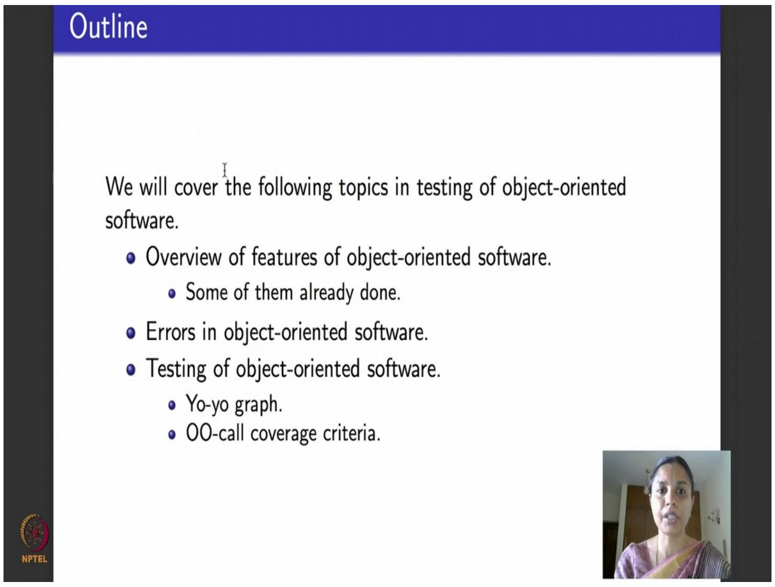
Software testing

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

Lecture - 55
Testing of Object-Oriented Applications

Hello there, welcome to the first lecture of week 12. If you remember in week 10, I had thought you about testing web applications and testing object oriented applications, system level testing of object oriented applications. This was the outline of how I began. We saw overviews of features of object oriented software, some of which were done in the middle of the course.

(Refer Slide Time: 00:24)



Outline

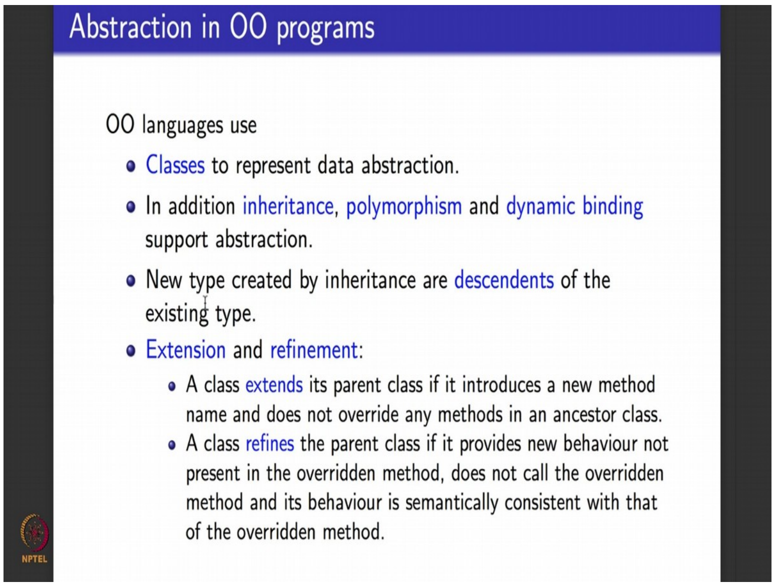
We will cover the following topics in testing of object-oriented software.

- Overview of features of object-oriented software.
  - Some of them already done.
- Errors in object-oriented software.
- Testing of object-oriented software.
  - Yo-yo graph.
  - OO-call coverage criteria.

Then we saw an error overview of all kinds of errors or anomalies is that can occur in a object oriented software. Especially during integration testing of object oriented systems. Then in we also saw what is a yo-yo graph. What I had not done is a part of those lectures was to do object oriented call coverage criteria specific to integration testing. In week 11, I decided to drop object oriented testing without doing this and give you an overview of symbolic and concolic test selection techniques.

Mainly because I wanted that to be a complete set of 5 lectures covering one week's details. So, what I will now do is catch up on what I did in week 10 and finish object oriented testing.

(Refer Slide Time: 01:25)



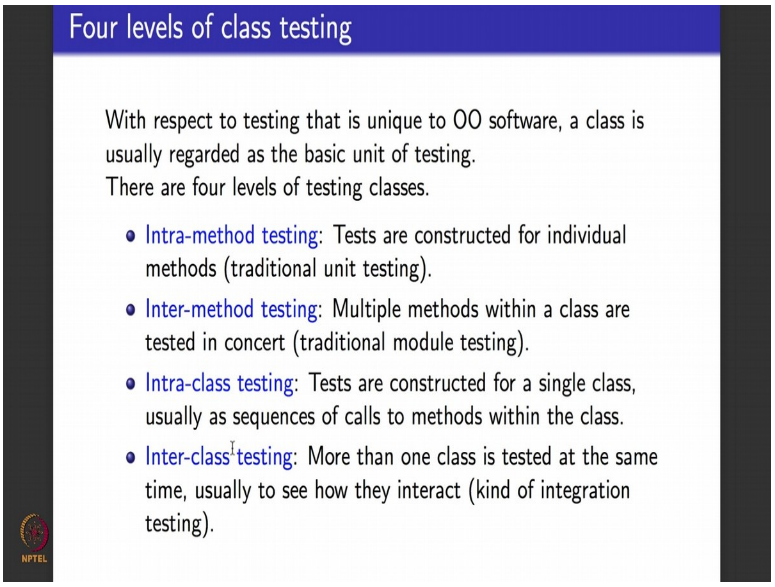
Abstraction in OO programs

OO languages use

- Classes to represent data abstraction.
- In addition inheritance, polymorphism and dynamic binding support abstraction.
- New type created by inheritance are descendents of the existing type.
- Extension and refinement:
  - A class extends its parent class if it introduces a new method name and does not override any methods in an ancestor class.
  - A class refines the parent class if it provides new behaviour not present in the overridden method, does not call the overridden method and its behaviour is semantically consistent with that of the overridden method.

