

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

Lecture – 34
White-Box Testing

Hello and welcome to the NPTEL MOOCS course on design and implementation of human-computer interfaces, lecture number 29. At this stage, let us take a pause and reflect on what we have learned and what remains. As we have repeatedly mentioned in all the lectures almost we are learning on the development of interactive systems. Human-computer interfaces are nothing but interactive systems and we are learning a stage wise process to develop such systems.

At this stage, it may be useful to just recollect why we are going for this stage wise development. We are going for the development because for interactive systems an important consideration is usability of the system that means whether the users perceive it to be usable. Now, that requires us to first properly collect and analyse the user requirements, specifically requirements from the end users, not the clients of the product because these two need not be the same.

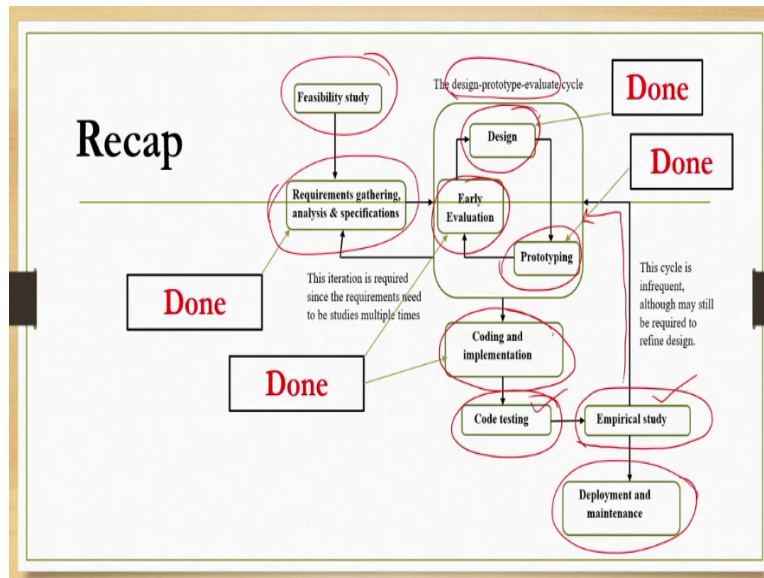
Once the user requirements are gathered, it is also important to convert these requirements to a proper design of the system and equally important are to test the system to know whether the system meets the usability requirements. So, we need to ensure that the system is usable as well as we need to ensure that the system is executionable, that means the system is executable. So, there are two requirements usability and executability.

Executability of course is required for every software product, but usability is required primarily for interactive systems or human-computer interfaces. In order to ensure that the system is usable, we make use of several methods. Also, in order to ensure that the product is executable we make use of several methods. Now, together these methods can be considered to represent a stage wise process of development of the interactive system.

In other words, what we are talking of is an interactive system development lifecycle. There are slight differences between interactive system development lifecycle and regular software

development lifecycle because in interactive system development lifecycle we specifically take care of usability. One such interactive system development lifecycle we are currently discussing which comprises of several stages. What are those stages? Let us have a quick look.

(Refer Slide Time: 04:04)



One is the feasibility study stage, which we have not covered in details in this course because that is not very relevant, although important but we have not covered it. Next is the requirement gathering analysis and specification stage. In typical software development lifecycle this stage is present, in interactive system development lifecycle this stage adds to what is already there in typical software development lifecycle in terms of gathering requirements for usability.

So, we need to gather requirements for software system as well as gather requirements from a usability point of view. Then there is this design prototype and evaluation cycle. Now, this cycle is required to take care of primarily usability concerns. So, objective is to design usable interfaces and interaction for interactive systems or human-computer interfaces. In order to do that, we need to first come up with a design based on the requirements that we have gathered in the earlier stage.

Then that design needs to be prototyped and evaluated, where evaluation takes place with experts. If some usability issues are found in the evaluation, then we refine our design and the cycle continues till we arrive at a stable interface design. Once a stable design is arrived at,

we go for converting this design to a system design or design of the code that is the second type of design that we are bothered about in this interactive system development lifecycle.

In code design, we primarily try to create a hierarchy of modules and submodules and then we define the relationships between them and the interfacing between those modules and submodules. Once system is designed, we go for implementing it with the coding and implementation stage that is the next stage. The implemented system is tested, then the whole system is tested again but not for testing the code rather testing the usability of the whole system.

Note the cycle here, at this stage if you find some usability issues then it may be required to go back to the design phase and the subsequent stages may need to be executed again. However, this cycle should not be frequent, should be maximum once or twice, otherwise the whole turnaround time maybe impractical. Once the system is found to be executable after code testing and usable after empirical study, we go for deployment and maintenance.

Now, these are the stages of an interactive system development lifecycle for systematic development of the human-computer interfaces. Among these stages, we have already covered requirement gathering analysis and specification stage. So we have learned how to gather requirement, broadly two types of requirements, functional and non-functional. We have learned how to gather functional requirements, we have learned how to gather non-functional requirements.

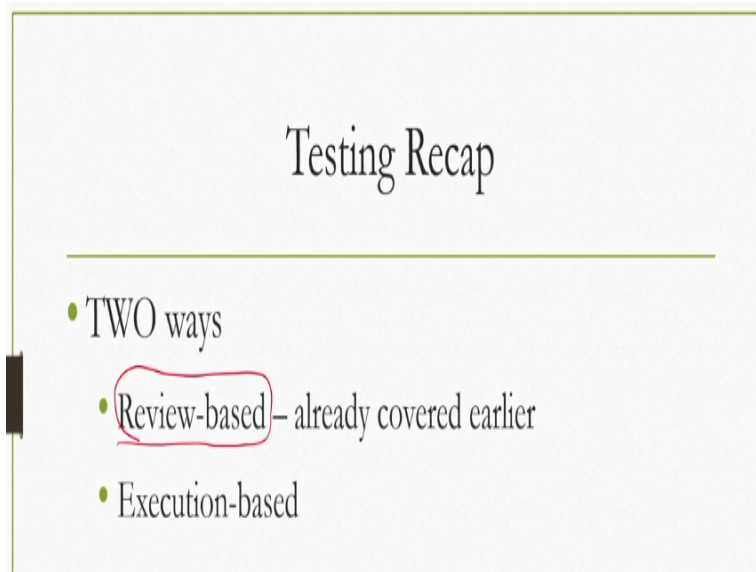
In this context, those non-functional requirements or usability requirements with techniques such as contextual inquiry and we have also learned how to convert non-functional requirements to functional requirements. Then we have discussed the design, prototype and evaluate stages. In the interface design stage, we have learned to use design guidelines. Then we talked about specific guidelines like Shneiderman's eight golden rule and Norman's seven principles.

