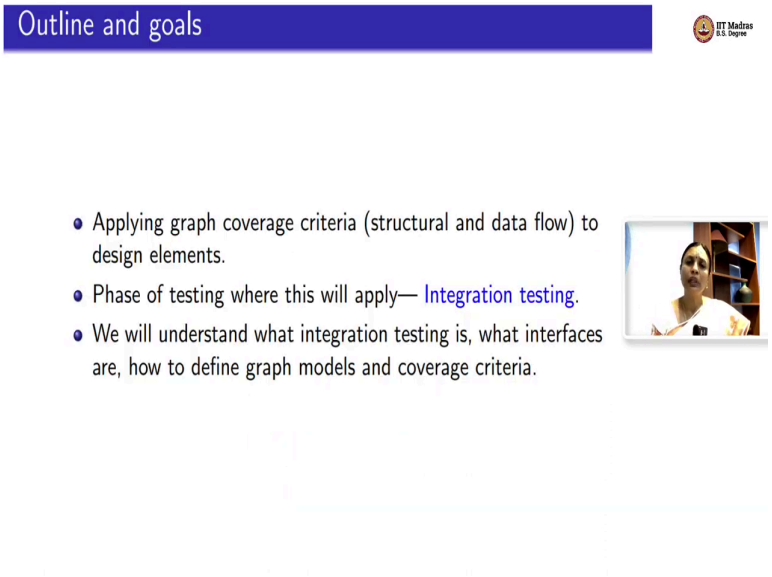
Software Testing
Professor Meenakshi D'Souza
Department of Computer Science and Engineering
Indian Institute of Technology, Bangalore
Software Design and Integration Testing

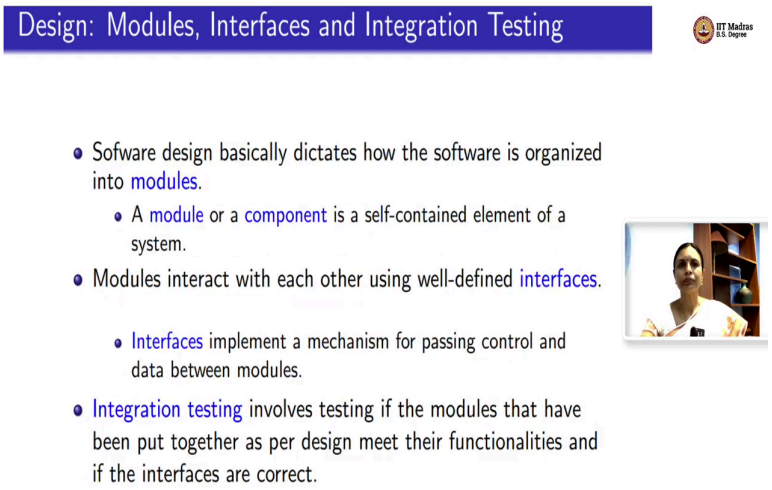
Welcome back. This is the first lecture of week 4. We continue with Graph Based Testing. But we move beyond unit testing and look at other things. So, the thing that I am going to talk to you about today is the next level of unit testing, which is about the software design and how to put together individual functions or modules and do software integration testing using graphs, though, that is going to be the focus of the first two lectures this week.

(Refer Slide Time: 00:48)



So, we learned graph coverage criteria, structural and data flow. We applied it at the unit level within a method, within a function. Now, we are going to apply it for call interfaces or at the design integration level where one method calls another method. And we would still like to consider structural more importantly, data flow graph coverage criteria on them. Before we look at graph coverage criteria and what they mean, it is helpful to do this module on understanding just what is integration testing.

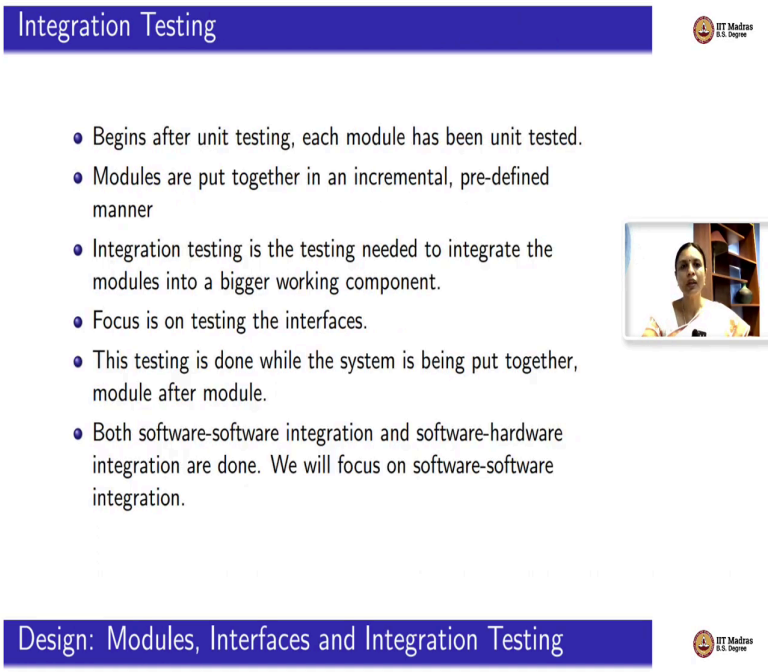
(Refer Slide Time: 01:23)

