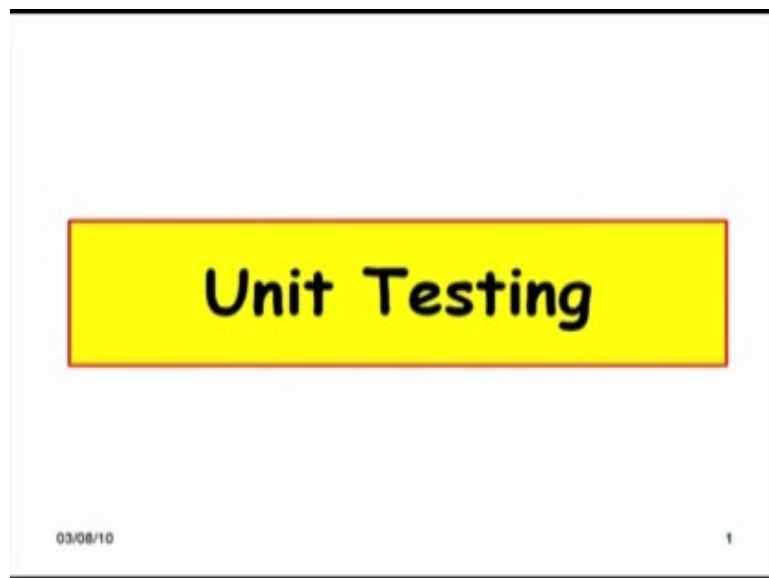


Software Testing
Prof. Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 05
Unit Testing

Welcome to this session. So, we will continue our discussion on Unit Testing. We were saying that in unit testing each unit is tested independently of the other units.

(Refer Slide Time: 00:29)




So, what it really means is that, if a unit needs to be called from some other unit and also, it calls some other units, then since you have to test it independently, we need to write small software here called as a drivers and stubs.

(Refer Slide Time: 00:48)

Unit Testing

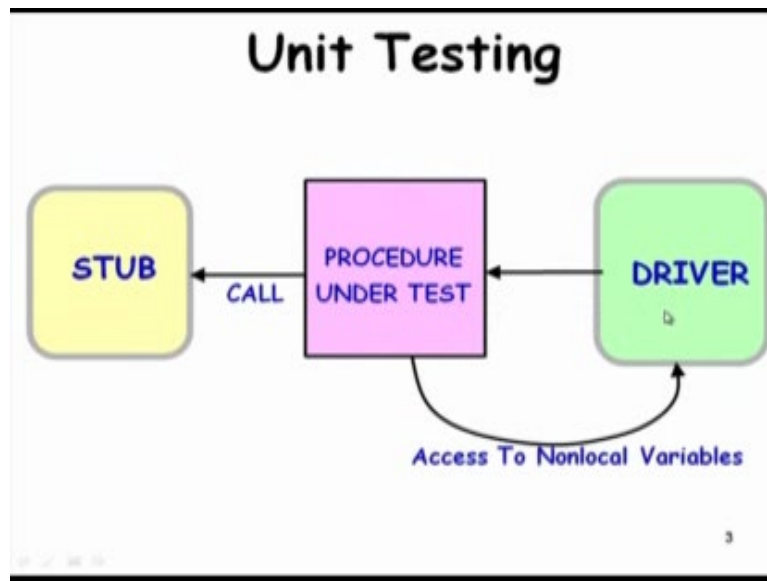
- **Testing of individual methods, modules, classes, or components in isolation:**
 - Carried out before integrating with other parts of the software being developed.
- **Support required for for Unit testing:**
 - **Driver**
 - Simulates the behavior of a function that calls and possibly supplies some data to the function being tested.
 - **Stub**
 - Simulates the behavior of a function that has not yet been written.



A driver is one which simulates the behavior of the function or the unit that calls this unit and possibly supplies some data to the unit being tested; whereas, the stub, it simulates the behavior of a function that has not yet been written or it is not available.

So, we are testing this unit, independent of the other unit and therefore, we use a stub here, which simulates the behavior of the other unit, possibly we have just stored some values here, that, if 2 is called, this other unit is called with the data 2, then it returns the data 11 or something. So, may be just a look up. These are very simple software, the driver and stub and these simulate the behavior of the other units, because we are testing this unit in isolation.

