Lecture – 16
Software Design and Integration Testing

Hello again, this is the second lecture of the fourth week. What I wanted to do today was to continue with graph based coverage criteria, we saw how to test code based on a graph coverage criteria. The basic idea was to take the control flow graph model it as a graph and then consider the various structural coverage criteria to be able to do graph coverage and then we augmented the CFG with definitions and uses, looked at D u pairs, D u paths and tested it for data flow criteria.

(Refer Slide Time: 00:47)

And what did we achieve by doing this? It was basically white box testing of code, we could cover the code see what statements were executed, how many times loops were executed where loops skipped and so on, and through examples we saw that it was indeed useful to find the errors in codes.
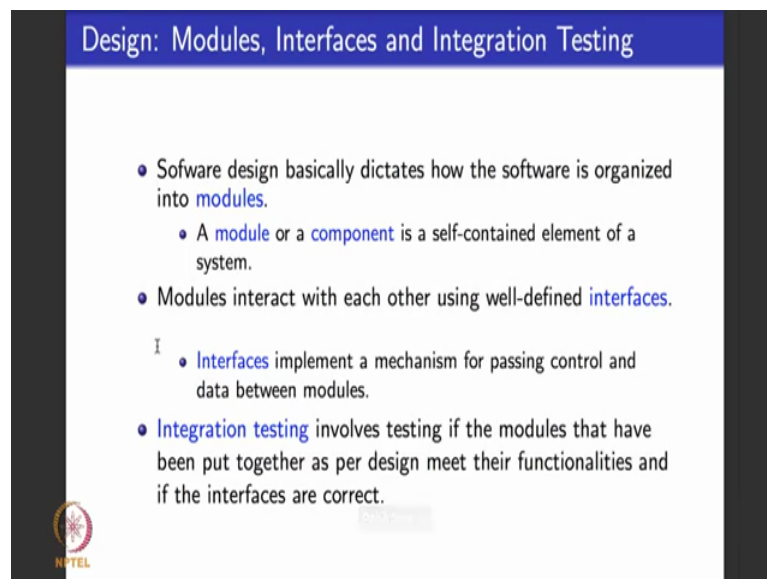
Now, I want to continue to use graph coverage criteria, but instead of considering code I want to be able to work with design, graphs for design models right. When it comes to design the phase of testing that we are looking at is what is called integration testing, and integration testing basically assumes that design puts the software into various modules it

puts together these modules and test set. So, what I want to spend some time on with this lecture is to help you understand the traditional or the classical view of integration test.

What is integration testing, how has it been defined in the popular textbooks available on software testing, how do graphs corresponding to integration testing look like, right. So, today we will look at the classical view of integration testing any old book on software testing that is existed for many decades will use this view of integration testing. In the next lecture what I will tell you is I will tell you graph models for design and how to apply structural and dataflow coverage criteria on those graph models we have to define slightly new structural and dataflow coverage criteria to be able to cater to graph models for design we will do that in the next lecture.

This lecture we will just spend on understanding integration testing as it is always been classically presented in several text books and software texts.

(Refer Slide Time: 02:30)

So, when I say software design what is it? Software design for a large piece of software is basically tells you details about how its large piece of software is split into modules since their software runs into several thousands of lines, of code or even millions of lines of code it is all not going to be one flat piece of code right; it is going to be split into several components each component is going to be designed and implemented by maybe by separate team, separate individuals and then they are all going to be put together modular way to be able to constitute the whole piece of software.
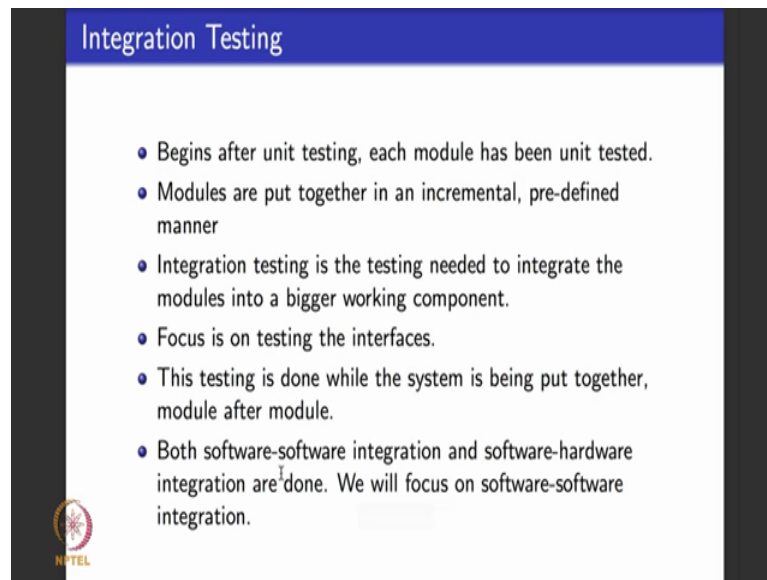
Typically for software popular design languages are UML unified modeling language, system ML system modeling language and some proprietary embedded software are modeled using proprietary design languages or even languages like simulink state flow etcetera. In this lecture we really will not consider each language for design and look at models for these languages instead I will assume that design is just a way that tells you how the software is split across modules and then we will see how these modules are put together and tested. So, what is a module? A module or a component these terms will be used interchangeably it is a self contained element of a piece of a software or a system.

