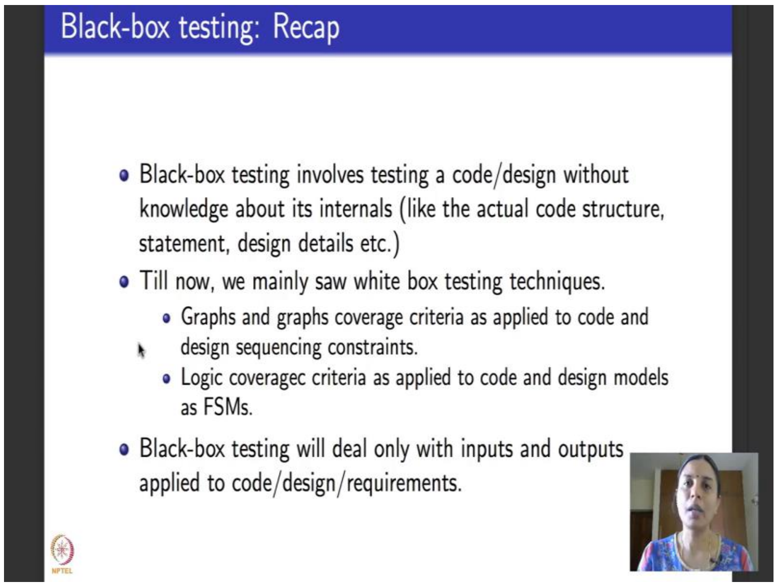
Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 31
Functional Testing

Hello all. This is week 7, second lecture. Last time we saw how to solve assignment 6. Now I am going to begin a completely new module in this course.

So far what did we see? We saw testing based on graphs and how it applies for source code, for design, for requirements. Then we saw testing based on logical predicates, how it applies to source code, for specifications and for finite state machines. Now we are moving into what is called, popularly called, functional testing or black box testing. And we will see this technique called input space partitioning, coverage criteria based on input space partitioning and look at an example of how to use these things.

(Refer Slide Time: 01:01)



Black-box testing: Recap

- Black-box testing involves testing a code/design without knowledge about its internals (like the actual code structure, statement, design details etc.)
- Till now, we mainly saw white box testing techniques.
  - Graphs and graphs coverage criteria as applied to code and design sequencing constraints.
  - Logic coveragec criteria as applied to code and design models as FSMs.
- Black-box testing will deal only with inputs and outputs applied to code/design/requirements.

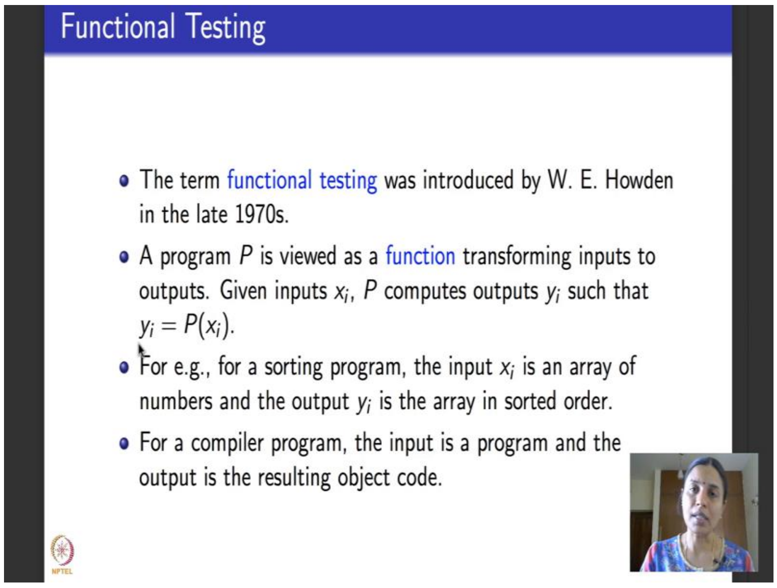
So, when we talking about functional testing, we are basically within the domain of black box testing. So, in the first week if you remember I had told you that there is this popular categorization in testing: white box and black box. So far the kind of things that we looked at graphs and logic. Whenever we apply it to code we were doing white box testing because we were talking about coverage criteria which exercised certain parts of the code and whenever we were applying it to specifications, we were doing what is called black box testing.

So, black box testing involves testing of the code or a design without knowing about it is internals. You do not know the language in which it is written, you do not know it is structure, you do not know how many loops it has where they are. You purely look at inputs and outputs in the code and test it based purely on the expected relationship between the inputs and the outputs. Black box testing typically deals with inputs and outputs, I could apply it to code, but without looking at the structure of the code. I could apply it to design without looking at the internals of the design. I could apply to requirements by directly it considering what the requirements state about the inputs and outputs.

So, this week we will fully be doing what is called black box testing. Even when we apply it to code we will purely focus only on the inputs that are there in the code, not really look at the structure of the code at all and see how to design test cases based only on the inputs. Of course, if we apply to requirements then there is no code and no need to see any structure and similarly when we apply to design, we will again focus only on the inputs and outputs.

So, we could apply to any software artifact, but remember that we are not going to design test cases based on the internal structure of the artifact. We are going to design test cases purely by looking at the inputs and the outputs of the artifact and what the artifact, which is code or design or requirements, is supposed to do while transforming the inputs to the outputs.

(Refer Slide Time: 03:02)



Functional Testing

- The term functional testing was introduced by W. E. Howden in the late 1970s.
- A program P is viewed as a function transforming inputs to outputs. Given inputs x_i, P computes outputs y_i such that $y_i = P(x_i)$.
- For e.g., for a sorting program, the input x_i is an array of numbers and the output y_i is the array in sorted order.
- For a compiler program, the input is a program and the output is the resulting object code.

So, the black box testing, if you refer to some other books popularly specially the old books, they will popularly call it as what is called functional testing. They practically mean the same. In fact, the term functional testing was introduced by this person called Howden, he was working for his statistics research institute in the late 1970s.

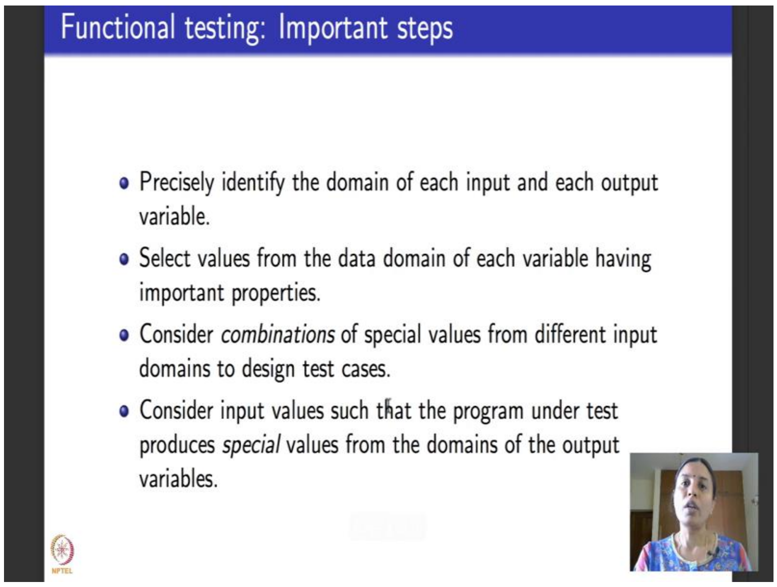
So, the idea is basically this. I have a program or a piece of code call it P, I will view it as a function that transforms the input that the program takes to produce outputs. So, this is a program that takes some inputs and produces outputs. When I functional test the program, I will view the program as a function that takes inputs, transforms the inputs to produce outputs.

