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

Lecture - 20
**Testing Object-Oriented Programs (Contd.)**

Welcome to this session, we shall continue our discussion on Testing Object - Oriented Programs. Remember that in the last session, we said that there is lot of expectation in the object oriented paradigm, in the sense that it was expected that the object oriented paradigm will make testing a very simple because the object orientation is found on very well formed principles.

(Refer Slide Time: 00:40)

But we were seeing that the different features introduced in object oriented programs actually make testing either difficult, or they cause ( ) new types of bugs requiring different testing strategies.
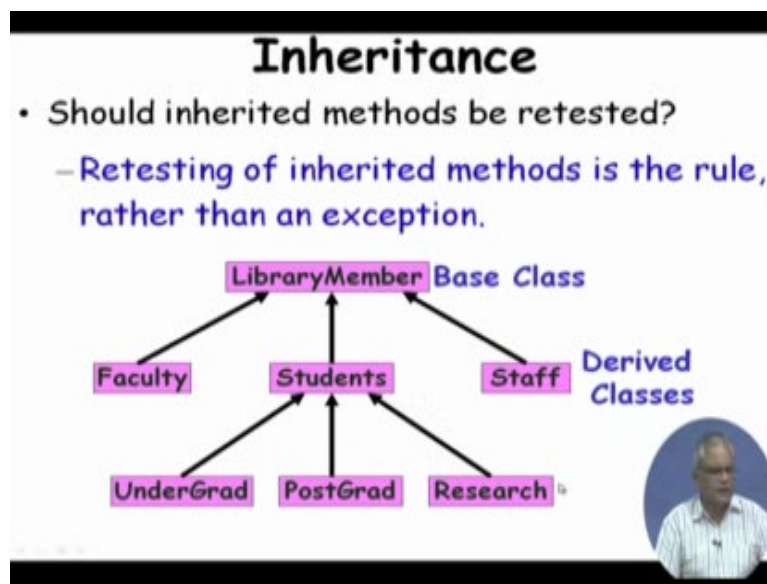
Let us continue from where we are discussing last time. Inheritance is a prominent feature of object oriented paradigm and inheritance helps code reuse. Once code is returned in the base class the code is reused in the derived classes. So, ( ) the question is that if the methods have been tested in the base class. So, those methods once they are inherited in the derived class should they be tested again?
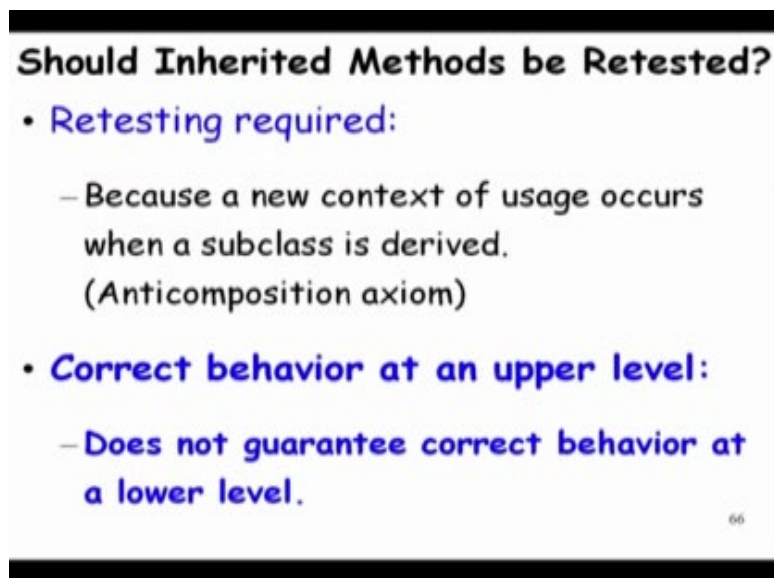
We had seen in the last session that retesting of the inherited methods is the rule rather than exception. So, all the inherited methods even though they worked fine in the base class there is no guarantee that they will not work correctly in the derived class and therefore, the inherited method have to be retested in the derived class.

So, if this is your class hierarchy, there is a base class and then in the first level derived classes new data and methods are introduced and then both of these methods are inherited by the second level derived classes and all the methods that are inherited by these base level classes; leaf level classes all have to be retested.