What do I mean by self contained? By self content I mean that it takes inputs executes, and produces outputs. For its execution it is not dependent on another piece of software running or calling another piece of software or availability of data from another piece of software right. So, it executes as a standalone entity, stops its execution produces outputs and stops. Simplest view of a module would be a procedure in C or a method in Java right, individual method. Modules interact with each other using what are call interfaces, we will see what are the various kinds of interfaces, but interfaces always have to be well defined.

What is an interface? An interface you can think of it as implementing a mechanism for passing control and data between modules. So, one module can call another module or procedure or a function can call another procedure or function in which case it passes control to the other called procedure or function, and when it calls it can call it with certain data right it says you, please return this data to me or run it with this input and return the following input. So, it also exchanges data between the call procedure and the calling procedure, right.

So, interfaces are basically mechanisms that facilitate this call and return, control of call and return of data. Integration testing is basically a phase of testing that deals and tests if all the modules have been put together properly as per design and whether all the modules put together meet the functionality that they are supposed to meet together.
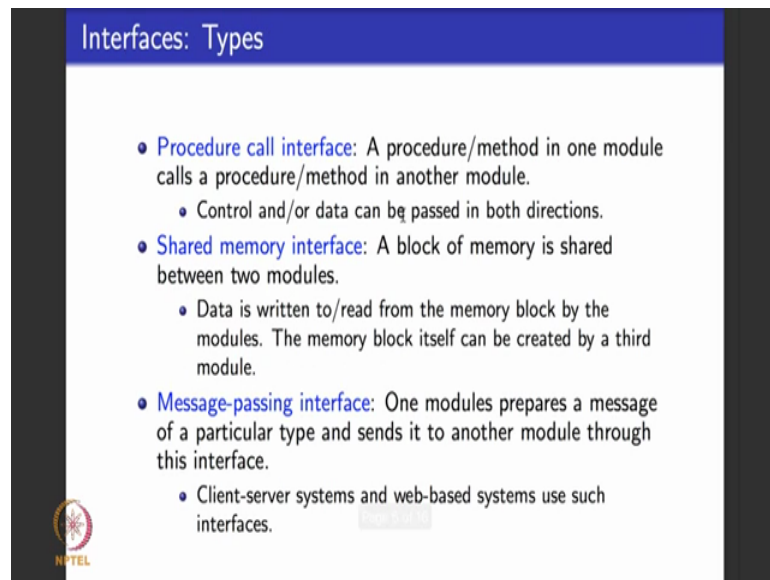
(Refer Slide Time: 05:36)

Integration Testing

- Begins after unit testing, each module has been unit tested.
- Modules are put together in an incremental, pre-defined manner
- Integration testing is the testing needed to integrate the modules into a bigger working component.
- Focus is on testing the interfaces.
- This testing is done while the system is being put together, module after module.
- Both software-software integration and software-hardware integration are done. We will focus on software-software integration.

When its integration testing typically done if you remember the phases of testing and the levels of testing that we saw right in the first week integration testing immediately follows unit testing right. When I mean integration testing here I mean integration testing of software modules, there is also what is called software hardware integration testing which involves putting the piece of software in the hardware and then testing. That is not the focus for today's module. We mean software-software integration, putting together individual code components and testing the software right.