So, given a vector of inputs x_i when I say a vector of input xi the program could have 3 inputs right like for example, it could have 2 integers and then it could have let us say a Boolean number. Another piece of program could have an array of numbers and let us say an index to search for or index whose value is to be return. So, different programs could have different inputs I put all the inputs of a program together and call it as a vector xi. So, that is like a set of all inputs of the program that are at any given point in time, that the program is suppose to process and produce outputs.

Outputs again a given program can produce more than 1 output. So, I put all the outputs together and consider it as a vector yi. So, what is a program? Program P can be thought of as a function that given xi returns yi. For example, suppose I have a program the sorts numbers then the input xi for the program is an array of numbers, and the output yi is the array, but in the sorted order. So, for example, suppose another example, let us take a program which is basically a compiler, let us say a C compiler. C compiler is a program right it could be a fairly large program, but it still a program it is a piece of code what is the input to a program like a C compiler? It is another program itself.

So, I can still represent it as a fairly large input vector to a program and then what is the output to this program? The output to this program is the compiled code or in the other words ,the object code. So, the input xi is another program and the output is the compiled or the object code.

(Refer Slide Time: 05:33)



Functional testing: Important steps

- Precisely identify the domain of each input and each output variable.
- Select values from the data domain of each variable having important properties.
- Consider *combinations* of special values from different input domains to design test cases.
- Consider input values such that the program under test produces *special* values from the domains of the output variables.

So, when I do functional testing, I consider every program as a function that takes inputs and produces outputs. So, what is functional testing? What are the important steps that people do in functional testing? They have to identify, remember I told you when we do functional testing we focus on inputs and outputs, we do not really look at the insides of a piece of program or a code.

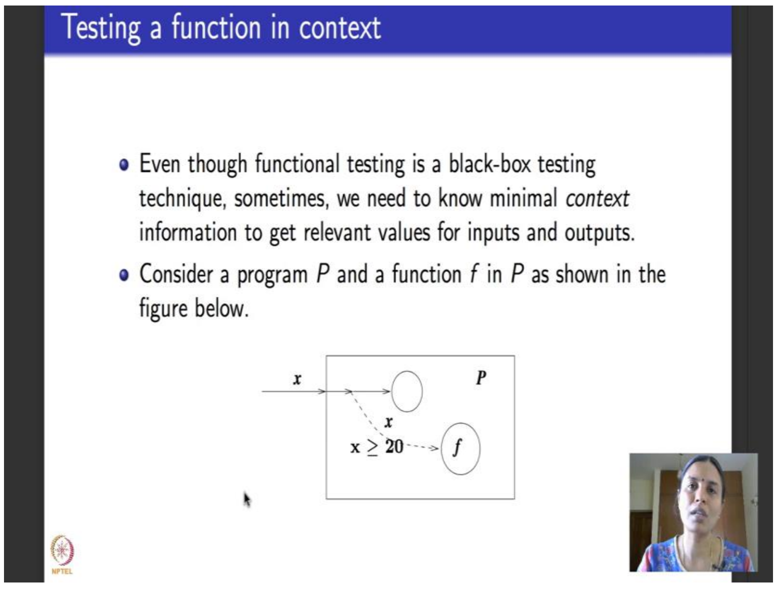
Now, I have to understand the inputs in detail first. I have to understand what and all constitutes the input, what is the domain of each constituent in the input. So, I have to identify the domain of each input. Similarly I need to know what and all constitutes the output, what is the domain or the type of each entity in the output. So, that is the first thing I know and understand. After I do that I select values from the domain of each input, and I select what will be the expected output for these values selected.

Several common input domains like integer, let us say typed input or a floating point typed input, let us say or an array of integers I have practically or fairly large number of values potentially infinite number of values. So, when I test for a program for the input output relationship, I cannot exhaustively test a program for all these values. The best that I can do is to select a sub set of values from the input output space. This whole week what we will be seeing is; what are the various ways in which we can select to test the program for it is input output relationship.

So, this selection or the combination of selections is what this week is going to focus on. Once we select values we consider combinations of the specific values from different input domains to test, design test cases and we consider input values such that this combination that we consider for each different combination that we consider, the program we expect, produces different-different outputs.

Let us say you consider the same combination again and again for which the program produces the same output again and again, then you would not been effective in testing the program. Ideally the combinations that you select should be different in the sense that they should produce different behavior in the program, hopefully resulting in different outputs. So, that is what we will focus on.

(Refer Slide Time: 07:52)



Testing a function in context

- Even though functional testing is a black-box testing technique, sometimes, we need to know minimal *context* information to get relevant values for inputs and outputs.
- Consider a program P and a function f in P as shown in the figure below.

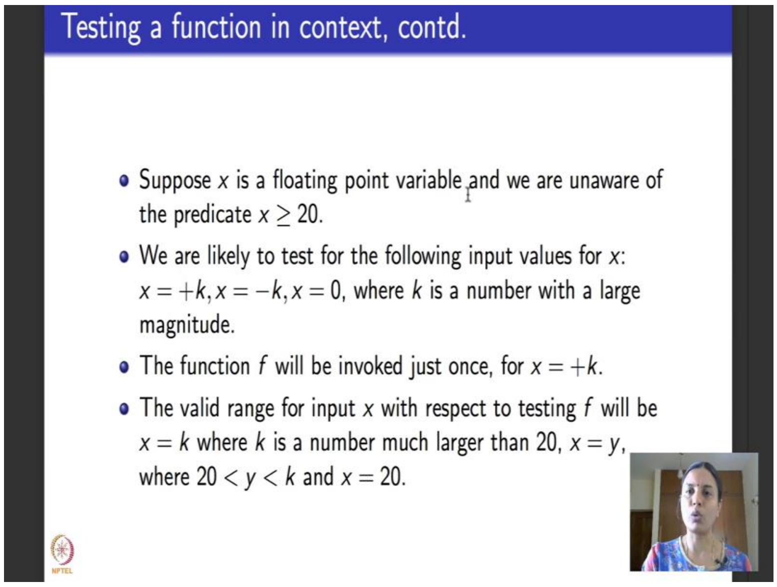
How to select and how to write combinations. Before I move on I would like to tell you that sometimes it may not be possible to purely focus on the inputs and outputs. Sometimes you might have to know a little bit about the insides of a program. Why is that so? Here is an example. This is popularly called doing functional testing in the context of something else. So, functional testing I told you is a black box testing.