So, to recap, we saw all the features of object oriented software: abstraction, inheritance, polymorphism, 2 kinds of inheritance, what is the polymorphic method, which is the level of object oriented testing that we are going to do, which is basically interclass and intraclass testing.

(Refer Slide Time: 01:31)

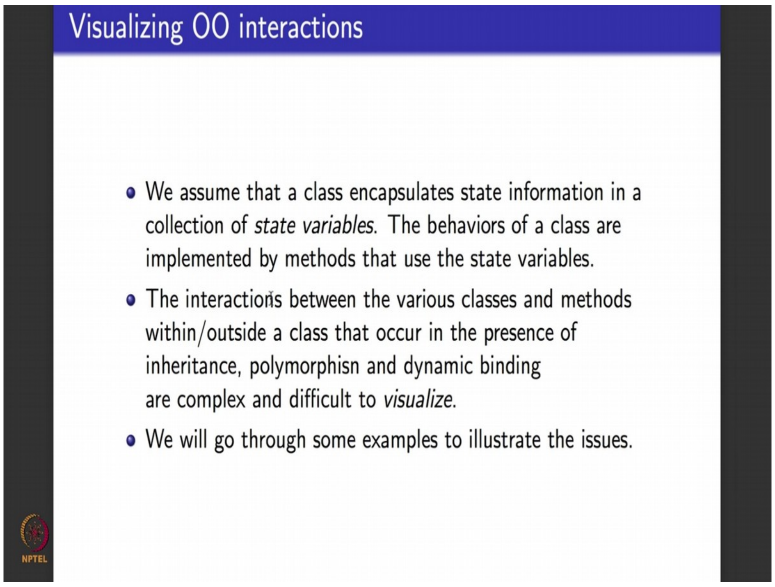


Four levels of class testing

With respect to testing that is unique to OO software, a class is usually regarded as the basic unit of testing.
There are four levels of testing classes.

- Intra-method testing: Tests are constructed for individual methods (traditional unit testing).
- Inter-method testing: Multiple methods within a class are tested in concert (traditional module testing).
- Intra-class testing: Tests are constructed for a single class, usually as sequences of calls to methods within the class.
- Inter-class testing: More than one class is tested at the same time, usually to see how they interact (kind of integration testing).

(Refer Slide Time: 01:38)

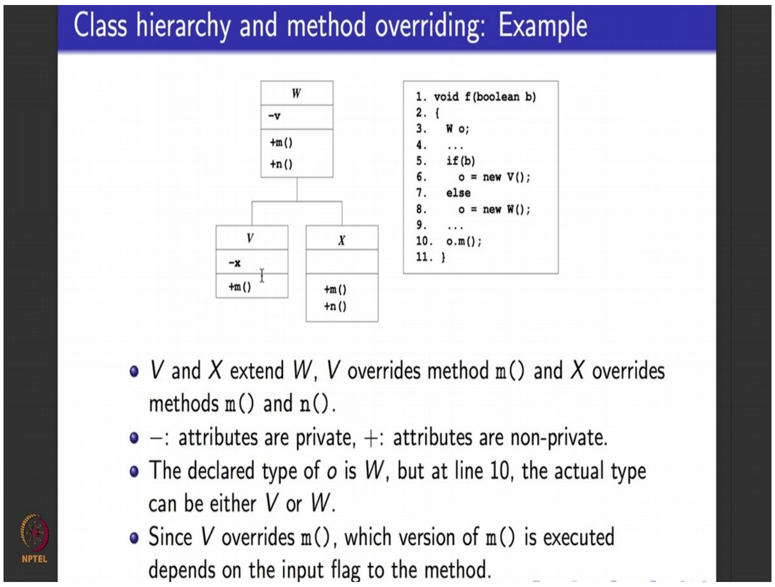


Visualizing OO interactions

- We assume that a class encapsulates state information in a collection of *state variables*. The behaviors of a class are implemented by methods that use the state variables.
- The interactions between the various classes and methods within/outside a class that occur in the presence of inheritance, polymorphisn and dynamic binding are complex and difficult to *visualize*.
- We will go through some examples to illustrate the issues.

And how to difficult, the problem of difficulty in visualising object oriented interactions in the presence of class hierarchies.

(Refer Slide Time: 01:44)



Class hierarchy and method overriding: Example

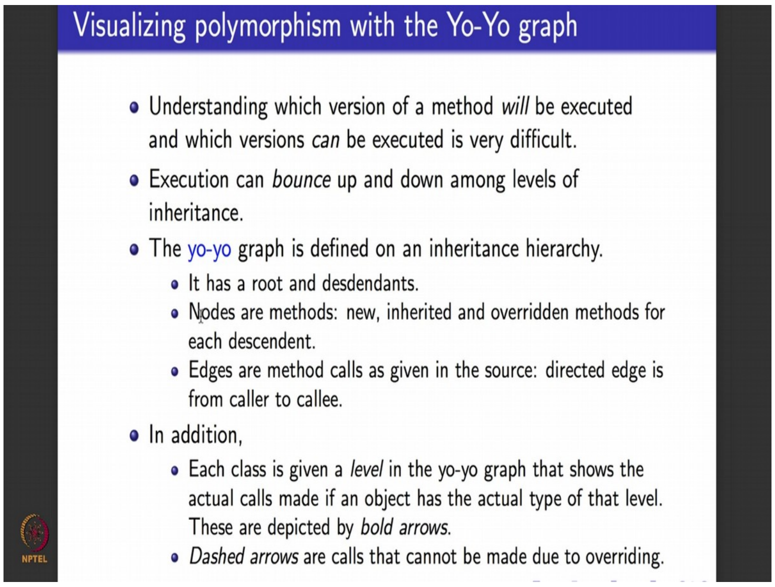
W
-v
+m()
+n()
V
-x
+m()
X
+m()
+n()

```
1. void f(boolean b)
2. {
3.   W o;
4.   ...
5.   if(b)
6.     o = new V();
7.   else
8.     o = new W();
9.   ...
10.  o.m();
11. }
```

- *V* and *X* extend *W*, *V* overrides method `m()` and *X* overrides methods `m()` and `n()`.
- −: attributes are private, +: attributes are non-private.
- The declared type of *o* is *W*, but at line 10, the actual type can be either *V* or *W*.
- Since *V* overrides `m()`, which version of `m()` is executed depends on the input flag to the method.