Modules can be put together in several different ways, but whatever; however, they are put together the way of putting them together is predetermined and its incremental it is not like I suppose I have 15 modules, we develop all the 15 modules and one fine they just sit and put together right we do not do that. We have a well defined incremental way of putting together all the modules. Integration testing basically what it does is that it test if the modules that have been put together are working fine, while testing for this it focuses on testing the interfaces - how are the calls and returns happening, the focus is on that designing test cases to see if there are any kind of errors on these interfaces. So, we spend some time understanding what are the various types of interfaces and how we will be dealing with them
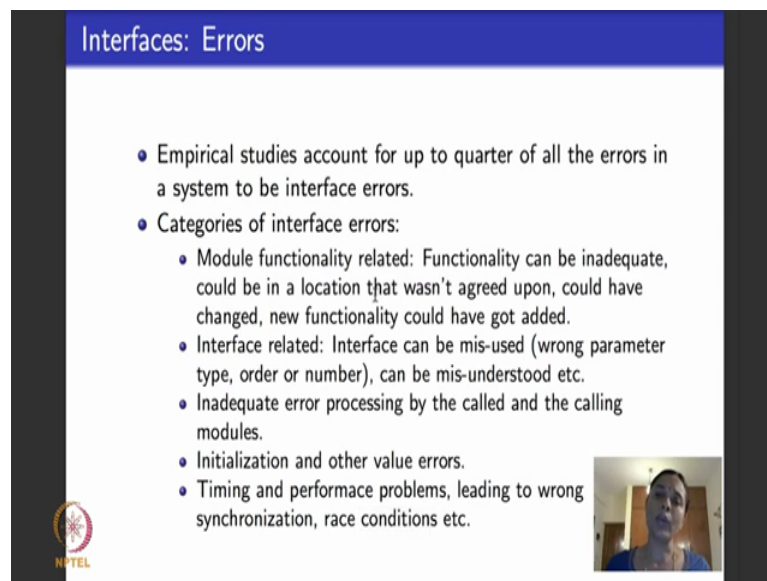
(Refer Slide Time: 06:55)

The most common interface is what is called procedure call interface. So, here we assume that a software procedure a function or a method in one module calls another procedure or a method in another module right, this call could be called parameter, passing call by value right several different ways that you would study in any standard C programming would apply to this kind of interfaces. Both control and data can be passed I mean both the direction. So, when it calls another module it transfers control to the other module and while it transfers control it also passes some data and when the call module returns back the control to the main module while returning back it could pass some data.

Second popular type of interface is what is called a shared memory interface here we assume that there is a block of memory that is available at some place. This block of memory could be within one of the modules that are sharing the interface or could be available or created by a third module in a different interface. How do these two modules communicate? They communicate by reading from that block of memory and writing onto that block of memory. So, this is a common shared global location and these modules communicate by reading from and writing to that value. Typically several systems software, like operating systems, embedded software they all use these shared memory interfaces.

The third popular interface is what is called a message passing interface. Here what we do is that we assume that there are dedicated channels or buffers available and various components communicate by sending and receiving messages on these channels. Popular examples of systems that communicate with message passing interface are what are called client server systems right, IoT is a very popular system these days that basically says that each individual IoT device is a client and then there are aggregate or nodes, common nodes. So, it can be thought of as a kind of a client server system. Web apps, web applications that we all popularly use right, social networking applications like Facebook, Twitter and all other things they also exchange a lot by using message passing interfaces. So, this gives you a broad idea of what the various interfaces could be.

But the testing methodologies that we would see today are independent of these kind of interfaces this is just to educate you a bit about various interfaces that could actually exist. When we look at testing techniques for integration testing for today's lecture we will abstract out and not worry about which is the kind of interface that is in play.

(Refer Slide Time: 09:47)

And why is it important to test for interfaces it is important to test for interfaces because a lot of errors could come from interfaces. In fact, empirical studies in software engineering, empirical studies or studies based on collecting factual data right from various organizations which contribute to these data we use lot of statistical tools to be able to come to conclusions. Such empirical studies actually indicate that almost 25

percent, at most 25 percent the total errors that come in software are basically related to interface errors, wrong calls wrong returns whatever they are. What could be these kinds of errors?

So, here is a broad category of the various kinds of interface errors that can happen. It could be the error could be related to module functionality in the sense that let us say one module is calling another module it could assume that the module that is calling offers some kind of functionality for sure. But then it might be the case that the module that is being called has inadequate functionality is not being able to offer the kind of functionality and its assumed to offer, there could be errors because of that and there could be errors in where the module that is being called is located right, it could be in the wrong place, it could be in a place where it is not being able to call it easily right or the module that is calling could have been removed, could have been changed a lot of things could have happened.
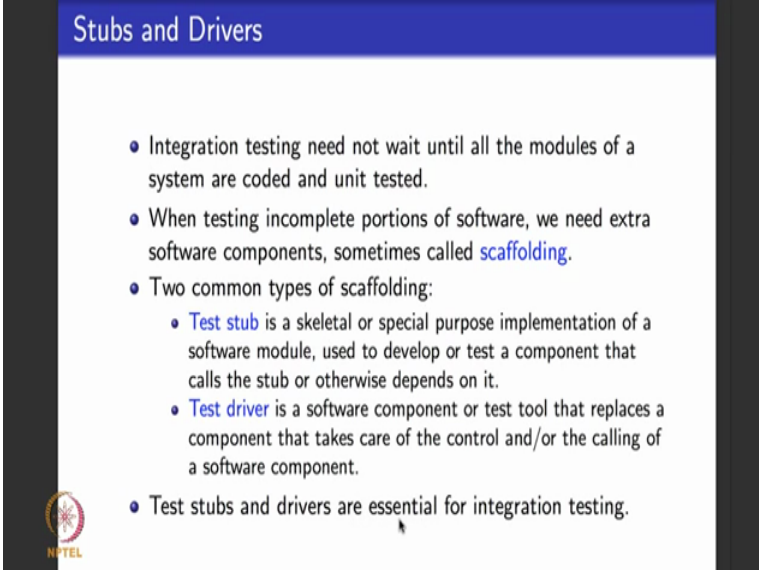
The next kind of error is the actual interface error itself in the sense that the interface can be misused. Suppose a procedure is calling another procedure let us say the first one procedure call interface, it could be the case that you are passing the wrong parameter type you are passing parameters in the wrong order, you have missed passing certain parameters, you are trying to pass more parameters any kinds of errors could happen right.

