

Lecture – 15
Mutation Testing

Welcome to this session. So far we had discussed about black box testing techniques and several white box testing techniques and in the last session, we were discussing about the mutation testing which is a fault based testing technique.

(Refer Slide Time: 00:58)

Mutation Testing

- Example primitive changes to a program:
 - Deleting a statement
 - Altering an arithmetic operator,
 - Changing the value of a constant,
 - Changing a data type, etc.

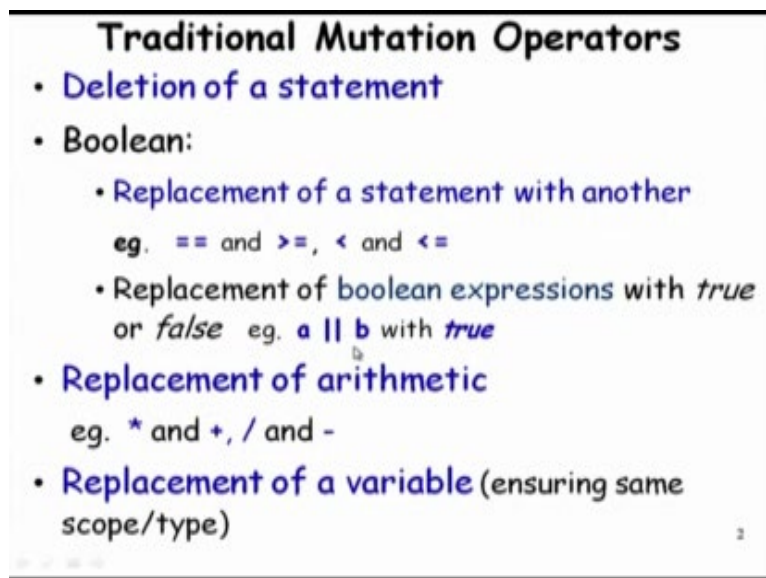
We had said that in mutation testing we have to first identify various fault categories that typically occur in a program and then, we generate mutated programs, where we introduce a specific type of fault that we want to check whether our test cases are doing a good job of that and then, we run the test cases and check if the error that we had introduced is caught by the test cases. If it is caught we say that the mutant is dead and our test cases are good, but if the mutant is live, that means all test cases that we had initially designed pass without catching this bug we () introduced.

Then, we say that the mutant is live and we need to add additional test cases to our test suite to kill the mutant. So, we strengthen our test cases. Now, let us see what are the typical types of mutants that we introduce into the program. So, example of mutation operators are deleting a statement. So, when we delete a specific statement, we get a mutant. So, we try to check because these are typical

program errors committed by a programmer. Possibly he just was wanting to write some statement he did not.

So, we introduce various types of simple programming bugs that a programmer commits while writing a program and some examples are deleting a statement, altering an arithmetic operator because these also a typical programming error where the programmer by mistake wrote a different operator, then he intended possibly he just interchanged a plus with minus or something like that, changing the value of a constant, changing a data type, changing a relational operator, changing the bound of a relational operator and so on.

(Refer Slide Time: 03:25)



