**Software Testing**
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 03
Software Testing: Testing based on models and criteria

Hello everyone. This is the lecture on the second module of first week. So, what will be seeing in today's lecture; we will look at it what it means to test the software.
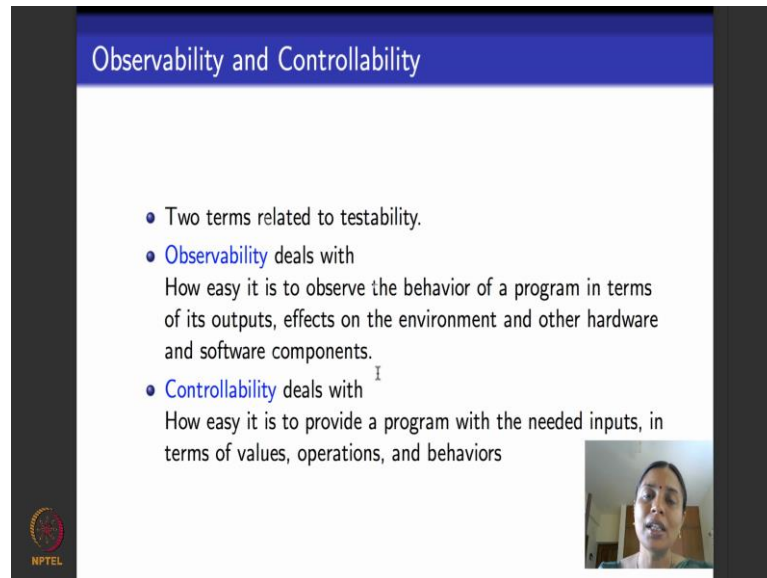
(Refer Slide Time: 00:22)
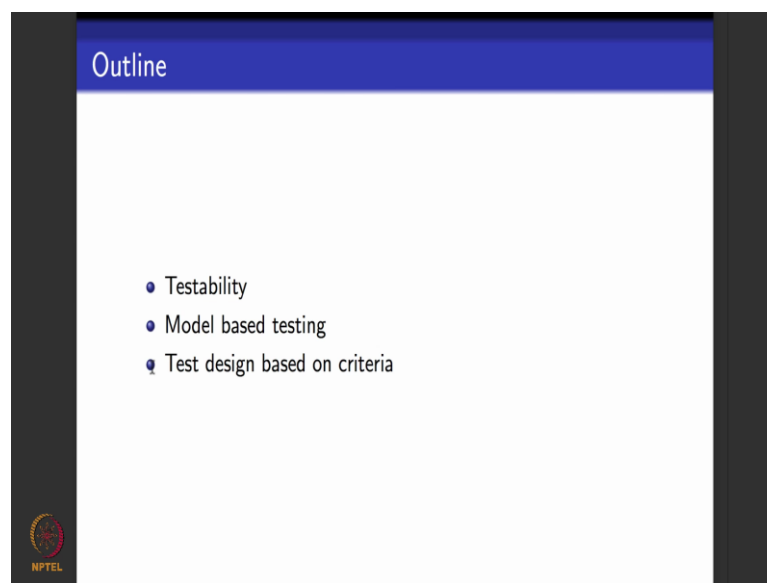
What is testability; we will also look at what is model based testing as its commonly perceive in the testing world, and how we will use model based testing when we design algorithms for test cases. As I told you lot of this course is going to be on algorithms and methodologies for test case design. And in fact we will be looking at what is call criteria based test case design. So, we will formally define what is criteria, and look at the various artifacts on which we will be defining these criteria to design test cases form.

So, I begin with introducing what is testability. So, what is testability? If you look at it in simple English terms it simply answers the question is the software testable or not. So now, if I ask the same question again in term what is it mean for the software to be testable right? So, we say can I give inputs to the software; inputs a test cases to software. After giving the test cases to the software can I execute the software an observe the outputs. You might think it is a very obvious question when I write a piece of core

what is the difficulty about giving inputs to the software, executing the software an observing its outputs.

(Refer Slide Time: 01:37)