We have also learned how to create prototypes. What are the different prototypes that are possible and we have learned how to evaluate quickly those prototypes with experts through methods such as cognitive walkthrough and heuristic evaluation. Next, we learned about system design where we learned about the modular design idea as well as languages to

express designs such as DFD or data flow diagram to express functional design or procedural design and UML or Unified Modelling Language to express object oriented design.

We learned about the coding and implementation stage, particularly talked about how to write good code, what are the good practices in coding. So, once coding is done, we need to test it for finding out errors with the code and rectify those errors. For that, we need code testing, currently we are discussing code testing. And in this lecture, we are going to discuss the remaining part of the code testing at the unit level that is white-box testing.

(Refer Slide Time: 09:54)



So in testing, if you can recollect we have broadly two ways of testing, one is the review based and the other one is the execution based. We have covered in details the review-based code testing. What we do in review-based code testing, here instead of executing the code with a computer, we manually go through the code, we just inspect the code, hand execute the code to find out faults with the code.

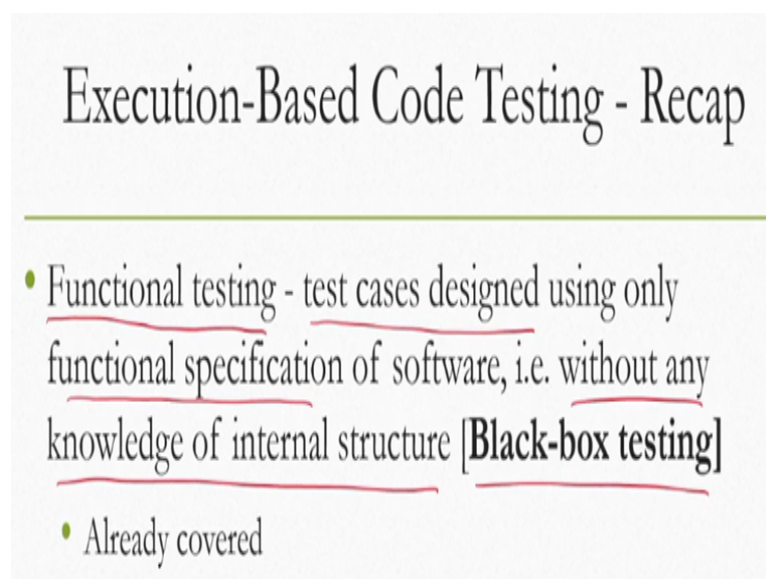
The purpose is to get a quick evaluation of the code done with a team of evaluators. So, this is again broadly of two types. One is code walkthrough in which there are some test cases designed and those test cases that means the input output pair are used to hand execute the code and see whether the output of execution matches with the expected output, if not, then there are some errors which are flagged.

In inspection-based code testing, the evaluators go through the code line by line to check for errors in terms of coding standards conventions, whether those were followed or not, whether

some common mistakes have been made like uninitialized variables or type mismatch and so on and so forth. So, you also saw how to create a document when we go for review-based code testing. The other code testing type is execution based.

In execution-based code testing, we actually execute the code with a computer with the help of a set of test cases or a test suit. And then see the output produced by the code after execution and output that should happen after execution that is expected output and observed, we see both are matching, if not then there are some issues. We note down the corresponding input cases and rectify the errors.

(Refer Slide Time: 12:45)



Execution-Based Code Testing - Recap

- Functional testing - test cases designed using only functional specification of software, i.e. without any knowledge of internal structure [Black-box testing]
- Already covered

Now, execution-based testing which is also a more formal and rigorous way of code testing can be done in two ways. One is functional testing. In this testing method, test cases are designed using only functional specification of the code without any knowledge of the internal structure how the function is written. This is also called black-box testing. We have gone through the details of the black-box testing in the previous few lectures. So, this testing method is already covered in details.

(Refer Slide Time: 13:19)

Execution-Based Code Testing

- Structural testing - test cases designed using knowledge
of internal structure of software [white-box testing]

There is another execution-based testing that is structural testing. In this case, test cases are designed using knowledge of the internal structure of the code. So, here we know the instructions that are part of a piece of code and based on that knowledge we try to design the test cases. This is also called white-box testing and we are going to talk about white-box testing in this and subsequent few lectures.

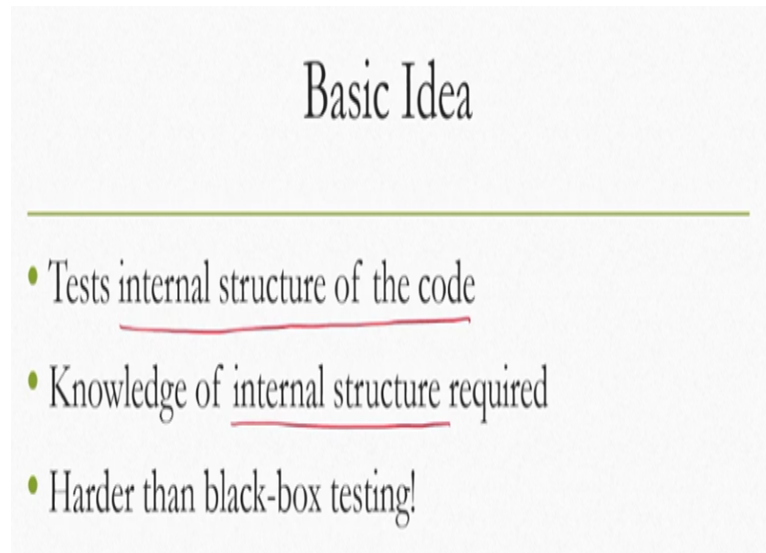
So, white-box testing is the focus of this lecture. Also at this stage, it may be useful to recollect that after every stage of the development lifecycle, we produce a document. Now that document can be the code itself or some reports. After requirement gathering analysis and specification stage, we produce a document that is called SRS or Software Requirements Specification. After design phase, we produce a design document.