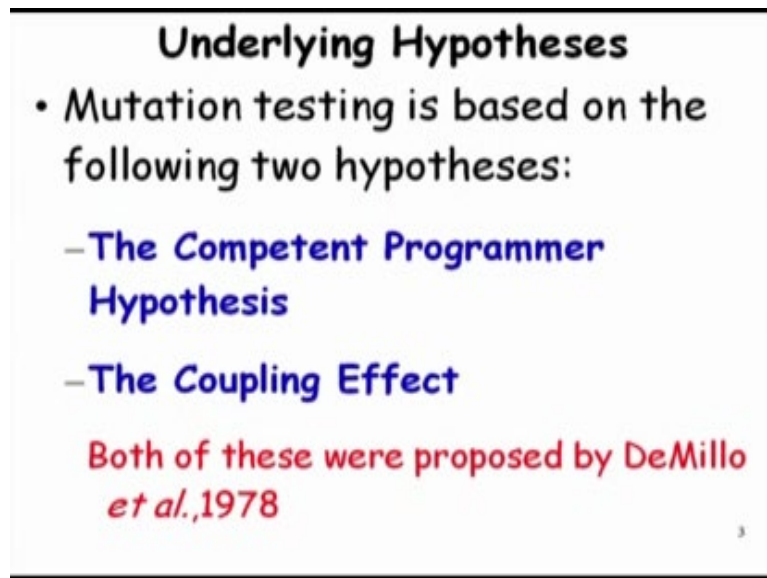
Traditional Mutation Operators

- **Deletion of a statement**
- **Boolean:**
 - Replacement of a statement with another
eg. `==` and `>=`, `<` and `<=`
 - Replacement of boolean expressions with *true* or *false* eg. `a || b` with *true*
- **Replacement of arithmetic**
eg. `*` and `+`, `/` and `-`
- **Replacement of a variable** (ensuring same scope/type)

2

So, replacement of a statement with another statement for example equal to operator with greater than equal to or less than equal to replacement of a expression with a true. So, that expression turns true always. Replacement of arithmetic operators, replacement of a variable, this is also a common programming error where a programmer instead of writing `a = b + c` possibly wrote `a = d + c`. So, all these different types of mutation operators are carried out which generate a large number of mutants.

(Refer Slide Time: 04:30)



Underlying Hypotheses

- Mutation testing is based on the following two hypotheses:
 - The Competent Programmer Hypothesis
 - The Coupling Effect

Both of these were proposed by DeMillo *et al.*, 1978

3


Now, let see why the mutation testing works. There are actually two hypothesis which leads to the success of the mutation testing. One is called as competent programmer hypothesis and the second is called as the coupling effect. The competent programmer hypothesis says that programmers they write almost correct programs. They do not write junk statements, they right meaningful programs excepting that one or two places, they commit small errors. So, that is one of the assumption and the second assumption or the second hypothesis is that when we introduce several simple errors that can behave like a complex error introduced by the programmer, a set of simple errors is equal to some complex error.

So, both these where proposed by DeMillo, 1978 and these two are the underlying theory behind mutation testing. Why mutation testing works are based on these two assumptions. the competent programmer hypothesis. The programmer writes almost correct programs. They do not write junk programs which are riddled with hundreds and thousands of bugs and some statements are totally meaningless. They do not write programs like this. They write the right programs which are almost correct except in for some minor problems and the second assumption is that even if a programmer introduced a complex bug that is equivalent to a set of simple bugs.

(Refer Slide Time: 06:49)

The Competent Programmer Hypothesis

- **Programmers create programs that are close to being correct:**
 - Differ from the correct program by some simple errors.




The competent programmer hypothesis as we said that the programs are close to correct and they differ from the correct program by some simple errors.

(Refer Slide Time: 07:04)

The Coupling Effect

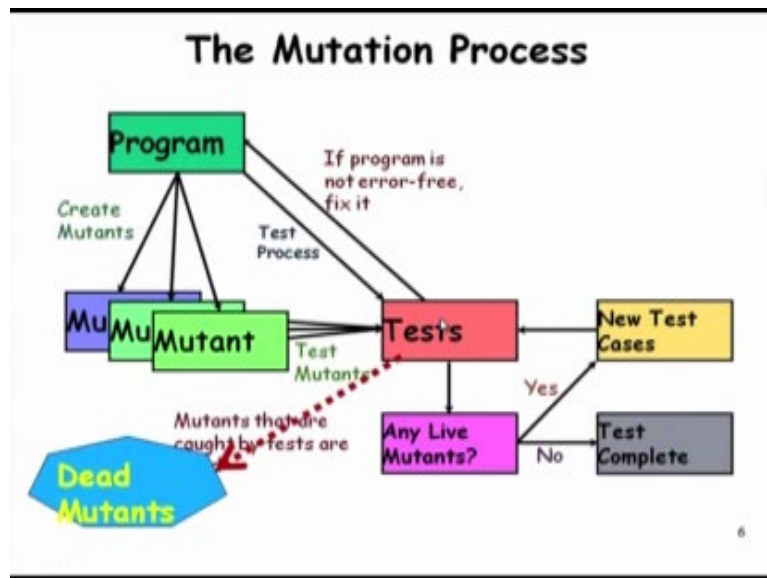
- **Complex errors are caused due to several simple errors.**
- It therefore suffices to check for the presence of the simple errors



