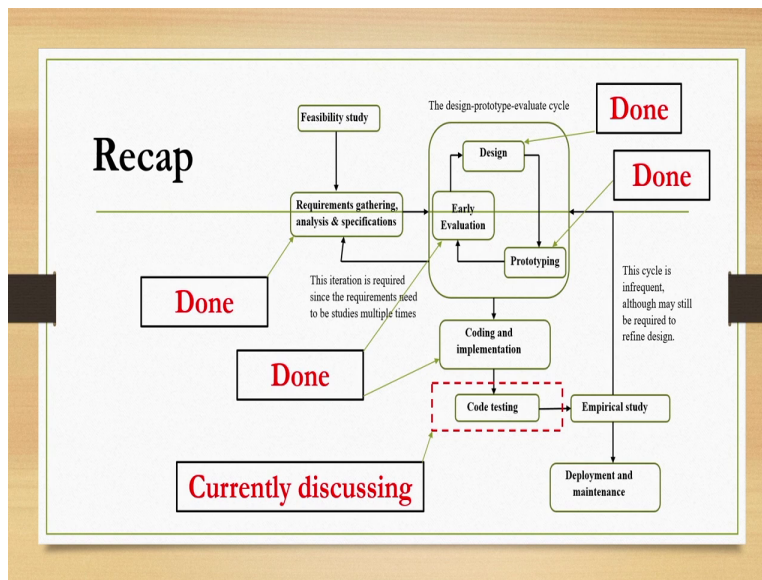


Design and Implementation of Human – Computer Interfaces
Prof. Dr. Samit Bhattacharya
Department of Computer Science & Engineering
Indian Institute of Technology, Guwahati

Lecture: 31
Black-Box Testing I

Hello and welcome to the NPTEL MOOCS course on design and implementation of human computer interfaces lecture number 27 where we will continue our discussion on code testing. Before we go into the subject matter of this lecture let us quickly recap what we have learned and what we are going to discuss.

(Refer Slide Time: 01:03)



So, if you may recollect we are discussing the interactive system development life cycle. This life cycle is a stage wise development of interactive systems which are of interest to us essentially when we talk about human computer interfaces. We are actually referring to interactive systems and it helps if we follow some systematic stage wise development process to design and develop such systems.

One such stage wise development process we are discussing here that is the interactive system development life cycle which consists of several stages. As you can see here the life cycle is consisting of requirement gathering stage design prototype evaluation stage coding and implementation stage code testing stage empirical study stage and deployment and maintenance

stage. Among them we have already learned about requirement gathering analysis and specification stage.

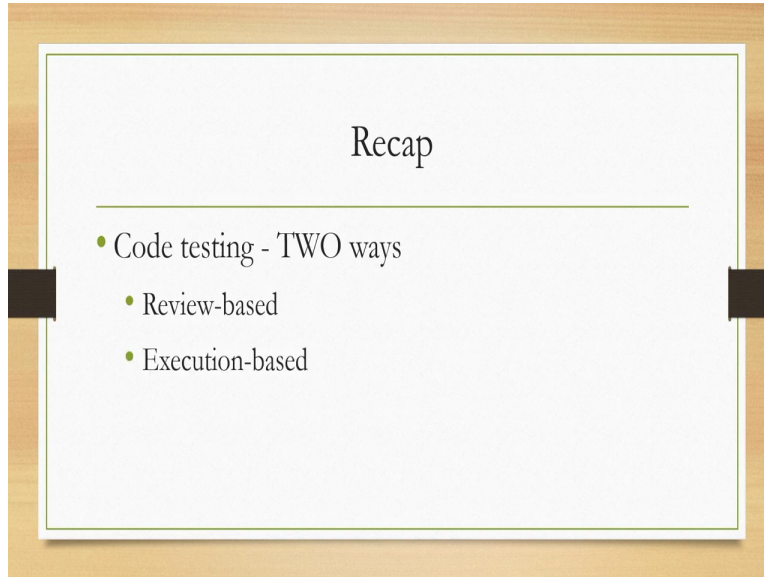
We also learned design stage prototyping stage evaluation stage. In fact if you may recollect this design prototype evaluate cycle actually refers to stage wise development of interfaces keeping in mind usability issues. Design also refers to design of the code we have also learned code design as well code or system design. We also have learned coding and implementation stage where we learned about good coding practices and do's and do not's while going for writing codes for implementing a system.