(Refer Slide Time: 03:13)

If you remember, we had said that retesting is necessary because there is a new context of usage. By the term the new context of use, what we mean is that there can be new class variables introduced in the derived class. There can be new methods in the derived class and that is the new context with which the inherited methods will be used and therefore, they need to be tested and this is what follows from the Weyukar's Anticomposition axiom.

So, the correct behavior of a method at upper level does not guarantee that the method will behave at a lower level class. Let us look at an example, we have a base class A where a class variable x is assigned value to 100 and then we have a method m which manipulates the value of x, but then makes sure that the value is always greater than 100 and for the method m to a work correctly, it

always needs to be 100, but then when it was later extended into a class B, a new method m1 was introduced, but then the programmer here, who extended the class A into class B did not pay attention to the invariant that x would be 100.

So, he assigned x is equal to 1 and therefore, m which work correctly in A once m 1 is called, will fail, the method m will fail in the extended class B. So, the execution of the m 1 causes the A bug in m and therefore, unless we test m here in the extended class B, we would not be able to discover that bug. Here is another example, here we have a class A which was two methods m1 and m2 and m calls m2 methods and class B which extends A overrides the method m2.
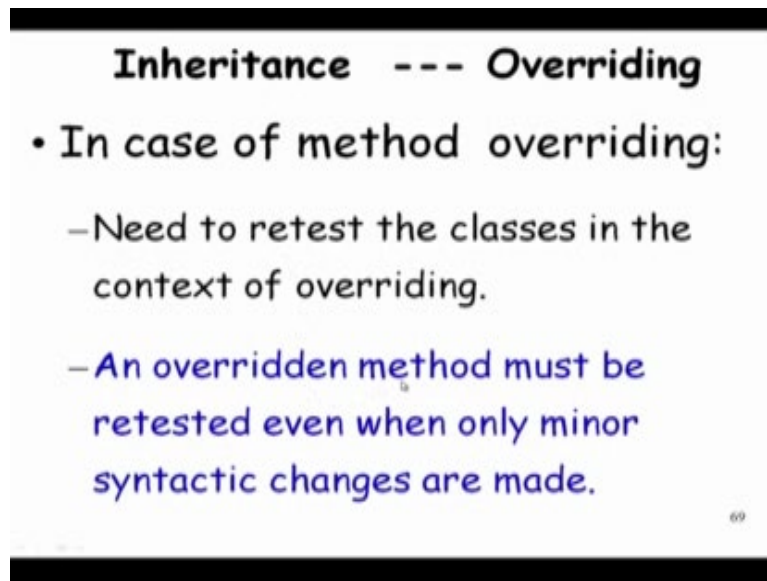
(Refer Slide Time: 05:46)

So, now when m is called in the context of class B due to a dynamic binding, the m 2 of class B will be called. So even though m worked correctly in the context of class A the same m when inherited in class B and m is calling m 2, but m 2 will bind to the m 2 in the class B and therefore, m can fail in class B. So, we can construct many other examples which, so that the method which works correctly in a base class fails to work correctly in the derived class.

I request you to construct few examples yourself, where a method works correctly in the base class, but then when extended derived and the method fails, so that will give you an understanding of how, why retesting of the methods is required. Please try and of course, if there is an overridden method in a derived class then obviously, the overridden method has to be tested.
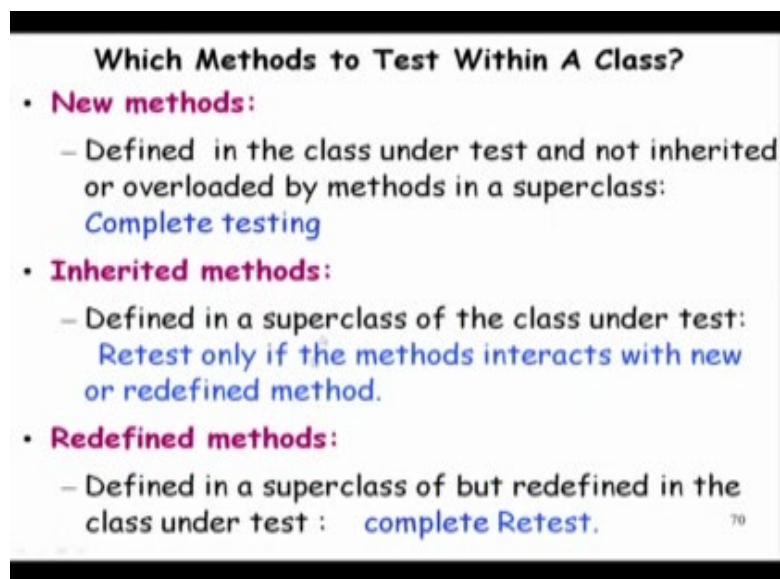
Even though there may be on small change introduced by the derived and overridden method. So, if we look at it in a derived class, we need to test all the new methods obviously, because these are will be retested first time and all the inherited methods have to be tested because these inherited methods will be used in the context of the derived class.