The coupling effect on the other hand as he said that all complex errors are caused due to several simple errors and therefore, if we check for simple errors, that should be enough to check for the complex errors as well we need not have to introduce very complex error into the program. Our simple errors which we introduce would be good enough to catch even the complex errors that a

program programmer can commit.

(Refer Slide Time: 07:40)



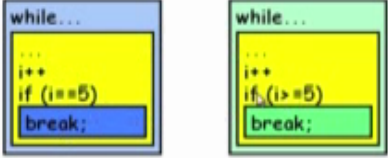
Now, this diagram shows the mutation testing process. So, we have the program and then, based on that we design some initial test cases based on coverage base testing and then, we run the test cases and then, if the test cases find some bugs, we fix to the program and now the test cases satisfy our targeted coverage criterion may be MCDC, may be independent or basis path coverage or both and then, we start the mutation testing process. So, generate large number of mutants and then, using our test cases, we test the mutants and if some test cases fail for a mutated program, then we say that the mutant is dead, our test cases that we designed are good enough. Then, our test cases if they pass, then we say that the mutant is alive and our test cases were not good enough to catch that. We need to augment the test cases.

So, there are live mutants. We say that there is problem with our test cases. They are not good enough. We create additional test cases and that is the way it goes on until we have generated a large number of mutants. Basically, the initial test cases that we had designed, we strengthen them by checking whether the initial test cases are good for various categories of faults and so, mutation testing just helps us to strengthen the test cases and ensure higher reliability of the programs than the coverage based test techniques and finally, once we have generated large number of mutants, we say that our mutation testing is complete and fortunately generating mutants running the test cases on the mutants can be largely automated.

(Refer Slide Time: 10:22)

Equivalent Mutants

- There may be surviving mutants that **cannot be killed**,
 - These are called **Equivalent Mutants**
- Although syntactically different:
 - These mutants are **indistinguishable** through testing.
- Therefore have to be checked 'by hand'



7

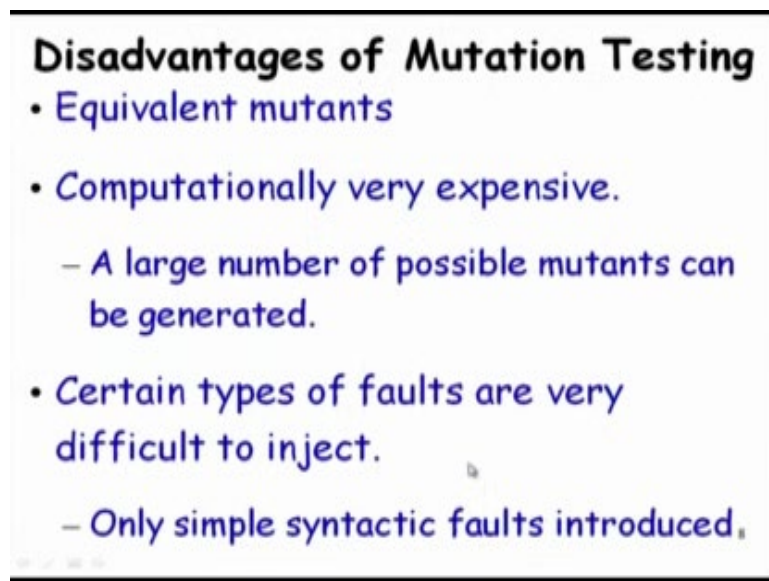
But, so far we looked at only very positive aspects of mutation testing. There are some problems with mutation testing. We did not mention the problem as follows. One problem is called as equivalent mutants. The problem here is that we generated a bug, we created a mutant, may be we replaced the value of a constant or may be we interchanged two statements, but then the interchange statement is not really a bug because there is no dependency between the statements. May be just to be an example or may be we replaced a equal to operator with a less than operator, but still may be there wont be any bug because less than also would work there.