The next is what is called inadequate error processing. So, here what we assume is that lets say suppose two modules are there one module is calling another module, you are also supposed to do some elementary debugging or error processing as a developer. Let us say the call the module is error prone, the calling module is supposed to have a handler that is supposed to handle an erroneous message from the module that is returning the value, maybe that was not correctly done right or it was assumed that certain kinds of errors would be there and those kinds of errors were not possible. So, there could be interface errors due to incorrect error handling.

The next is initialization and other value errors. You could pass wrong initial values, you could pass wrong data values, you could have missed initializing values anything can happen.

The last kind of popular interface errors are related to timing and performance. Sometimes you not only call a module you want to call module to respond to you fast enough, you wait for some time and if there is no response your timeout right and there could be errors because you have decided to timeout early or you waited for too long there could be errors related to these. But the basic summary is that interfaces are important, they can cause up to 25 percent of the errors that occur in typical software, so it is important to be able to test interfaces and integration testing focuses only on that.

(Refer Slide Time: 13:13)

Before we move on and look at various approaches or methods of doing integration testing its useful to understand two other terminologies related to testing. As I told you right integration testing puts together the modules and test them as and when they are ready. The typical believes is that integration testing need not wait until all the modules are ready because that will be too late right, suppose you had the typical large systems may have 100s of modules it does not make sense to wait through all the modules are ready. So, you test the system as and when the modules become available, right.

So, when you are testing the system as and when the modules become available you may not have the full test system that is test ready. So, what you have to do is a process in testing that is referred to as scaffolding. So, what happens in typical scaffolding is that whichever portion of the software is missing or is incomplete you try to substitute for it
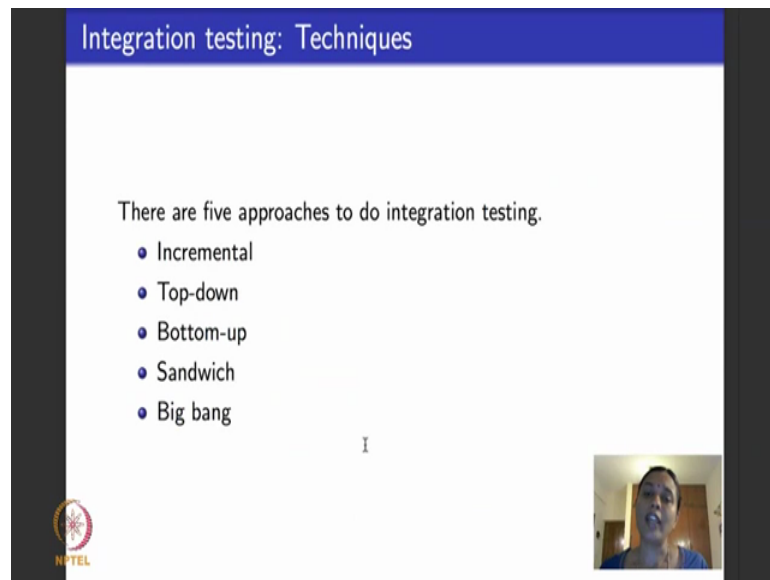
right. There are two kinds of popular scaffolding that is available one is what is called test stub and the other is what is called a test driver.

What is a test stub? The word stub says that suppose you had a module which you wanted a module and you did not really have it. So, you create a dummy module which behaves as if the module that you wanted was there right it takes some values it returns maybe some dummy values, but it does not represent the module. It is just a dummy entity that is being substituted for the actual module right. The dummy entity could need not represent the actual module it can differ a lot from the actual module that is all right, but we still need a dummy entity to be able to go ahead and test. So, when I substitute an actual module with a dummy entity that behaves like the actual module in terms of interacting with the interfaces I have developed what is called it test stub.

The next kind of scaffolding is what is called it test driver. What is the test driver? That is the software component that replaces a component that is supposed to take control of the calling of the software component right. Suppose you have three modules that have to be tested for integration testing, you want to put together and test. It so happens that three modules are being called by the parent module let us say one after the other, but the parent module itself may not be ready. So, then what you do is that you create a dummy parent module it is called a test driver and make this dummy parent module just call these three modules right. So, when you do that then that kind of scaffolding that you develop is what is called a test driver. They are very very needed for integration testing it cannot integration testing cannot happen without stubs and drivers.

