which are common. And then we record the actual testing details.

I hope you enjoyed the lecture and you have understood the things that needs to be mentioned while you create a report. Remember that this is only one of several documentations that are part of this execution of the interactive software development lifecycle for better maintainability and refinement of the overall system. That is all for this lecture. Looking forward to meet you all in the next lecture. Thank you and goodbye.