(Refer Slide Time: 02:09)



This just explains that little bit. So, this is the unit and then the stub is the one which this unit needs to call and the stub we have written here, for some sample data, we look up; and for, if your data, if it is called with some value, just look, looks up the result here and returns that. Whereas the driver, it not only calls the procedure under test that is a unit, it possibly might have other global data and so on, which this procedure might need. The unit might need some global variables and so on; the driver would have that.

(Refer Slide Time: 02:58)

The slide is titled "Quiz" and contains a question: "Unit testing can be considered as which one of the following types of activities?". Below the question are two options: "Verification" and "Validation", both preceded by a square bullet point. In the bottom right corner, there is a small circular video inset showing a man with glasses and a light blue shirt.


Now, let us have a small quiz. We know the unit testing, at least what is its purpose. Now, what do you think, would unit testing be a validation activity or a verification activity? We had actually discussed this issue sometime back that, whether unit testing will be a verification activity or a validation activity. Actually, the answer is that, unit testing will be a verification activity; it is not a validation activity, because validation means, we are testing the working of the entire system, with respect to the requirement specification; whereas, in unit testing, we are testing only one of the functions or unit, with respect to the design.

The high level design or the detailed design, sorry, the detailed design will have the specification for this unit. We are checking, in during unit testing, whether the, the code is written in conformance with the detailed design. And therefore, unit testing is a verification activity.

(Refer Slide Time: 04:23)

Design of Unit Test Cases

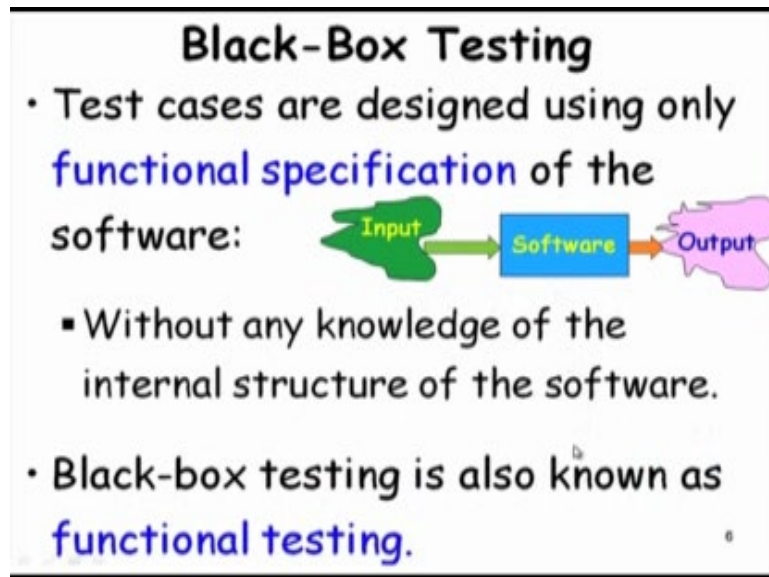
- There are essentially three main approaches to design test cases:
 - **Black-box approach**
 - **White-box (or glass-box) approach**
 - **Grey-box approach**



Now, let us see, how we design test cases for unit test. There are essentially 3 main approaches for designing test cases. One is black box approach, white box approach and grey box approach. The black box approach, as the name says, we do not have, we do not need to look at the code of the unit; we just look at the input, output behavior of the unit; that is, the specification, detailed specification, where we will have this unit, what does it do; given a data, what data does it, what result does it produce. And then we have this white box testing, where we actually have to look at the code. We look at the code, to design the test cases, based on whether some code elements are getting covered, decisions, conditions, and statements and so on.