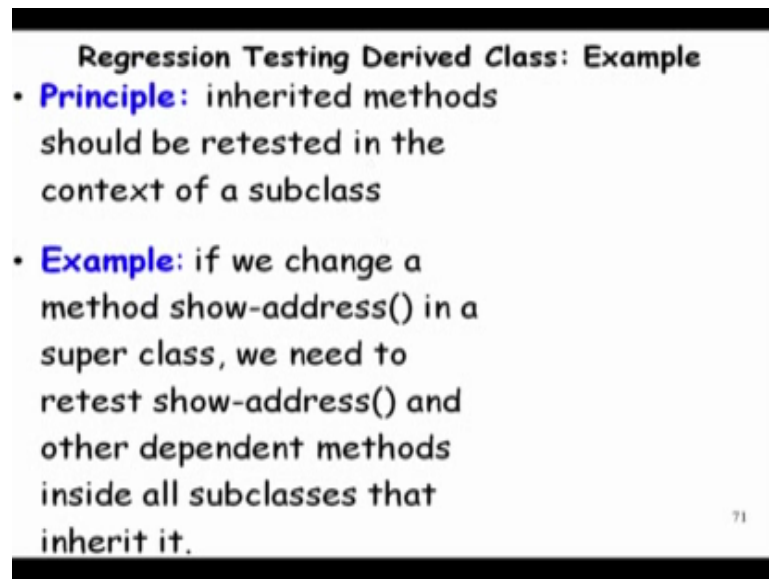And also all the overridden methods, the redefined methods have to be tested. So, what really it means that all the methods of a derived class have to be tested. So, object orientation, it is for a testing is concerned does not give any benefit in reduction of the testing effort as far as inheritance is concerned. Now, what about regression testing?

(Refer Slide Time: 08:27)

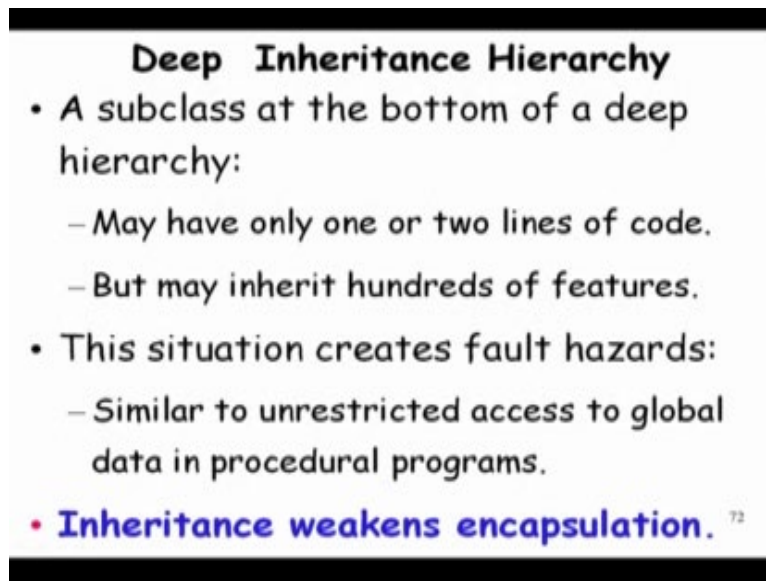
Regression Testing Derived Class: Example
- **Principle:** inherited methods should be retested in the context of a subclass

- **Example:** if we change a method show-address() in a super class, we need to retest show-address() and other dependent methods inside all subclasses that inherit it.