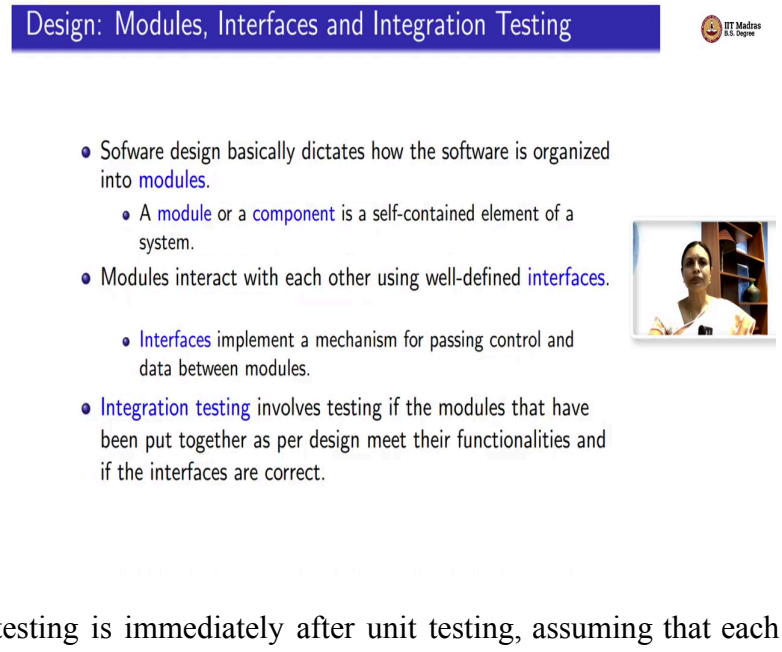


So, software design, which could be a UML diagram or any other design document or any other language that you use for design basically consists of how your code is organized into various modules and what are the levels. A module or a component in software is typically a self-contained element of a system. So, it could be one procedure, a function, a single method, or a whole class, it is depending on what is the unit that we want to refer to as a module.

Modules interact with each other using well defined interfaces. They exchange information, call each other, pass information to each other using these interfaces. There could be several inter kinds of interfaces between the modules. And they implement a mechanism where control is passed from one module to the other. And the data can also be passed from one module to the other. Module could also mean multi-threaded code where each thread can be thought of as one unit. It is up to us to interpret what a module could be. It could be a method inside a particular class or an object or it could be a thread in a multi-threaded programming and interfaces are all method calling another method, a function calling another function or two threads interacting with each other by sending and receiving messages and so on.

(Refer Slide Time: 02:55)

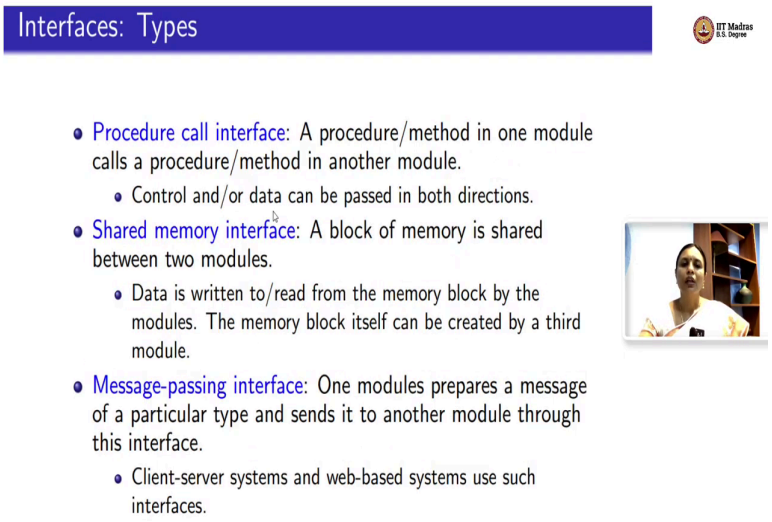




So, integration testing is immediately after unit testing, assuming that each thread has been tested, each method has been tested. And the individual modules let us not distinguish between calling them as methods or threads, we will just call them as modules as per this interpretation that has been defined here. So, an individual module has been developed and unit tested, may or may not be ready, but let us say it is ready. And the modules are put together in a fixed way and tested for their interfaces. That is one module calling another module work fine the interface between them.