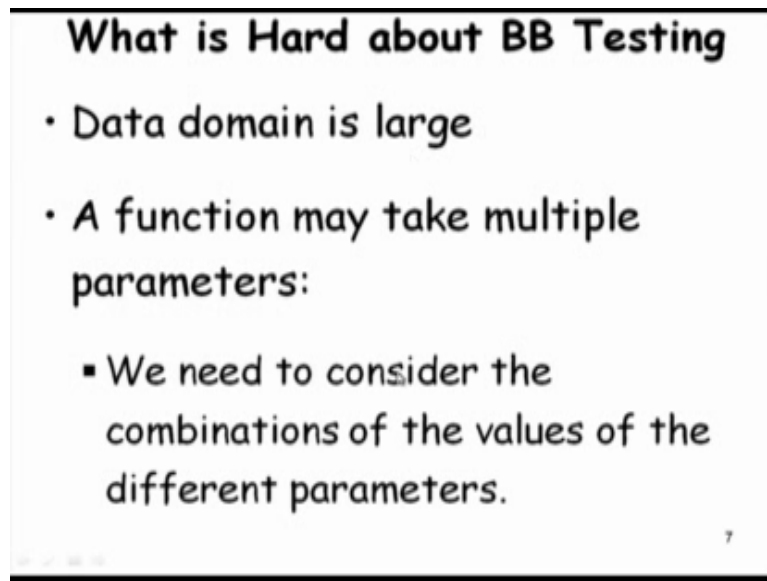
The third one is a grey box approach, where we look at the design of the unit and we come up with the test cases. So, in a black box, we just look at the input output behavior and come up with the test case. In white box, we look at the code and based on the source code, we design the test cases. In grey box, we do not look at the input output specifications, neither the white box, oh sorry, the code; we just look at the design of that and based on that, we come up with the test cases.

(Refer Slide Time: 06:14)



First, let us look at the black box approach. In the black box approach, the software is actually, the unit that we are considering, is considered as a black box. We do not know how the code has been written; we just know what is the input that it takes and what is the output it produces. So, this is also called as a functional specification of the software. So, the test cases are designed here, looking at the functional specification; we do not need to look at the code here. And, for this reason, the black box testing is also called as functional testing.

(Refer Slide Time: 07:01)



But, before we look at the black box testing strategies, let us answer a very basic question that, what is hard about this black box testing. We know the input data; we know the output result that is to be produced. So, what is hard about this? The hard about this black box testing, it is very hard because typically, for a practical software, the data for a unit will be extremely large, the data domain. So, testing, with respect to all elements of the data domain, will be very difficult; and not only that, it might take, unit might take multiple parameters and each parameter will have its data space; and, when we do testing, we will not only have to consider the data values that are taken by each parameter, but also, we have to check the different combinations of the different parameters.

Just having a parameter taking all values is not enough. We will have to check for each value of one parameter, what are the results when all other values for the other parameters are considered. So, that is the combinations; the combinations of the values for different parameters. So, this is the hard problem; not only, even if it takes one parameter, number of test cases can be very large and then typically, you will have to consider multiple parameters

(Refer Slide Time: 09:00)

What's So Hard About Testing?

- Consider `int check-equal(int x, int y)`
- Assuming a 64 bit computer
 - Input space = 2^{128}
- Assuming it takes 10secs to key-in an integer pair:
 - It would take about a billion years to enter all possible values!
 - Automatic testing has its own problems! *

Just to elaborate the problem, we will take a very simple example; that is we have written a very simple code, takes 2 integers as argument. Our unit here is a function; name of the function is check equal; taking 2 integers as parameters and then returns zero, if they are not equal and returns 1, if they are equal. Very trivial function and we are trying to test it. But then if we want to test exhaustively, whether it works for all possible values of x and y, we will have to consider how many values are, what is the domain size for x and y. So, the domain size for x, assuming a 64 bit computer, is 2^{64} ; and similarly, for y is 2^{64} . And then we have to consider all possible combinations of these 2 parameters; and therefore, it becomes 2^{128} . And then if we really test with this data value, 2^{128} , let us just, for our curiosity, check how long it will take. If we take 10 seconds, keying it very fast, each value, pair of value, it will take billion years to enter all possible values.


But then you might say that, why not automatically generate all this possible values and let the computer execute. So, that also will take long time to generate all possible values, because each execution takes finite time and it will keep on running for long time; and not only that, such automatic testing has its own problem; because if the result is produced, we do not know whether it is a correct result. So, only thing that we can detect is a crash.