So, these are called as equivalent mutants. When we make a change to a program, but that does not really introduce a bug into the program and then, we have a problem because we will assume after running the test cases because obviously there are no bugs and the test cases will not detect any problem. All test cases will pass and we will say that the mutant is alive and we may try without meeting any success writing large number of test cases trying to kill this mutant, but actually it is not a bug. Even though it is mutant, it is an equivalent mutant, the program in a mutated form is also equivalent to the original program which is the correct program does not introduce any bugs and therefore, even though it is syntactically bit different, but it is not buggy. It is not a program with having bugs and therefore, the mutated program and the original program will be indistinguishable by testing.

So, when we have equivalent mutant, our testing process, automated testing process will run into problem. It will flag that the mutant is live and when need to argument the test cases, but then

whenever they research a flag, we need to check manually whether we are getting a equivalent mutant and we do not worry. We will just ignore this alert raised by our mutation testing process that the test cases are insufficient for this specific type of bug. This is just one example in this case. If i equal to 5 breaks and you changed here equal to greater than 5, greater than equal to 5. Now, they are actually same because the test case that passes here, we will also pass here and we say that these two are equivalent.

(Refer Slide Time: 14:07)



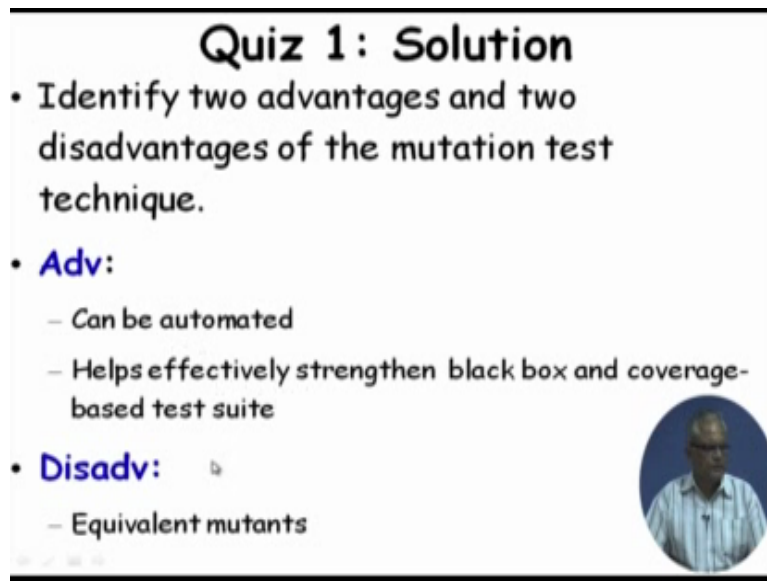
Disadvantages of Mutation Testing

- Equivalent mutants
- Computationally very expensive.
 - A large number of possible mutants can be generated.
- Certain types of faults are very difficult to inject.
 - Only simple syntactic faults introduced.

So, the problems with the mutation testing, one big problem is equivalent mutant. The second problem is that for a non-trivial program, we can generate billions or trillions mutants and then, even though the process can be automated, still it can take huge number of, huge time and also many of the errors flagged may be due to equivalent mutants and also, we can inject only simple syntactic faults and certain type of faults. For example, algorithmic faults and so on. We cannot really introduce these and for those types of faults, let say algorithm faults and so on, the coupling effect may not really hold.