So, we expect not to look at the insights of a program or a code, but sometimes we need to know a little bit. We need to know something like a minimal contextual information to be able to get relevant values for input and output. I will illustrate this through an example for you. Consider a program P, I have not really written the code for P, but assume that this is the space of program P, this is where the code of P resides. Somewhere in it is code P uses a function f. There is an input x to the program P and this x within P is passed on to call the function f whenever x is greater than or equal to 20.

So, the program P uses the input x, program P also has an internal function or a method called f. This program P uses the input x as input to the function f to call the function f whenever this condition is true. That is whenever x is greater than or equal to 20. Now let us say I want to be able to do functional testing for this program, which means what I really do not know this much that the program has a function f, that it is going to call f with x when x is greater than or equal to 20. Why I do not know about this because I am doing black box testing and I am not supposed to know. All that I know is x is an input to the program P.

So, I make an attempt to design select values for x such that I can test how the program P transforms the input x into corresponding output.

(Refer Slide Time: 09:48)



Testing a function in context, contd.

- Suppose x is a floating point variable and we are unaware of the predicate $x \geq 20$.
- We are likely to test for the following input values for x: $x = +k, x = -k, x = 0$, where k is a number with a large magnitude.
- The function f will be invoked just once, for $x = +k$.
- The valid range for input x with respect to testing f will be $x = k$ where k is a number much larger than 20, $x = y$, where $20 < y < k$ and $x = 20$.

Suppose x is a floating point variable and as I told you we are not aware of the presence of this predicate. We are not supposed to be aware, because we are doing black box testing. So, we are likely to test for the following input values of x. We will take x to be let us say one positive number, x to be one negative number and maybe we will test how the program behaves when x is 0.

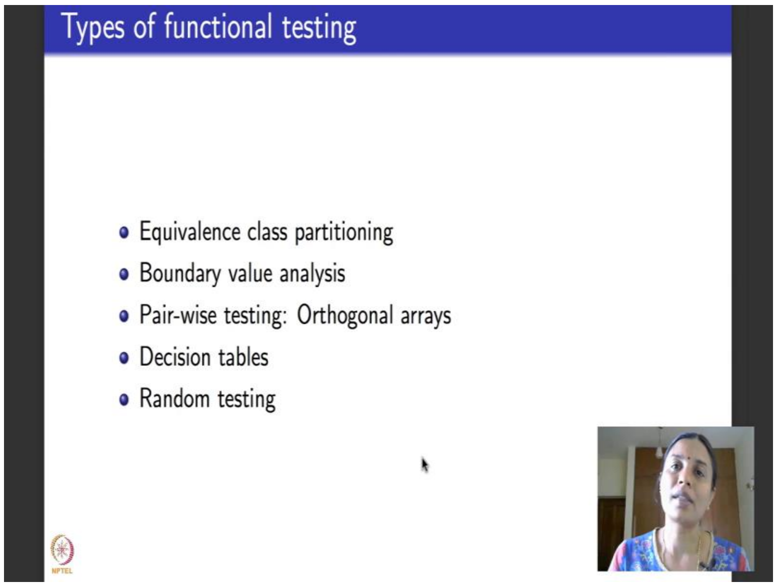
I have written k as a number with a large magnitude because we are unlikely to test it for small magnitudes, because it is a floating point number you if you choose 0 then you would choose a large number and you would choose a large positive and a large negative number. Now if you see these 3 choices that we have selected for input x: +k, -k, 0 which of these 3 choices will actually make the program call the function f? It will only be this +k because for the other 2 choices x is -k and x is 0, this predicate x greater than or equal to 20 which makes the program call the function f, that predicate becomes false.

So, only for one choice I would have call this. Now suppose I want to be able to focus on functional testing of the function f, I have not really exercised f except for only once. So, suppose I know the context in which f is called. Context meaning this predicate, that x greater than or equal to 20, then I will be in little more effective in designing my test case inputs to test the function f. If I had known the context I would have designed test cases that look like this. I would say x is k, where k is a number much larger than 20 and I would say x is y, where y is a value between 20 and k and then I would call it for x equal is equal to 20.

In all the 3 cases the function is called in this case at 20 and then in this case for a number larger than 20 and then that is in middle case for a number in between 20 and some large number, that is for a number close enough to 20. So, what I am trying to tell you through this slides is that even though we are focusing on black box testing, where we do not look at the internals of a program that looks like this, but suppose we want to do black box testing or functional testing focusing on a particular function f of the program, we might have to look at little bit inside the program just to know the context in which that function is invoked or called. Knowing the context will help us design select input values, such that we can exercise the calling of the function f itself correctly to be able to do black box testing for that function.

We still need not know the entire code for this program, it is enough to know the context in which this function is called for us to be able to help us to deign good test cases.

(Refer Slide Time: 12:33)



Types of functional testing

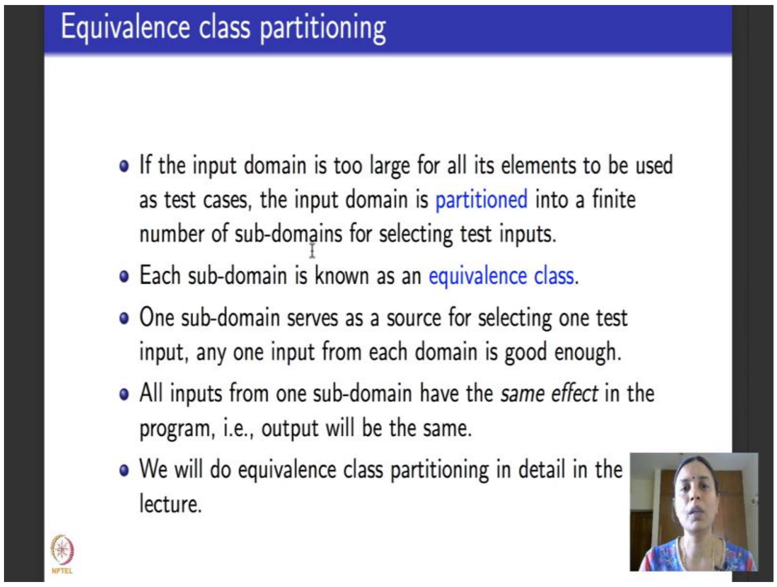
- Equivalence class partitioning
- Boundary value analysis
- Pair-wise testing: Orthogonal arrays
- Decision tables
- Random testing

What are the various types of functional testing that you will find in the testing literature? Here is a broad classification: in several books you will find this term very popular term called equivalence class partitioning or ECP or ECC, just EC. Then there is a related technique called boundary value analysis abbreviated as BVA, then you would have heard of the term pair wise testing.