So, we will see what are the particular hiccups that can I raise while doing this. Two related notions of testability are what are called observability and controllability. So, what is observability mean? Observability it tries to answer the following question right. So, it says the suppose you give a software certain kinds of inputs and executed, how observable is its behaviour, how observable are the outputs that the software produces.
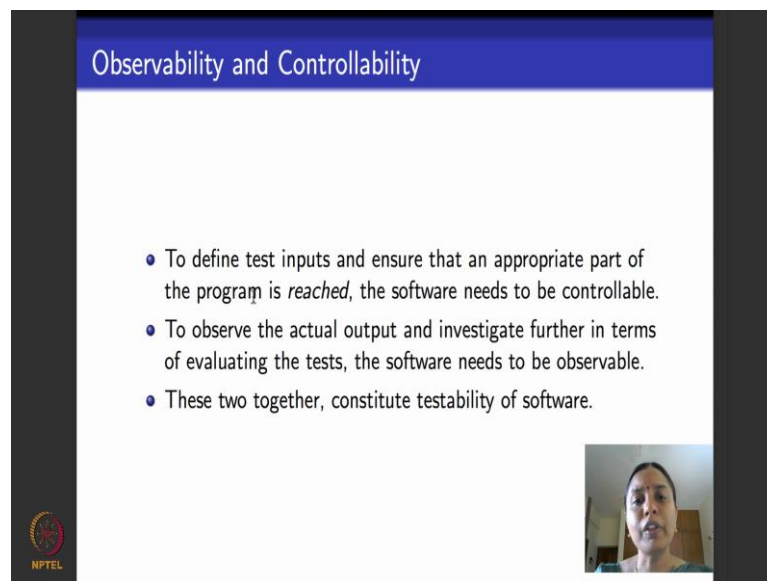
It could do we doing a lot of things silently and producing only the final output, whereas as a tester you might be interested in some intermediate outputs that the software produces. So, observability tries to answer this question about how observable the outputs produced by the software are and how much we can tweak the outputs that we want the software to produced towards testing it.

So, when I test the software I might want to observe a little more about its outputs, not actually the actual outputs of software. But I might want to know the values of specific internal variables, specific call and return values observability deals with all this. Another related term for testability is what is call controllability. What is controllability? Controllability says is a software controllable by giving it in some input and actually executing the software.

Again this might seem like a trivial question you might ask why is it so difficult to give inputs to a software. Let us take a particular case for example, where piece of software runs for some time and then at particular point and core it calls a particular function. So, I want to be able to let us say integrate test this function call. So, what I am interested in controllability is the fact that I should be able to give inputs to the softwares such that I can ensure that the particular function of procedure that is embedded deep inside the code is actually called for the values of the inputs. And not only is the function called now wont to be able to observe what the function returns in turn.

So, controllability and observability help you to answer these questions and software. And they are one of the several different quality attributes broadly call elatives the people worry about while testing and writing software.

(Refer Slide Time: 03:51)

So, to define test inputs and to ensure that in appropriate part of the code is reached, like I told you an appropriate function is called and I will be able to in turn observe the value return by the function I say that the software needs to be controllable. When I want to observe the value that is return by the function and see how it gets passed on to the main calls a program then I say that the software needs to be observable. Observability and controllability put together constitute testability of a software.

Obviously, for a software to be testability its testability quotient must be very high; it should be observable, it should be controllable. So, we little more in detail understand

what this notion are; when people say a software is testable they usually talk about four parameters for observability and controllability put together.

(Refer Slide Time: 04:38)

So, these four parameters are what are called RIPR model. So, what is R? R stands for reachability. Reachability mean suppose I am testing white box testing a piece of software and I want to check whether a particular statement is reached, where a particular function call is reached whether a particular code segment is reached; I should be able to give test inputs that guarantee the execution of software in such a way that this particular target place in the software of goal is reached.

Now the next important thing is I: I stands for infection, what did you mean by infection? Not only do I want to reach that place I want to be able to execute those statements in the software. And suppose there is an error that is observed I want the error to be exercised or I want the error to be reached; I want the software to reach its faulty state. And I want to be able to give inputs in such a way that I can infect that particular statement in the software and actually pull out the error of the faulty states the software was in. Suppose I manage to do that, but the software still manages to run file and execute such that it act normally, then even though the software is erroneous I as a tester have not really observed the error.

So, the next parameter that I am looking for is P with stand for propagation. So, I say not only much particular statement be reached and the error be revealed as infected; the error

should also be propagated to output observable output such that I know that there is an error that has occurred in the software. And it could be propagated and for some simple reason like you might just forgotten to write a printer statement or forgotten to track this particular values the propagated error might not actually get revealed to the tester. So, at the last parameter in this RIPR model that I am looking for is for the error to be revealed; this tester will be able to observe the incorrect part in the program and correctly isolator identify which part of the code was the error in.

So, to summarise we say that if I am targeting a particular a piece of code that I suspect to be erroneous or that I want coverage to be achieved somebody told me you please test this piece the code thoroughly. You please test this function and the function called thoroughly. So, I am looking for four properties the software should satisfy for it to be observable and controllable.

The first one is that the particular target code fragment or segment should be reachable. The particular target code fragment or segment if it has an error should infect and the particular test case that I give should make sure that the error state is actually occurred. This occurred error intern ship should propagated to the software producing a different kind of output that reveals the error. And the revealed error should in turn help the tester to be able to identify and isolate the error as having occurred in that piece of software. So, software that needs this RIPR model property is supposed to be highly testable.

(Refer Slide Time: 07:48)

So, here is an example of an RIPR model: you consider the small code fragment that as given here I am not given the full piece of code here, I have just given a small fragment of the code. In this small fragment of the code there are two inputs x and y, and then they could be other statements lot of other commands and the software. But let us focus on this particular thing fragment of the code which says that for there are two nested if statements here. The first statement check if x is less than 10 that it sets the value of z to be x plus 1. And then there is one more if inside that which says that if y is less than z then there is an error in the software. I want to actually written what the error could be I have just written it like a stub which marked it out as error.