So, this is effective. The mutation testing is very effective on simple programming bugs which the programmer committed simple errors while writing the program, but more complex types of bugs like algorithmic bugs performance bugs and so on mutation testing may not be that effective.

(Refer Slide Time: 15:42)



Quiz 1: Solution

- Identify two advantages and two disadvantages of the mutation test technique.
- **Adv:**
 - Can be automated
 - Helps effectively strengthen black box and coverage-based test suite
- **Disadv:**
 - Equivalent mutants

So, let us just ask you one question. What is the advantage of mutation testing compared to other testing techniques and what is one important disadvantage? The answer can be that advantage is that it is highly amenable to automation and it is a technique which can strengthen our test suite. Once we have completed testing, we can do mutation testing to strengthen our test cases and improve the reliability of the program and disadvantage we saw that one very prominent disadvantage is equivalent mutants. So, this is the solution that written here can be automated effectively strengthen the black box and coverage based test suite and disadvantage is equivalent mutants.

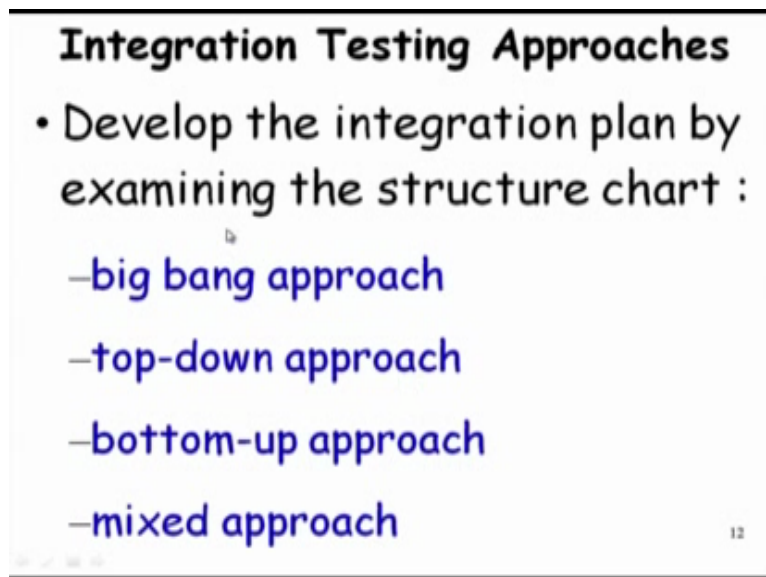
Now, let us look at integration testing. So far we have been looking at unit testing, both black box and white box testing where we test a program unit. A program unit is typically a function. It can be a module also. Now, assuming that we have tested our functions modules, now let see how do we carryout integration testing and at the start of our discussion, we had said that why we need to do integration testing, why do unit testing and then do integration testing and then, do the system testing. We had said that these help us to effectively debug any problems in unit testing. We have very limited place to look for the bug during debugging process and therefore, the debugging is cost effective. If we had bypassed unit testing, then those bugs which could have been eliminated very cheaply, we would incur tremendous testing cost, debugging cost for eliminating those bugs.

The next thing to do would be the integration testing where we have carried out unit testing on modules and now, we check whether the modules interface properly. So, let me repeat that

statement that during integration testing we check whether the different modules that were unit tested interface properly. We know that by doing unit testing, we have eliminated most of the bugs there. Almost there are no bugs, but then when we put two modules together, bugs may arise in the interface. So, that gives us a hint that the type bugs that will be targeted by the integration testing. So, if we ask that what is the type of bug that is detected during integration testing, these are the interface bugs interfacing between two modules.

What is interfacing? Interfacing is that one function calls another function with a wrong parameter or may be it calls a parameter, it calls a function with less number of parameters or more number of parameters or there is a parameter type mismatch. Suppose to call the function with an integer value, but it is calling with a floating point value or a string was required and it just passes a character and so on. So, these are the examples of interface bugs and these are the target of integration testing.