So, just to succinctly repeat what I said till now. What is a stub? A stub can be thought of as a module that is like a dummy, that represents the actual module. What is a test driver? The test driver is a dummy, but not for replacement of any module it is a dummy module that calls the other modules that are in the lower level than this
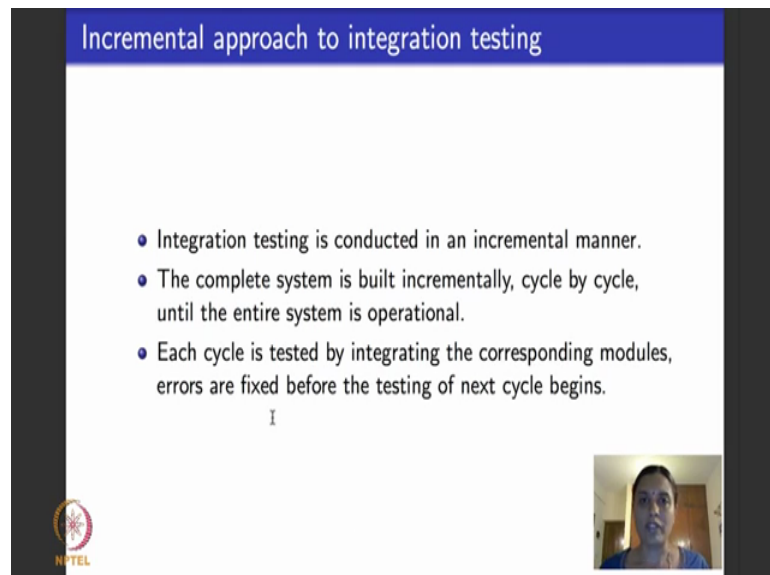
(Refer Slide Time: 16:37)

So, there are five broad approaches to classical integration testing. Most of the books in software testing would refer to these approaches, the older books. The five broad approaches are incremental testing top down integration testing, bottom up integration testing, sandwich testing and big bang testing.
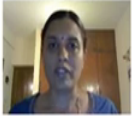
(Refer Slide Time: 17:00)

So, we look at them one after the other what is incremental testing approach to integration testing. So, as the word says incremental means you do it in an incremental fashion as and when you move you keep integrating and testing right.

So, the complete system is built incrementally phase by phase or cycle by cycle as and when the modules of one cycle are ready I put them together to integration testing. The next cycle is ready I put them together do integration testing and each cycle is tested for modules working in that cycle together, errors are fixed then and there and what is done later on is incremental testing, the cycle that has been put together first is not retested all over again.

(Refer Slide Time: 17:50)

Another popular approach to integration testing is what is called top down integration testing. So, here it assumes that the system design that breaks up the software into its various modules is hierarchical. So, it assumes the modules are organized in a hierarchy of levels.

So, here is a small example if you refer to this figure here. There are 7 modules in this figure A B C D E F G and as you see the modules are arranged as if they were the vertices of a tree right, this vertices of a tree represent implicit hierarchy that is available in the module. So, here what it means is that there are three levels of hierarchy modules E F and G are at the lowest level of the hierarchy, C calls these modules E F and G, modules B C and D are at the next high level of hierarchy and module A is at the topmost level of the hierarchy and that calls all the three modules B C and D.

It so happens that in this case the figure looks fairly balanced and complete at every level there are equal number of modules, but it may not be like that. What I meant is that there

are several modules organized into levels at each level; the modules from the previous level call the modules in this level right. So, it has a tree or a directed acyclic graph structure to it right. So, the terminal modules are the modules that occur in the leaves of a graph in this case B E F G and D are the terminal modules, A is a top level module, C is an intermediate module right. This is my reference document to begin the next phase which is top down integration testing and bottom up integration testing for that matter.

So, I have such a structure what do I do when I do top down integration testing. So, there is a top and I start from the top which is the module A and I go down as I do integration testing. So, that is what is top down integration testing.