And overriding methods and polymorphic methods, and we the also saw this yo-yo graph which depicts how object oriented interactions occur.

(Refer Slide Time: 01:53)

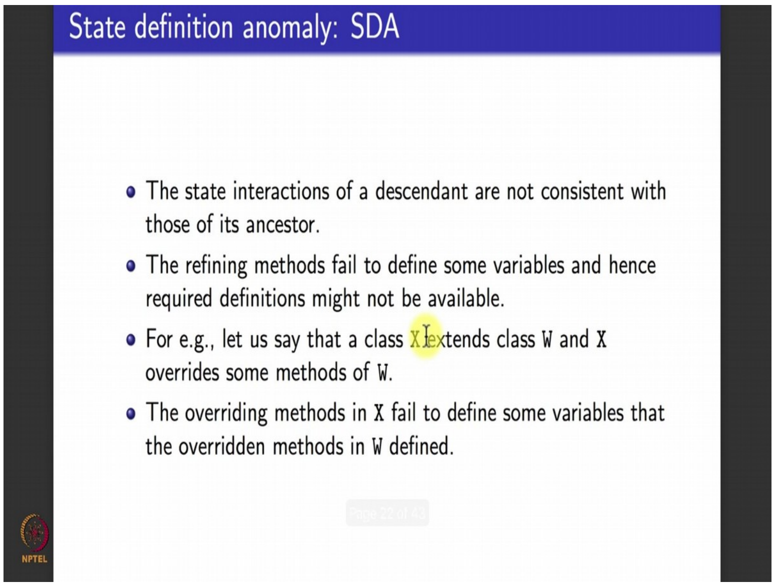


Visualizing polymorphism with the Yo-Yo graph

- Understanding which version of a method *will* be executed and which versions *can* be executed is very difficult.
- Execution can *bounce* up and down among levels of inheritance.
- The yo-yo graph is defined on an inheritance hierarchy.
  - It has a root and desdendants.
  - Nodes are methods: new, inherited and overridden methods for each descendent.
  - Edges are method calls as given in the source: directed edge is from caller to callee.
- In addition,
  - Each class is given a *level* in the yo-yo graph that shows the actual calls made if an object has the actual type of that level. These are depicted by *bold arrows*.
  - *Dashed arrows* are calls that cannot be made due to overriding.

And in week 10, I had also introduced you to the various kinds of object oriented faults.

(Refer Slide Time: 02:04)

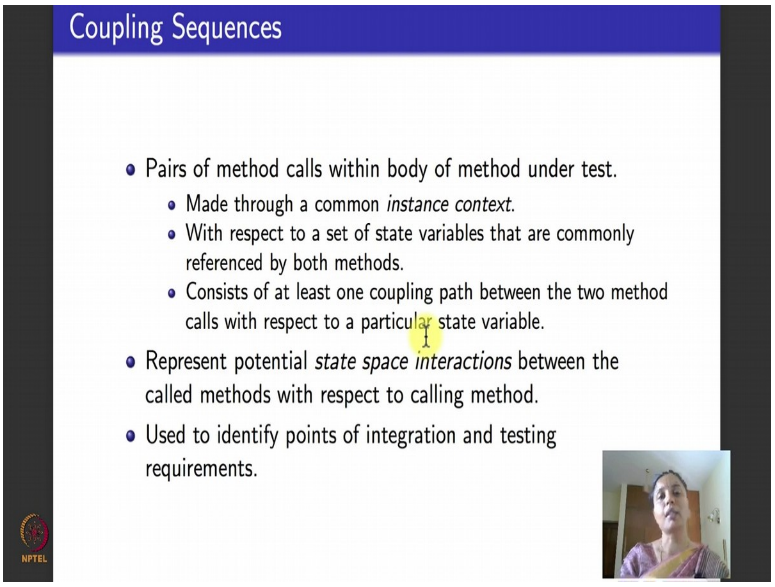


State definition anomaly: SDA

- The state interactions of a descendant are not consistent with those of its ancestor.
- The refining methods fail to define some variables and hence required definitions might not be available.
- For e.g., let us say that a class X extends class W and X overrides some methods of W.
- The overriding methods in X fail to define some variables that the overridden methods in W defined.

State definition anomaly, state definition inconsistency, state visibility fault, all other kinds of faults.

(Refer Slide Time: 02:18)

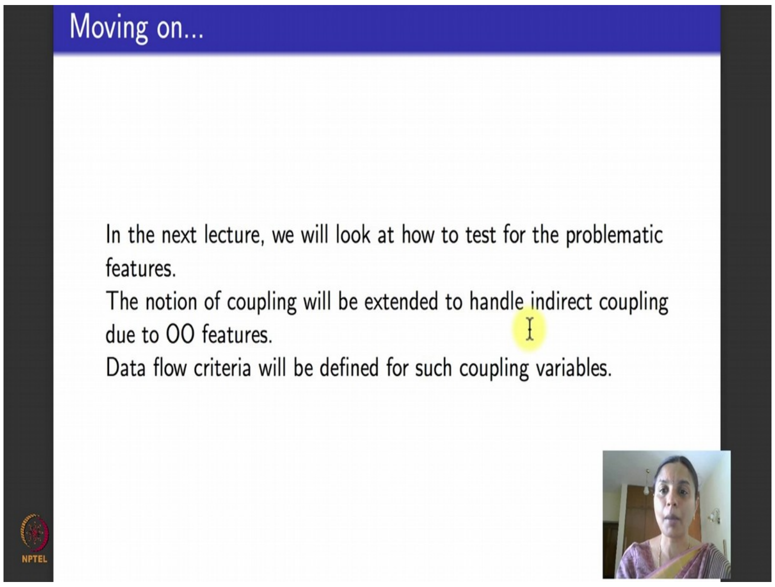


Coupling Sequences

- Pairs of method calls within body of method under test.
  - Made through a common *instance context*.
  - With respect to a set of state variables that are commonly referenced by both methods.
  - Consists of at least one coupling path between the two method calls with respect to a particular state variable.
- Represent potential *state space interactions* between the called methods with respect to calling method.
- Used to identify points of integration and testing requirements.