(Refer Slide Time: 20:20)



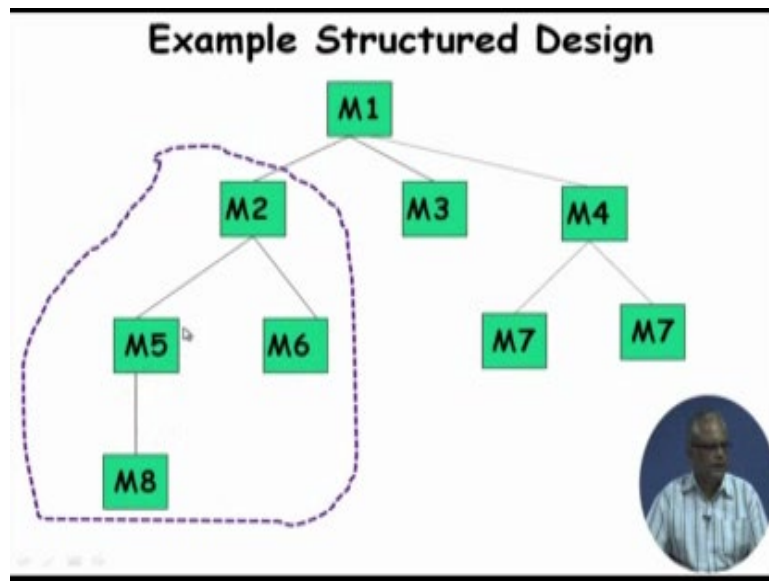
Integration Testing Approaches

- Develop the integration plan by examining the structure chart :
 - big bang approach
 - top-down approach
 - bottom-up approach
 - mixed approach

12

Now, the integration testing is based on how the modules or the units call each other and one example representation of how they call each other is a structure chart and once we have this module structure available to us, the module structure is how the modules call each other. Based on that we can have various types of integration testing, we can have big bang approach, top down approach, bottom up approach and a mixed approach or a sandwich approach. Now, let us look at these approaches.

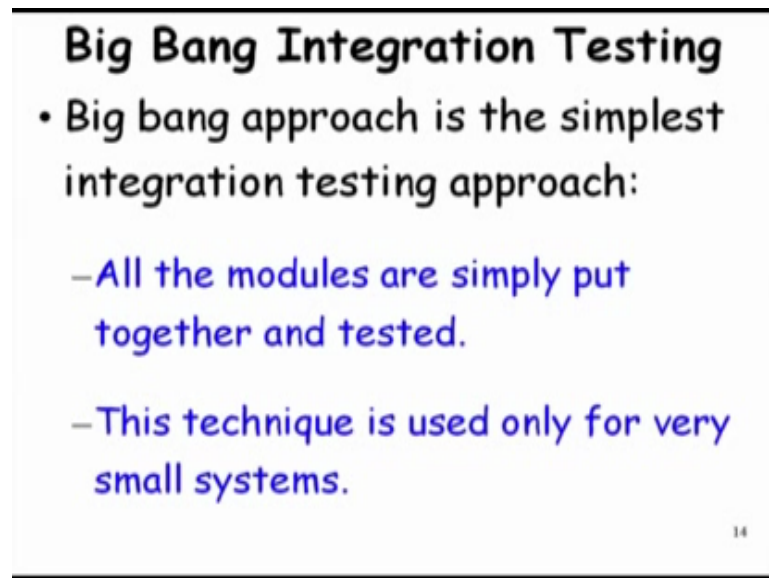
(Refer Slide Time: 21:05)



This is an example of a structure design or an example of how the modules call each other. M1 calls M2, M3 and M4 and M2 calls M5, M6, and M5 calls M8 and M4 calls M7 and M9. This is a good design is a layered design where the modules are arranged in a hierarchy, but we might also have situation where there is a call relation from M8 to M6 or M3 and so on, may not be in a hierarchy for not so well designed program this is an example of a well design program, where the modules call each other in a hierarchy, a top level module calls lower level module and no call from a lower level module to higher level module exists and therefore, its a very good design, but then we could also have other types of design where we have some cases of a lower level module calling higher level module, but let us see what are the different ways you can do the integration testing based on the module calling structure.