And here, the focus is not on unit testing the module, but only on testing the interfaces. Typically, people also test for software put on a particular hardware. But for now, the focus of this course is to put together two software modules and test them.

(Refer Slide Time: 03:52)



So, let us understand spend a little time understanding what would be the types of interfaces that occur between the modules. The most standard thing is procedure call interface, one function or a procedure or a method, as I told you will use these terms synonymously in one module just calls another procedure or a method in the other module. So, let us say procedure A calls procedure B in the other module. And the process of this call, it uses some variables. So, A calls B by passing two parameters x and y and B does some computation on them. And B returns the value which A receives.

So, this is what we call procedure call interface. And when A calls B control is passed from module A to module B, the parameters that are used in the call is the data that is passed from module A to module B. When B finishes executing, B returns values back to A. So, control gets transferred back to A. And the data also gets transferred back to A. The next is shared memory interface where let us say there is a global block of memory. It could be two common global variables, three comma and global variables, a common database, a common file that is global or persistent in nature.

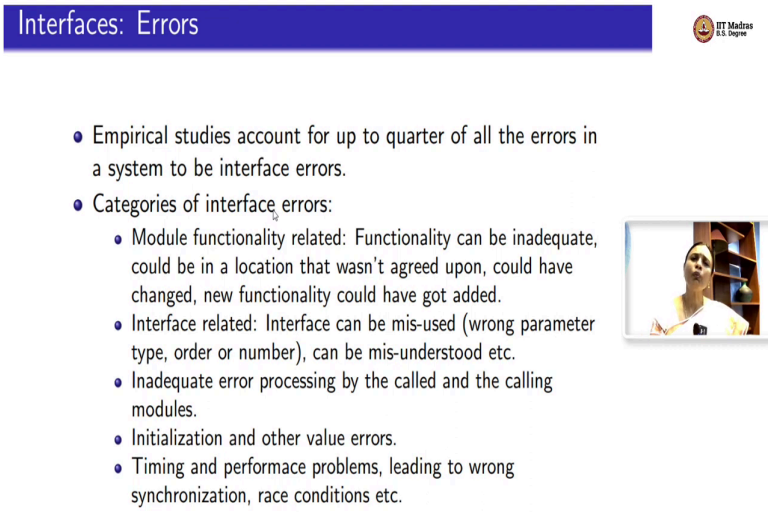
And these two modules, two methods or two threads share that. They can read from that common variable. They can update back or write back values to the common variable. The memory block itself could be owned by a third module.

The third kind of interface that you can see is Message Passing Interface, where there are dedicated buffers or queues or channels between two modules. One module puts data or messages to be sent across the buffer on the queue or in the interface. And the other module

when it is ready picks up the data at the other end of the queue or the interface. In the message passing interfaces need not be queues but channels of a particular kind.

Classical examples are web clients talking to web servers are generally software clients talking to the main server. They all are message passing interfaces. These are common interfaces that you will find while programming that and that can be used for integration testing.

(Refer Slide Time: 6:14)