After prototyping phase, we may produce details of the prototype. After evaluation of prototype phase we may produce the evaluation report. After code design phase, we may produce detailed code design document expressed in either DFD or UML. After coding phase, the code itself is the document along with that there may be some technical manual and user manual to understand the code.

After testing phase, we generate testing reports for the code. After review-based testing, we can produce one document, we have already seen how to create those documents, what should be the content of that document. Similarly, after black-box testing, we have seen case studies where such documents are discussed to give you some idea of the content of such documents.

Similarly, after learning about white-box testing we will see how to generate a white-box testing report, what should be the content in subsequent lectures. But first, we will learn the basics of white-box testing.

(Refer Slide Time: 15:52)



Basic Idea

- Tests internal structure of the code
- Knowledge of internal structure required
- Harder than black-box testing!

So, let us begin our topic for this lecture on white-box testing. As we have already mentioned, white box testing is meant to test internal structure of the code. So, in black-box testing, we tested each function assuming it to be a black box, we do not know what is inside the function, so there is no test of the internal structure. In white-box testing, we do the opposite.

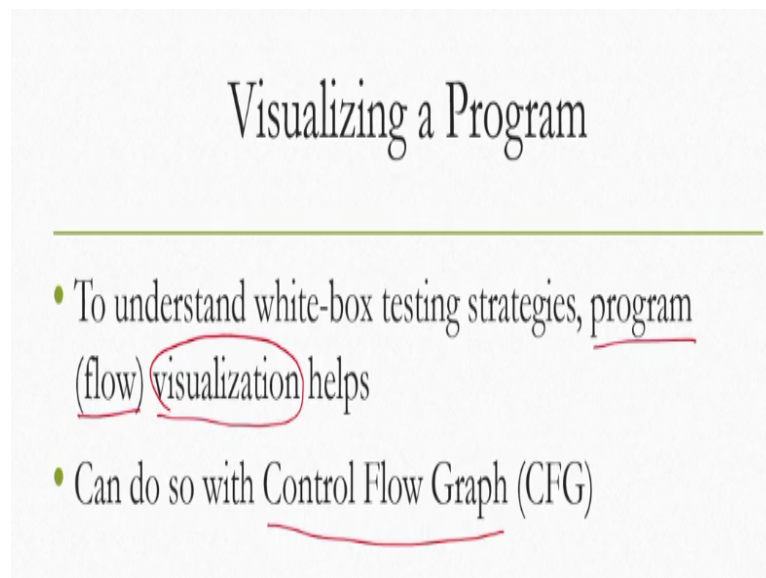
We basically first learn about the internal structure of the code that is the set of instructions that are written as part of the code and then based on that we design test cases to test the structure. Obviously, if we want to do that, then we need to know the internal structure of the code, how the code is written that was not required in case of black-box testing. Since we have to know the internal structure, it may sometime appear to be harder than black-box testing.

Because here the internal structure can be complex and accordingly design of test cases may be complex like in black-box testing where we need to only bother about the input domain and output domain rather than how the code is written. So, that is possible that the white-box testing turns out to be harder to do than black-box testing for complex pieces of codes. In

order to make our job easier, in order to make the design of test cases easier, we need some method.

A complex code is very difficult to understand we all know. Now, if we want to design test cases based on the knowledge of a complex code, then somehow we have to make sure that the code is understandable easily so that we can go ahead with design of the test cases. Now, how to make a code understandable easily?

(Refer Slide Time: 18:06)



One way to do that is to visualize the code. In other words, if we can visualize the flow of the program, then that makes it relatively easier to design the test cases. So, the key thing here is to be able to visualize the program and the program flow in a code. So, there is one approach through which we can perform this visualization that is known as Control Flow Graph.

So, this is a graph like structure which can be used to express program or rather the flow of a program to help us visualize the flow. So, in order to be able to understand white-box testing approach, it is helpful to know about the control flow graph and how to use it to design test cases.

(Refer Slide Time: 19:16)

What?

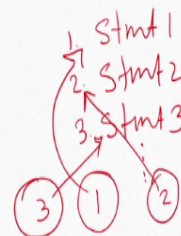
- Graphical representation of sequence of instruction execution
- How the control flows through the program

So, let us quickly learn about control flow graph or CFGs. So, as the name suggests this is a graph which represents control flow in a program. So, it is a graphical representation of sequence of instruction execution. So how the instructions get executed, in which sequence is represented graphically with the control flow graph. In other words, it represents how the control of the program flows through the program. So, both are same. To visualize the sequence of instruction executions and to visualize the control flow are the same concepts and control flow graph helps us to do just that visualize the flow of control in a program.

(Refer Slide Time: 20:10)

CFG – How to Construct?

- Assign numbers (in sequence) to all statements of a program
- Create a 'node' in the CFG for each numbered statement



So, the definition is simple. A control flow graph is a visual representation of the flow of control in a program. Now, since it is a graph, so it consists of nodes and edges, so there will be some nodes and there will be edges and since it shows some direction, so edges will be

directed. So, in other words it is a directed graph. Now, what should we consider to be a node and what should we consider to be an edge?

How to create edges between nodes these are some of the issues that we should understand before we are able to construct a control flow graph. So, then first what can be the nodes in a control program or CFG? What we can do is we can assign numbers in sequence to all the statements of a program. So, program consists of instructions or statements, each statement can be assigned a number, a unique number of course.

Then, we can create a node in the CFG for each numbered statement. So, if we have a program say statement 1, statement 2, statement 3 and so on, so we can assign numbers 1, 2, 3, etc., and then can create a node like this a circle with this number 1, circle with this number 2, circle with the number 3. So, each will be a node representing a statement. So, these represent the statement, this one will present statement 1, this one will represent statement 2 and so on.

So, number each statement in the program, for each numbered statement create a node. In the node, a node is nothing but a circle with a number inside it where the number corresponds to the corresponding statement in the program.

(Refer Slide Time: 22:32)

CFG - How to Construct?