So, suppose I want to actually; my goal is to be able to be actually reach this error statement and reveal the fact that this error as occurred. So, what is RIPR parameters do for this? So, let us look at reachability first. So, what is it mean for the first if statement to be reachable; any value of x will reach that statement assuming that there are no code fragments that change the value of x any value of x that is given here will reach the first statement which is x less than 10. So, reachability can be thought of as been true always true there is nothing specific that I need to do.
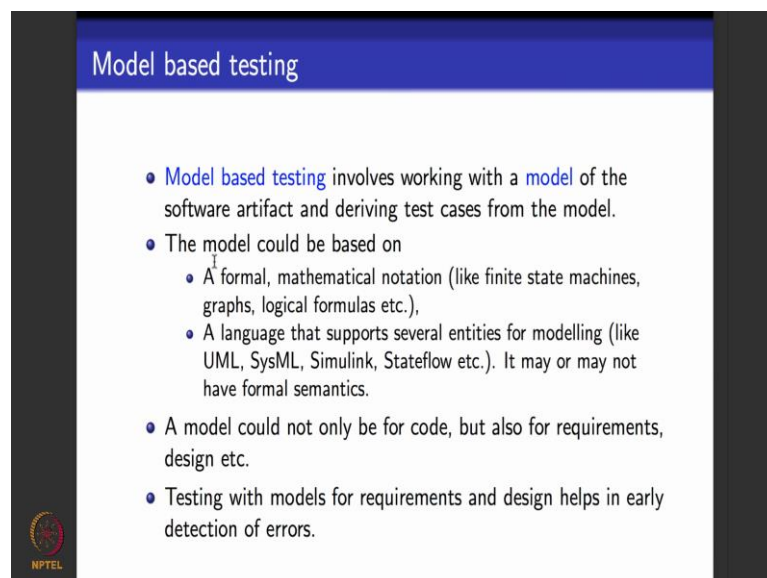
But to be able to reach the second nested if statement; suppose the first if statement test negative the sense suppose I give a value of x that is not less than 10. So, this if statement this predicate will written false and the code will exist this if statement it will never enter this if statement. If it does not enter this if statement it is not going to be able to test the second if make it positive and then reach the error statement. So, reachability for the second if statement will happen only if the first if statement returns true and the first if statement will return true if I give a value of x that is less than 10.

So, the predicate x less than 10 should be true for me to be able to reach the second if statement. So, what is infection? Infection means I not only reach the statement I manage to execute that statement and make it true or false. So, again an infection for the first if statement is any value of x less than 10 like for example, or any value of x greater than 10 or even x is equal to 10 will manage to infect this first statement. For the second statement remember z is set to x plus 1. So, you must give a value of y that is less then x plus 1 to be able to test the second of statement. Now moving on when it comes to propagation I should be able to go past the first if statement makes it true, go past the second if statement make that also true, and then only I will reach the error statement.

So, both these predicates x less than 10 and y less than z will have to be true for me to be able to reach the error statement.

I hope what reachability infection and propagation is very clear. So, in this piece of the example it is quite small, so it easy to be able to do what are the conditions for reachability, infection and propagation and so on. But for large fragments of course where I have thousands of lines of code it is not an easy problem to be able to come up with less list of predicate for reachability for infection and propagation and to be able to solve then. In later modules when we look at testing criteria we will see how what it means to solve the predicate or a particular piece of code for reachability, for infection, and for propagation in detail.

(Refer Slide Time: 11:20)

And now move on to looking at model based testing. What is model based testing mean? In the English sensor the term model based testing means that you have a model of the software artefact; the artifact could be requirements it could be design, it could be architecture, it could be code, and then you design test cases by looking at the module. There are of course, several other uses of the models for software, well done models could be used to auto generate codes they could be used to early validate the system, there are several other uses for model based design and model based testing.

There are broadly two kinds of models: the model could be based on a formal mathematical notation. Here are some popular modelling languages: finite state

machines, graphs, logical formulae. As I told you in the last module there are several different verification techniques, like model checking theorem proving they all use mathematical model software artifact to be able to verify a software.

Models it could also mean a particular domain language or specifically designed language that specifies all the artifacts that particular software is concerned. You might have heard popular modelling language like UML which stands for unified modelling language, SysML which is systems modelling language, Simulink, Stateflow which are modelling languages that are proprietary to a company called mathworks which makes a popular tool called MATLAB.

So, these modelling languages support several diagrammatic notations for me to be able to module various software arifacts like use cases requirements design control laws and so on. They may or may not have formal semantics. So, it is to be noted as I told you that the model need not be only for code; it could be for requirements, it could be for design, and so on. Testing with models for requirements and design: suppose I manage to have good well defined models for requirements and design and I am able to design test cases for them, then remember that I am doing this kind of testing right at the requirements level or design level before I write codes.