Currently we are discussing the code testing stage also at this stage it is helpful to remember that the outcome of each of these stages is a detailed document. For example for requirement gathering analysis stage at the end of this stage we produce an SRS document. We have seen such document formats and some case studies on how to create such a document design prototype evaluate cycle also leads to documents.

So, we have seen case studies on such documents design documents at the end of this stage design document we have also seen design document for system design. So, both for interface and interaction design as well as system design documents we have discussed. At the end of coding stage we get the codes which are the documentation part as output of this stage.

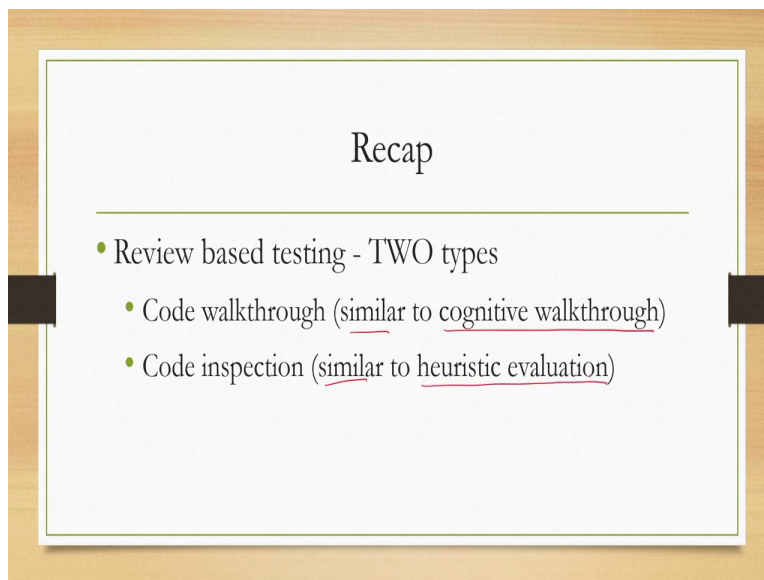
After code testing also we produce testing reports which are the documentation parts we have already seen such reports for review-based code testing and the other code testing methods that we are going to learn in subsequent lectures will also produce such testing documents as we shall see in subsequent discussions. So, at the end of each stage we should remember that we are supposed to produce some documentation.

(Refer Slide Time: 04:46)



Now in code testing what we have learned earlier broadly there are two types of code testing one is review based and one is execution based. So, we have discussed in details the review based code testing.

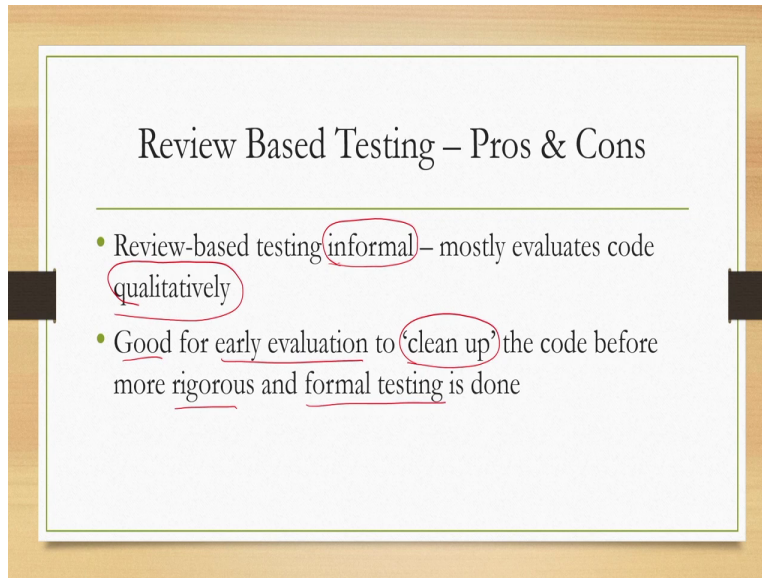
(Refer Slide Time: 05:03)



In review based code testing we have seen that there are two types code walkthrough and code inspection. We have also seen how code walkthrough is similar to the cognitive walkthrough method that we have learned earlier in the context of prototype evaluation. Similarly code inspection is similar to the heuristic evaluation method that we have learned earlier in the context

of prototype evaluation. One issue with the review based code testing is that which we have mentioned earlier as well.