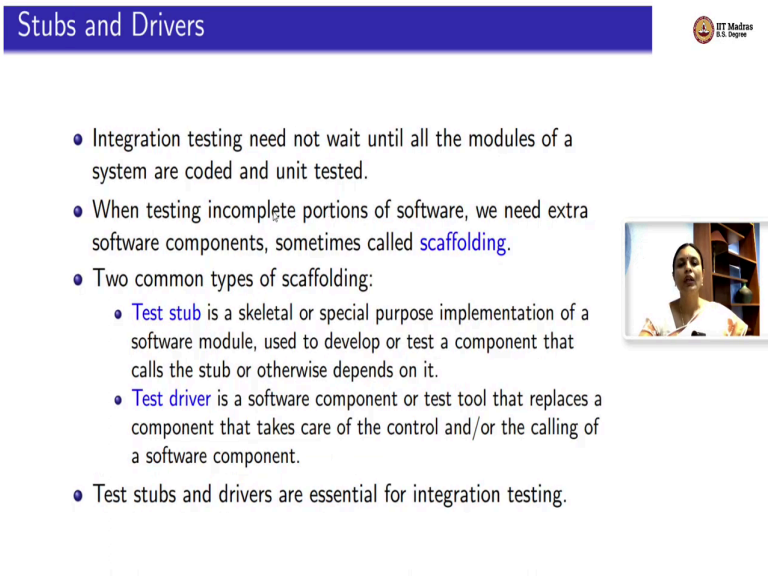


Why are we worried about interfaces? Empirically, studies in software engineering have shown that errors on interfaces constitute almost something like 25 percent of the total kinds of software errors that can occur. So, here are some example categories of what the interface errors can be. It could be that the particular module that is interfacing or calling the other module is implemented wrongly and the wrong functionality is actually affecting the interface. It could be that or the particular module could have gotten changed and the changed functionality is affecting the interface. Or it can be the case that you are passing the parameter of wrong type, wrong order and wrong number of parameters or the order in which they have to pass the parameters that could be wrong.

For example, Module A passes parameters x and y to Module B, there could be integers in Module A, they can be declared as strings in Module B, obviously, this is not possible. The next could be that the particular one or more modules have inadequate error processing. That could also be a problem. It could be related to initialization of variables. It could also be related to there being delays in the queues network delays when there are message passing

interfaces, which results in timing and performance related delays. Several reasons are there and interface errors constitute almost one fourth of the total errors that can occur. So, it is important to understand how to test for interfaces.

(Refer Slide Time: 07:50)



So, while testing for interfaces, the other thing that you commonly tend to come across is, so let us say a team is developing a piece of software. And together various sub-units in that team, it could be individual members or sub-teams are responsible for developing individual modules. So, let us say totally four modules are being developed. Let us call them A, B, C and D. But let us say out of this module A and C are fully ready. B and D are not yet ready. You have been you have developed A unit tested A and A interfaces with B. B is not yet ready, but I would still like to test the interface.

So, we create a dummy module for B. And this process which behaves exactly as B would behave, but replaces B for the purpose of testing the interface between A and B. This process of writing incomplete replacing incomplete portions of software with extra code special purpose code that is written for the purpose of testing interfaces is called scaffolding in testing. There are two kinds of scaffolding that you can do. One is called a test stump, the other is called a test drive.

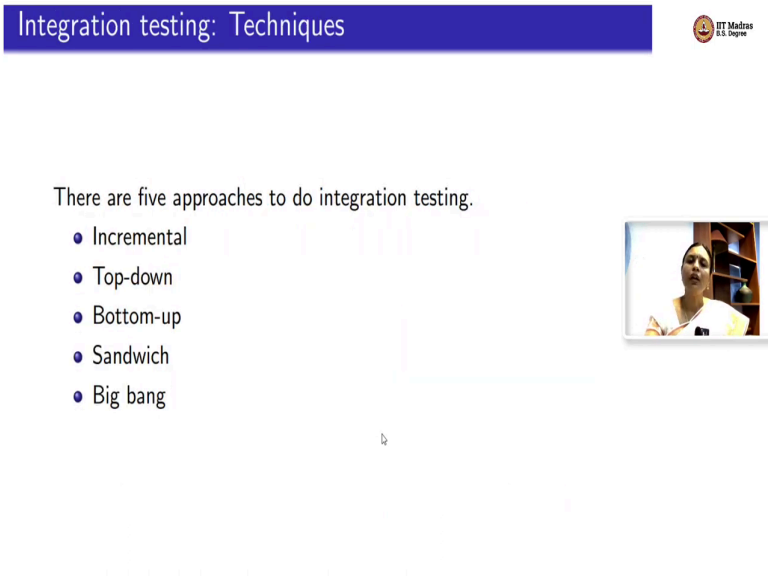
So, let us say module A calls module B. Module A is fully ready unit tested. And you would like to test the interface between A and B. Module B is not yet ready. So, you write a special purpose implementation for B. And when you write that, that is called a test stub. Suppose module same structure holds, module A calls module B. Module B is ready but A is not yet

ready. And you would like to test the interface between A and B. You write a special purpose implementation for A, the caller module, then that is called test driver. Those are the two definitions that are given here.

A test stub is a special purpose or a basic skeletal implementation of a software module that is otherwise not ready. That is used, developed, written just for one purpose. It is to develop or test a component that calls the stub or the module that the stub represents or is otherwise dependent on it. But test driver is the other end is a special purpose software component, or a test tool that replaces a component that takes care of the control and calling of a software component.