So, if I find an error I find it early on in the life cycle and that is considered to have lots of benefits. So, what is the process of model based testing how does it work?
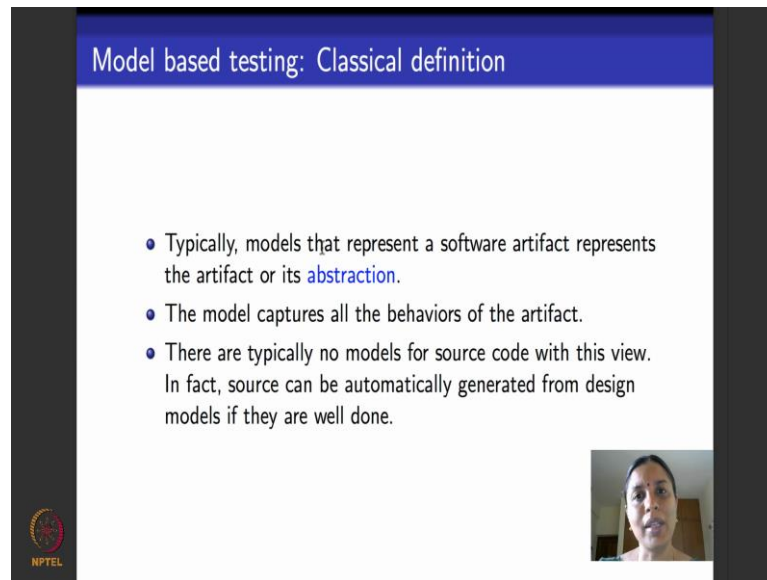
(Refer Slide Time: 13:32)
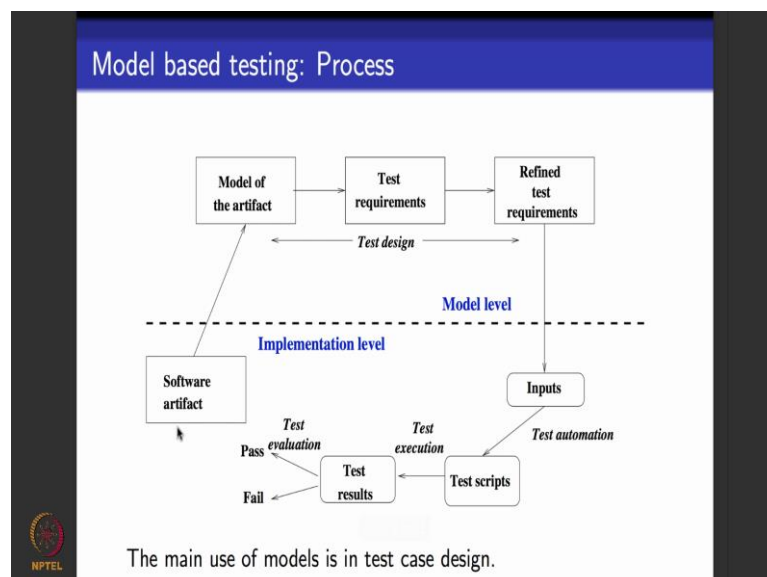
Let us begin here with this part: I have a software artifact with me under consideration that would be code that could be a requirement document that could be a design document. Code is typically executable, but many software companies have requirements of design document written in English. So, English document cannot act as models. So, I take the software artifacts and come up with the model of the artefact. The model of an artifacts is designed in a language that I have predisposal decided a suitable for the class of software that I am working with.

Once I have the model of this software artifacts I do what is called model based design and I can subject this model to several different uses, but what we will focus on today is in this particular slide is the use of the model when it comes to designing test cases. So, I take this model of the artifacts and then I also have a set of test requirements that have been given to me. Test requirements could say something like you please there are some highly critical requirements you please design test cases to exhaustively test for all the requirements. Test requirement could say something, like you cover this particular fragment of the model or a piece of code.

So, I take these test requirements and refine them with reference to the model that I have in mind. After I refine the test requirements I have done my test case design and I am ready to give my test case as inputs. Remember once I have a test case that has to be actually executed in code. So, I assume that I have an implementation ready and I move on to the implementation of the software. Once I have given my test case design I past these input make it ready for execution; make it ready for execution you remember in the last module we saw this step called test automation. So, I do test automation and get that test scripts which are ready for execution then I use the tool to execute this test scripts and observe the results. And then I conclude the test cases have passed or failed.