(Refer Slide Time: 20:05)
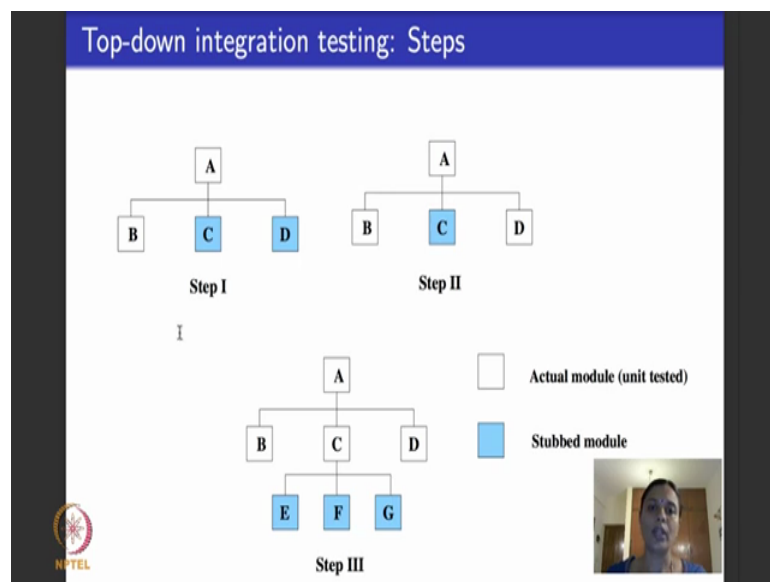
So, what I do? I will explain it in these figures. So, this is the total software design, there are three levels 7 modules I want to put them together one after the other, one after the other and test, and I want to put them together and test in a top down way. How do I go about doing it? I initially, I start with the topmost module which is the module A, module A calls modules B C and D right, it is not wise to put all B C and D together and test A, I would want to test how A works with B, how A works with C and how A works with D independent of each other and I want to test how A works B C and D together.

So, what I do first is that I will keep A, A is unit tested ready and keep B which is a unit tested ready actual module, what I do is I create stubs for C and D, I create test stubs for C and D. So, even if actual module C and A are ready I do not put into them now, I just

made them a stubs, dummies and why, because my focus is to test the interface between A and B alone. I test this part and now what I do assuming that have tested all bugs if found fixed. Now I move on to testing the next interface I could test the interface between A and C or A and D, in this particular example I am testing the interface between A and D, order does not matter, you could do A and C before you do A and D there is no problem.

So, if I test interface between A and D it is not wise to stub B now, because B is tested integrated with A, so I retain it as it is I remove the stub for D, I replace D with the actual module right and then I test whether the interface between A and D as added to the interface between A and B which was already tested works fine. And the third step what I do is I replace C with the actual model right. Now remember in that graph in this figure C is a module that in turn calls E F and G right. So, when I replace C with an actual module it comes as it calls for E F and G, but for now I still want to focus in test if the interface between A and C is working fine. So, I do not keep the module C E F and G actual, I stub them out or they may not be ready I do not really worry about it I just stub it out now.

Now, at this phase what am I testing? I am testing the interface between A and C to be working fine after having tested the interface between A and B, and A and D right, I still have to go on because I have to remove these three stubs and carry on right.
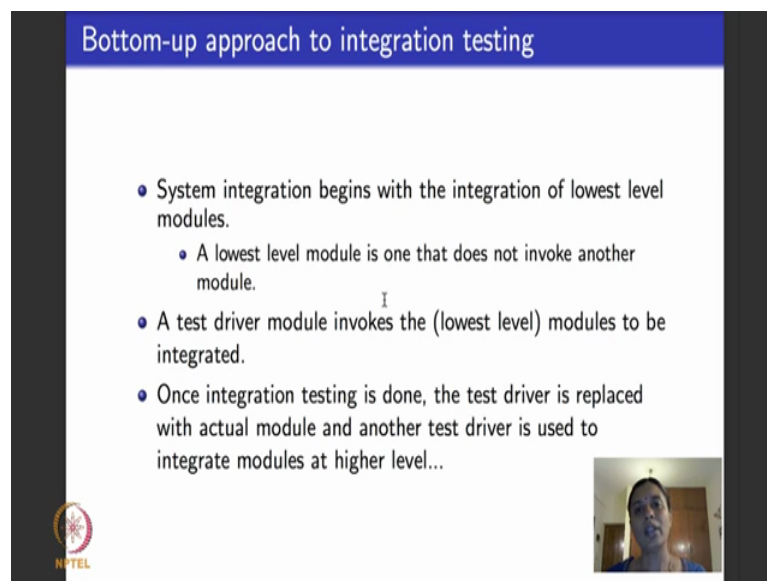
(Refer Slide Time: 23:10)
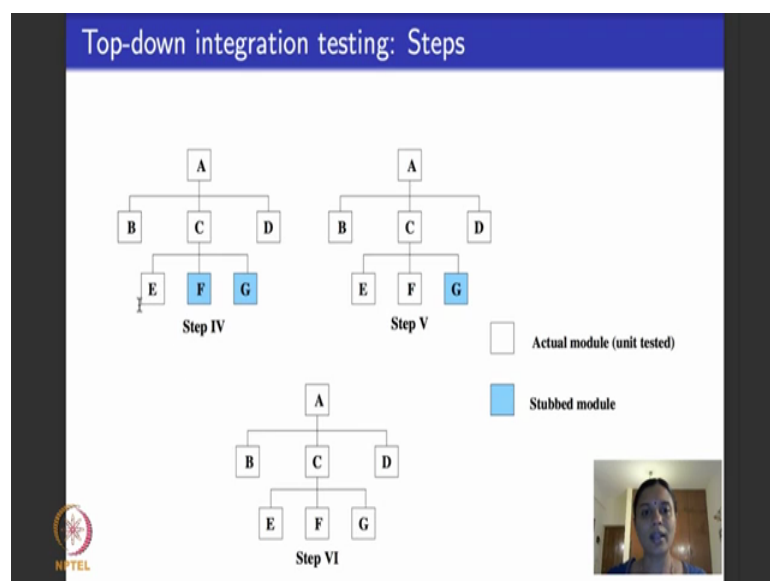
So, I can remove them in any order, here I have removed this stub for E, replaced it with the actual module for E, now I am testing this interface between C and E. And I move on I remove the stub for F replace it with the actual module for F test the interface between C and F and the last step I remove the stub for G replace it with the actual module G and interface test the whole system. So, this is what is called a top down approach to integration testing.