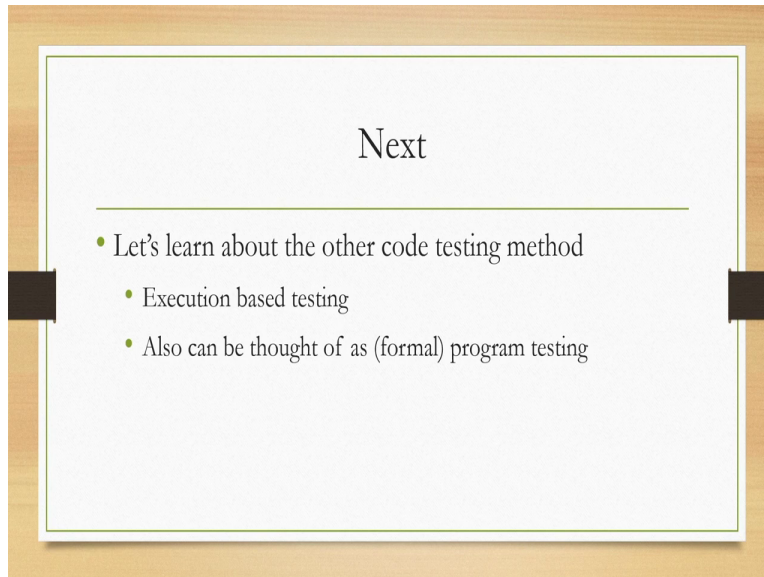
(Refer Slide Time: 05:35)



Such testing methods are informal meaning they mostly evaluates code such methods mostly evaluate code qualitatively. So, that is one major issue with any review based code testing if that is the case then why we go for it why we do not go for more quantitative approach because quantitative measurements are supposedly to be better than pure qualitative measurements. Now such methods are typically good for early evaluation when the code is built but when we want to quickly check whether there are some issues with the code.

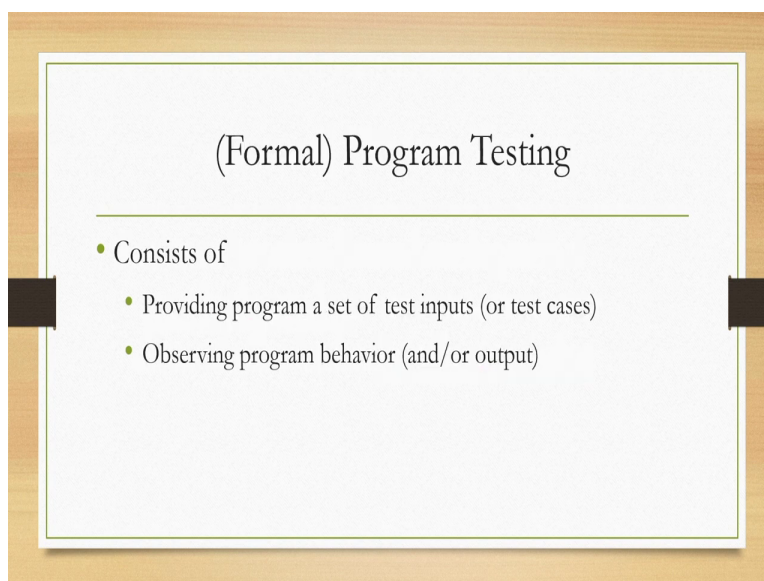
So, this early evaluation stage is where we apply this review based code testing methods to clean up the code before more rigorous and formal testing is done. So, one thing to be noted here is that we do not rely entirely on review-based code testing rather we use review-based code testing to clean up code before such codes are subjected to more rigorous and formal testing methods why that is done because review based testing's provide quick way of detecting errors.

(Refer Slide Time: 07:05)



So, apart from review based code testing we have execution based code testing. Now execution based testing is more rigorous and formal testing of the program that you are writing. So, we can think of this execution based testing as formal program testing methods.