Anyway, what we really wanted to point out here is that, the hard thing about black box testing is that, the number of possible data items are extremely large. So, our objective in test case design is to design a set of test cases, which should be as effective as testing with all possible values, but we should use a minimum number of test cases.

(Refer Slide Time: 11:42)

Solution

- Construct model of the data domain:
 - Called Domain based testing
 - Select data based on the domain model




So, how do we do that? The main idea in black box testing, we will look at several black box techniques, but one thing that is common to all this is that, we construct a model of the domain; because the data domain is very very large, you cannot really test it effectively, by looking at the domain itself, other than taking all possible values. We construct a model and then this model, we call as the domain model and based on this domain model, we select test data from this model. So, that is the main idea here. Now, based on that, let us see, how we design the test cases.

(Refer Slide Time: 12:39)

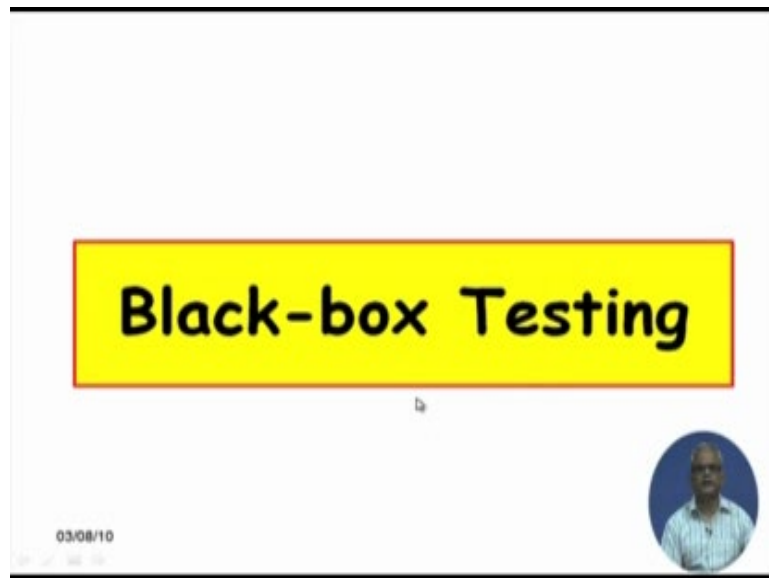
White-box Testing

- To design test cases:
 - Knowledge of internal structure of software necessary.
 - White-box testing is also called structural testing.



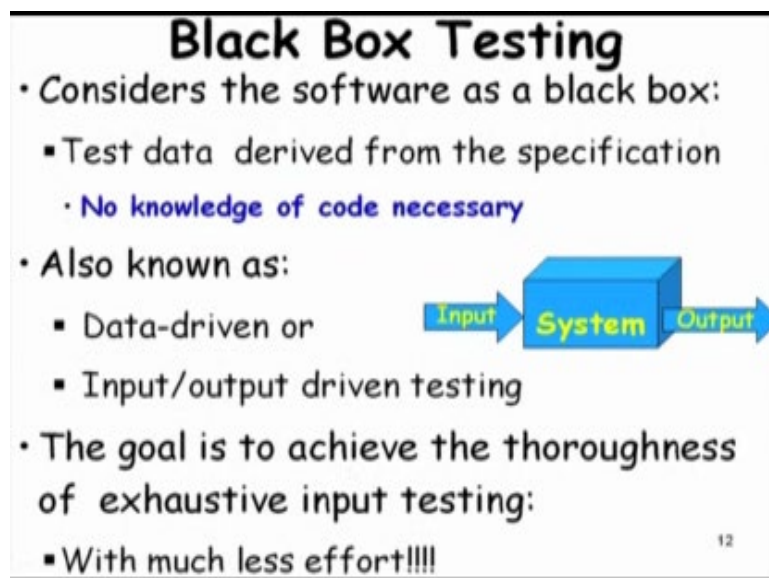
Before that, let us have one basic idea about white box testing. White box testing, we look at the code and based on that, we know what is the internal structure of the software and therefore, the white box testing is called as the structural testing.

(Refer Slide Time: 13:02)



Now, let us look at the black box testing and how do we design test cases; what are the different strategies available for designing the black box test cases.