- Assign numbers (in sequence) to all statements of a program
- Create a 'node' in the CFG for each numbered statement
- Add an 'edge' from one node to another if execution of the statement representing first node results in transfer of control to other node

1. Stmt 1
2. Stmt 2
3. Stmt 3

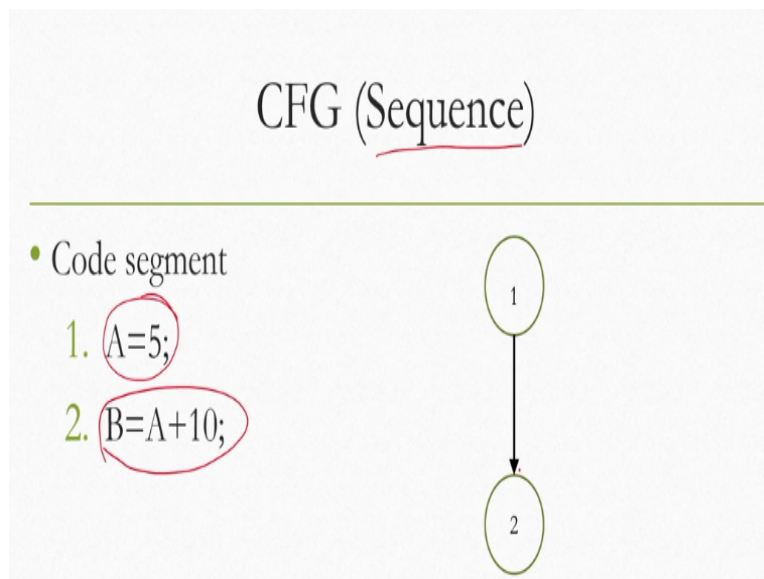
Then, what we can do is so once the nodes are created like 1, 2 3 etc, we can add an edge from one node to another node if execution of the statement representing the first node results in a transfer of control to the other node. In other words, suppose after statement 1 which is

numbered as 1, statement 3 happens numbered as 3, in between there is another statement, statement 2 numbered as 2.

Now, if I create these three nodes 1, 2 and 3 in my program, after 1, 2 does not happen, 3 happens so then I should not add a node between 1 and 2. These nodes should not be there, instead there should be a node between 1 and 3. So, whichever statement gets executed after the previous statement an edge should be added between those two corresponding nodes that is the basic idea.

So, we first number each statement, create a node for that number or the number statement, a node means a circle with a number inscribed inside which represents the statement and then add an edge between nodes when execution of the first node results in transfer of control to the second node that is the program, then executes the second node after the first node is executed. So, this is a simple idea for constructing a CFG that is create node and add edges as per the flow of control.

(Refer Slide Time: 24:52)



Let us see few examples about different types of control flow in a program and the corresponding construction of CFGs. The most simple form of control flow is of course as we all know sequential control flow. So, the statements are listed in sequence and they are executed also in sequence. Suppose, this is a code segment consisting of two statements, an assignment statement equal to 5 and some computers and statement B = to A + 10.

Now, if these statements are executed in sequence that means after 1, 2 happens, so what should be the CFG for these two statements? For 1 we have a node a circle with 1 one inscribed inside, for 2 again we can have a node that is another circle with 2 inscribed inside and since they are sequential 2 gets executed after 1 is executed. So, we simply add directed edge from 1 to 2. So, this indicates that 2 gets executed after 1.

(Refer Slide Time: 26:18)

CFG (Selection/Condition)

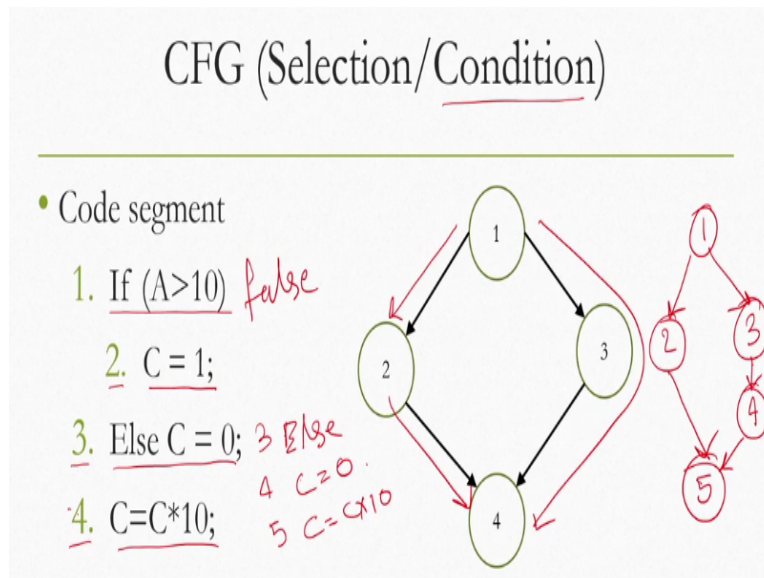
- Code segment
 1. If (A>10)
 2. C = 1;
 3. Else C = 0; False
A B C = 0;
 4. C=C*10; 5

Now, let us move to a slightly complex code segment that is conditional execution. So, now suppose there are these 4 statements. First statement is an if statement that is if followed by some condition A greater than 10. Second statement is what happens if this condition is satisfied that is C is assigned to 1. Three says else C is assigned to 0. Four says C = C into 10. Here one point is that sometimes we tend to write else C = 0 in two lines that is instead of writing it in a single line you can write else C = 0.

So, then one confusion may come with that this should be 3 and this should be 4, then whether it should be the last one should be 5 or both should be considered as a single statement like we have done here as 3. Since, only else has no significance, so it is preferable that we consider them together as we have done here, although even if we considered them separately then also there is no effect on the CFG.

Only thing is it gets more nodes and the resulting CFG becomes complex to manage. So, considering them in single line is perfectly alright rather than considering them as two separate nodes. So, that is the thing that often may create some confusion, so you should keep this in mind.

(Refer Slide Time: 28:08)



Now, let us see assuming else $C = 0$ represents a single node in the CFG let us see these 4 statements and how they can be converted to a CFG for visualization. So, for statement number 1 we have this node with 1 inscribed inside. For statement number 2 we have this node with 2 inscribed inside. For statement number 3 we have another node where 3 is inscribed inside and finally we have statement number 4, so there will be another node, but before that let us see how the control flows.

