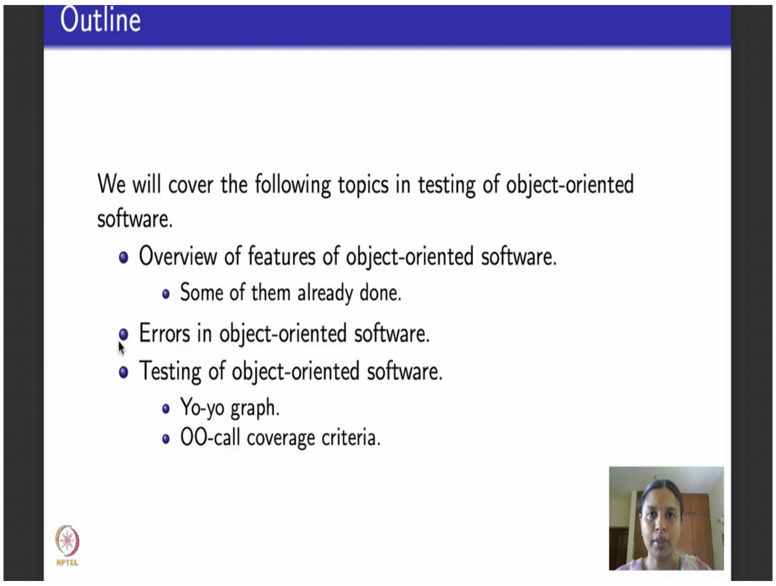
Software testing
Prof. Meenakshi D’Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 49
Testing of Object-Oriented Applications

Hello again, this is a last lecture of week 10, last time I had introduced to you to testing of object oriented applications.

(Refer Slide Time: 00:21)


Outline

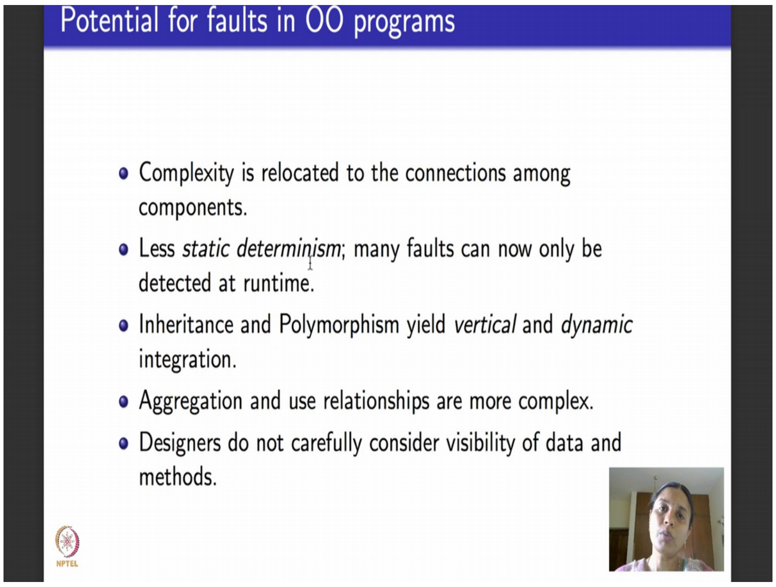
We will cover the following topics in testing of object-oriented software.

- 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.

This was an outline of the talks that I had discussed with you we will over look at on overview of some of the features of object oriented software we already done some in the context of mutation testing for object oriented integration. And then we said we will discuss about what are the kinds of faults or errors that will come an object oriented software will follow it up with testing of object oriented software Yo-yo graph is something that we already looked at in the last module.

And then in the next lecture I will ended with object oriented call coverage criteria today what I am going to focus on is what are the errors in object oriented software we already done overview in of object oriented features. So, I had introduced you to inheritance polymorphic methods and what is the class hierarchy what could be the various anomalies with object oriented testing and we had ended last class by looking at this graph called Yo-yo now moving on what I am going to do today's lecture is focus on faults or errors that comes specifically related to object oriented programs.

(Refer Slide Time: 01:22)

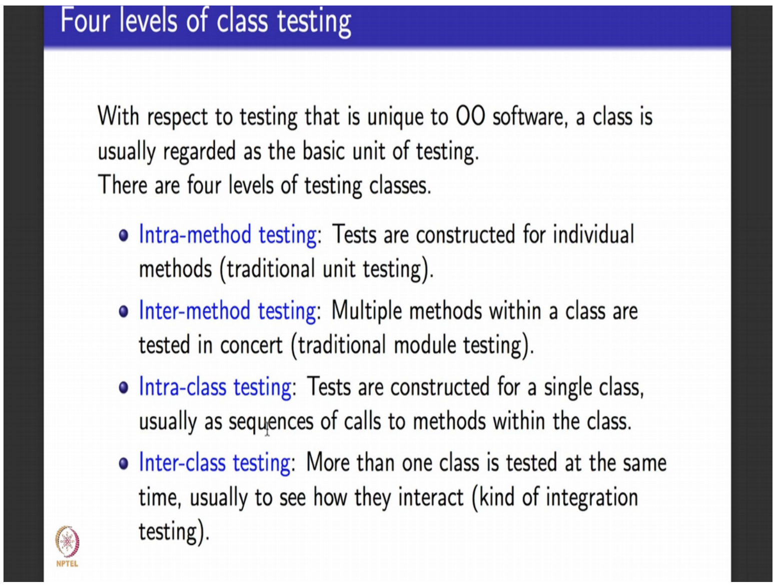


Potential for faults in OO programs

- Complexity is relocated to the connections among components.
- Less *static determinism*; many faults can now only be detected at runtime.
- Inheritance and Polymorphism yield *vertical* and *dynamic* integration.
- Aggregation and use relationships are more complex.
- Designers do not carefully consider visibility of data and methods.