(Refer Slide Time: 07:28)

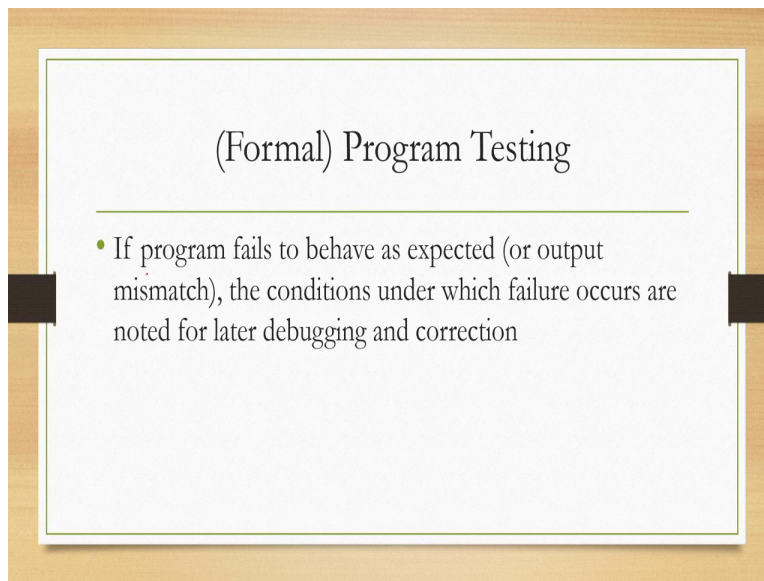


So, when we are talking about any formal testing method what are its characteristics. So, any formal program testing method consists of providing the program which is being tested a set of test inputs or test cases and observing program behaviour along with or the output. So, in review based testing we do not really execute the code with a computer in code walkthrough method that is done manually.

In inspection based method there is no execution at all instead we check for qualitative characteristics of the code such as whether they follow coding standards whether some common errors have occurred just by looking at the code rather than actually executing it. In contrast in execution based testing we need actual execution of the code. Now that execution takes place with respect to some inputs.

And during execution the code behaviour is also checked along with the output or only the output is checked.

(Refer Slide Time: 08:47)



If during execution the program fails to behave as expected or there is a mismatch of output then the conditions under which such failure occurs are noted for later debugging and correction. So, essentially here we provide some inputs and check execution. If we notice some anomaly there in terms of execution or in terms of output then we note down the input which serve as the condition test condition and then we make corrections in the code to take care of such anomalous conditions.

(Refer Slide Time: 09:33)

Terminology

- Test case – a triplet $[I, S, O]$
 - I = data input
 - S = system state at input time
 - O = expected output
- For simplicity, we will consider only the doublet $[I, O]$

Just to quickly recap while trying to formally test a code we make use of some terminologies which we have mentioned earlier. Two important terminologies are test cases and test suits a test case is by definition a triplet where I indicates data input, S indicates system state at the time of input and O indicates expected output. So, it is a triplet of three terms I, S and O. However for simplicity in this and subsequent lectures we will assume that test case represents only a doublet I and O and we will not consider S in our discussion.

(Refer Slide Time: 10:24)

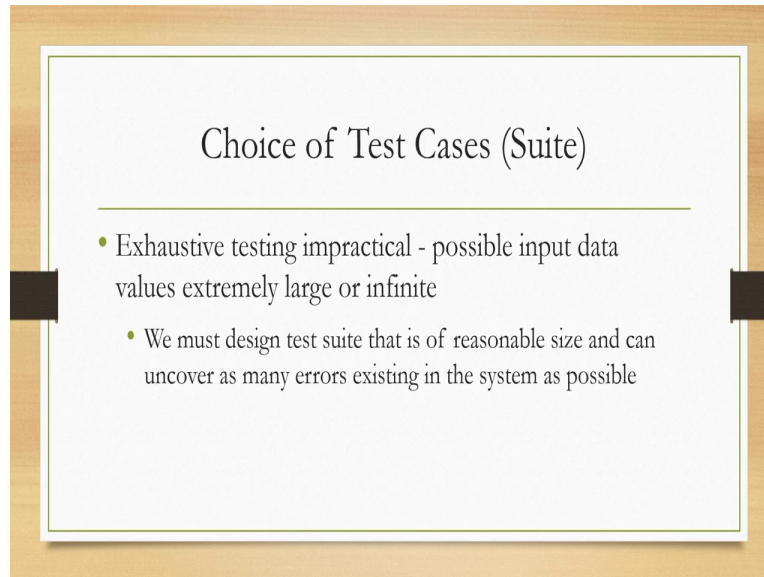
Terminology

- Test suite - set of all test cases with which a given software is to be tested

The other term that we should remember is test suit that is set of all test cases with which a given software is to be tested. Now note that we are going to use the terms program code and software

interchangeably in this context of this lecture and subsequent lectures in fact in the context of this course these terms will be used interchangeably and synonymously. So, they will refer to the same concept.