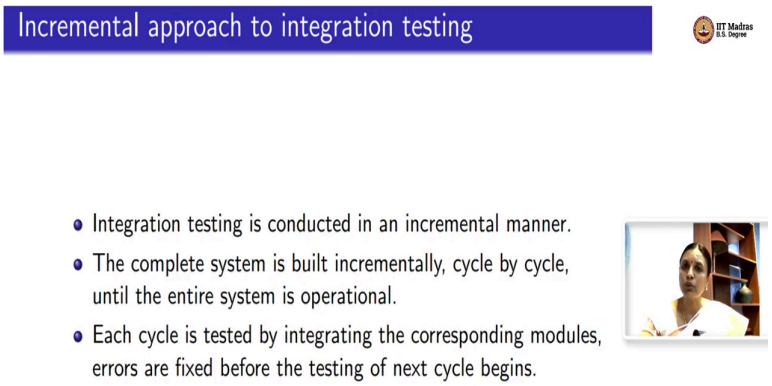
So, just to repeat module A calls module B, if A is not yet ready, but B is ready, you would like to test the interface you write a driver for A test driver for A. But if A equals B and B is not yet ready. A is fully ready, you would like to test the interface between A and B. The skeletal implementation for B becomes a stub. Writing stubs and drivers is a very important part of doing integration testing.

(Refer Slide Time: 10:55)



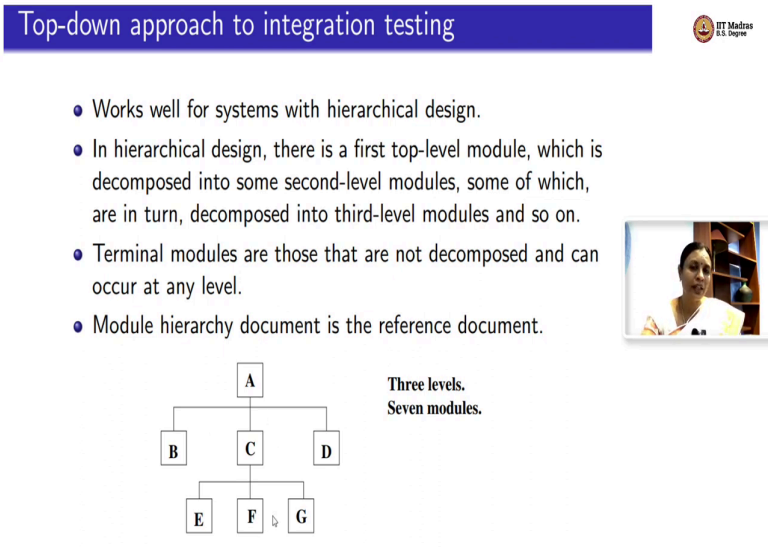
Broadly, based on the organizations that exist, these are considered the five broad categories of doing integration testing four actually, because incremental testing can either be top down or bottom up, these are subcategories of incremental testing. And then we have sandwich testing and big bang testing which are more or less the same.

(Refer Slide Time: 11:15)



So, what is an incremental approach to integration testing? So, the word incremental says you do it one at a time. And as you do you make progress with the testing. So, integration testing is conducted in an incremental manner. System is built in parts incrementally could be sprint after sprint, cycle after cycle. And the eventual goal is to get the entire system operational. But as and when it is built in parts, I unit test the module and a test for interfaces. So, in each cycle, I can do integration testing also incremental.

(Refer Slide Time: 11:53)



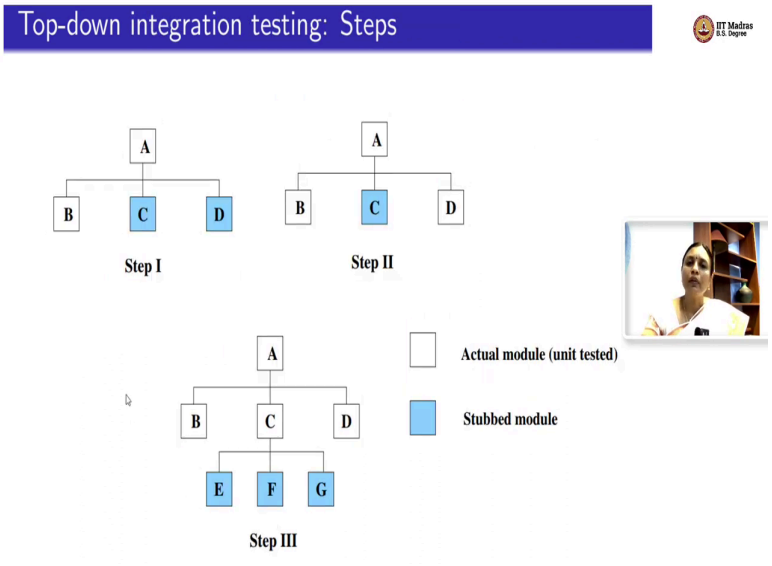
So, there are two ways to do incremental testing. One is top-down and the other is bottom-up. So, people usually develop a document that looks like this that is represented by this figure that you see at the end of the slide. Read this as follows. What does this picture say? It represents what we call a module hierarchy document. So, it says that a particular piece of

software has 7 modules, let us name them A, B, C, D, E, F and G. These 7 modules could be threads, could be methods, could be procedures, let us not worry about that. It is just a self-contained unit of code.