One may be that we can decide to integrate together these modules and then, test whether these modules are working correctly, but if we integrate 4 or 5 modules in one step, then again we will have a problem is that if there is a failure of a test case, then we will have to look at many modules. So, if the modules are non-trivial, reasonably large modules supporting many different interfaces, then we do a incremental integration where at a time we integrate only one module.

(Refer Slide Time: 23:49)



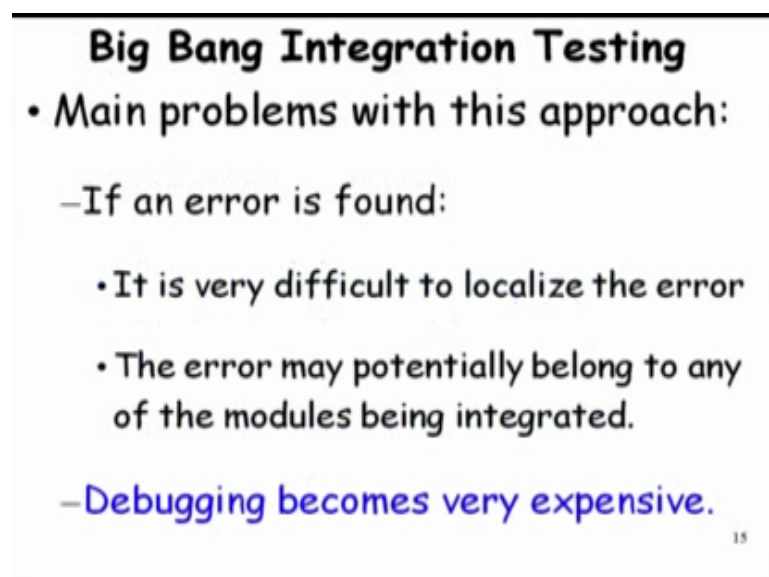
Big Bang Integration Testing

- Big bang approach is the simplest integration testing approach:
 - All the modules are simply put together and tested.
 - This technique is used only for very small systems.

14

First let us look at the big bag integration testing here as the name says that all modules are integrated in one step. So, all the modules are linked, compiled and then, we run the test cases for this, but the problem here is that the big bang testing can work only if we have two or three simple modules. If we have several dozens of modules and we do a big bang testing, then the disadvantage is that our debugging process will be enormous. It will incur very high cost. So, the big bang integration testing where the integration is done in just one step is applicable to only very trivial programs, toy programs where we have just two or three modules, simple modules.

(Refer Slide Time: 25:04)



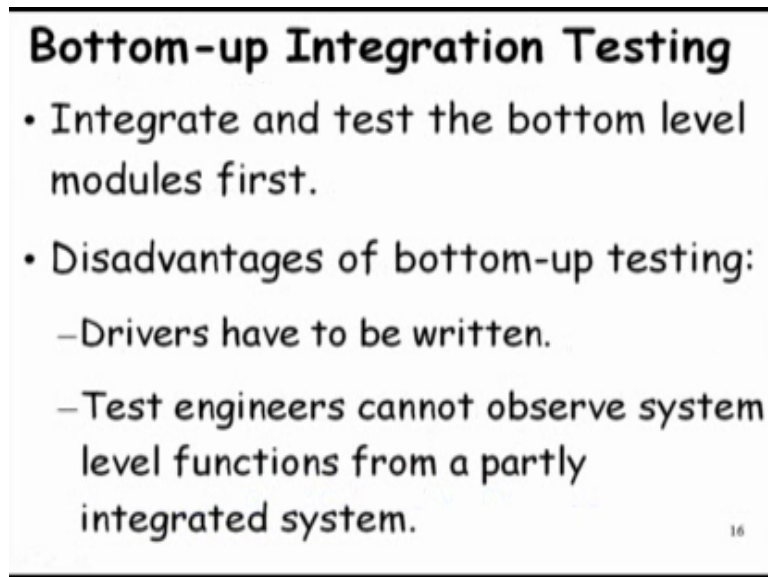
Big Bang Integration Testing

- Main problems with this approach:
 - If an error is found:
 - It is very difficult to localize the error
 - The error may potentially belong to any of the modules being integrated.
 - Debugging becomes very expensive.