In fact, there is a very popular technique which was introduced by an Indian statistician Mr. Rao called orthogonal arrays. Then you would have heard of the term decision tables and another usually popular way of doing functional testing is what is called random testing. What we will be focusing for a lot of this week is on some of these techniques, especially on equivalence class partitioning. I will call it input space portioning because that is what the text book by Amman and Offutt calls it as. Boundary value analysis we will see today. Pair wise testing we will see as one of the coverage criteria when we do equivalence class partitioning or input space partitioning. I will tell you what decision tables are today I will also tell you what random testing is today. We will do look at random testing in little more detail towards the end of the course when we do symbolic execution.

So, for the rest of this lecture, we will look at these techniques one at a time, and I will tell you through examples how to use these techniques. Equivalence class partitioning is one what we begin with.

(Refer Slide Time: 14:01)



Equivalence class partitioning

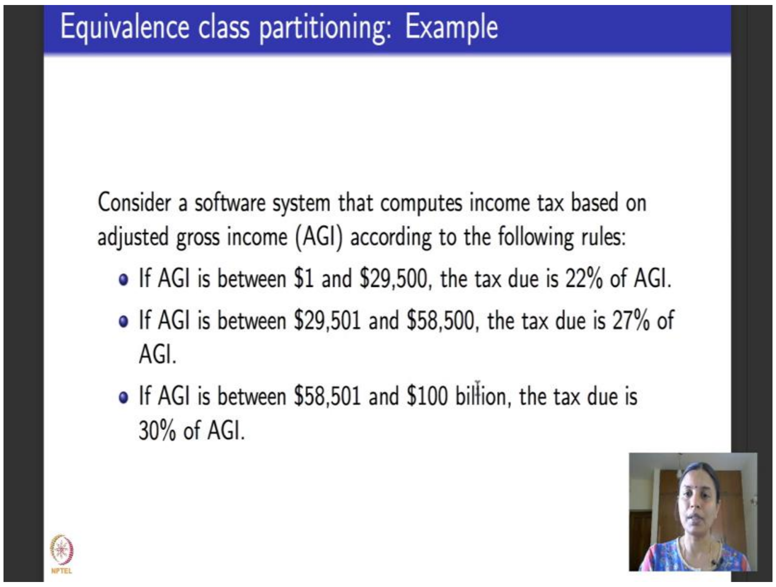
- If the input domain is too large for all its elements to be used as test cases, the input domain is partitioned into a finite number of sub-domains for selecting test inputs.
- Each sub-domain is known as an equivalence class.
- One sub-domain serves as a source for selecting one test input, any one input from each domain is good enough.
- All inputs from one sub-domain have the *same effect* in the program, i.e., output will be the same.
- We will do equivalence class partitioning in detail in the lecture.

So, equivalence class partitioning basically assumes that the input domain that I want to test is fairly large. Let us say the input is a floating point number, the input is an integer so it is practically impossible to test the program for all values of the inputs. So, I want to be able to partition the set of inputs such that the number of partitions is finite, and each such partition is known as an equivalence class. The word equivalence class comes because as far as the program is concerned, it is expected to behave similarly for each element that chosen from the partition.

So, we will make this very precise and clear. I will define it to you what a partition is and how to select inputs from partitions, what are the properties that partitions should satisfy everything in the next lecture. But for now equivalence class portioning means you take an input domain it is a fairly large set, I partition the input domain into several sub sets, a finite number of them such that I pick one input value from each partition. And the inputs have the property that from each partition if I pick one input value then the program when it runs on that randomly picked input, will produce the same output as any other input picked from the same partition.

So, as I told you we will do equivalence class partitioning as input space partitioning in detail in the next lecture.

(Refer Slide Time: 15:32)



Equivalence class partitioning: Example

Consider a software system that computes income tax based on adjusted gross income (AGI) according to the following rules:

- If AGI is between \$1 and \$29,500, the tax due is 22% of AGI.
- If AGI is between \$29,501 and \$58,500, the tax due is 27% of AGI.
- If AGI is between \$58,501 and \$100 billion, the tax due is 30% of AGI.

So, here is a small example that motivates equivalence class partitioning. So, remember I am doing black box testing. So, I do not look at the program at all, I will just look at the requirements or what the program is supposed to do. So, here is some program that to calculates income tax for some country. So, here are the rules based on which the program is supposed to calculate the income tax. To calculate income tax the program considers an input called adjusted gross income abbreviated as AGI and it says if the adjusted gross income is between 1 and 29500 then the amount of tax that is payable by that person is 22% of the AGI.

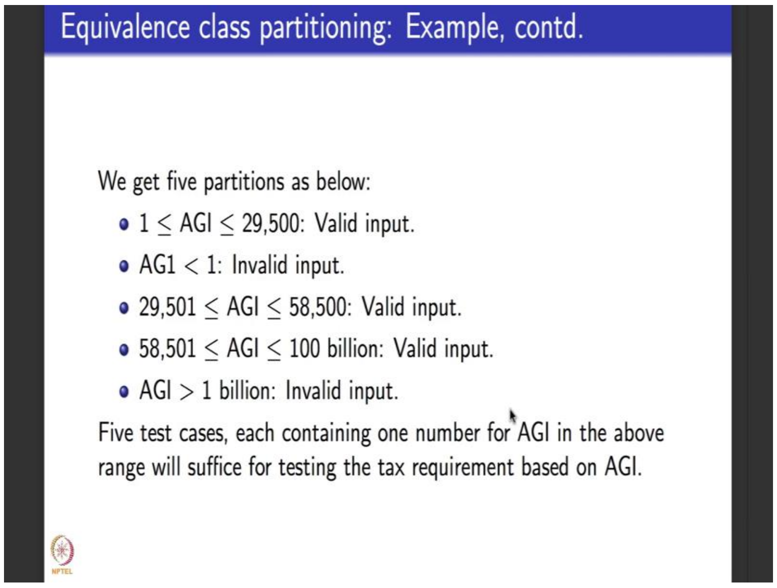
If the AGI is between 29501, that is it exceeds this 29500, but is within 58500 then the amount of tax to be paid by that individual is 27% of the AGI. If the AGI exceeds 58500 dollars, that is it is at least 58501 and if it is less than 1 billion which is a lot of money, then the tax to be paid by the individual is only 30% of the AGI. So, these are some rules it says based on these rules you take the input as AGI, the annual gross income and apply these rules to calculate the amount of tax to be paid by an individual.