So, this is the normal process of testing that we saw the only difference here is that while doing designing of test cases I have a particular modelled of the software and hand and I design test cases for a set of test requirements on that specific model of the software. Here my main use of models is to be able to do test case design.
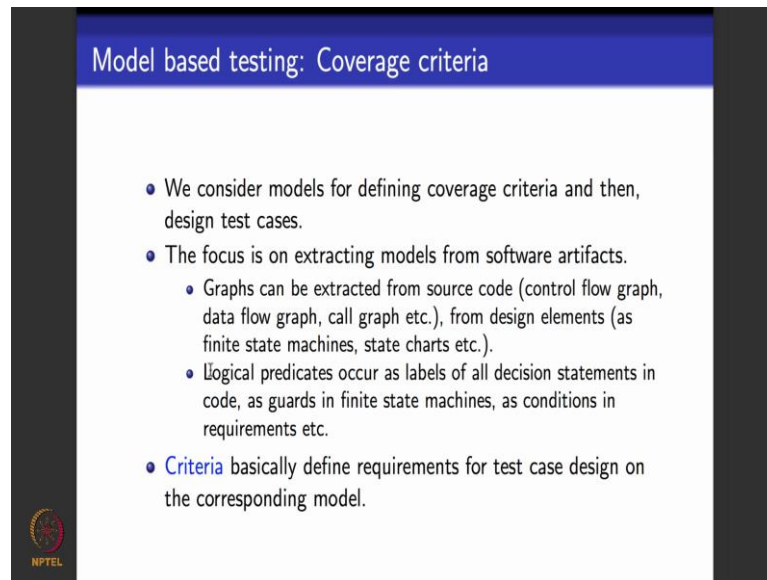
(Refer Slide Time: 16:03)

Model based testing: Classical definition

- Typically, models that represent a software artifact represents the artifact or its abstraction.
- The model captures all the behaviors of the artifact.
- There are typically no models for source code with this view. In fact, source can be automatically generated from design models if they are well done.

So, the classical view of model based testing or model based design is to say that models could represent a very high level abstraction of the software, it could need not represent the actual code. In fact, code is almost always never represented as a model. Code is represented in the programming language of your choice and is always an executable entity. Models may or may not be executed, they could be static artifacts that represent the code. Typically a model is supposed to capture all the behaviours of the particular software artifacts and it is for the user to ensure that the model of the software is accurate and good enough to meet the required expectations of the model. If the model that you do of the software artifacts is itself wrong, then obviously the test case and everything else that you do with the model is also going to be wrong.

So, is up to the user to ensure that he or she knows the modelling notation well and is model software well enough to be able to design all the test cases. So, we do not view the classical view of model checking in this particular course, we will view model based testing for doing coverage criteria based design in this course.

(Refer Slide Time: 17:13)

So, what do we do? We first focus and represent the software artifacts as models, we typically work with several different models. So, where do these models come from? So, graphs are one kind of models that I want to work with. Where do I get graphs from? I can get graphs from source code. You might have heard of control flow graph or a flowchart corresponding to a piece of code. In fact, there is something called data flow graph which represents the control flow along with tracking of the variables that are involved at a particular piece of statement in the code.

And then there is an notion of call graph in an inter procedural call graph when the codes modularized into several functions or procedures or method. I can also get graph from design; lots of designs these days are represented using UML notations which are basically finite state machine, state chart all these are some specific kinds of graph. They have several different specific parameters to each of them, but they are all basically graphs.

Another class of models that we will be working with what are called logical predicates. Where do logical predicates come? If you see a typical code or an implementation is started with them, at every decision point in the code. What is the decision point in the code, it could be an if statement, a while statement, a for statement which says that there is going to be some kind of the branching in the execution of a code, some kind of the choice in the execution of code. At every decision point in the code there is a predicate.

Based on whether the predicate is true or false the code execution takes one path or the other.

So, what we will try to do is we will try to work with graphs, logical predicates and a few other things is a models of software and design test cases based on these models, based on certain criteria.
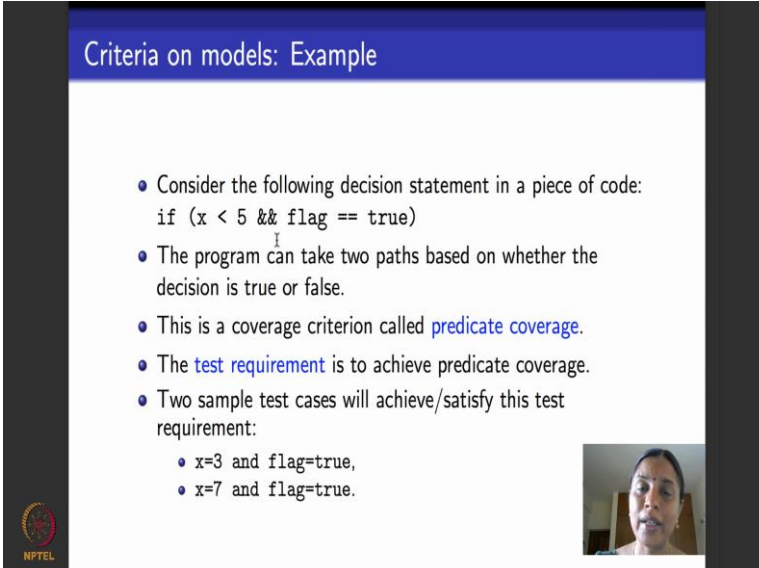
(Refer Slide Time: 19:09)

So, what are the four different model structures is that we will be working with as a part of this course. We will first begin with graphs next week. So, graphs typically in software define control flow, they also define data flow and as I told you little earlier they also could be graphs that depict calls of procedures, one procedure calling the other. So, we look at graphs as models representing such entities about a piece of software and then we will define criteria based on that and understand how to design test cases based on criteria.