So, depending on the condition it can go to either 2 that is if this turns out to be true, then it can go to 2, otherwise it can go to 3 and for the fourth statement there is another node with 4 inscribed. Now from both 2 and 3, 4 can be reached. So, if this condition is true, if this is true then the path followed is 1, 2 and 4. If this is false, in that case the path followed is 1, 3 and 4. So, depending on the condition, a specific path is followed.

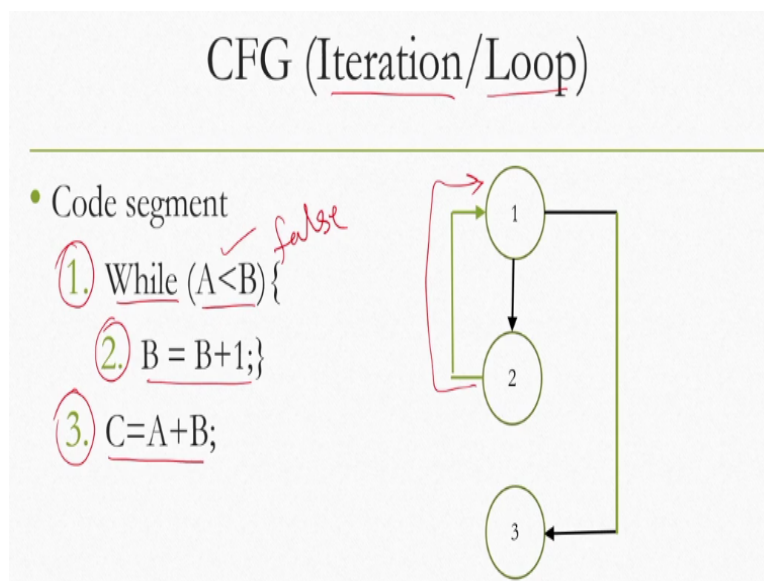
So, when conditional statements are there then we can have this type of CFGs. Now, let us have a quick look at what would have happened if we would have considered this statement 3 not as a single statement but as two statements. Three says else, then 4 says $C = 0$ and 5 says $C = C * 10$. So, in this case the last statement becomes 5 instead of 4. So, then we will have 1 as before 2, 3, 4 and 5.

Now, if condition is true then we will go to 2, if condition is false then we will go to 3. Now, since this is simply an else statement, then 3 and 4 are sequential, so 3 will be followed by 4 and then we will come to 5. So, there will be slight modification with the addition of one

node and one edge. So, earlier there were 4 nodes if we would have considered 3 as a single statement. Now, there are 5 nodes.

Similarly, earlier there were 4 edges, but now, if we consider 3 to be not a single statement but two statements then there will be 5 edges. Now, if we extrapolate it to the whole program, then the edges and nodes will increase manifold and the resultant CFG may become more complex to visualize. So, it is preferable to go for single statement when else type of statements are there. So, that is about conditional statements and how to construct the CFG out of those statements.

(Refer Slide Time: 31:45)

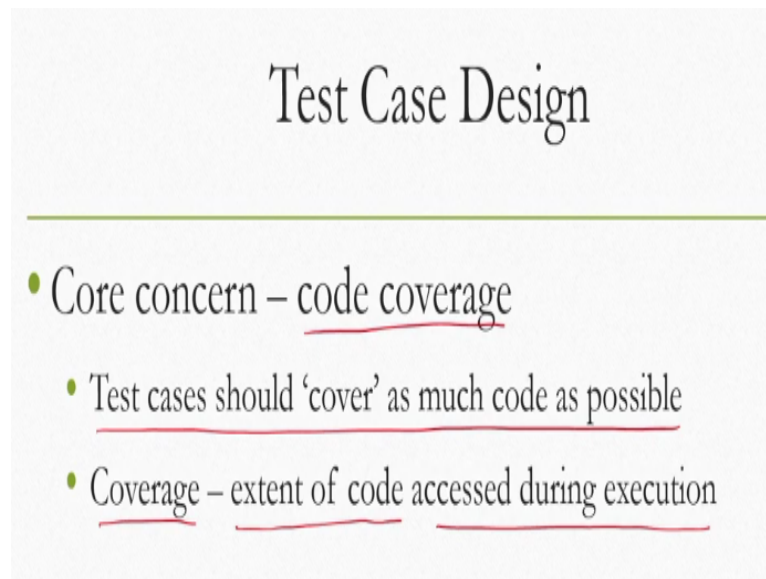


Next, let us think about another important construct in a program that is loop or iteration. Let us consider this segment while followed by a condition then some statement $B = + 1$ followed by another statement. These three statements are numbered as 1, 2 and 3. So, then how the CFG will look like? For statement 1, we will construct one node as shown here. For 2 we will construct another node.

Now, there will be a connection from 1 to 2 which we can generally expect when the condition is satisfied. So, there will be one directed edge from 1 to 2, but then it is a loop or an iteration. So, this condition will be checked repeatedly that means this instruction or node 1 will be visited repeatedly from 2 so after node 2 gets executed, so then we will add a loop edge from 2 to 1. For statement 3, there will be another node and from 1 we can go to 3.

Note that here from 2 there is no edge, only from 1 when the loop condition becomes false we can come to 3. So, this is typically how we can construct a CFG for a loop. So, you have seen three important constructs and the corresponding CFGs namely for sequential statements, for conditional statements and for iterative statements. So, with that basic knowledge now let us move forward and see how the internal structure of a code can help us construct test cases when we visualize the code using CMGs.

(Refer Slide Time: 34:00)



Test Case Design

- Core concern – code coverage
- Test cases should cover as much code as possible
- Coverage – extent of code accessed during execution

So, when we go for test case design for white-box testing, our core concern is coverage of the code as much of the code as possible that means as many instructions in the code as possible. For that of course, we need to know how many instructions are there, how the control flows from one instruction to another that is the internal structure. So, the concern is test cases should cover as much code as possible where the coverage means extent of code accessed during execution.

So, when we provide a test case, a set of statements in the code gets executed. So, our objective should be to come up with test cases that forces the system to execute as many statements as possible.

(Refer Slide Time: 34:50)

Test Case Design

- Many approaches
 - Statement coverage ✓
 - Branch coverage ✓
 - Condition coverage ✓
 - Path coverage ✓
 - Control flow testing ✓
 - Data flow testing ✓

Keeping this objective in mind, we can approach the test case design problem in several ways. There are many approaches. We can have an approach called statement coverage approach. We can have branch coverage approach. We can have condition coverage approach. We can have path coverage approach. We can have control flow testing approach, data flow testing approach and so on.