(Refer Slide Time: 10:57)



So, in formal program testing or execution based testing then what we need to do is we have to have a test suit consisting of test cases which will be provided as input we will consider the test suit as the input conditions will check program behavior and the output and then if we detect some problem with these inputs then we will note down that inputs and take corrective actions. Now the question is how to decide on this test suit or rather how to decide the test cases.

Generally a program or a practical real life program takes as input data or information from a very large domain. So, typically what we find is that possible input data to a program in real world real life are generally extremely large or infinite in size. We must design test suit that is of reasonable size and can uncover as many errors existing in the system as possible. So, input data values can come from a very large domain.

And our objective for formal testing is to decide on a test suit which is of reasonable size and which can uncover as many errors as possible. Now these two conflicting conditions one is large input size another one is uncovering as many errors as possible leads to some practical issues.

(Refer Slide Time: 12:57)

Choice of Test Cases (Suite)

- Randomly selected test cases not necessarily contribute to significance of test suite
 - Need not detect additional defects not already detected by other test cases

First of all if we randomly select the test cases and form our test suite will that be good. Randomly selected test cases are not necessarily good they not necessarily contribute to significance of test suit that means randomly selected test cases need not detect additional defects not already detected by other test cases. So, we already have a test suit and we want to add some test cases in the suit.

So, we random decided on some test cases and added that added those cases in the test suit and used those cases as input data to uncover some more errors but that need not necessarily lead to uncovering of more errors. Whatever errors we have detected so, far with the earlier available test suit need not be enhanced with randomly decided test cases.

(Refer Slide Time: 14:10)

Choice of Test Cases (Suite)

- Number of test cases not indication of **testing effectiveness**
- Large number of test cases selected at random does not guarantee all (or even most) of the errors will be uncovered

Also number is not a very significant factor here number of test cases is not an indication of testing effectiveness that means if we go for a large number of test cases which are selected at random does not guarantee that all or even most of the errors will be uncovered. So, two things we have to keep in mind one is we did not decide on test cases randomly and secondly we should not go for large test suits where most of the test cases are decided at random both these approaches need not lead to uncovering of most or significant errors in the program.

We have already seen in earlier lectures some examples which actually tells us that these randomness in test cases or randomly decided numbers are not necessarily good approaches.

(Refer Slide Time: 15:22)

Choice of Test Cases (Suite)

- Implication - test suite should be carefully designed (not decided randomly)
- Require systematic approaches

So, these two things imply that our test suit should be carefully designed not decided randomly. So, we should do something to design the test suit. This in turn implies that test should be designed following a systematic approach. So, we should take records to systematic approaches to design our test suits rather than going for random choice of test cases.

(Refer Slide Time: 15:54)

(Systematic) Code Testing

- Broadly of TWO types

Now there are broadly two types of systematic approaches using which we can decide on the test cases.

(Refer Slide Time: 16:02)

(Systematic) Code Testing

- Functional testing - test cases designed using only functional specification of software, i.e. without any knowledge of internal structure [**Black-box testing**]

One is functional testing in this case what we do is we design test cases using only functional specification of the software that means without any knowledge of the internal structure of the software. So, we visualize the software as consisting of a set of functions each function is treated to be a black box we do not know how the function the output or how the function gets its work done. So, we have no knowledge of the internal structure we are only bothered about the input and output of the function. Such an approach is called a black box testing approach or a functional testing approach.