(Refer Slide Time: 13:14)




Black box test case, as we are saying we just look at the input output behavior. We know what will be the input, corresponding output and system; we do not have any knowledge about that; appears like a black box to it; and therefore, it is also called as input output driven testing or data driven () testing. So, our goal in test case design is to achieve the thoroughness of exhaustive input testing, with as little number of test cases as possible.

(Refer Slide Time: 13:48)

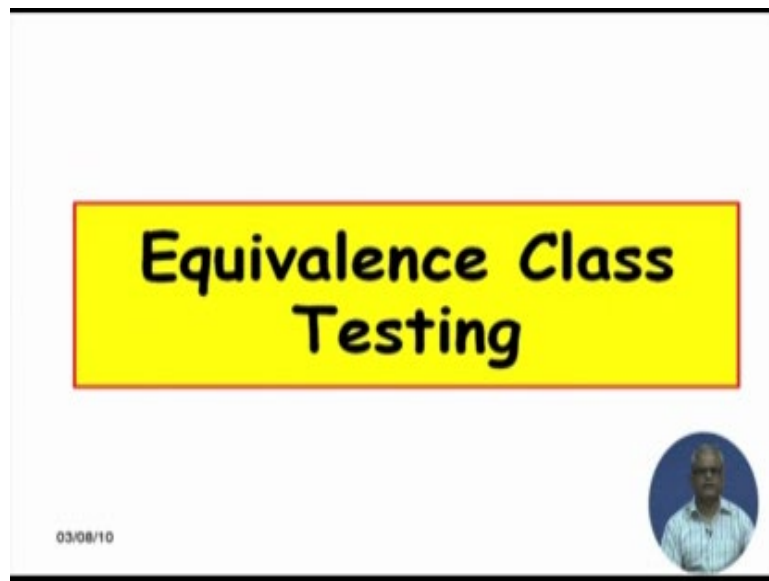
Black-Box Testing

- Scenario coverage
- Equivalence class partitioning
- Special value (risk-based) testing
 - Boundary value testing
 - Cause-effect (Decision Table) testing
 - Combinatorial testing
 - Orthogonal array testing



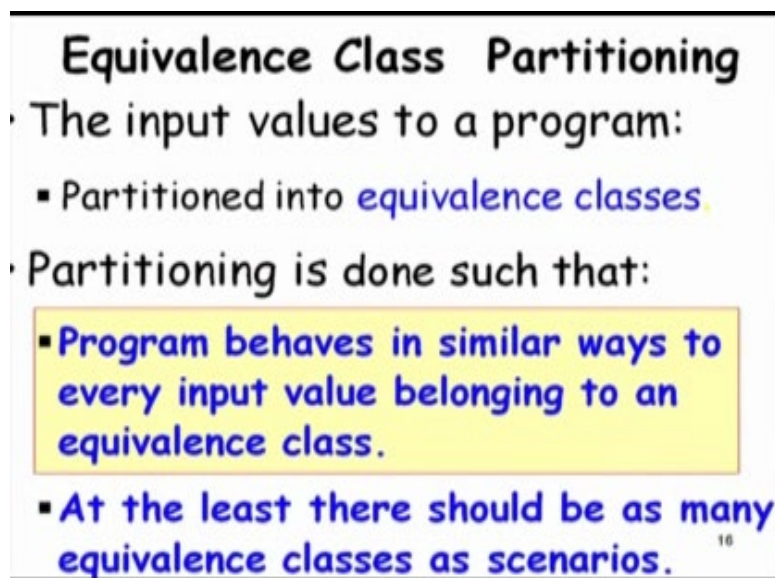
There are many black box test strategies that exist; Scenario coverage, Equivalence partitioning, Special value testing, Boundary value testing, Cause-effect testing, Combinatorial testing orthogonal array testing and so on. Let us look at these techniques.

(Refer Slide Time: 14:19)



First, let us look at the equivalence class testing.