If we in a class hierarchy, if we make a small change to a method in a top level class, what do you have to do in the other classes, can we just test the method in the top level class and just leave it there or do we have to test it in all the class hierarchy the same method in every class in the class hierarchy? So, unfortunately if we make a small change let say show-address() is a method in the base class which is inherited in all these classes.

Now, let us say after I have developed an application, we make a small change to this method in the base class. So, do we test it well in the base class and leave it there? No, we have to test the same method in every class in the hierarchy.

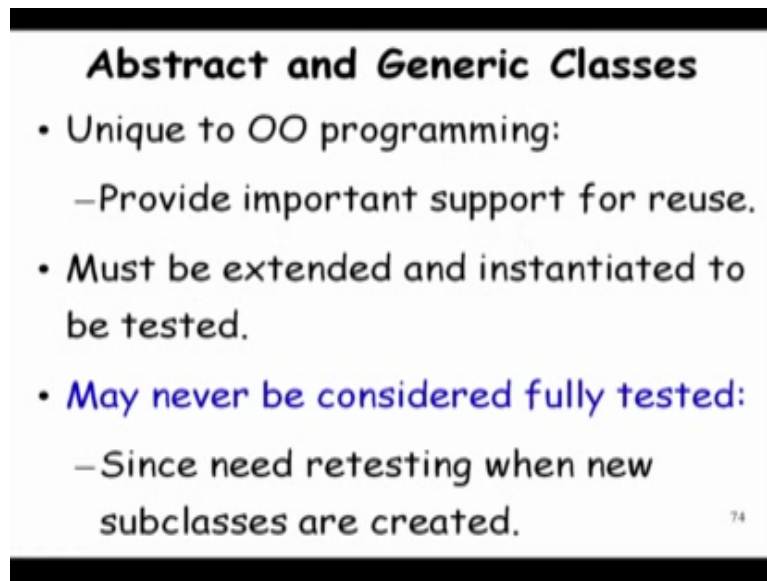If we have a deep class hierarchy we will have lot of testing to do. We might in a derived class, we might just add one method or two methods or few data variables, but all those inherited methods have to be tested again and also we have more chances of faults occurring here because too many methods interacting with each other accessing a global variables, it is something like a procedural program where we have global variables and all methods are inter interacting through them.

In case of deep hierarchy the encapsulation is weakened. So, if we have a deep class hierarchy what may be the solution, one thing is that a deep class hierarchy may introduce bugs, result in low reliability of the software, reduce testability because too many test cares testing has too much of testing has to be done and also incorrect initialization and forgotten methods may result. So, one possibility is that we may flatten the class hierarchy, but then that also has its own problem. So, a deep class hierarchy is not good and also if we have multiple inheritances as in a language such as C++, then we might have too many contexts to test.

Now, what about abstract and generic classes? The abstract classes provided in object oriented programs to help increase reuse of code, but then we can extend abstract class in many ways and therefore, an abstract class may never be considered as fully tested. So, we do not, since abstract class cannot be directly tested, it cannot be instantiated therefore, we test it only through its derived classes and each time we have a derived class we have a case for testing it.

Now, what about polymorphism? This is another prominent feature of object oriented programming. So, we know that polymorphism is ( ) present in object oriented programming either a static binding or is a dynamic binding. So, for each case separate testing is needed. For at least statistic binding,
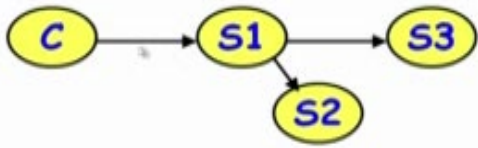
(Refer Slide Time: 13:05)

we know that, what are the methods to which it can be bound? But, in case of dynamic binding there may be many classes method, in many classes where it can be bound dynamically and therefore, finding all the bindings may be difficult because classes may also get extended by different programmers.

(Refer Slide Time: 13:41)

Just look at this example, where the class c has a method which gets a mount to method of S1, S2 and S3. So, we have difficulty in integration testing because unless we have S2, S3 and S1 tested and integrated. So, we cannot test C. So, the server classes which provides service to the class C, C is the client class unless these have been tested and integrated, we cannot test C. So, the dynamic binding is a cause for concern in testing because we do not know, what are the bindings that may occur? We cannot use just code analysis statistic analysis to find all the dependencies that may occur in a program just to give an example.