I begin with the topmost module A then I go and test the second level modules B C and D I can test them in any order, but typically I would want to do B and D because it testing C will involve moving to the third level. Between B and D I can test them in any order and then I move on to the next level which is testing C and here there are three more modules to be tested and because there is no further calls I can test them in any order. So, I go from top all the way down and integrate the modules one after the other and test them. So, this is how top down integration works.

(Refer Slide Time: 24:17)

The next approach to integration is what is called bottom up integration. So, here as the name says I start from the modules at the lowest level and keep moving up in the hierarchy till I reach the top most level right. What is the lowest level modules it is a module that looks like this E F G B D a lowest level models, they do not invoke any other modules right. Now to be able to put together the lowest level modules if you go back and see this figure E F and G are the lowest level modules right. Suppose I want to

be able to test the interfaces for E F and G how do I test them? E F and G do not really call each other as for this figure right. Who calls E F and G? It is actually C that calls E F and G right.

But my goal is to go bottom up if I use this test directly C E F G and violating the approach of bottom up integration testing I test the bottom most level with one level top which is C, I do not want to do that. So, what I now do is I create a dummy for C right, what is the dummy for C? I need a dummy that behaves like C in terms of calling modules E F and G such a dummy is what is called a test driver. So, I create a test driver module that invokes the lowest level modules to be integrated. I test for that level lowest and one level up and then this test driver module is actually replaced with the full module and then I go one level high and introduce the next test driver. So, that is what is illustrated in this figure.
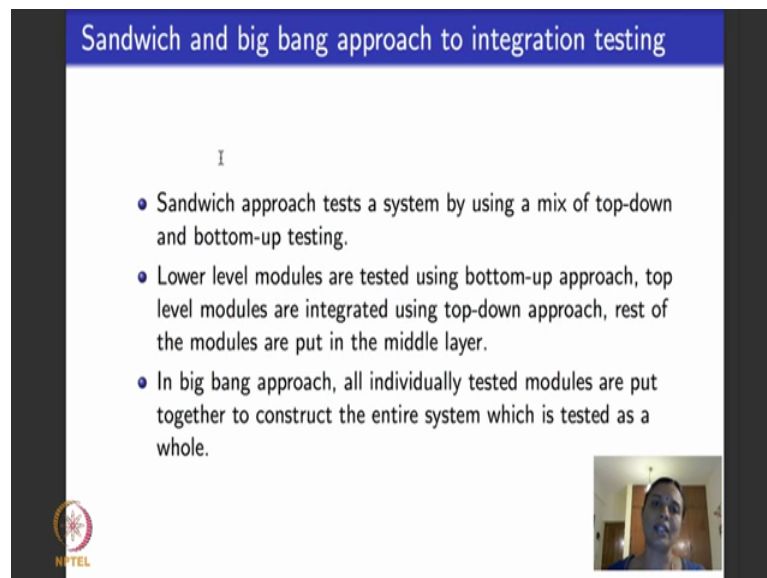
(Refer Slide Time: 26:01)
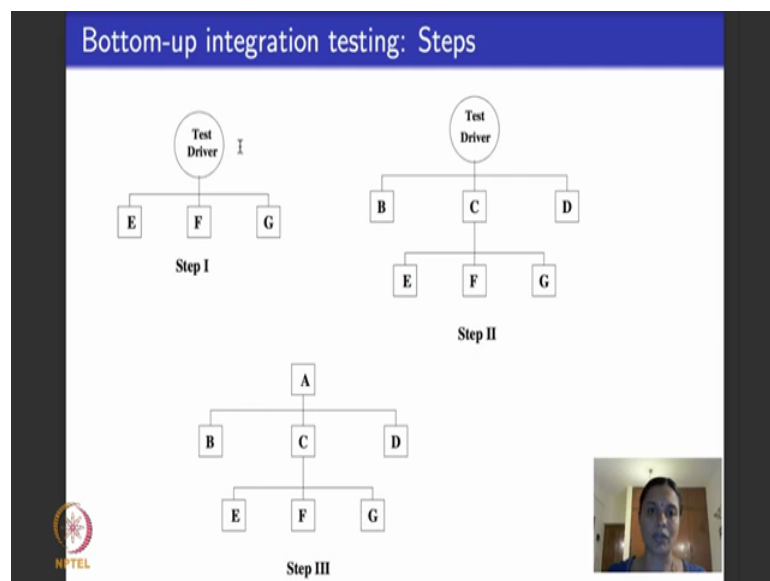
My goal is to do bottom up I start from the lowest level module. So, I start from E F and G, I do not want to put C as it is. So, I write a test driver for C that calls modules E F and G. And in the next step I replace C the test driver with actual full module C, now my idea is to integrate the test B C and D I still do not want to include A because A comes at next level above in the hierarchy. So, I now write a test driver for A and then test the integration of modules B C and D at the second level finally, I replace the test driver for