And then what I did not do is the last part, which dealt with coupling of variable specific object oriented software.

(Refer Slide Time: 02:21)

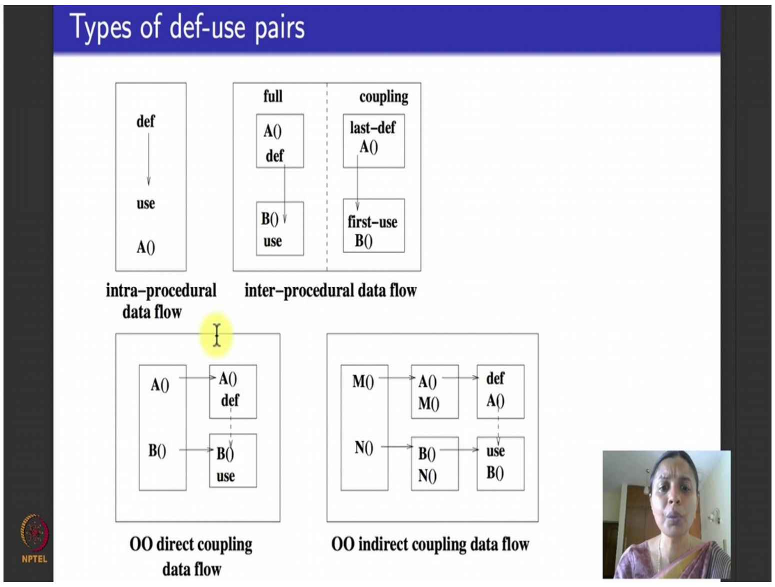


Moving on...

In the next lecture, we will look at how to test for the problematic features.
The notion of coupling will be extended to handle indirect coupling due to OO features.
Data flow criteria will be defined for such coupling variables.

How does data flow work in the presence of coupling variables, and what are the object oriented specific coverage criteria that we would see? So, that is what I am going to do in today's lecture. Coupling variables is not a concept that is new for us in the course. We have seen coupling when we did design integration testing. Coupling variable is a variable that is used in one and defined in one module and used in the other. But now the problem is in object oriented software, how does a coupling variable occur? There could be a pair of method calls within the body of method that is being testing. This pair of method calls can be made through a common instance of an object with respect to set a variables that are commonly referenced by both the methods. And they consist typically, of at least one coupling parts between the 2 method calls for one state variable.

(Refer Slide Time: 03:36)

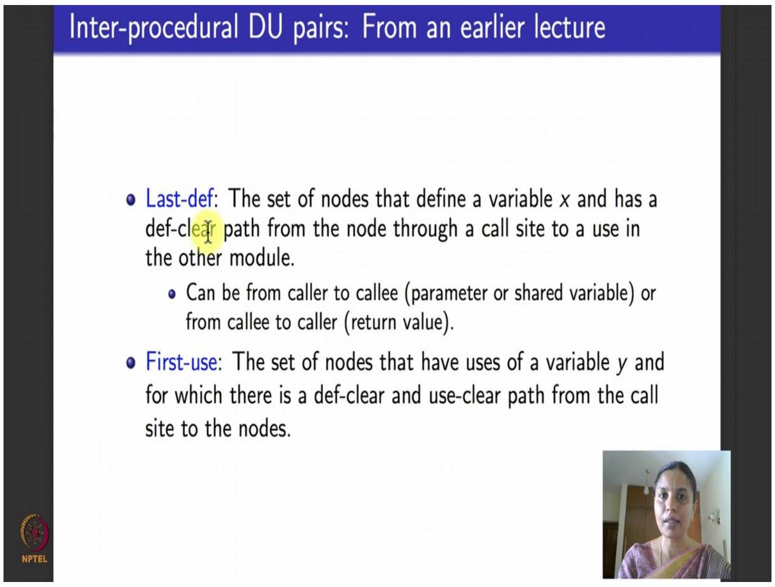


Types of def-use pairs

So, these represent potential state space interactions between the called methods with reference to the calling method. So, we have to identify which are the points of integration and what are the testing requirements in terms of coverage criteria for us to test on. So, we look at the various kinds of definition-use pairs that we have seen till now. This is one thing that we saw right in the beginning when we did data flow testing along with graphs. Within one method or a procedure A, a variable can be defined and used. So, this is intra procedure or intra method, within a method, normal kind of definition-use data flow. This we saw when we did data flow coverage criteria during graph base testing.

The next is inter procedural data flow which is from one procedure A to another procedure B, a particular definition happens in A and is used in B. If the definition happens to be the last definition in A and the first use of a particular variable in B, then such a variable is called a coupling variable and we consider coupling data flow. Now when it comes to object oriented data flow, what we talking about is instances of these a and b and definitions occurring in the instance of A and in instance of B. This is direct object oriented coupling. In indirect object oriented coupling what happens? There is a method m, there is another method n, there is a method a that uses m, there is method b that uses n and the definition and use happens in a and b for an m and n. So, we see an example where this becomes little more clear.

(Refer Slide Time: 04:59)

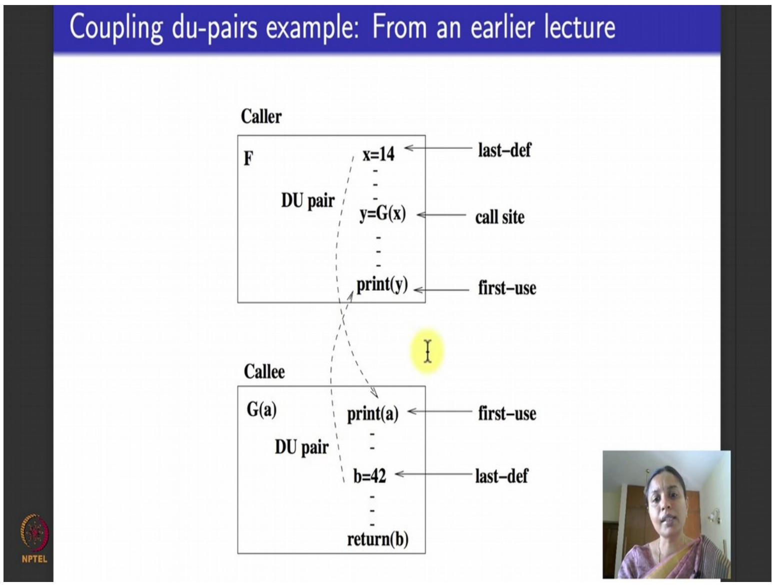


Inter-procedural DU pairs: From an earlier lecture

- Last-def: The set of nodes that define a variable x and has a def-clear path from the node through a call site to a use in the other module.
    - Can be from caller to callee (parameter or shared variable) or from callee to caller (return value).
- First-use: The set of nodes that have uses of a variable y and for which there is a def-clear and use-clear path from the call site to the nodes.