And it so happens that A has call interfaces with B, C, and D, those are the connections that you see. And C in turn has called interfaces with E, F, and G. Those are the other three connections that you see. So, we would like to test these call interfaces. And sometimes it might happen that particular parts of these modules are not fully ready. So, this call interface here A calling these three modules B, C, and D. C in turn, calling the modules E, F, and G is hierarchical.

So, it is amenable to doing hierarchical integration test. It so happens that E, F and G and B and D, so called leaves of this tree like graph, do not have any further call interfaces in this example, so they could be left as it is. Once they are fully developed, the goal is to just test the interfaces. But module hierarchy documents in general look like this. And they could be at any level have any number of modules, any number of dependencies.

(Refer Slide Time: 13:50)

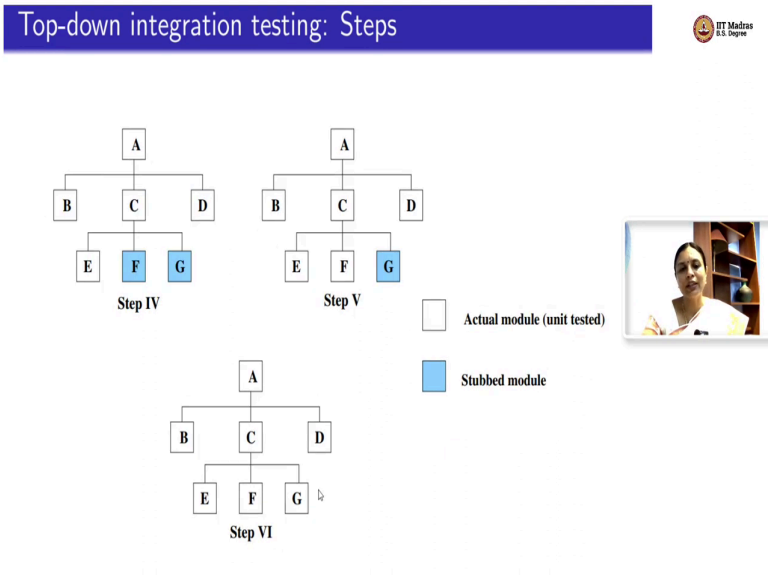


So, if I have to do top down integration, this is just one scenario that can happen while integration testing. So, please remember this module hierarchy document. Now it could be the case that modules A and B are ready. So, I can test the interfaces from A to B, but let us say C and D are not yet ready. So, I developed some stubs for C and D to be able to test the interface between A and B. I may not develop stubs for C and D if the interface between A

and B is a standalone interface not dependent on C and D. But if it is dependent on that I develop stubs. Let us say D also eventually gets ready. So, I work with the stub only for C.

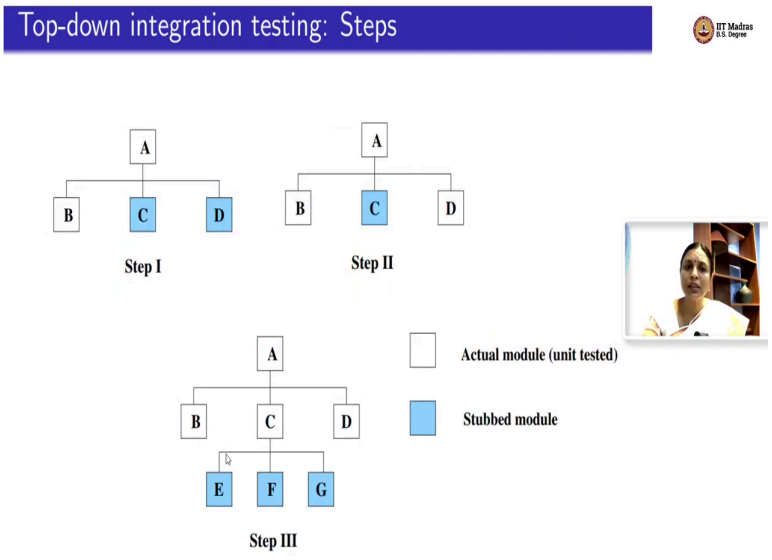
Now C also gets ready. But the collie modules of C, E, F and G are not yet ready. So, I work with stubs for E, F and G.