Then we will look at logical expressions as the occurrence software, as the occurrence code, as they occurrence guards in design that represented as finite state machines, as they occur in requirements. And we will design test cases that check when a logical predicate could be true or when a logical predicate could be false. We will also for the point of view of black box testing look at sets of inputs that are given to a software and design test cases based on partitioning of those sets. So, sets would be another structures that we would be working with in this course.

And finally, when we look at coverage criteria we will also look at the underline syntax of programming language. So, every programming language as you know it could be C, it could be Java, it could be Python as an underlining grammar which tells you what is allowed in terms of writing in that programming language, how to write programs in the programming language. And when I compile a piece of software in the process of compiling I also check at the particular program adheres to the underline syntax or not. So, you can do mutation testing which exploits the syntax of a particular piece of programming language and design test cases by manipulating inputs that adhere to the underline syntax or grammar of that particular language.

So, to summarise this slide what we will do is we will look at software artifacts as four different structures or models: we will first consider them as graphs, and then will consider the logical predicates that occur in the code, then we will look at sets of inputs and outputs, and finally we will look at the grammar or the syntax of the programming languages. For each of these we will design what are called testing criteria and also see algorithms on how to design test cases test for these criteria.

(Refer Slide Time: 21:39)

So, here is a simple example to understand what criteria means. Let us say somewhere in your piece of code there is this following decision statement. How does this read? It says if x is less than 5 and a Boolean variable called flag is true then you do something. We are not interested in what we do I just want to focus on this particular if statement.
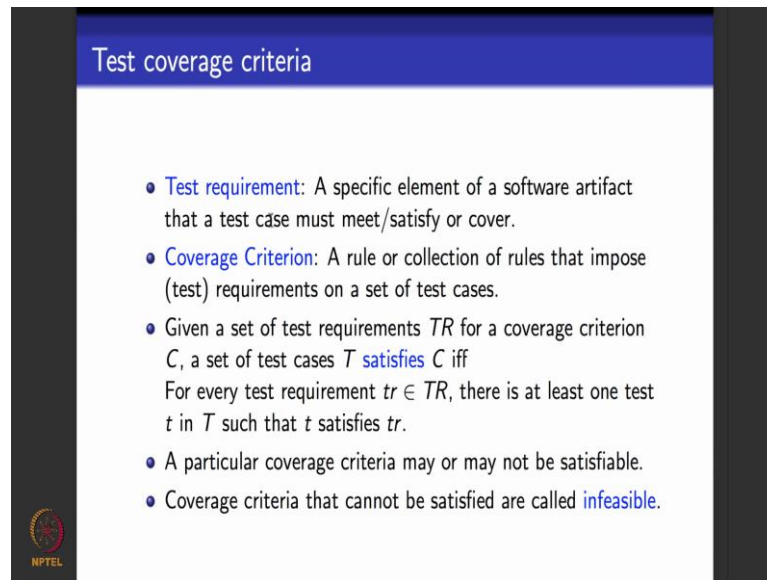
Now, what is the label of this if statement? The label of the statement is this particular thing right x less than 5 and flag is equal to true. It involves two kinds of variables: it involved maybe an integer variable x whose values compared to 5, and it involves a Boolean variables flag which is tested to be a equal into true. So, this whole thing is what we call a predicate. So, let say suppose this predicate turns out to be true that is x is indeed less than 5 and flag is also equal to true then you say that the if statement takes that then branch of the code. And suppose this predicate is false then you say that the if statement takes what is called the else branch of the code.

So, how will I make this predicate true? So, I say you write two kinds of test cases one that makes this predicate true and one that makes this predicate false; that way I would have exercise this if statement for predicate coverage or branch coverage. So, when I cover a predicate I also cover then branch of the if statement and predicate is true and I cover the else branch of the if statement when the predicate is false.

So, for this particular if statement: suppose I set x to be equal to 3 and the value of the variable flag to be equal to true then the whole predicate with this and operator evaluate to be true. So, this would exercise the then part of the if statement. And suppose I give a value of x to be 7 and let say I set the Boolean variable flag to be true, then this predicate would evaluate to be false, because this first condition will evaluate to be false. So, then this will mean that this whole if statement will have to exercise the else part of the code that is present after the if statement.

So, I say this is my test requirements; my test requirement is to be able to achieve predicate coverage on this if statement and these are the two test cases that achieve this test requirement. So, we will see how to define such coverage criteria based on graph, based on predicates, based on sets of inputs and how to automatically design test cases that will tell you whether the particular coverage criteria is achieved or not, and if it is achieved how is it achieved.