(Refer Slide Time: 14:26)



In equivalence class testing, we need to construct a model of the domain, called as the equivalence partitions. We partition the input domain to equivalence classes. The idea is that, the partitioning is done such that. So, we will partition the data, input data, into a number of, number of classes, such that, each class will be, when you consider value from any class, the result will be the same as considering any other value from the same class. So, the program behaves in a similar way for

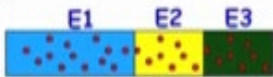
every input value belonging to an equivalence class.


When you consider, we have developed the equivalence classes; for any equivalence class, all the values are equivalent to the other values, in the sense that, checking the software with any one value, is as good as checking with all other values of that class. If you are doing a scenario based testing, in scenario based testing, it is very simple, where we just execute its scenario of the software. Let us say, book return example, the scenarios can be that, the book was returned successfully; another scenario can be, there are somebody who had reserved the book and therefore, the book return could not be done. So, whether it could not, it was really that, it could not be done when somebody else had reserved. So, that is a scenario.

The third scenario, may be the membership of the member had expired and when renewing, it said that, the membership has expired; you cannot renew the book; please return the book. So, whether this scenario is correctly implemented? So, in a scenario based testing, the requirement specification document, typically documents all possible scenarios of operation, of various functionalities. And, in scenario based testing, we just write test cases to execute each scenario. So, in equivalence partitioning, obviously, each scenario is a different equivalence class. This says that, if we know the scenarios, know the specification for this unit and we know what are the scenarios for this, then we will have as many equivalence classes at least, at least as many equivalence classes as there are scenarios of operation, for this unit. Of course, there will be more equivalence classes; let us look at those.

(Refer Slide Time: 17:56)

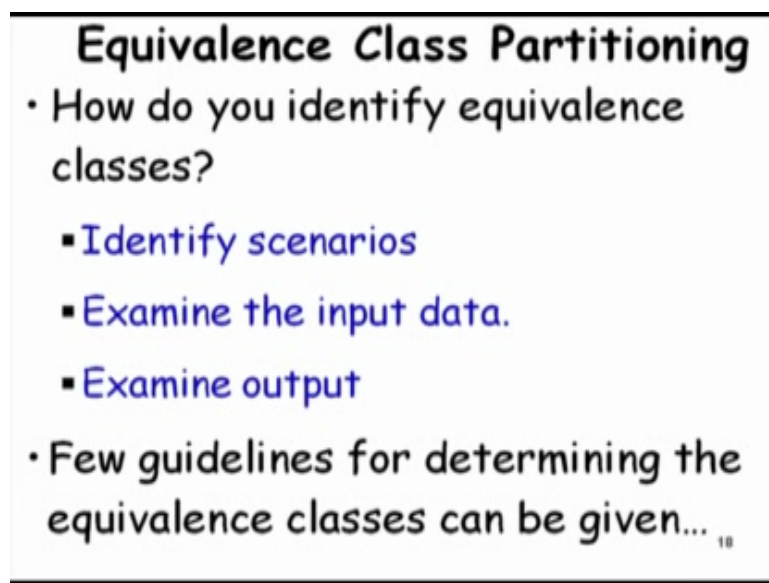
Why Define Equivalence Classes?

- **Premise:** 
- **Testing code with any one representative value from a equivalence class:**
- **As good as testing using any other values from the equivalence class.**



The premise of equivalence class testing is that, the input domain, we are partitioning into different equivalence classes. So, this input domain is partitioned into 3 equivalence classes and the main premise is that, if we take any one value from this equivalence class that should test the system as good as any other value taken from the same class. So, for equivalence testing, we just consider one value. So, what is the logic behind this assumption that, any value is as good as any other value? The assumption is, the basis for this assumption is that, once one feature, the software is exercised with one of this test data, it executes certain elements; and all other elements are executing the same set of program elements; and therefore, we will observe if it is success for one, then it will be also success for all other; if it is a failure for this and there will be failure for all other. The main assumption here is that, any one element here will be exercising the same set of program elements as any other data here.

(Refer Slide Time: 19:38)



Equivalence Class Partitioning

- How do you identify equivalence classes?
 - Identify scenarios
 - Examine the input data.
 - Examine output
- Few guidelines for determining the equivalence classes can be given... 18

Now, the hardest problem here, in equivalence partitioning is that, given a unit and its input output description, how do we design the equivalence classes? So, one thing is, we identify the scenarios; we examine the input data; we examine the output that might be produced for different data; and then based on that, we design the equivalence classes. So, we can have few guidelines about how to design the equivalence class partitions.