So, that is a program that is as it and we are doing black box testing for that program. So, we do not get to see the program, we do not get to look at, we have to still be able to design test cases purely based on this information. What is the information that is given in the slide the program has one input AGI, the program is suppose to produce one output which is the amount of tax to be paid. Here are the rules if AGI is between 1 and 29500 tax is 22 %. If AGI is between 29501 and 58500, tax is so much. If AGI is above 58501 and less than 1 billion then the tax is so much. We know this much, so we know inputs, we know outputs, and we know how the program is expected to compute the output given input.

Now, if you see this input there could be so many individuals who have different-different AGIs, I clearly cannot test the program by giving each individual number as annual gross income and test and it is not necessary also. Why is not necessary, because if you see the requirements here it clearly states that if the annual gross income is between one dollars and 29500 dollars, which means for any value between these 2 numbers the tax is always 22 % of that value. So, it is enough to pick up one value from this number to be able to test it right.

And then, similarly it is enough to take up one value between the second range to test it, is enough to pick up one value between the third range to test it, and we also typically pick up values that are outside the range.

(Refer Slide Time: 18:47)



Equivalence class partitioning: Example, contd.

We get five partitions as below:

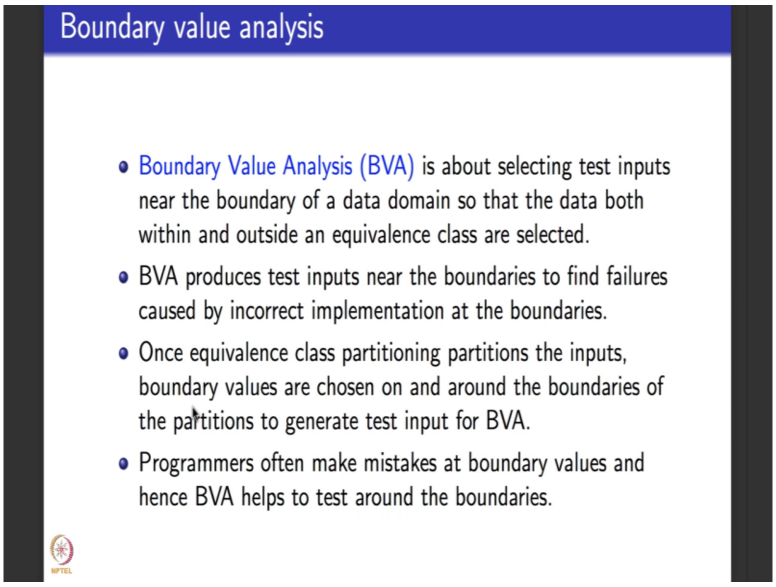
- $1 \leq AGI \leq 29{,}500$: Valid input.
- $AG1 < 1$: Invalid input.
- $29{,}501 \leq AGI \leq 58{,}500$: Valid input.
- $58{,}501 \leq AGI \leq 100$ billion: Valid input.
- $AGI > 1$ billion: Invalid input.

Five test cases, each containing one number for AGI in the above range will suffice for testing the tax requirement based on AGI.

So, I will give a number less than 1 dollars and I will give one number greater than 1 billion which constitute the boundary of these equivalence class partition. So, totally we get 5 partitions: one partition is for between one, annual gross income between one and 29500, it is a valid input, another partition is for the annual gross income being less than 1, it is an invalid input, because annual gross income is a negative number let us say or let us say 0.

The third partition which is again valid is annual gross income it is between 29501 and 58500, the fourth partition which is another valid input it is annual gross income is between 58501 and 100 billion and there one final outside the value or invalid input, which is annual gross income will greater than 1 billion. Maybe that person does not have to pay any tax. So, 5 test cases one for this value one for this range, one for this range, one for this range and one for this range is enough if fully test the program for correct functionality.

(Refer Slide Time: 19:41)



Boundary value analysis

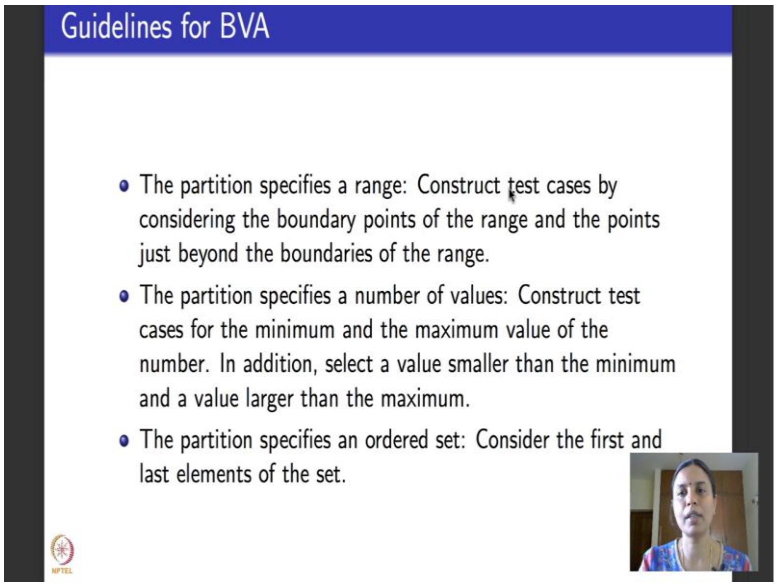
- Boundary Value Analysis (BVA) is about selecting test inputs near the boundary of a data domain so that the data both within and outside an equivalence class are selected.
- BVA produces test inputs near the boundaries to find failures caused by incorrect implementation at the boundaries.
- Once equivalence class partitioning partitions the inputs, boundary values are chosen on and around the boundaries of the partitions to generate test input for BVA.
- Programmers often make mistakes at boundary values and hence BVA helps to test around the boundaries.

Moving on, there is another popular technique in functional testing called boundary value analysis abbreviated as BVA. What is BVA about? It is about selecting test inputs nearer the boundary of a data domain so that the data within and outside the boundary is well taken care of. For example, if you go back to the same example, what constitutes boundaries? Boundaries are what if the AGI is one rupee, one dollar sorry, what if the AGI is exactly 29500 dollars, what if the AGI is let us say 29500.5 dollars it is still more than this, but it is less than this right.

So, boundaries values constitute the boundaries of the various partitions and the idea is to be able to use the values at the boundaries to be able to produce test inputs, because usually programmers makes small mistakes there or people who writes specification makes small mistakes there. They would not have taken care of the boundary values correctly, they would not have put the less than or equal to when they had to put it. They would may be put less than instead of less than or equal to. So, it is useful to pick up input values exactly at and around the boundary to be able to functional test a program and boundary value analysis is about doing that.

Once equivalence class partitioning partitions the input, we choose boundary values so that they are on or around the boundaries of the partition to generate test inputs for BVA. Why, because we believes that typically programmers make mistake at the boundaries and those errors are easier to catch and the test inputs design for BVA will help us to catch those errors. So, how do you do boundary value analysis. Here are the guidelines: the equivalence class partitioning specifies a range like we saw here right, here is a range first range what we one and 29500 second range says less than 1 and so on.