(Refer Slide Time: 24:21)

So, here are some definitions. So, what is the test requirement? A test requirement is basically a requirement that you say about a test case, you gave a think like please design a test case to cover this if statement, to execute this if statement once for the then part once for the else part. You might say please design a test case that will execute this while statement at least three times. You know these are all test requirements. They basically give you requirements on how to design test cases.
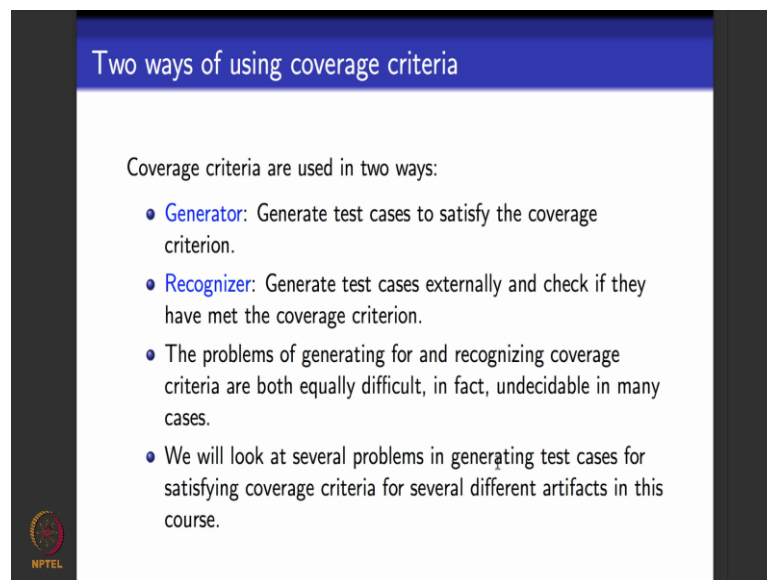
What are the criteria? A criterion is a rule or a set of rules that impose certain requirements on a set of test cases. Like in the previous slide if you say my test requirements is to test this if statement to be true once and to be false once then the criteria is what we call as predicate coverage. So, given a set of test requirements for a coverage criteria C this say that a set of test cases satisfies the coverage criteria C if every test case in that set of test requirements satisfies the coverage criteria C; right there is nothing more to it.

So, it is to be noted that a particular coverage criteria may or may not be feasible. Like for example: you take an if statement that occurs in something like this, let us say this if statement is labelled by a predicate that can never be made false for example right, it could be labelled by a predicate that is always true or a tautology. In which case I say that predicate coverage criteria on this if statement becomes infeasible, because predicate

coverage says that you make this predicate that labels and if statement true once and falls once.

The predicate is such that it can never be made false. If it can never be made false then I cannot achieve predicate coverage. When I cannot achieve predicate coverage then I say that the particular coverage criteria that I cannot achieve are what is called infeasible. So, it is undecidable problem to check or an arbitrary coverage criterion whether it is feasible or not. What we will see several heuristic or techniques that will let you decide implicitly whether particular coverage criteria is feasible or not; and if it is feasible to be able to design test cases that will satisfy the coverage criteria.
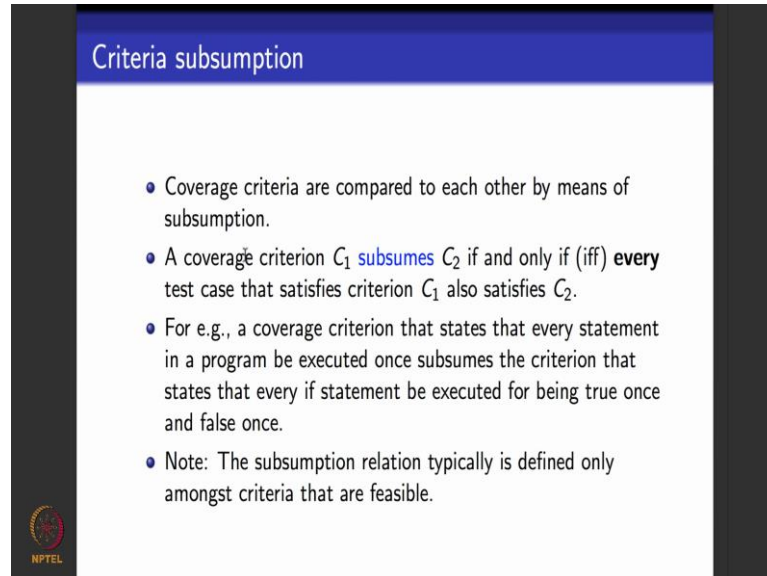
(Refer Slide Time: 26:39)

So, how do we design test cases that satisfy the coverage criteria? There are two base to do it: one is to automatically generate test cases that satisfy the coverage criteria, and the next one is to generate test cases externally; externally means you generated arbitrarily and then somebody gives you the coverage criteria. Now you take this set of generate test cases and the coverage criteria and you check if these test cases meet those coverage criteria so that is done by what is called a recognizer. And directly given a coverage criteria generating test cases for that coverage criteria is done by what is called generator.