15

Now, let us look at the other types of big bang integration, other types of testing. So, we have written here the same thing that we are saying that it becomes very difficult if you are doing a big bang testing. It becomes very difficult to debug or localize the error and therefore, huge amount of effort needs to be spent once a test case fail to find out where exactly is the bug and therefore, the debugging process becomes extremely expensive.

(Refer Slide Time: 25:40)



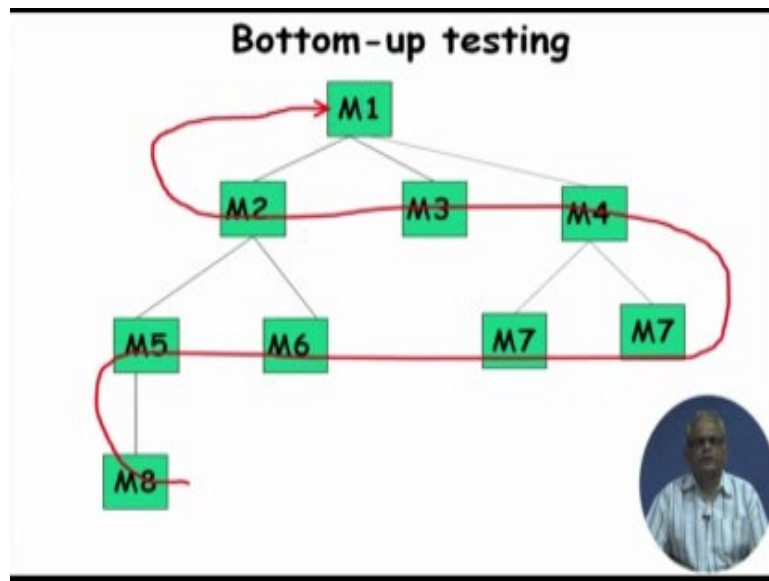
Bottom-up Integration Testing

- Integrate and test the bottom level modules first.
- Disadvantages of bottom-up testing:
 - Drivers have to be written.
 - Test engineers cannot observe system level functions from a partly integrated system.

16

Now, let us look at the bottom up integration technique. In the bottom up integration testing, we have this module structure that is a call structure of all the modules and will start at the bottom level and if we have a well designed program, then the bottom levels are well defined and then, we take the bottom most level and integrate one module with it, another bottom module with it, but then when we do this integration since these are bottom level module, then we should have some software called as drivers written which will actually call these bottom modules. So, one disadvantage here is that the test engineers cannot observe the system level functions that is the just states the bottom level modules, but there may be a huge case and they cannot really run the huge case. They just test some dummy test cases here, very simple test cases or low level programs. Sorry low level modules and as you might already know that the lower level modules are typically the input-output modules which may be a simple GUI which take a user input. It may be a simple file reader which read some data from a file or write some data to a file or write display something on the user interface. So, these are the lower level modules which are at the bottom of the hierarchy. They do only some simple input output and therefore, we have to just check whether those input-output are working. We cannot really run system level test cases.

(Refer Slide Time: 28:14)



Let us see how the bottom up testing will work. So, we take one module here and then, integrate it with another module which is possibly at the same level since there are no modules at the same level with integrated to the higher level module and then, we integrate with another module, this level and then the next module and then, the next module. Then, we integrate the next higher level module and so on until we complete the route level module. So, we just looked at some very basic integration testing techniques. There is more integration testing techniques which we will look in the next session.

Thank you.