(Refer Slide Time: 21:46)



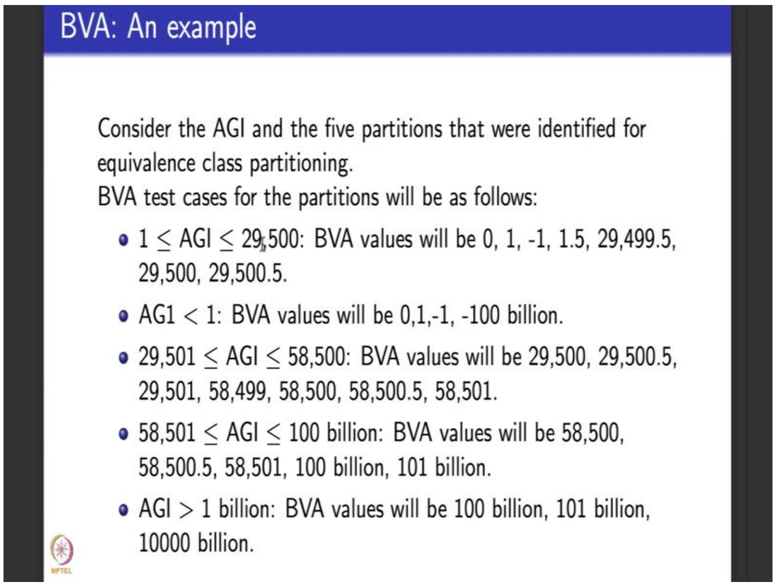
Guidelines for BVA

- The partition specifies a range: Construct test cases by considering the boundary points of the range and the points just beyond the boundaries of the range.
- The partition specifies a number of values: Construct test cases for the minimum and the maximum value of the number. In addition, select a value smaller than the minimum and a value larger than the maximum.
- The partition specifies an ordered set: Consider the first and last elements of the set.

So, you take that range and you construct test cases by considering the boundary point of the range, and the points just beyond the boundary of the range and if the partition specifies number of values, we will see examples of this as we move on, you construct test cases for the minimum value and for the maximum value as you mean that the values are order.

If the partition specifies an ordered set then you consider the first and the last element of the value. This is how you pick up test case input values while you are doing boundary value analysis.

(Refer Slide Time: 22:13)



BVA: An example

Consider the AGI and the five partitions that were identified for equivalence class partitioning.
BVA test cases for the partitions will be as follows:

- $1 \leq AGI \leq 29{,}500$: BVA values will be 0, 1, -1, 1.5, 29,499.5, 29,500, 29,500.5.
- $AG1 < 1$: BVA values will be 0,1,-1, -100 billion.
- $29{,}501 \leq AGI \leq 58{,}500$: BVA values will be 29,500, 29,500.5, 29,501, 58,499, 58,500, 58,500.5, 58,501.
- $58{,}501 \leq AGI \leq 100$ billion: BVA values will be 58,500, 58,500.5, 58,501, 100 billion, 101 billion.
- $AGI > 1$ billion: BVA values will be 100 billion, 101 billion, 10000 billion.

So, we will take the same annual gross income and tax computing based on the annual gross income example. If you remember we had defined 5 partitions here is it these 5 partitions. What will BVA do? This was the first partition right one less than or equal to AGI less than or equal to 29500 boundary value analysis will pickup values around the boundary. So, let us take the left boundary here left boundary is one less than or equal to.

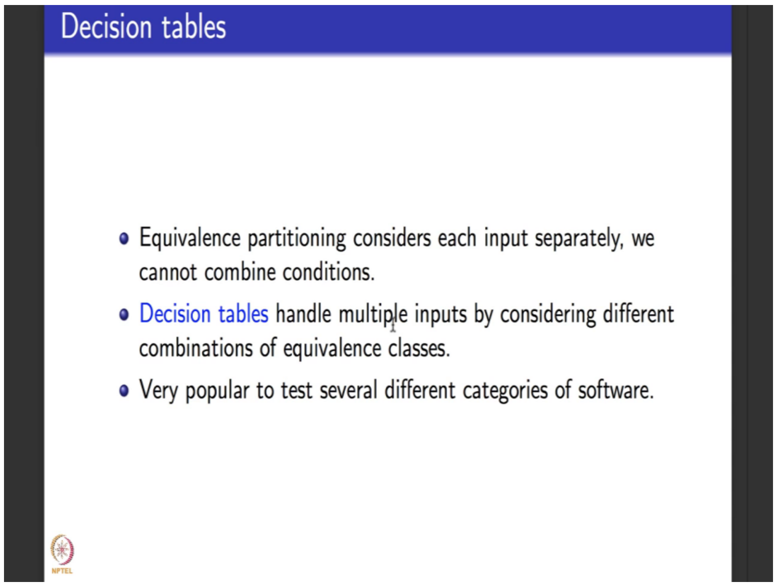
So, the values around the left boundary will be 0, 1 which is at the boundary, minus 1 which is before the boundary 1.5 which is just after the left boundary. Similarly for the right boundary what we do is we take 29500 and 99.5 which is just below the right boundary 29500 at the boundary 29500.5. Why did I choose this 0.5, mainly because if you see the next one assumes that it is 29501. So, let us say the annual gross income is a floating point number then the program should be well take up to handle 29500.5, may be you should over approximate it to be 29501 and then calculate tax, but the programmer should have taken care of it.

So, writing a test case like this will help to find an error if the program is not taking care of it. Now moving on, for the second partition which was this annual gross income less than 1, boundary value analysis will be 0, 1, -1 and something like -100 billion, which is like a fairly large negative number. -100 billion if you do not like it, you put it as -3, -7, it will be the same as for as the program behavior is cons. Again for the third partition you take this which is a third partition that was given in the equivalence class partitioning right boundary values 29500, 29500.5, 29501 which tests for the left boundary. For the right boundary you have 58499, 58500, 58500.5, 58501, this tests for the right boundary.

Similarly, if a next partition right values around the boundary the last partition right values around the boundary here the last value that I have written here I had given some fairly large number just to make sure that the program handles some corner cases, but if you do not like this fairly large number you could give something like 100 and 5 billion or something like that and stop it. So, is it clear?

Please, how BVA works? BVA assumes that equivalence class partitioning or input space partitioning has been done on the program and designs test cases by picking up input values on and around the boundary of each partition.