(Refer Slide Time: 14:47)
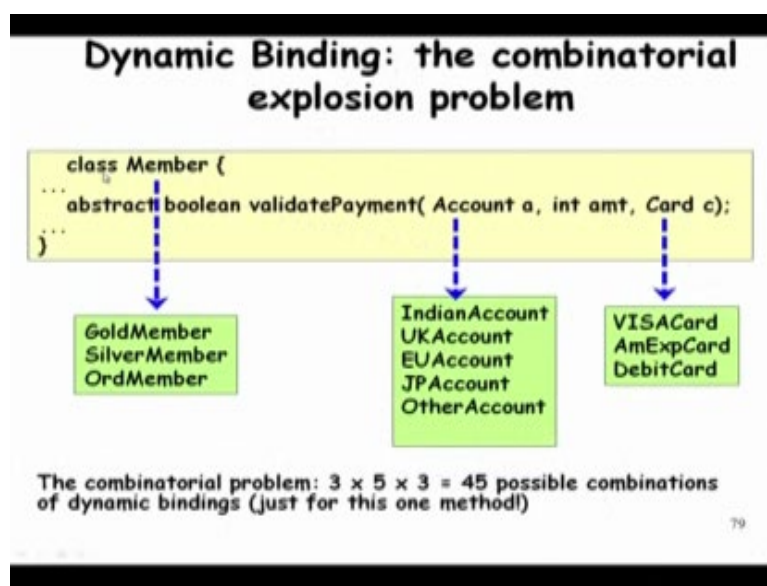
Let say, we have a class member and then we have a method here, let say Boolean validate payment. So, the member might have an Indian Account, UK Account, European Union Account or Japanese Account or any other account and we might pay it through a VISA Card, American Express Card or a Debit Card.

Once the method validate payment is ( ) called, first the member since it can be bound to the derived members Gold members, Silver member and Ordinary member. So, first is the binding that may occur with respect to the member. Now, when we invoke the function member dot validate payment. The validate payment also the account in the card can get bound to any of these. There are 3 here and there are 5 here and then there are 3 here. So, we have 45 possible combination to test and we might have more parameters here and it can the number of test cases can blow up. One possibility is that we might try the pair wise testing.
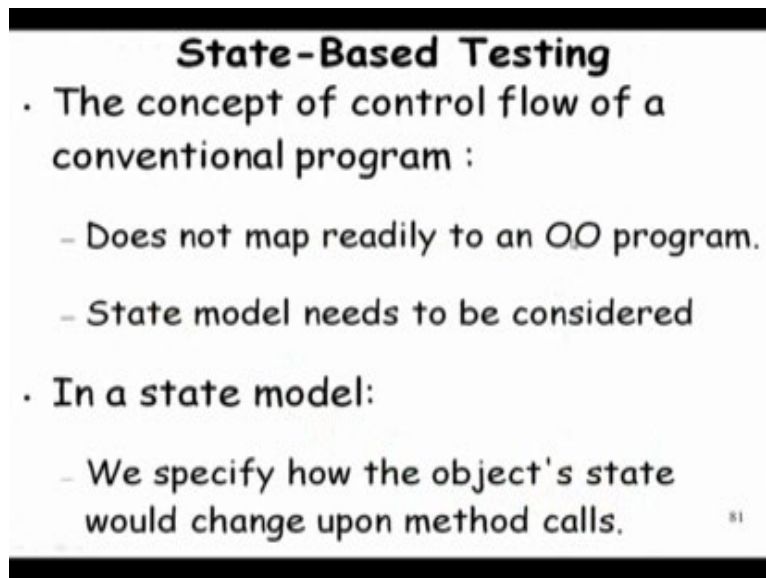
(Refer Slide Time: 16:16)
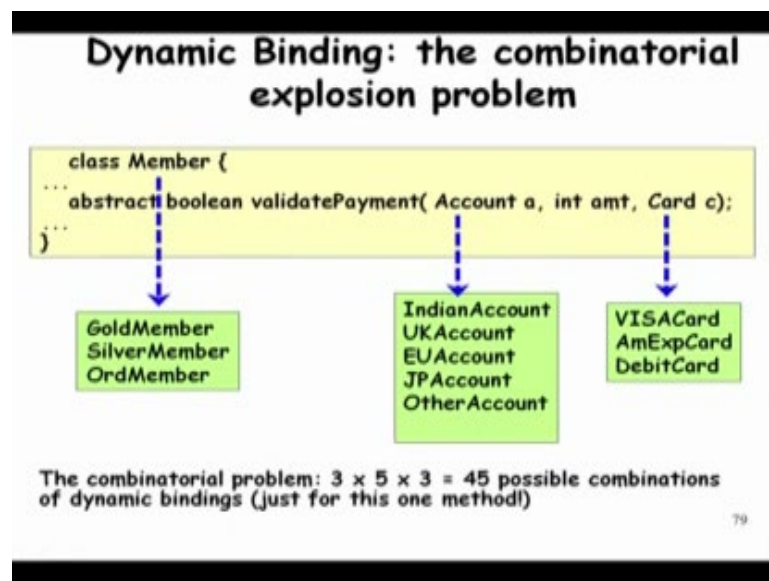
So, if we draw the diagram here for pair-wise testing. So, if we have a class hierarchy like this and m, these objects can be placed in place of m. So, they can be bound, the method can be bound to any of these. So, how do we derive the pair wise test cases?