So, this is again a recap from the earlier lecture. What is the last def for a last definition? It is a set of nodes that define a variable x and has a def clear path from the note through a call site to a use in other procedure or module. That can be from the caller to a callee or can be from callee back to the caller, in which case it is the value that is returned. What is the first use? It is a set of nodes or statements that have uses of a particular variable y for which there is a def clear and a use clear path from the call site to the node that contains this use.

(Refer Slide Time: 05:42)

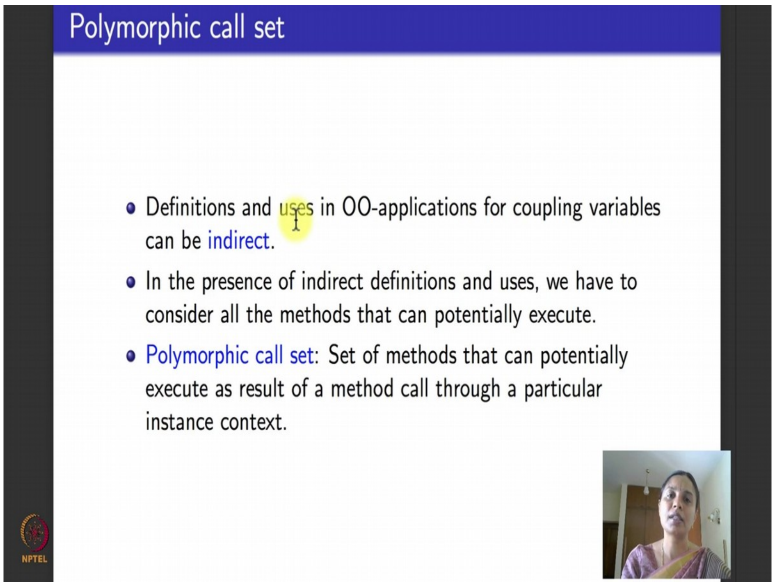


Coupling du-pairs example: From an earlier lecture

This is what we saw a long ago when we did design integration testing. And this if you remember is a exact recap of a example that I used in the earlier lecture. There is a procedure F that it some point in it is code calls another procedure G, and it calls G with x the value that G returns is passed to y. So, this statement is what is called the call site.

This the last definition of x that occurred just before this call is the last definition, when it is passed it becomes the first use and sum calculation happens and the value that is a returned, the statement that this causes the value that to be returned, is what is called the last definition.

(Refer Slide Time: 06:30)



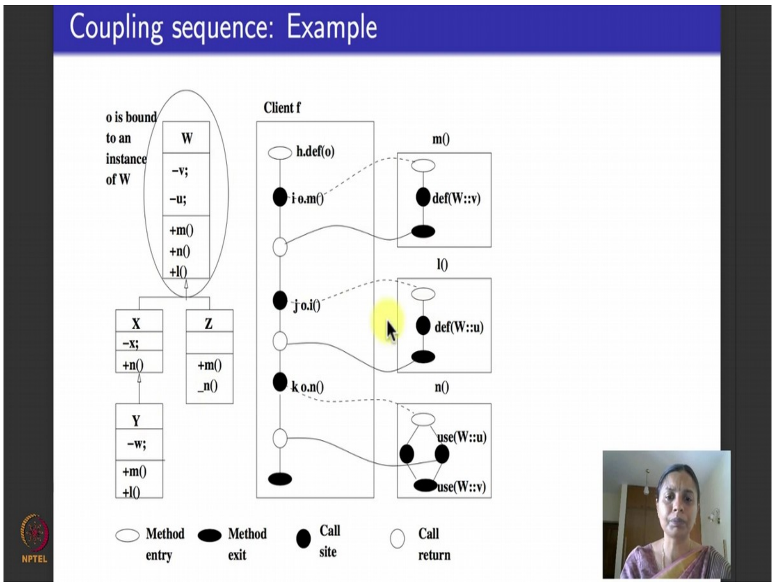
Polymorphic call set

- Definitions and uses in OO-applications for coupling variables can be indirect.
- In the presence of indirect definitions and uses, we have to consider all the methods that can potentially execute.
- Polymorphic call set: Set of methods that can potentially execute as result of a method call through a particular instance context.

So, this is what we saw as last definition and first use. Now we will see the same concept, but with reference to object oriented testing. So, to understand that we first need to understand when we have polymorphic methods, what is a polymorphic call set? So, here in the presence of the polymorphic methods, the first thing to observe is that the definitions and use for such coupling variables like these can be indirect. So, what do we mean by indirect? Indirect means you do not know which version will be executed in which method that is being used. So, in the presence of indirect definitions and uses, we have to consider all the methods that can potentially execute.

So, polymorphic call set is a set of methods that can potentially execute as a result of a method call through a particular instance context.