(Refer Slide Time: 24:51)



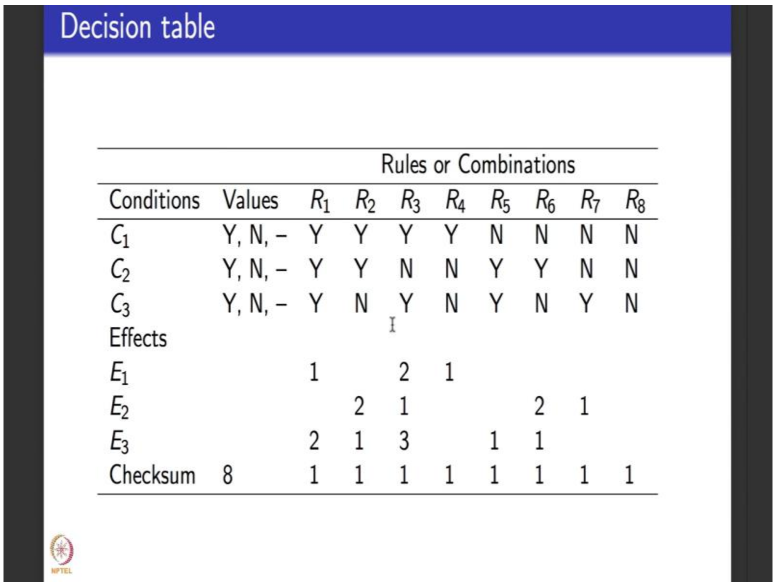
Decision tables

- Equivalence partitioning considers each input separately, we cannot combine conditions.
- Decision tables handle multiple inputs by considering different combinations of equivalence classes.
- Very popular to test several different categories of software.

The next popular functional testing technique that you would have heard of it is one of the most popular functional testing techniques that I know of is what is called decision tables. Why do decision tables come into picture? They come into picture for the following reason. A technique like equivalence class partitioning considers each input separately, it cannot handle a combination of inputs. In fact, in the example that we saw which involved annual gross income, there is exactly one input which is the annual gross income.

So, suppose there was an annual gross income and age and the tax rule says that if the annual gross income is so much and the age is above 60, let us say the person is a senior citizen then the tax is slightly less. Equivalence class partitioning may not be able to handle it for that you need what is called decision tables.

(Refer Slide Time: 25:48)



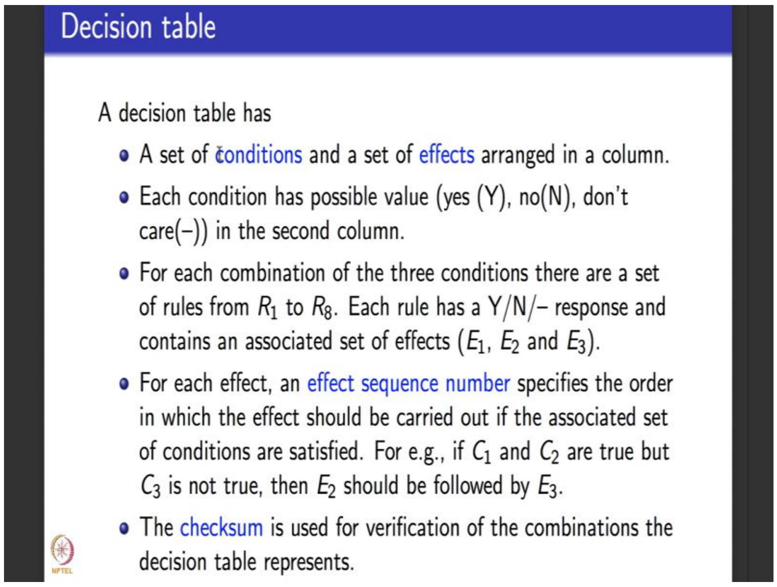
Decision table

		Rules or Combinations							
Conditions	Values	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8
C_1	Y, N, –	Y	Y	Y	Y	N	N	N	N
C_2	Y, N, –	Y	Y	N	N	Y	Y	N	N
C_3	Y, N, –	Y	N	Y	N	Y	N	Y	N
Effects									
E_1		1		2	1				
E_2			2	1			2	1	
E_3		2	1	3		1	1		
Checksum	8	1	1	1	1	1	1	1	1

Decision tables can handle multiple inputs by considering different combinations of equivalence classes. So, how does a decision table look like. This is how it will look like. There will be a fairly large table. What do each of these things mean, we will see. So, what are the entities that are there in the decision table. There are something called conditions, then there are things called values. Values are typically yes abbreviated as y, n abbreviated as no, I mean sorry no abbreviated as n, and this dash rewrite as do not care. What do I mean by do not care? Do not care means, it is the value is immaterial of whether it is yes or no and these rules are basically populated with yes, no and do not care symbols for each condition.

So, it is says condition one should be satisfied, that is what this yes mean in rule one. Condition 2 a condition one should be satisfied in rule 2, condition 2 should not need not be satisfied in rule 3. So, that is why it is written as a no. So, these rules basically say for each of the conditions whether each condition is needed ,not needed or I do not care. In this particular example I have just filled it up with yeses and nos I have not given do not cares, but you can write do not cares also. Then the later half of the decision table has what a called effects. Let us say here there are 3 effects E 1, E 2, E 3 and then there is a checksum which tells you what happens to each of these effects.

(Refer Slide Time: 27:19)



Decision table

A decision table has

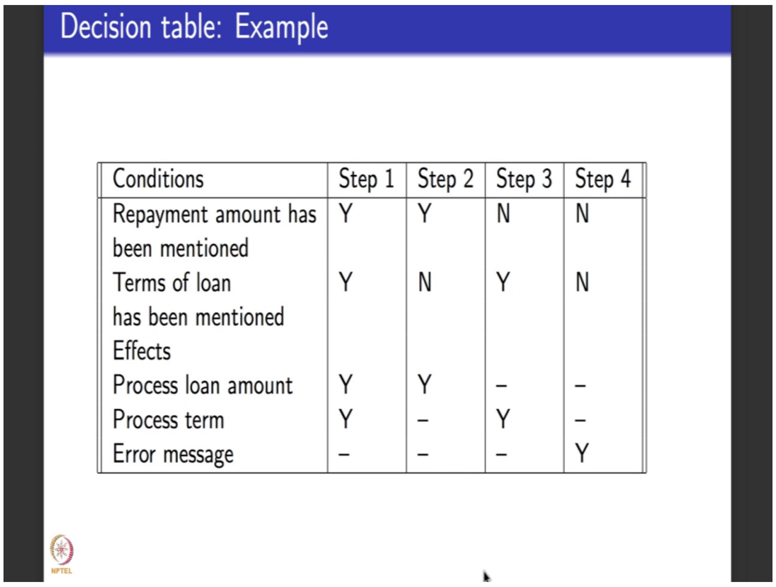
- A set of conditions and a set of effects arranged in a column.
- Each condition has possible value (yes (Y), no(N), don't care(−)) in the second column.
- For each combination of the three conditions there are a set of rules from R_1 to R_8. Each rule has a Y/N/− response and contains an associated set of effects (E_1, E_2 and E_3).
- For each effect, an effect sequence number specifies the order in which the effect should be carried out if the associated set of conditions are satisfied. For e.g., if C_1 and C_2 are true but C_3 is not true, then E_2 should be followed by E_3.
- The checksum is used for verification of the combinations the decision table represents.