(Refer Slide Time: 20:19)

Guidelines to Identify Equivalence Classes

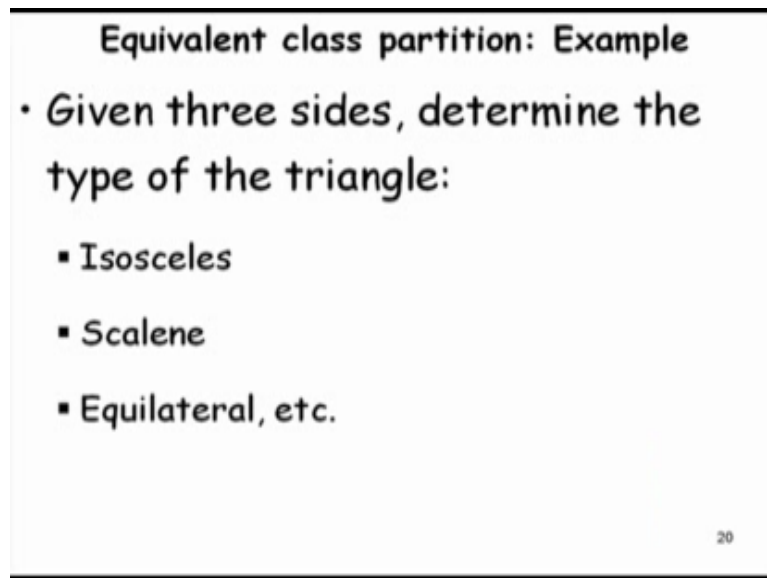
- If an input condition specifies a range, one valid and two invalid equivalence class are defined.
- If an input condition specifies a member of a set, one valid and one invalid equivalence classes are defined.
- If an input condition is Boolean, one valid and one invalid classes are defined.
- Example:
 - Area code: range --- value defined between 10000 and 90000
 - Password: value - six character string.

19

One is that, if an input data is specified by a range of values that, it can take between zero to 1000 or something, and then we have 1 valid and 2 invalid equivalence classes. If it specifies a specific number of test data, then we have 1 valid and 1 invalid equivalence class. If it is, input is a Boolean, and then we have 1 valid and 1 invalid equivalence class. For example, let us say, the area code for a telephone number is value between 10000 and 90000. So, what will be the equivalence classes? So, there will be 3 equivalence classes; 2 invalid; less than 10000, more than 30000, sorry, 90000 and valid equivalence class between 10000 and 90000.

Similarly, you might have a password 6 character string. So, here also, we will have 1 equivalence class, which is less than 6 characters and another equivalence class which is exactly 6 characters; third equivalence class, which is more than 6 characters.

(Refer Slide Time: 21:52)



Equivalent class partition: Example

- Given three sides, determine the type of the triangle:
 - Isosceles
 - Scalene
 - Equilateral, etc.

20


Now, let us look at some examples based on what we discussed and see if we can identify the equivalence classes; and later, we will have some practice also given, so that, you can test whether you can have practice design equivalence classes; and depending on the unit that under test, it can be very very challenging to design the equivalence classes. Let us look to start with, let us look at a very simple problem. Our unit or a function, takes 3 integers, denoting 3 sides of a triangle. Now, our function just writes down the type of the triangle, whether it is isosceles, scalene, and equilateral.

So, our function that we have written, takes 3 integers and then prints out or displays, what is the type of the triangle, whether it is a isosceles, scalene, equilateral, not a triangle and so on. So, how do we design test cases?