Both generator and recognizer for testing based on coverage criteria are the most generality undecidable problems. In fact several times the algorithms for decidable fragments also have high complexity, but we will look at algorithms none the less

without worrying about what the complexities and see how to use them to be able to generate test cases.

(Refer Slide Time: 27:47)

Now, suppose I have a particular thing and I define several different coverage criteria on a particular test requirement. So, I should know how do each of these coverage criteria compared to each other. Like for example, I in a piece of code I could have two different coverage criteria: one coverage criterion says that- you design a set of test cases that will execute every statement once. Another coverage criterion says that- you design a set of test cases which will execute every branch for the then part and for the else part once.
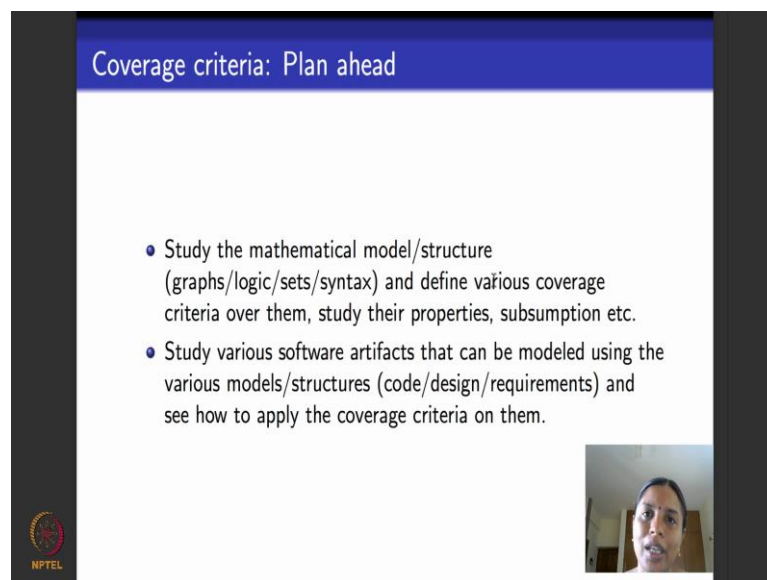
So, how do I know which of this is better and the work that I do for achieving one coverage criteria, implicitly also meets another coverage criteria. So, the notion of subsumption comes to a rescue here. So, we say that a particular coverage criteria C 1 subsumes another coverage criteria C 2 if every test case that satisfies criterion C 1 also satisfies criterion C 2. Like for examples suppose I told you there was a criteria with says executive every statement once right, which means executive every statement in the program. And let us say there was another other criteria which says that is executed in every branch that you encounter, execute the true part executive the false part.

It is obvious that the first coverage criteria with says executive every statement one subsumes the second coverage criteria, because if I execute every statement once I would also be executing the statements of the then branch and the statements of the else branch.

So, this is how I use the notion of subsumption to be able to compare coverage criteria. It is important to be able to compare coverage criteria to reduce the burden on you repeatedly generating test cases for different coverage criteria that are actually subsumed the other coverage criteria. So, it is important to know if I generate test cases for one kind of coverage criteria what other parts of other different coverage criteria I have already achieved because of this. So, the nation of subsumption will help you to answer these kinds of questions.

And a one important note to observe that is that suppose a particular coverage criteria is infeasible then I really do not look at what subsumption for that coverage criteria, I define subsumption only for feasible coverage criteria.

(Refer Slide Time: 30:14)

So, what is the plan a head for the next weeks? What we will do is that we will consider software as various mathematical models structures. So, what are the four structures that I said we will look at it: will look at graphs as model software artifacts, we look at logical predicate that occur in software artifacts, we will look at sets of inputs and outputs that are given to the software, and finally we will look at the underlining grammar or the syntax of software.

So, each of these models are structures we will define coverage criteria and discuss algorithms that will let us you design test cases to achieve coverage criteria. Of course, we look at subsumptions of coverage criteria and so on and so forth. So, what we will do

first is we look at the model at an abstract level. Just as I say as a graph, as a predicate, and then define algorithms for coverage criteria. Post that we will look at various software artifacts; so we will see how to take code and model it is a graph, and what are the coverage criteria that we look at the graphs and how do those help to test various aspects of the code. Then we will take design and model it as a graph, and then we will see what are the coverage criteria that will looked at for graphs that will be relevant to design test cases for these designs.

So, similarly when we go to logical predicate we look at logical predicates as their occur in code, look at coverage criteria and then see how these coverage criteria means testing which parts of the code. So, we look at these mathematical structures look at coverage criteria separately, then we look at how to model software artifacts using these structures and what do the various define coverage criteria mean for these structures.

So, next week when I begin my first module we will look at graph as models of software artifacts and define coverage criteria based on graph, so that will be the next lecture.

Thank you.