(Refer Slide Time: 07:11)



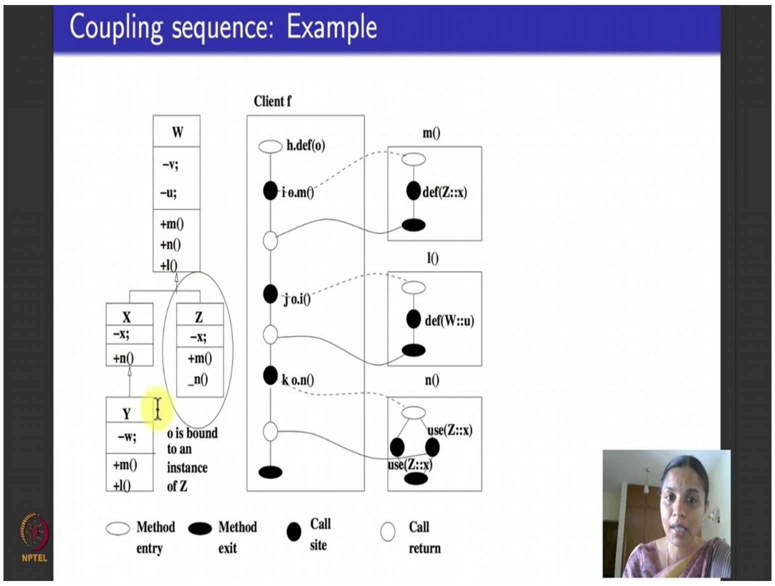
Coupling sequence: Example

I will illustrated with the example. So, this is a slightly big figure, but we will go through it slowly. On the left hand side is what we have the class hierarchy. There are 4 classes w, x, z and y. w has 2 private variables v and u, 3 methods m n and l. x has a variable small x method n again over ridden. z has no variables that matters to us for now. So, we will ignore the other variables that do not matter to us in z, I have not depicted it. And z also has 2 methods m and n, and y has a local private variable w and 2 methods m and l. Please remember that m occurs here, here and here. And n occurs in w, x and z, and l occurs in w and y. So, here is a typical code that uses this class hierarchy and the methods from these various classes.

Let us say it is the code of some client f. So, there is an object o. Let us say now that o is bound to an instance of the class w for our example in this code. So, what happens now let see at this point the method m is called in o and m is executed if you go to m m let say defines a v now remember object is bound to an instance of w. So, it is w's definition of v that is used and passed back. And then moving on, so how do I read this figure, what is the legend? These white eclipses are method entries, these black eclipses are method exists. So, there are method entries here, here and here for m, l and n. There is a main method entry here and method exit here. These black circle ones are call sites where the method calls happens and the white circles are when the method finishes executing, it is the call return.

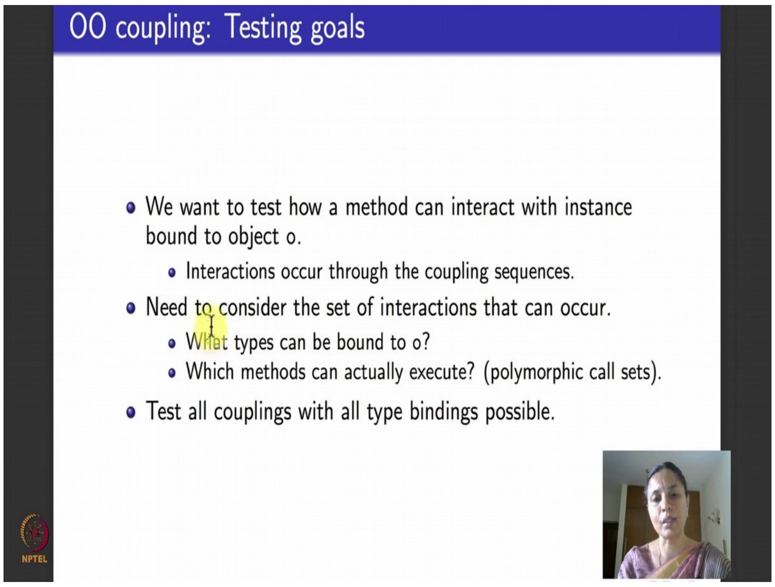
So, from this figure I hope it will be clear. So, method m is call which defines v and the value is returned and the method i is called which defines u and then the values returned and then the method n is called which defines which uses u, and then let us say it also uses v and it is returned. So, this dotted, dashed lines and the solid lines represent the coupling sequence of variables and the call sites and call returns. This happens for this class hierarchy when the object o is bound to the class w.

(Refer Slide Time: 09:49)


Coupling sequence: Example

Let us say an object o is a instead for the same class hierarchy, is bound to an instance of the class z. Then what happens? Right up front when this client call this method i, instead of taking ws definition let us say z's definition of x is taken, we are introduced to variable x that matters to us in z. And then let us say the next call w's definition of u is taken because n, l does not belong to z. And then x's use in z is considered and x's use in z is again considered here right. So, in which case the problem is because z does not have an instance of l w's l is considered and it could cause the coupling sequence being different. There is branching here which did not happened here during the use. When we come to object oriented coupling, what are a testing goals? We want to be able to test the following.

(Refer Slide Time: 10:40)

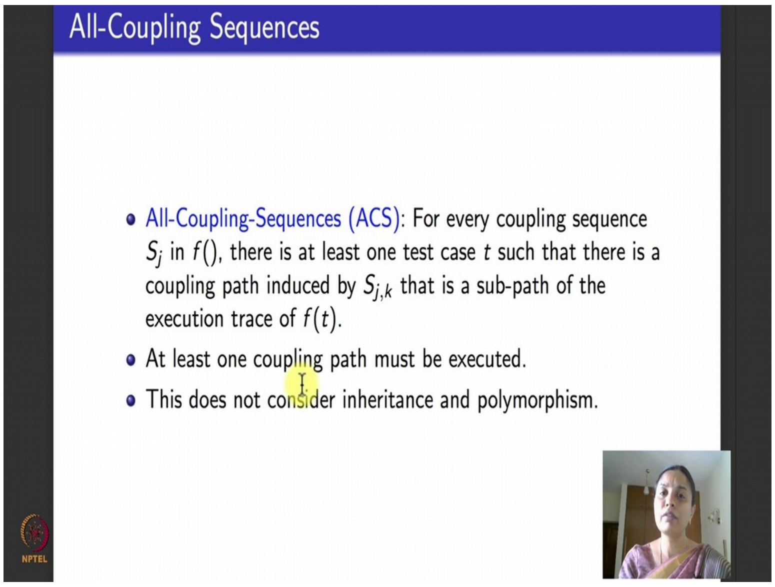


OO coupling: Testing goals

- We want to test how a method can interact with instance bound to object o.
  - Interactions occur through the coupling sequences.
- Need to consider the set of interactions that can occur.
  - What types can be bound to o?
  - Which methods can actually execute? (polymorphic call sets).
- Test all couplings with all type bindings possible.