(Refer Slide Time: 23:21)

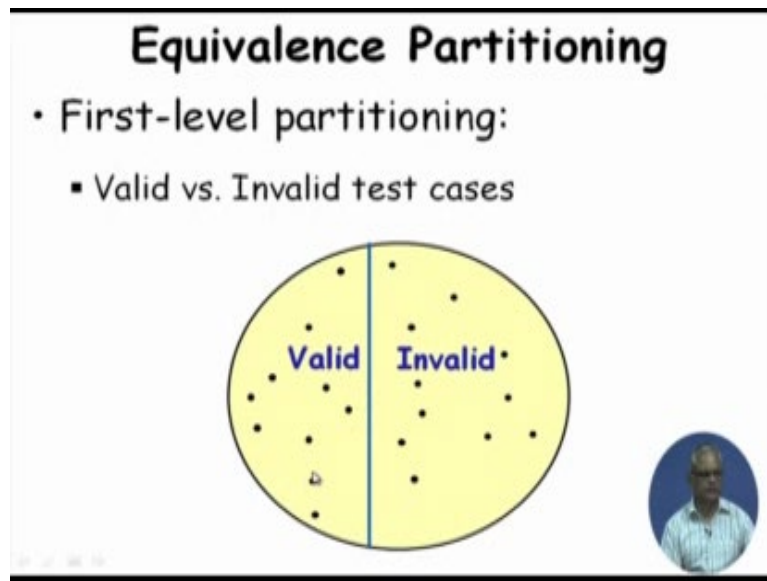
Equivalent class partition: Example

- Given three sides, determine the type of the triangle:
 - Isosceles
 - Scalene
 - Equilateral, etc.
- Hint: scenarios expressed in data in this case.



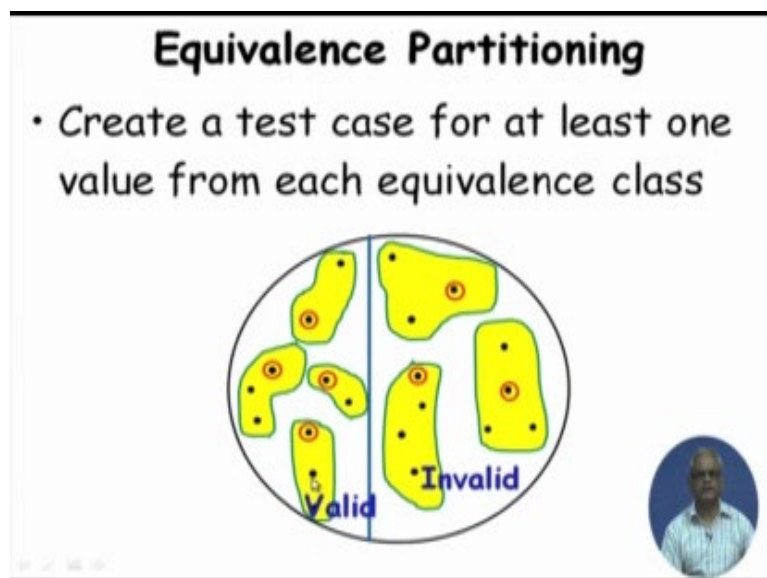
We do a scenario coverage that is the first thing. Just see here that, these are the different scenarios of operation. In one scenario, it writes down isosceles; another scenario, scalene; third scenario, equilateral; fourth scenario, not a triangle and so on. We must have test data, sorry, equivalence classes, defined corresponding to isosceles, scalene, equilateral, not a triangle, because different elements of the program are exercised, when we declare, where the function declares the 3 sides to be isosceles, scalene or equilateral. So, we just take 1 value from here, 1 value from here, 1 from here. So, these are, once we have the equivalence classes corresponding to this, we have those values.

(Refer Slide Time: 24:29)



So, the main idea here is that, the first level in the equivalence class, we have 2 sets of values that we design; 2 sets of equivalence classes; 2 equivalence classes here, 2 sets actually, the valid set of equivalence classes and the invalid set of equivalence classes.

(Refer Slide Time: 25:10)



First, we have 2 here, valid, invalid; and, for each of this, we design, we look for further equivalence classes, in the valid. So, these are set of valid equivalence classes and these are set of the invalid equivalence classes. And then once we have done the equivalence partitioning, we found

out all the valid equivalence classes and all the invalid equivalence classes; we just pick one value from each of them, randomly and this will form our equivalence class test suite. So, will stop here and we will continue from this point in the next session.

Thank you.