(Refer Slide Time: 16:56)

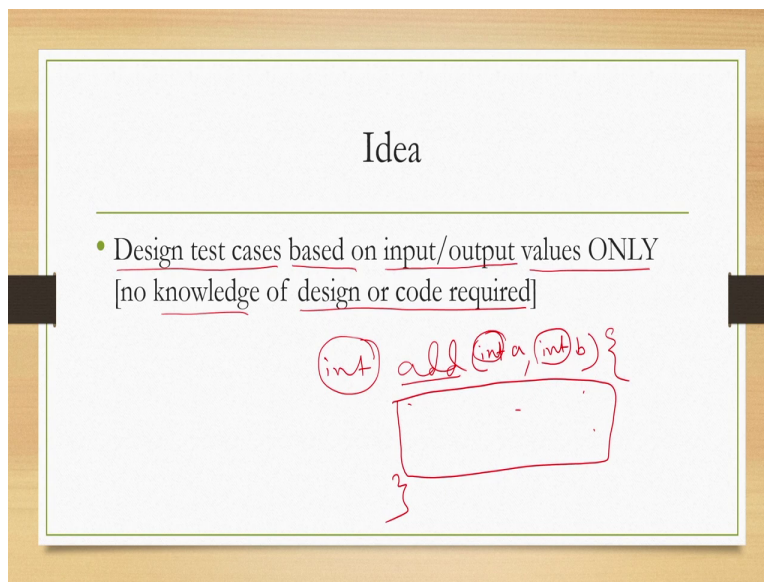
Black-Box Testing

The other systematic method for deciding on test cases is the structural testing method. This is just the opposite of functional testing in functional testing we are not aware of the internal working of the functions or the code whereas in structural testing we are fully aware of the internal structure of the software that means how the codes get executed and test cases are designed using this knowledge of the internal structure.

This approach the structural testing approach is also known as white box testing approach. So, in this course we are going to learn about these two methods for formal program testing where the way we approach the problem decides the design of test cases. In black box testing we approach the problem as testing a software which consists of functions but we do not know the internal working of the function we only know the input to the functions and the output of the functions and based on that knowledge we decide the test cases.

In the other approach the structural testing approach. We are fully aware of the internal working of the software the functions and based on that knowledge we decide the test cases.

(Refer Slide Time: 18:21)



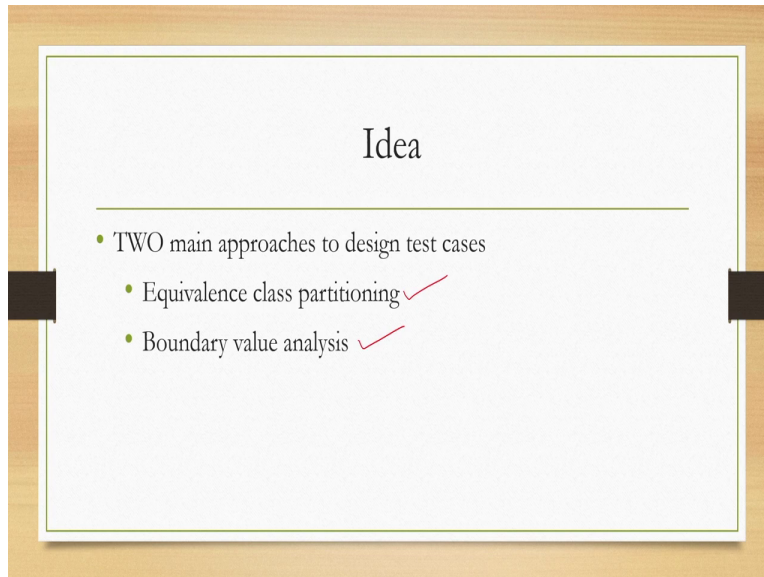
Idea

- Design test cases based on input/output values ONLY
[no knowledge of design or code required]

```
int add(int a, int b) {  
    .  
}
```

Let us first learn about the black box testing approach.

(Refer Slide Time: 18:26)



So, in black box testing we go for designing of test cases based on input and or output values only. So, we do not assume any knowledge of the design or code. So, suppose I am trying to test this function add int a int b let me write it properly int a, suppose we want to test this function if we are going for a black box testing we do not know how this add function works how it produces the output.

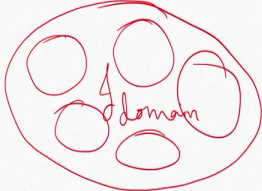
So, this part the internal structure of the function is unknown to us it is as good as it does not exist. Instead what we know is that there is a function which takes two integer values as input into a and in b and produces some integer values only this much knowledge we have and based on this knowledge we go for testing it. So, we do not know how the internal things happen when we design test cases based on this approach this is a this is called functional approach or the black box testing approach.