(Refer Slide Time: 14:43)



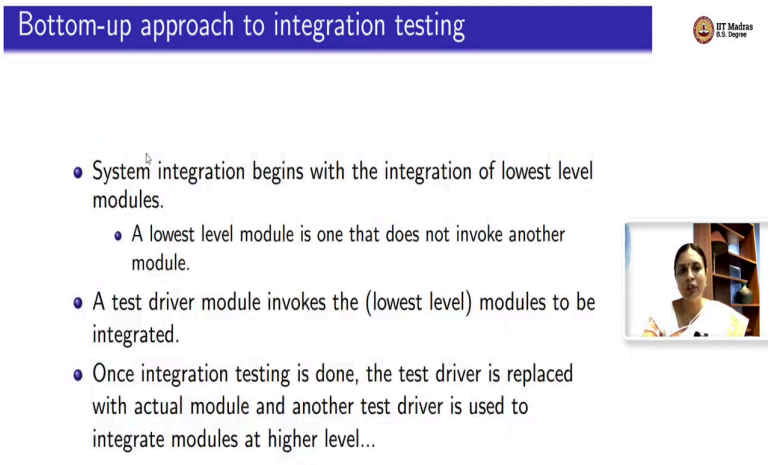
And I move on like this. Let us say E gets ready. So, I work with stubs for F and G, both E and F for ready. G is only stub. Finally, at the last step, all the modules are ready by which I would have tested it for all the interfaces.

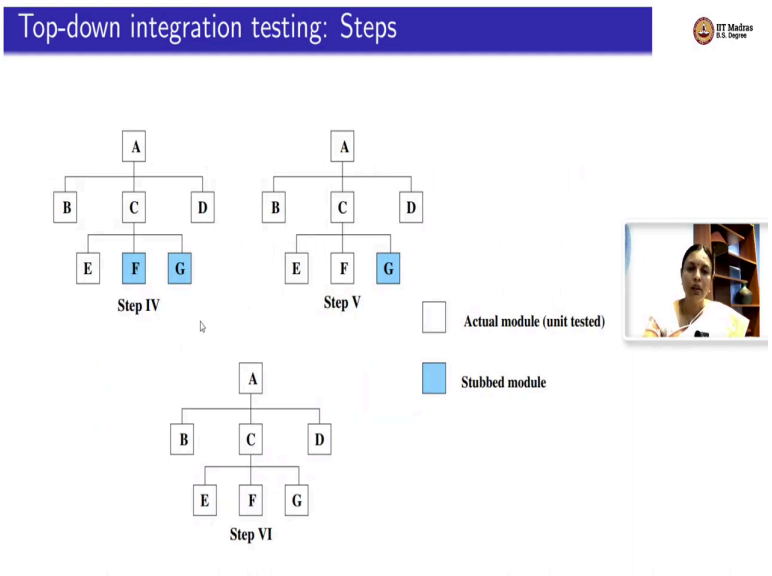
(Refer Slide Time: 14:58)



So, this is a top down approach. Is this clear, firstly is ready.

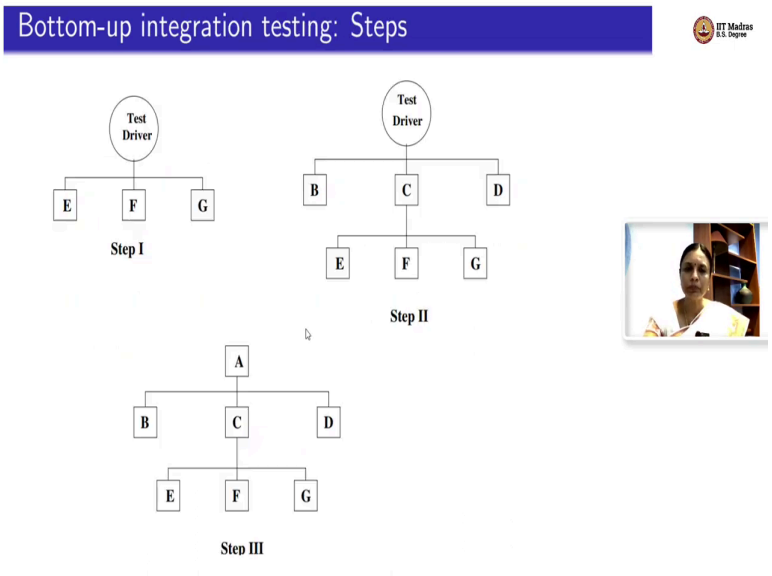
(Refer Slide Time: 15:05)





The same interpretation can be given in a bottom-up way. So, system integration begins with the integration of the lowest level modules or lowest level module in a module hierarchy document as we saw is one that does not invoke other modules. Like for example, here, B, E, F, G and D are lowest level modules. So, if I do bottom up, I begin first with they being reading. And then I write test drivers that will invoke these lower-level modules and go in a bottom-up way.