So, please think about it, we had discussed about pair-wise testing some time back and we can use the same context, I mean same concept here and derive the pair-wise test cases. The other complicacy that arises is that unlike functions which do not have attributes permanently storing values, the class attributes store values permanently and therefore, we can consider a class to be having states.

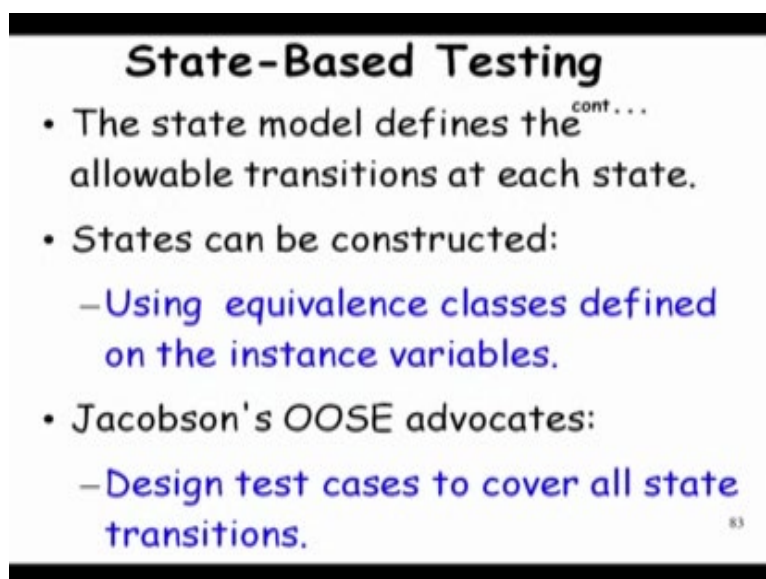A class can be represented; the state behavior of a class can be represented with a state model in a procedural program. We could test it by method calling or a function calling another function and all possible calls among the functions, we could test their integration, but here first of all, a class is a unit of testing and when we try to test a method invoking other methods, we must also consider that just testing a method all possible ways another method can be invoked is not enough. We have to also test it in different states of the class, the states of the class is given by a state model and which specifies how the object state changes upon method calls.

Whenever a method is called for an object, some attribute may change and therefore, the object can change its state and therefore, we have to first construct the state model of a class and the state model can be thus different states can be constructed by using equivalence class is defined on the instance variables.

So, what we really mean that the instance variables of an object they actually define the states and let us say, an instance variable can assume values 1, 2, 3 and in that case we might say that there are three states when the instance variable value is 1 or 2 or 3. So, we can define equivalence classes on the instance variables that is, those values for which the class remains in the same state they are equivalent.

So, we can define equivalence classes and that will help us determine the states of the class and once we have the state model Jacobson's object oriented software engineering advocates that we have to have test cases to cover all state transitions and not only that in each state all the methods have to be tested. So, it is not just enough to test the methods once, we have all the states covered, design test cases to cover all state transitions and then in each state we need to test the methods.