So, how do I read it? I read these as do it in order of priority, means if, we will see an example. So, what is a decision table have it has a set of conditions and a set of effects which are arranged in a column like this, the first column. Each condition has possible values yes no do not care, in the second column that is what is written here. For each combination of the 3 conditions there are rules, in this case there are 8 rules, it need not be 8 rules all the time rules are flexible, you can write as many as you want to be able to exhaustively capture the conditions on the inputs. Each rule can be a yes no or a do not care and with each rule, there are a set of effects.

Like for example, with rule one which are the 2 effects that are associated with rule one effect one E 1 and effect 3 E 3, effect 2 is left blank which means effect 2 does not matter for rule one and what is this order say, 2 and 1? It is says if this is true which means all the 3 conditions are satisfied in which case rule one applies effect one occurs first and then effect 2 occurs. So, for each effect there is a sequence number which is what is this part, effect sequence number which specifies the order in which the effect should be carried out if the associated set of conditions are satisfied. For example, if C 1 and C 2 are both yes and C 3 is a no, then E 2 should be followed by E 3.

So, let us see if C 1 and C 2 are both yes, which is rule 2, and C 3 is a no, then E 3 is followed by E 2, is it clear. So, what is checksum? Checksum is used to the standard use of checksum is to verify if the combination of the values is correct. So, this is how a decision table looks like.

(Refer Slide Time: 29:04)



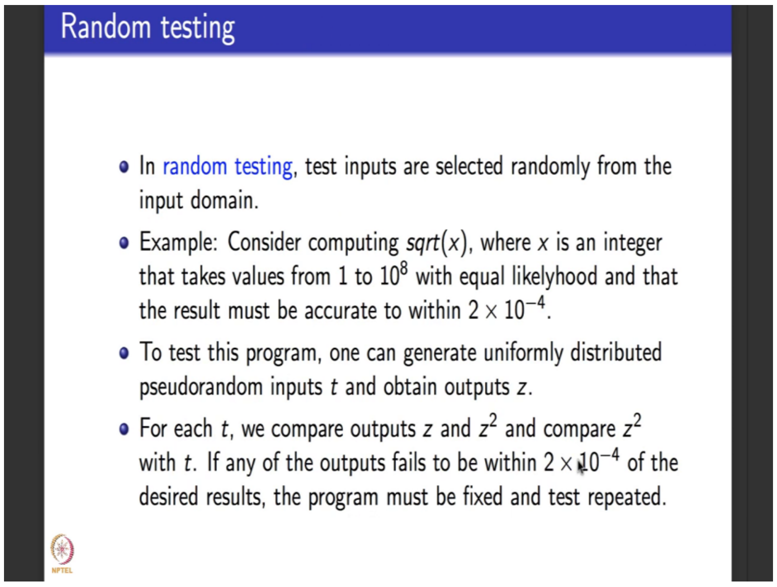
Decision table: Example

Conditions	Step 1	Step 2	Step 3	Step 4
Repayment amount has been mentioned	Y	Y	N	N
Terms of loan has been mentioned	Y	N	Y	N
Effects				
Process loan amount	Y	Y	–	–
Process term	Y	–	Y	–
Error message	–	–	–	Y

So, we will see a small example. So, let us say there is a loan management system and that calculates what is the amount of loan taken what is the amount to be repaid and what are the what is the term is to be paid over 3 years, over 5 years, over 15 years what is it rates. So, there are 2 kinds of inputs repayment amount is one input term of a loan which is a number of years with which we repay the loan is another input. So, the condition says, has the repayment amount been mentioned, yes or no? Similarly as the term of the loan been mentioned yes or no and what is the effect? If the repayment amount has been mentioned and the term of loan has been mentioned, then you can process the loan amount and you can process the term. But let say if the repayment amount has been mentioned and the term of loan is not been mentioned, that it is a no, then you can only process the loan amount you cannot process the term.

Similarly if the amount is not mentioned, but the term has been mentioned then you cannot process the loan amount you can only process the term. If both have not been mentioned that is, it is a no then you have to give an error message. So, this is how you write requirements for a design decision table. Based on these requirements you write test cases. So, if these rules are true in this case, you write a test case that will do processing now by taking input as repayment amount taking input as the term of loan and produce the expected output.

(Refer Slide Time: 30:47)



Random testing

- In random testing, test inputs are selected randomly from the input domain.
- Example: Consider computing $sqrt(x)$, where x is an integer that takes values from 1 to 10^8 with equal likelyhood and that the result must be accurate to within 2×10^{-4}.
- To test this program, one can generate uniformly distributed pseudorandom inputs t and obtain outputs z.
- For each t, we compare outputs z and z^2 and compare z^2 with t. If any of the outputs fails to be within 2×10^{-4} of the desired results, the program must be fixed and test repeated.

The second column, it will take input only as the repayment amount, the third column, it will take input only as the term of processing. In forth column it will it just handles error handling is the program done exception handling or error handling well that is what the forth case will test in the decision table. I hope decision tables are clear. Moving on we will bind up this module with brief look at random testing. What does the word random say? Random says pick a random input and test. You might wonder why is this useful. In fact, it turns out to be a very popular testing technique. There is a popular class of tools called monkey runners which is like imagine monkey producing inputs to a program.

So, this tool will produce inputs that are as random as that. These are very useful when you test programs if it has handled exceptions and all kinds of error conditions very well. So, random testing basically, it selects a test input randomly from the input domain and gives it to a program. But they also see how do we use randomly selected inputs to know if a program has an error, here is an example that illustrates that. Consider a program that computes this square root of a number I have written it like this sqrt(x), where x is an integer. Let us say x takes values from 1 to 10^8 with equal likelihood, and we say that the result this program produces must be accurate up to within this number $2x10^{-4}$.

So, to test this program we can generate a uniformly distributed pseudo-random input, let us say t, and let us say the program a produces output z which is supposed to be this $\sqrt{t}$. So, for each such t what do we do? We take z and we take z^2. So, if z is actually the $\sqrt{t}$ then z square should be close enough to t. In fact, it should be within this likely permissible range $2x10^{-4}$. If z^2 fails to be within this range of t then the program has an error so that must be fixed and then this test can be repeated. So, this is how random testing happens. So, to random testing may or may not be effective all the time. Later in the course we will see one very effective way of over coming random testing called symbolic testing.

So, this modules goal was to give you an overview of the various functional testing techniques from next class onwards, I will focus only on equivalence class partitioning or called input space partitioning. We will see how to define partitions of a set, how to design input space partitions for various kinds of programs, and what are the coverage criteria based on these partitions.

Thank you.