Statement coverage, branch coverage, condition coverage, path coverage, control flow testing, data flow testing these are all methods to achieve the same goal that is to come up with test cases that help us execute as many statements as possible to unearth errors in the code. Among them, we will be focusing on the first four that is statement coverage, branch coverage, condition coverage and path coverage.

(Refer Slide Time: 36:04)

Test Case Design (Statement Coverage)

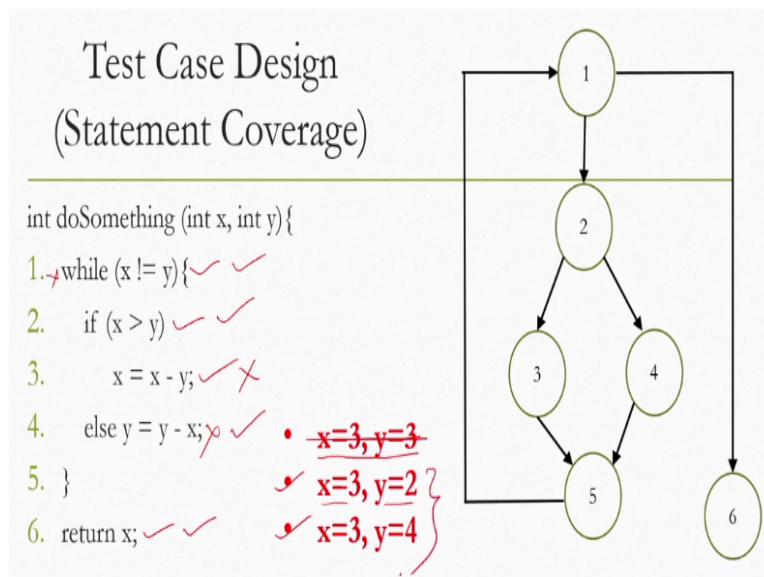
- Aims to design test cases such that every statement is executed at least once
 - Idea - unless a statement is executed, hard to determine existence of error in that statement
- Note - executing a statement once and observing 'correct' behavior for 'some' input does not guarantee 'correct' behavior for all input values

So, let us start with the idea of statement coverage what it means. So, when we are trying to design test cases using the statement coverage approach, our aim is to design cases such that every statement is executed at least once. So, when we are aiming for statement coverage objective is to have test cases that forces the system to execute every statement at least once. The idea is simple, unless a statement is executed, it is hard to determine the existence of error in that statement.

So, this is the most intuitive way of looking at testing a code. One note of caution though, executing a statement once and observing correct behaviour for some input does not guarantee correct behaviour for all input values. So, this is one very crucial thing to note that we may have come up with test cases which generates correct behaviour for all statements, but that does not mean that for all input this assumption will hold.

Remember that here we are not talking about first creating equivalence classes and then designing test cases which can be representative of all inputs in that class to manage the number of inputs. So, here we have to look at the problem in a different way.

(Refer Slide Time: 37:47)



Now, then with that basic aim, let us see a code segment and how we can design test cases by constructing a CFG from the code segment and then going forward to design the test cases. So, suppose this is the code segment, a function having six statements. Now to construct a CFG from these 6 statements, we need to take into account the nature of the statements. So, it is a while loop having an if else construct.

So, both, iterative sequential, so all the 3 constructs are there iterative, sequential as well as conditional. So, let us see how the CFG looks like. So, for each statement we have this node. So, there are 6 statements, so 6 nodes; 3, 4, 5 and 6. Now, since statement 1 represents an iterative statement, so there is a loop construct from 5 here to here because here the loop starts and here it ends.

Now, you may think that having a simple bracket does not necessarily require you to create a node, but that thing we have done here to let the discussion happen in a simple manner, otherwise we could have equally ignored the statement and the corresponding node but then showing or visualizing the code would have been a bit difficult. So, to avoid that to keep the matter simple, we just included this node here.

So, there is this loop construct from 5 to 1. Now, once this condition is true, as long as this condition is true from 1 the control comes to 2, now 2 is a conditional construct, so there are two parts. From 2 if it is true then it can go to 3, if it is false then it can go to 4. And then if it is false, then it comes back to the first statement and if it is true then it goes to the sixth statement from 1 and come back to 6.

So, this is the construct, I hope the construct is understandable to all of you the CFG how it is constructed with this knowledge now, suppose we want to execute each and every statement, then what kind of test cases we should design? So, if we have inputs like $x = 3$ and $y = 3$, then which statement it executes? So, while fails, so statement 6 gets executed. If $x = 3$, $y = 2$ then it enters the loop.

Now $x > y$, so this gets executed, this gets executed and this gets executed but not this one for this second input and then 6 gets executed. Now, when $x = 3$, $y = 4$ then this gets executed, this gets executed, this one gets executed, this does not get executed and this one gets executed for this third pair of input. So, you can see that we can execute all the statements if we provide these three.

In fact, since 6 gets executed anyway for the second and third input, so first one we can simply ignore. So, we can have only these two.

(Refer Slide Time: 42:06)

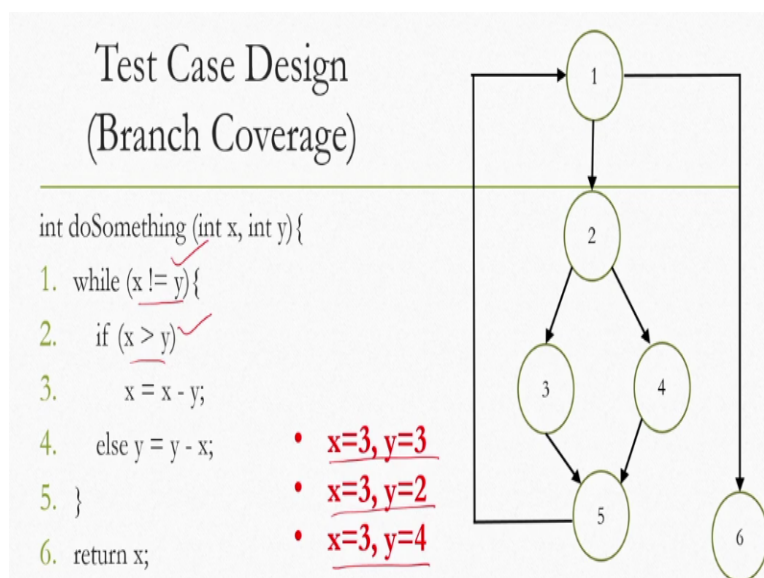
Test Case Design (Branch Coverage)

- Test cases designed to cover each branch condition (both true and false values) - also known as 'edge testing'
- Guarantees statement coverage - "stronger" strategy compared to statement coverage

Now, let us move to the next approach that is branch coverage. Here the idea is that the test cases are designed to cover each branch condition, both true and false values. Now, this approach is also known as edge testing. So, what happens here? It guarantees statement coverage, stronger strategy compared to statement coverage. So, in statement coverage, we have to look for inputs which executes each and every statement.