a with the actual module and integrate test the whole system, right. So, this is how top down and bottom up testing works.

For top down to recap I need stubs to test one interface at a time, stubs are dummy modules that replays other modules just returns some dummy values or fixed values that are consistent with the module values that it would return and I do top down start from the topmost modules and keep going down my hierarchy. In bottom up I start from the lowest level modules and keep going up then the hierarchy, but to integrate this the lowest level modules I need to be able to write test drivers which are dummies, which will for the name sake call the modules at each level that have to be integration tested, right. These two are the most popular integration testing techniques.

(Refer Slide Time: 27:41)
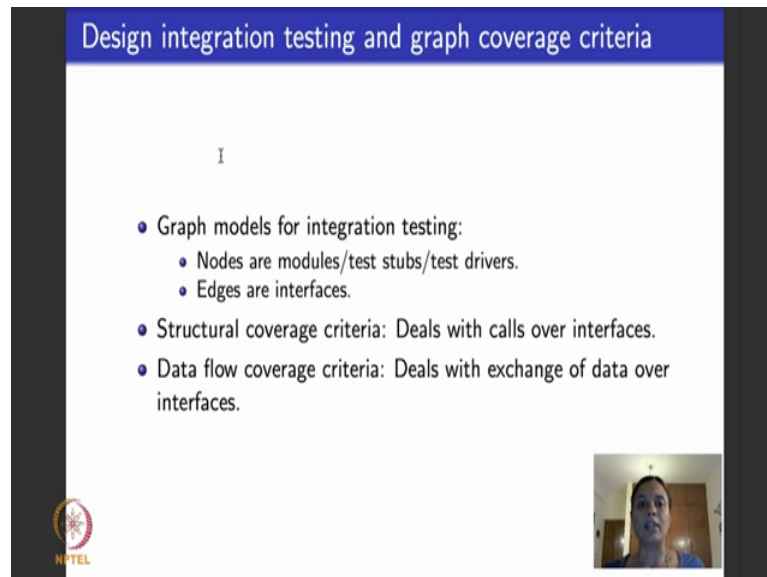

There are other popular techniques called sandwich and big bang. So, what is sandwich do? As the name suggests its somewhere between top down and bottom up. Sometimes it uses top down sometimes it uses bottom up sometimes it uses a mix of them right, it could be the case of the lower level modules are tested using bottom up the higher level modules are tested using top down and then they are put together in a sandwich (Refer Time: 28:09)

The next popular approach to integration testing is what is called big bang testing all individually tested modules are put together in one shot and tested. I am personally not a very big fan of big bang testing because I feel that it is very difficult to isolate a fault

when an error happens in a fairly large piece of software. I would advocate that when there is a clear cut notion of hierarchy, you follow either top down or bottom up based on which are the modules that are readily available for you to integrate and test.

(Refer Slide Time: 28:39)

So, now what we will do in the next lecture is we will go back to our favorite graph models, our focus is now to look at graph models for design and see how they apply to integration testing as we learnt today. So, what do graph models for integration testing look like? If you see the ones that we saw here right these are graphs, all these are also graphs. So, what do the nodes in these graphs represent? They represent modules sometimes they could represent stubs or drivers right. What do the edges represent? They represent interfaces.

As I told you I really do not worry about which is the correct interface for now, when we go later weeks look at object oriented software, web software at that time we worry about interfaces, but for now I consider them as just edges. Now we want to see how to apply structural coverage criteria which will deal with calls on the interfaces and how to apply data flow coverage criteria which will deal with parameter passing and return.

So, in the next module we will look at graph models for integration testing and specifically consider coverage criteria for all these things.

Thank you.