Now when we know of only the function input and output rather than the internal structure and try to design the test cases. So, we can approach this problem in either of the two ways. Two main approaches are there.

(Refer Slide Time: 21:00)

Equivalence Class & Partitioning

- Domain of input values partitioned into sets – each called 'equivalence class'

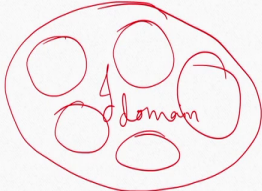


Equivalence class partitioning and boundary value analysis in fact they are not either or instead what we do is while designing test cases we approach the problem as a problem of equivalence class partitioning as well as a problem of boundary value analysis both are there it is not either or.

(Refer Slide Time: 21:24)

Equivalence Class & Partitioning

- Domain of input values partitioned into sets – each called 'equivalence class'

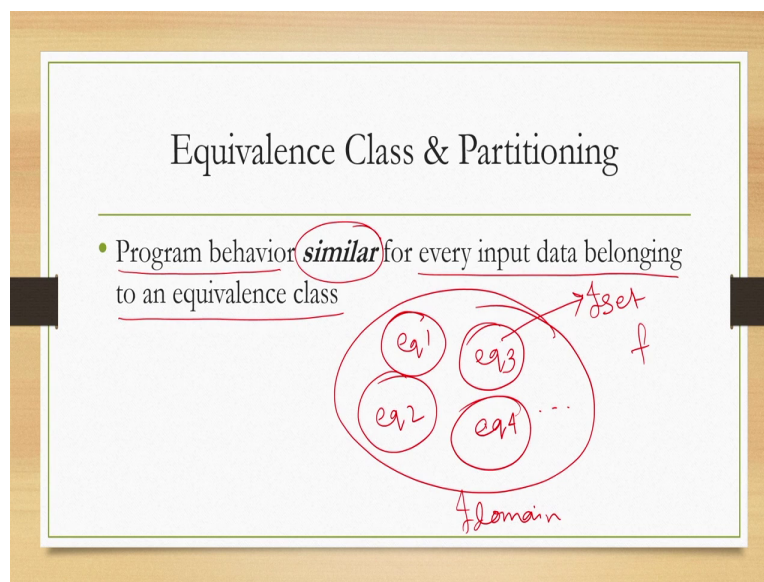


Let us first briefly try to learn about equivalence class partitioning concept. So, when we talk of equivalence class partitioning the first concept that comes to our mind is the term equivalence class and then comes partitioning concept. So, first concept that we should be our office equivalence class and second concept is partitioning. So, what is equivalence class? Remember

that when we talked about function input we said that the input value can come from typically a large domain of values.

Now suppose we represent it as a set. So, this is the input domain set this set we can partition into subsets like this generally non overlapping subsets those subsets are equivalence classes. So, the idea is that we have a domain of input or input domain which is a set and we divide it into or partition it into subsets each subset is called an equivalence class and generally these subsets are non overlapping.

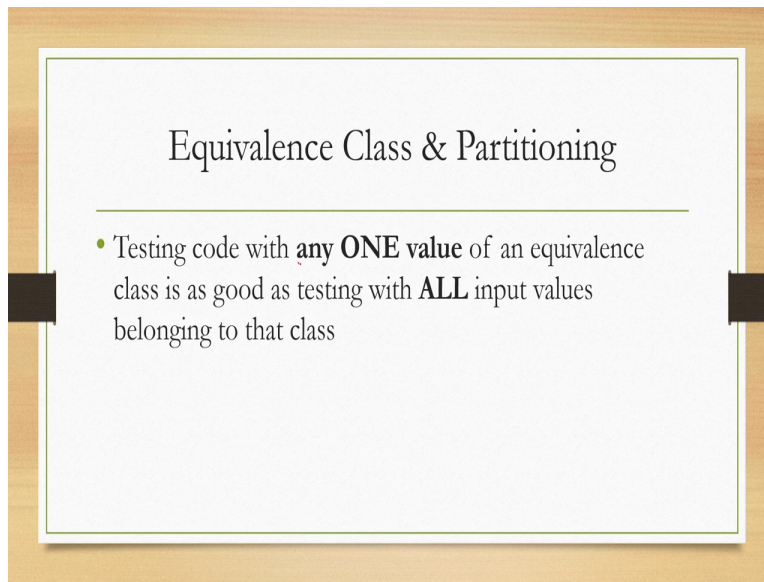
(Refer Slide Time: 22:58)