(Refer Slide Time: 20:43)
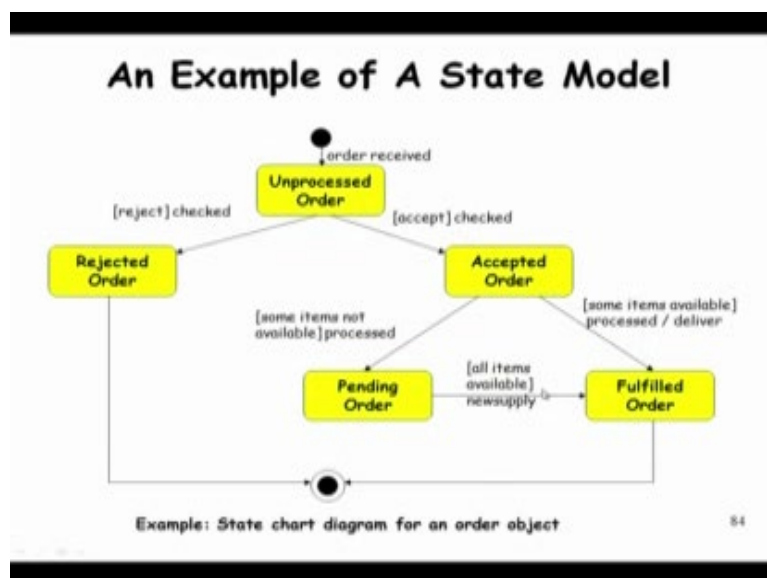
So, this is an example of a state model and we need to have the object assume different states through the different state transitions and then we need to do full testing of the methods in each of the states, but then if we have multiple classes to be tested each one has its own state.

How do we make each class? Assume different states different possible combinations of states that is, very difficult problem because we are no more invoking methods on one class individually every class, we are invoking a use case.

There are multiple classes we are just invoking method of one class and then the other methods are automatically in invoked. So, we may not be possible, it may not be easy to traverse all possible states in that case. So, the locus of the state control is distributed over the entire object oriented application each class has it is own state model, but we are calling method of one class and therefore, covering all the states of all classes may become difficult.

Difficulty With State Based Integration Testing

- The locus of state control is distributed over an entire OO application.
  - Cooperative control makes it difficult to achieve system state and transition coverage.
- A global state model becomes too complex for practical systems.
  - Rarely constructed by developers.
  - A global state model is needed to show how classes interact.

But whatabout test coverage analysis we had seen, test coverage analysis in the context of procedural programs. We had seen that several instance of test coverage analysis starting with statement coverage, condition coverage, branch coverage, path coverage and so on.
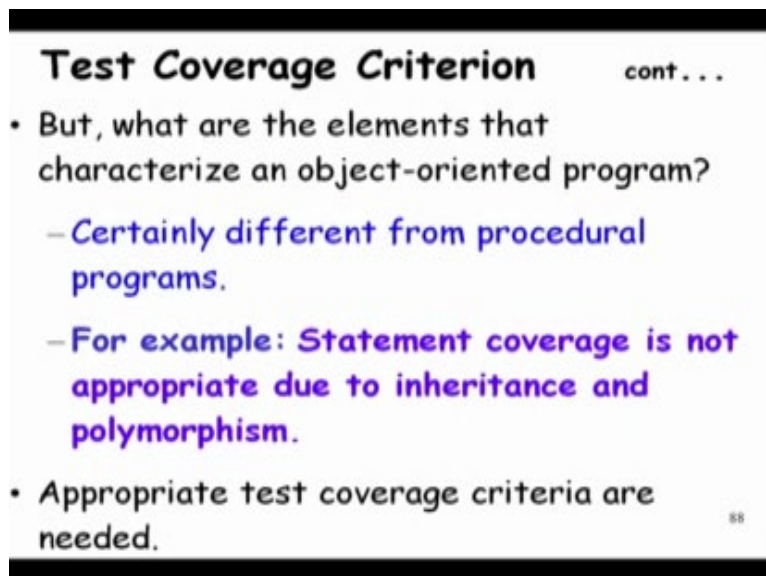
Test Coverage

- Test coverage analysis:
  - Helps determine the "thoroughness" of testing achieved.
- Several coverage analysis criteria for traditional programs have been proposed:
  - What is a coverage criterion?
- Tests that are adequate w.r.t a criterion
  - Cover all elements of the domain determined by that criterion.