In branch coverage if we look for inputs which only covers the branch conditions for both true and false situations, then it ensures the statement coverage as well. So, our effort in finding out test cases are less in that sense it is stronger, a stronger approach than statement coverage.

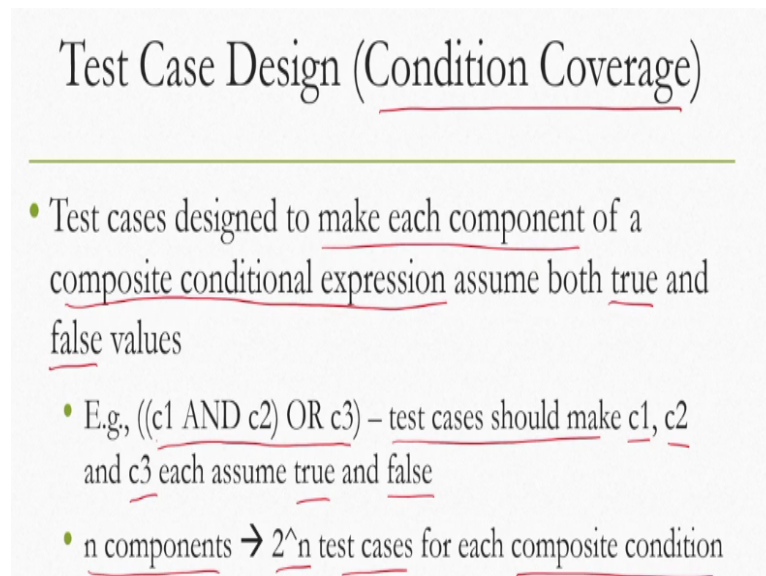
(Refer Slide Time: 43:06)



So, let us consider the same example. So, we have this CFG and if we are only focusing on the branch coverage, so this branching can happen here or here. So, true and false conditions both we have to check. So accordingly we can decide on these three. Note that the input pairs are the same and they anyway perform the statement coverage. So, these three inputs are designed to take care of true and false conditions in both branching points here and here.

So, here we are only focusing on identifying test cases that result in true or false conditions in the branching statements, we are not bothered about whether that other statements get executed or not. So, we are restricting our search space into finding out branching conditions only, so our effort becomes less but at the same time it ensures that other statements get executed.

(Refer Slide Time: 44:13)



Test Case Design (Condition Coverage)

- Test cases designed to make each component of a composite conditional expression assume both true and false values
- E.g., ((c1 AND c2) OR c3) – test cases should make c1, c2 and c3 each assume true and false
- n components → 2^n test cases for each composite condition

Third is condition coverage. So here what happens is the test cases are designed to make each component of a composite conditional expression as you both true and false values. So here the focus is specifically on conditional expressions. For example, if this is a composite conditional expression, c 1 and c 2 or c 3, then test cases should make c 1, c 2 and c 3 each assume true and false values.

So, we are focusing only on the conditional statements and true and false values for each component of the conditional statements. Earlier in the branch coverage or edge coverage, we are considering true and false value for the entire statement, now here for each component we are considering true and false value. So, it is even more complex than the branch condition.

So, if there are n components, 2^n test cases for each composite condition has to be designed.

(Refer Slide Time: 45:27)

Test Case Design (Condition Coverage)

- Branch testing – simplest condition coverage strategy (true/false values considered for whole condition rather than individual components)
- Guarantees branch and statement coverage – “stronger” strategy compared to both (may be impractical if conditions are complex)

So, branch testing is one form of condition coverage approach, simplest condition coverage strategy, true false values are considered for whole condition rather than individual components. But, in general we have to focus on individual components rather than the whole condition. So, if we are doing that then that guarantees branch as well as statement coverage. So, this approach condition coverage is even stronger than the branch strategy or the statement coverage strategy.

We may note here that this approach may be impractical if conditions are complex that means the number of test cases need to be designed can increase to a very large number, which may become unmanageable.

(Refer Slide Time: 46:22)

Test Case Design (Path Coverage)

- Path - a node and edge sequence from starting node to a terminal node of CFG of a program
 - Note - CFG can have more than one terminal node

And finally, we have the idea of path coverage. What it says? It says that test cases should ensure all linearly independent paths in the code are executed at least once. So, it introduces a new concept linearly independent path, what is a linearly independent path in a CFG or control flow graph? First what is a path? As the name suggests in a graph path means a series of nodes and edges.

So, a node and edge sequence from starting node to a terminal node of a CFG of a program, of course CFG is corresponding to a program only, that is called a path. We can define more than one terminal nodes in a CFG, so accordingly we can have several paths.