Now what is the purpose of dividing the input domain into subsets or equivalence classes. We do that assuming that the behaviour of the program is similar for every input data belonging to an equivalence class that means suppose we have this equivalence class partitioning that means input domain is divided into equivalence class 1 2 3 4 and such subsets what we are assuming here or what is the motivation behind this partitioning is that in each equivalence class we get a set of input values input set.

For each input set the function behaves in the same way if the function is denoted by f then behaviour of f for input set is same for each element in this set. So, the program behaviour is similar for each element belonging to an equivalence class. Now what is the significance of that particular statement that is program behaviour is same for each element in an equivalence class.

(Refer Slide Time: 24:38)



Equivalence Class & Partitioning

- Testing code with **any ONE value** of an equivalence class is as good as testing with **ALL** input values belonging to that class

It means if we test our code with any one value of equivalence class of an equivalence class then that is as good as testing with all input values belonging to that class. Remember earlier we said that generally for real world programs the input domain size is very large number of possible inputs is very large it may be infinite as well. So, it is practically impossible to test for each and every possible input.

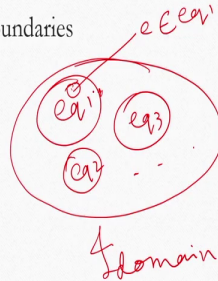
So, then what we can do is we can carefully select a finite number of input test conditions as our test cases and then test the code which is expected to reveal all our most major problems with the code equivalence class partitioning gives us one such method. Here by carefully analyzing the input set and partitioning it into equivalence classes what we can do is from each equivalence class we can choose only one input value and the matching output value as our test case.

This is because by definition equivalence class means the program behaves in the same way for all input values belonging to a to an equivalence class that means if we test the program with one element that is as good as testing the program with all the elements in the equivalence class. So, in that way we can keep the size of our test suit finite that is the objective of partitioning input domain into equivalence classes.

(Refer Slide Time: 26:41)

Boundary Value Analysis

- Check at the class boundaries



The other thing that we should consider is boundary value analysis this is closely related to the partitioning approach. So, input domain is partitioned into equivalence classes such as equivalence class one two and so on and so forth. Now while choosing an element say one element belonging to this equivalence class one sometimes we fail to recognize that at the boundary of the classes. Now each class is a set.

So, the set has boundary values at the boundary of the classes or the corresponding sets the behaviour may change. So, while we are choosing an element from an equivalence class to represent the whole class along with that the boundary values are also to be taken into consideration for those equivalence classes because the behavior may be different at the boundary values of such classes.

So, we have to take care of equivalence class partitioning and we have to take care of the boundary values for those equivalence classes while deciding on our test cases that is the important thing that we should keep in mind while going for black box testing.

(Refer Slide Time: 28:20)

Note

- In the next lecture, we shall learn more on black-box testing with examples

In the next lecture we are going to learn these concepts in more details in terms of several examples which will make it further clearer to you what are the equivalence classes how we can partition and how we can use the boundary value analysis to come up with appropriate set of test cases or the appropriate test suit which is likely to uncover most of the coding errors that we are interested in.

(Refer Slide Time: 28:53)

Reference

- Rajib Mall (2018). **Fundamentals of Software Engineering**, 5th ed, PHI Learning Pvt Ltd. **Chapter 10**
- Roger S Pressman (2020). **Software Engineering: A Practitioner's Approach**, 9th ed, McGraw-Hill Education, New York, **Chapter 19-21**

Whatever we are discussing in this lecture can be found in these books fundamentals of software engineering chapter 10 as well as software engineering practitioners approach chapters 19 to 21. Of course these content these chapters contain the concepts in more details you may go through

these chapters to learn those concepts. I hope you understood the material that we discussed today and you enjoyed it looking forward to meet you in the next lecture, thank you and goodbye.