We want to test how a method can interact with an instance bound to an object o. Like for an example here an object o is bound to an instance of w, the interaction is different from when an object o is bound to an instance of z. And we need to consider the set of interactions that can occur, what are the types that can be bound to o, which are the methods that can actually executed the presence of polymorphic call sets, and we have to test all couplings with all the type bindings.

(Refer Slide Time: 11:12)

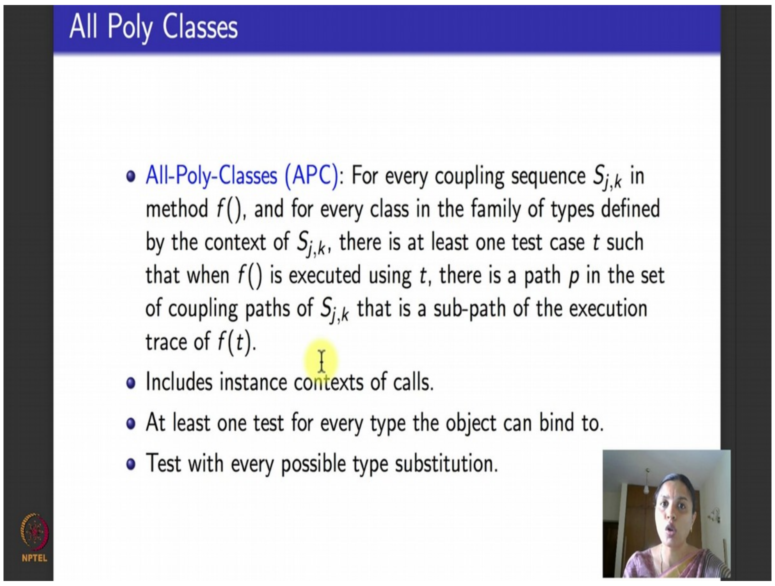


All-Coupling Sequences

- All-Coupling-Sequences (ACS): For every coupling sequence S_j in $f()$, there is at least one test case t such that there is a coupling path induced by $S_{j,k}$ that is a sub-path of the execution trace of $f(t)$.
- At least one coupling path must be executed.
- This does not consider inheritance and polymorphism.

So, we define 4 coverage criteria. The first one that we will look at is what is called all coupling sequences abbreviated as ACS, where for every coupling sequence s j in the particular method f, there is at least one test case t is needs to be written such that there is a coupling path introduced by s j, induced by s j k which is a sub path of the execution trace of f when it is executed in t. So, what it said is the there is a coupling sequences s j for example, like this kind of sequence, what is coupled with what, what is coupled with what, what is coupled with what. Every dash line is coupled with the following bold normal line. So, for every sub sequence in f, there is at least one test case that we must write such that when you execute the test case the coupling parts comes s j k that should be a sub path of the execution trace when f is executed with t. Which means what? At least one coupling path must be executed. Let us say there are 3 coupling variables, all it says is that per variable please execute at least one coupling path. No inheritance, polymorphism, polymorphic call set, it is not considered in this coverage criteria.

(Refer Slide Time: 12:26)



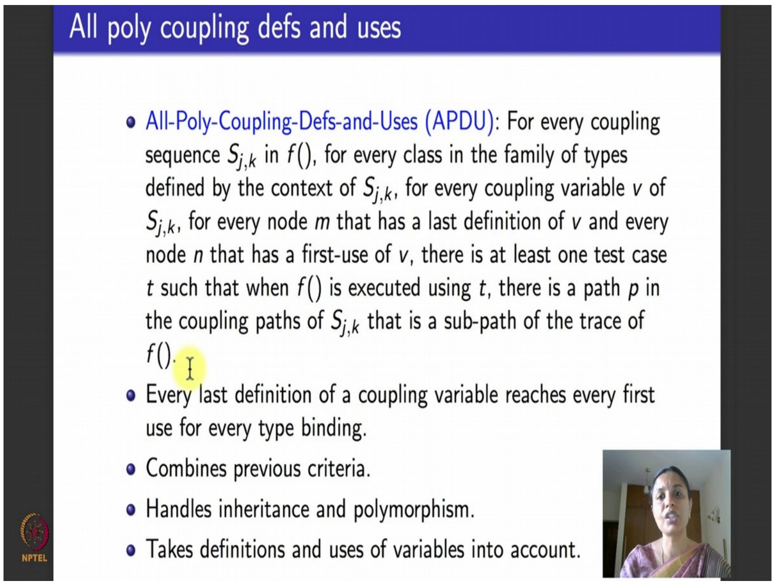
All Poly Classes

- All-Poly-Classes (APC): For every coupling sequence $S_{j,k}$ in method $f()$, and for every class in the family of types defined by the context of $S_{j,k}$, there is at least one test case t such that when $f()$ is executed using t, there is a path p in the set of coupling paths of $S_{j,k}$ that is a sub-path of the execution trace of $f(t)$.
- Includes instance contexts of calls.
- At least one test for every type the object can bind to.
- Test with every possible type substitution.

The next is all polyclasses abbreviated as APC. Here for every coupling sequence s j k, in the method that is the same as we saw here. And for every class in the family of types defined by the context of s j k, there is at least one test case t such that when is executed on f there is a path in the set of coupling paths that is the sub path of the execution trace of f of t. So, what is the difference between this and this? This includes all the coupling variables, this includes all the coupling variables in the context of the call that was made considering the binding also. So, it includes at least one test case for every type the objective can be bound to. If you go back to the example, it will include one test case for this case when o is bound to z, one test case for this case when o is bound to w. And it also test with every possible type substitution that can happen. Obviously, this is more effective or it subsumes the all coupling sequences criteria.