(Refer Slide Time: 47:16)

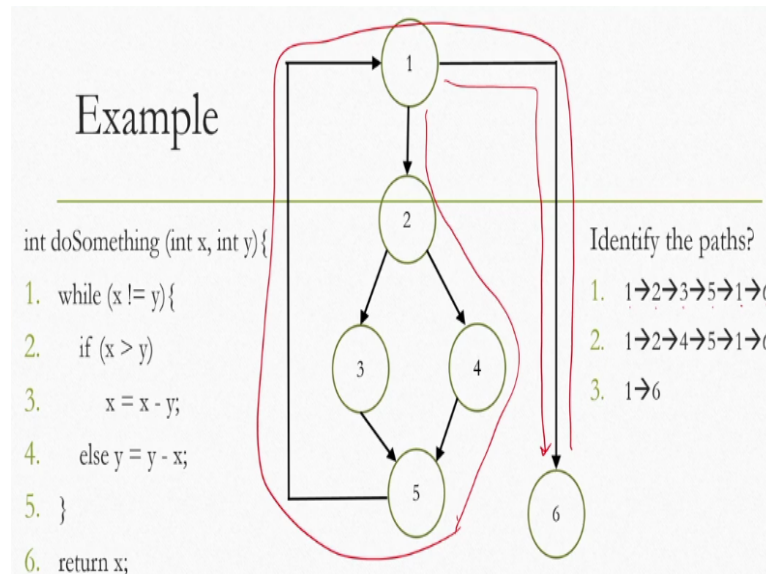
Test Case Design (Path Coverage)

- Linearly independent path - any path with at least one new edge/node not included in any other linearly independent paths of the CFG
 - Sub path of another path not linearly independent

Now, once we know what is a path, let us try to understand what is a linearly independent path. Any path with at least one new edge or node not included in any other linearly

independent paths of the CFG is defined as a linearly independent path. So, if we define a path which is not a sub path of any other path, then that path can be a linearly independent path.

(Refer Slide Time: 47:52)



So, in this example for this code segment that we are discussing, we have this CFG comprising of 6 nodes and how many edges are there? 1, 2, 3, 4, 5, 6, 7 edges. So there are 6 nodes, 7 edges to represent this code segment. In this CFG let us try to identify the linearly independent paths. So, first let us try to identify the paths, what are the paths? One path can be the sequence the node number 1, 2, 3, 5, 1, 6 this can be one path.

Let me just clear it. Then there can be another path 1, 2, 4, 5 then 1, then 6 can be another path. Simply 1 to 6 can also be another path from starting node to a terminal node. So, these are the paths and what are the linearly independent paths, how do I identify those?

(Refer Slide Time: 49:36)

Cyclomatic Complexity

- A measure of upper bound of number of linearly independent paths for a CFG

There is a concept called cyclomatic complexity. It actually is a measure of upper bound of number of linearly independent paths for CFG.

(Refer Slide Time: 49:55)

CC Computation

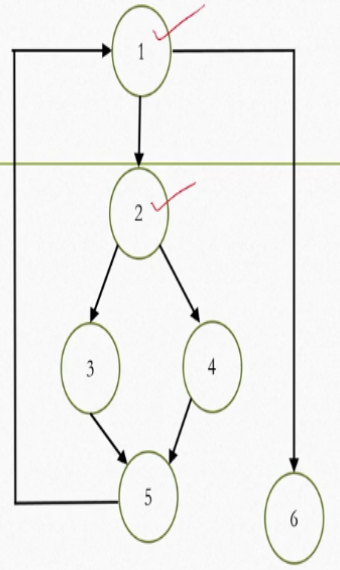
- One way to compute: $E - N + 2$
 - E = no of edges
 - N = no of nodes
- Another way: $D + 1$ (D = no of decision statements)

So, what it tells us is a way to identify linearly independent paths. So, let us see how we can compute this cyclomatic complexity. One way to compute is with this expression $E - N + 2$ where E represents number of edges, N represents number of nodes. Another way is with this explicit $D + 2$ where D represents number of decision statements. So, there are two ways to compute cyclomatic complexity. One is the expression $E - N + 2$ that is in terms of edge and node, number of edges and number of nodes. Other one is $D + 1$ where D represents number of decision statements, so in terms of decision statements that are part of the code.

(Refer Slide Time: 50:42)

CC Computation - Example

- 1st method
 - $E=7, N=6$
 - Complexity = $E - N + 2$
= 3
- 2nd method
 - $D = 2$ (nodes 1, 2)
 - Complexity = $D + 1 = 3$



Given that definition of cyclomatic complexity, let us try to understand how many linearly independent paths can be there. If we follow the first method, we can have the complexity value to be 3 because edge number is 7, node number is 6, so $E - N + 2$ gives us $7 - 6 + 1$ or the value 3. If we follow the second method, then the decision statements or decision nodes are 2, this node 1 and node 2, $D = 2$. Then the complexity is $D + 1$ or 3.

So both the methods lead to the same value that is 3. So, in this particular CFG we can have 3 linearly independent paths and those 3 paths we have already mentioned earlier. So, let us summarize what we have learned. So, in this lecture we covered the basic concepts of white-box testing where we mentioned that we need to understand the structure. Now, to understand the structure visualizes and helps, so there is a way to visualize the code that is called CFG or control flow graph.

We have seen different constructs like sequential construct, iterative construct and conditional construct and how to construct a CFG out of these constructs. And we have seen how to design test cases for white-box testing with a CFG in hand. So, essentially different approaches can be applied. We have learned about four approaches; statement coverage which is the most intuitive idea of designing of test cases.

Then branch coverage which is stronger than statement coverage in the sense that if we follow branch coverage, then anyway statement coverage will be done. Then there is condition coverage, which is even stronger than statement coverage, but sometimes it may

become impractical to follow it. And finally, we learned about path coverage which is very strong and it can cover the other approaches.

For path coverage, we need to identify linearly independent paths and design test cases to execute statements that are part of these paths. So, first task is to identify linearly independent paths. In order to do that, we need to first know how many such paths are there and then as per definition we need to identify those. To know how many sides paths are there in our CFG we can make use of the concept of cyclomatic complexity.

We have seen how to compute those and using that computation we can know the number and accordingly we can identify the paths. All these concepts we have learned with examples to make things easier for you.

(Refer Slide Time: 54:01)

Reference

- Rajib Mall (2018). Fundamentals of Software Engineering, 5th ed, PHI Learning Pvt Ltd. **Chapter 10** ✓
- Roger S Pressman (2020). Software Engineering: A Practitioner's Approach, 9th ed, McGraw-Hill Education, New York, **Chapter 19-21** ✓

Whatever I am discussing or we have discussed so far can be found in these books Fundamentals of Software Engineering chapter 10 or Software Engineering: A Practitioner's Approach chapters 19 to 21. I hope you understood the concepts and enjoyed it. Next lecture we will go through one case study of a white-box testing report as a document as the outcome of the stage like we did before. Looking forward to meet evolve in the next lecture. Thank you and goodbye.