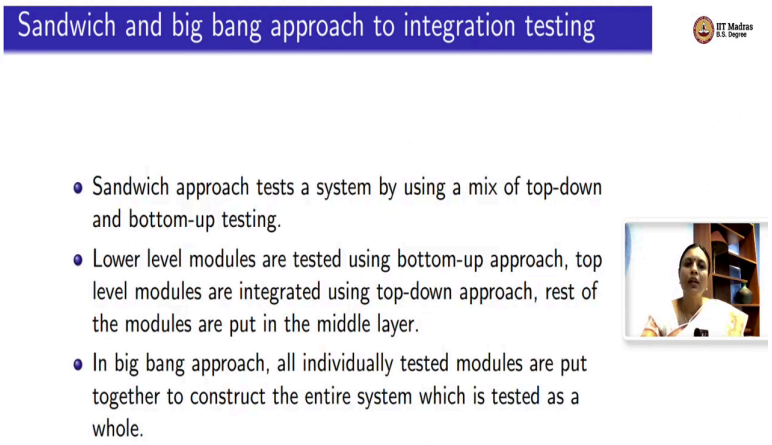
(Refer Slide Time: 15:36)



So, same picture is illustrated like this. Let us say the lower modules E, F and G are ready first. So, I write a test driver here for C. Let us say C is also ready. At some point, B and D are also ready. But A, the highest level module in this module hierarchy document diagram that we saw is not yet ready. So, in step 2, I write a test driver for A. Eventually A also gets ready. And here I have the full set of 7 modules and all the six interfaces that could exist between the modules that are fully ready.

So, this was top-down and bottom-up. Please remember that such a document module hierarchy document is not to be confused with other design documents. It is this is just for planning phase based on when each module could be ready. As and when they are ready. You try to write stubs or drivers to test each of them. That is what it means.

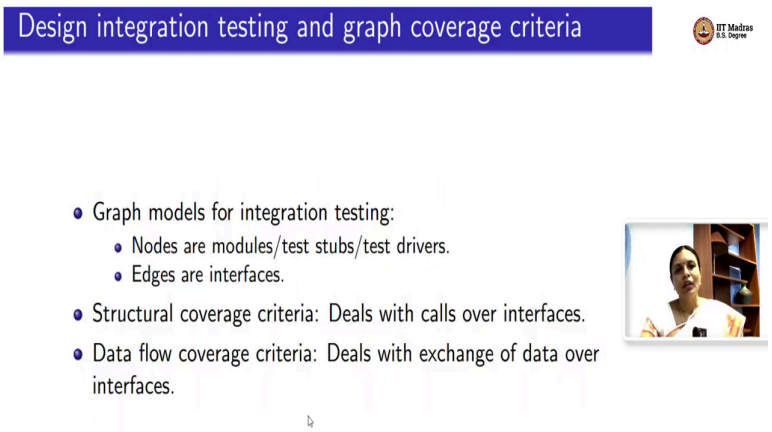
(Refer Slide Time: 16:31)



The other approaches are sandwich approach and Big Bang approach. As the name says they are neither top-down, not bottom-up. Sandwich approach just tests the system by using a mix of top down or bottom up. It is none of these. Lower level modules are tested maybe in bottom of A. Top level modules are tested in a top down way. You do not really adhere to any kind of rule.

In big bang approach, all the individually tested modules are basically put together and the entire system is tested as a whole, not a clean way of doing it. But if you have to do it really fast, this is useful.

(Refer Slide Time: 17:07)



So, now we learned the basics of what integration testing means and what do interfaces look like at a high level. So, we would like to move on and understand what the graph models for these testing of these interfaces look like. Graph models for integration testing, what will be the vertices or the nodes of your graph, they would be individual modules, or stubs or drivers developed for those modules and edges would be the call interfaces.

Structural coverage criteria would deal with each module as a whole and the call interfaces, data flow criteria would deal with how parameters are passed and received over these call interfaces. So, in the next lecture, we will actually formulate how control flow criteria and data flow coverage criteria makes sense for these kinds of graph models, what are the graph models and walk you through an example. I will stop here for now. Thank you.