But, are those test coverage metrics hold in case of object oriented programs?Unfortunately no. So, it is seen test coverage metric helps us determine the thoroughness of testing how well we have tested by computing those metrics.

So, the test coverage metric is typically defined by how many elements of interest are covered, the element of interest may be statements then we have statement coverage, the element of interest may be a branch and then we have branch coverage and so on. If we think of it obviously, statement coverage is not an appropriate coverage metric for object oriented program because we had already said that even if a statement is covered in the context of the base class already covered once, but then in the context of the derived class.
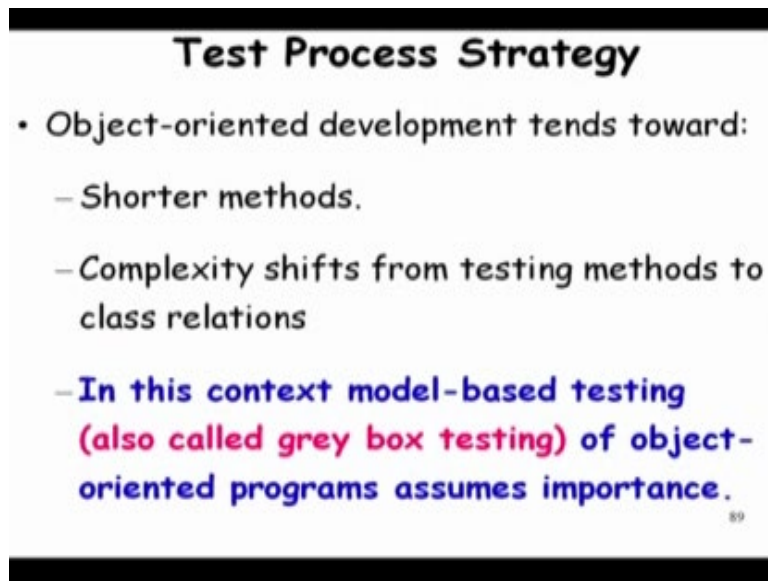
(Refer Slide Time: 23:36)

We have to again test it. So, just saying that the statement has been covered is not enough because of the inheritance hierarchy and inherited methods which need to be retested therefore, just having statement coverage each statement is executed once does not mean much because testing it, in the sense in the context of the base class is no guarantee of the statement working correctly in the derived class. It has to be again executed in the derived class and therefore, simple statement coverage is rather meaningless in the context of ( ) object oriented programs.

So, then what is an appropriate coverage criterion in the context of ( ) object oriented programs? These are still areas of research lot of results are coming out, but then we can make our own inference about what can be a suitable test coverage metric for object oriented programs.

Now, what about the test process strategy? In object oriented programs, the methods tend to be very short because that is the characteristic of object oriented programs, we write very short methods and then we have these inheritance and then method collaboration across different classes and in the same class the methods tend to be short and the complexity shifts from testing methods to class or relations because the methods are short, small methods, bugs are not very likely to be there the bugs are more likely to be there in the class relations association, aggregation, inheritance and so on.

In this context we have to test an object oriented program through its design model because the class relations and so on are represented using a design model and therefore, the traditional code based testing, white box testing is less significant for object oriented programs, what is much more important is the design based testing what is called as a grey box testing is very important for object oriented programs because bugs are much likely to be present in the class relations, which are represented using a design model rather than in the method body itself. What about integration testing? If you remember in a procedural program, we had a design hierarchy, the modules are present hierarchically and based on that we have top down bottom of mixed integration stregies.

But, here unfortunately in an object oriented program the design is not hierarchical. The objects interact in a very arbitrary ways. The integration testing strategies are also different we have thread based integration. So, we integrate classes as required to respond to an event.

A method is called on one class and then it needs to call other class methods. So, we need to integrate them, that is called as thread based testing or used based testing once a use case is executed what are the methods participate for different classes we need to integrate those classes and then the cluster testing. So, different classes which collaborate are integrated first, even the integration strategies here are different from ( ) procedural programs. So, we are running out of time we will stop here.

Thank you.