So, the next is all coupling defs and uses. So, here what happens? Abbreviated as ACDU, for every coupling variable v in each coupling pair s j k of a test case t, there is a coupling path introduced by s j k such that p is the sub path of execution trace of f of t for at least one case t. Which means what? Every last definition of a coupling variable reaches every first use. So, here it was like at least the first case, here it is every. This is again without inheritance and polymorphism. So obviously, the next one is going to be every last definition reaching every first use with inheritance and polymorphism.

(Refer Slide Time: 14:11)



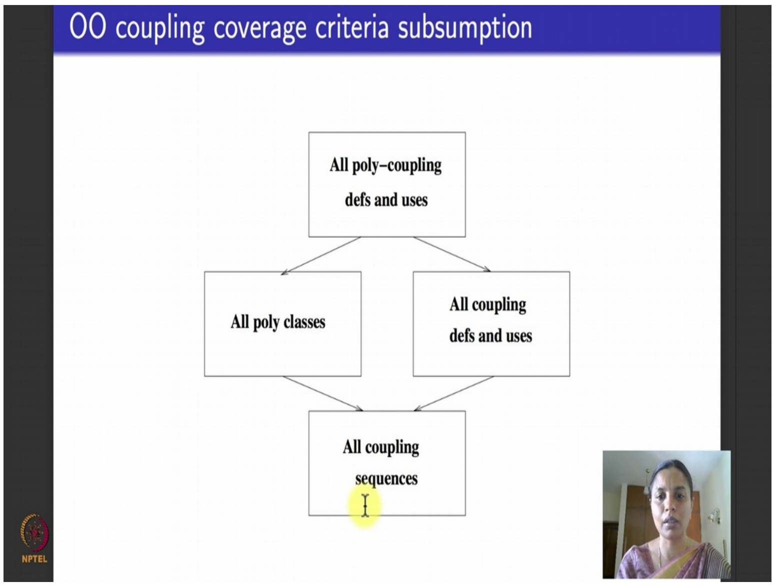
All poly coupling defs and uses

- All-Poly-Coupling-Defs-and-Uses (APDU): For every coupling sequence $S_{j,k}$ in $f()$, for every class in the family of types defined by the context of $S_{j,k}$, for every coupling variable v of $S_{j,k}$, for every node m that has a last definition of v and every node n that has a first-use of v, there is at least one test case t such that when $f()$ is executed using t, there is a path p in the coupling paths of $S_{j,k}$ that is a sub-path of the trace of $f()$.
- Every last definition of a coupling variable reaches every first use for every type binding.
- Combines previous criteria.
- Handles inheritance and polymorphism.
- Takes definitions and uses of variables into account.

And this is the mother of all criteria. It includes polymorphism, polymorphic call sets and it says every coupling variable, every definition to every use has to be covered.

Do not worry about this definition being long. What it basically says, is it says t if I have a coupling sequence in a function f, then for every class in the family of types define by the context of s j k, consider every coupling variable in that and then consider the nodes in which the last definition happen and the first use happen. You must have a test case for including that path, that is what it says. So, the simplest, to repeat, is all coupling sequences which says at least one coupling path must be executed. The next is at least one coupling path in the presence of instance contexts and object binding should be executed. This is every last definition to every first use, reachability should be tested, without inheritance and polymorphism.

(Refer Slide Time: 15:35)

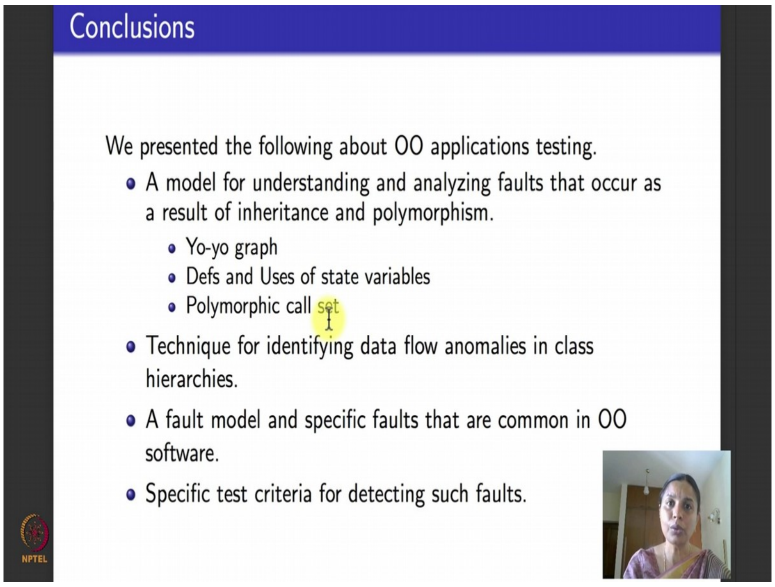


OO coupling coverage criteria subsumption

$$\begin{tikzcd}
& \text{All poly-coupling defs and uses} \arrow[dl] \arrow[dr] & \\
\text{All poly classes} \arrow[dr] & & \text{All coupling defs and uses} \arrow[dl] \\
& \text{All coupling sequences} &
\end{tikzcd}$$

This is every last definition to every first use with inheritance and polymorphism and type binding. So, this is the highest coverage criteria, the most difficult to subsume, and by definitions, this picture should be very clear. This says that test all coupling sequences the first one. This says with polymorphism. This says all definitions and uses, but without polymorphism. This is all definitions uses with polymorphism, with object binding. So, these are the 4 coupling coverage criteria that you can use for integration testing for object oriented software.

(Refer Slide Time: 16:02)



Conclusions

We presented the following about OO applications testing.

- A model for understanding and analyzing faults that occur as a result of inheritance and polymorphism.
  - Yo-yo graph
  - Defs and Uses of state variables
  - Polymorphic call set
- Technique for identifying data flow anomalies in class hierarchies.
- A fault model and specific faults that are common in OO software.
- Specific test criteria for detecting such faults.

So, to conclude the object oriented software lecture we understood in week 10, yo-yo graphs. In, sorry, errors and anomalies in object oriented testing. Today we understood polymorphic call sets, what are it is definitions and uses of state variables for integration testing and also looked at various coverage criteria that will detect faults that come because of object oriented features like inheritance and polymorphism.

So, I hope this lecture is beneficial to you. This will bring us to an end of object oriented integration testing.

Thank you.