Specifically with reference to the object oriented features that we have seen till now; now what will be the complexity of the main source of errors the main source of errors as I told you because we are focusing on which level of testing if you go back to this slide that I had looked at we are focusing here we are focusing on intra class and inter class.

(Refer Slide Time: 01:41)



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.

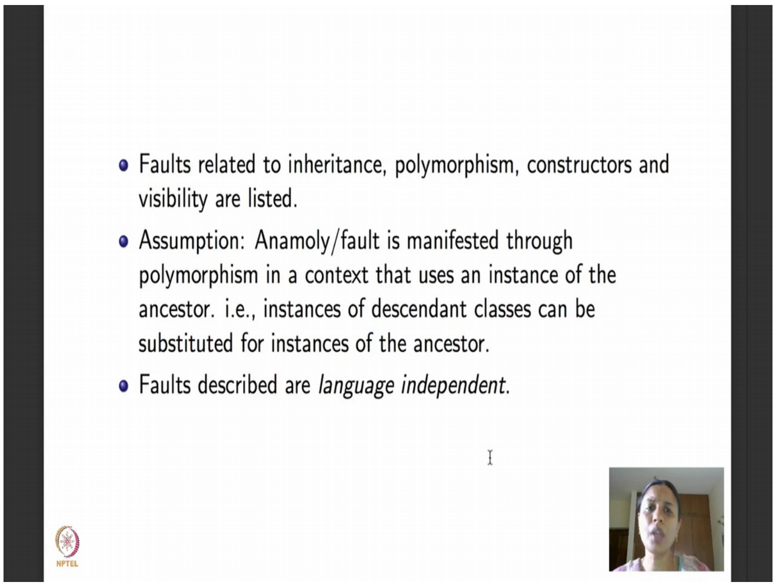
- Intra-method testing: Tests are constructed for individual methods (traditional unit testing).
- Inter-method testing: Multiple methods within a class are tested in concert (traditional module testing).
- Intra-class testing: Tests are constructed for a single class, usually as sequences of calls to 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).

So, which means what we are focusing on various components and how they are going to interact; so, our complexity will or focus will be on the connections amongst these components the problem with object oriented software as we have seen till now is that the amount of static determinism is very less.

In the sense that suppose I make a method call another method or let us say I make a function call and other function and see I know that this is how the call is going to happen nothing is going to be different where as that is not true for an object oriented programming language for method calls and other method. The method that is being called can be overridden the method that is called link can be overridden one of these methods could be polymorphic all kinds of interactions can happen nothing is statically determined lot of things depend on dynamic execution.

As I told you the complicated features that we recapped reasonably well like inheritance polymorphism they yield vertical integration dynamically changing integration and there are aggregation and use relationships that also add to the complexity and typically designers when they design and capsulation and abstraction they do not consider visibility on data and methods because they consider it as a testers problem.

(Refer Slide Time: 03:05)



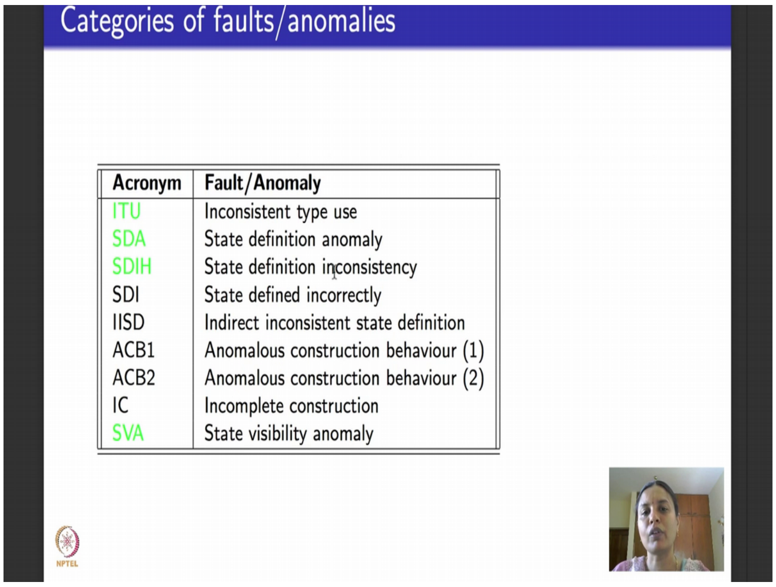
- Faults related to inheritance, polymorphism, constructors and visibility are listed.
- Assumption: Anamoly/fault is manifested through polymorphism in a context that uses an instance of the ancestor. i.e., instances of descendant classes can be substituted for instances of the ancestor.
- Faults described are *language independent*.

So, now what we are going to look at in these lectures just completely understand some of the faults that come because of these features inheritance polymorphism constructors visibility and so on.

While we understand these faults we will make one assumption will assume that the fault are anomaly is manifested through polymorphism in a context that uses in instance of ancestor that is instances of descendants descendant classes can be substituted for instances of an ancestor another thing that I would like to tell you is that the faults that I have described here are completely language independent it is not like these are exclusive only to java and they cannot be used for C++ they are generic faults that can arise in object oriented languages.

That use these features. So, they are independent of a particular programming language.

(Refer Slide Time: 04:02)



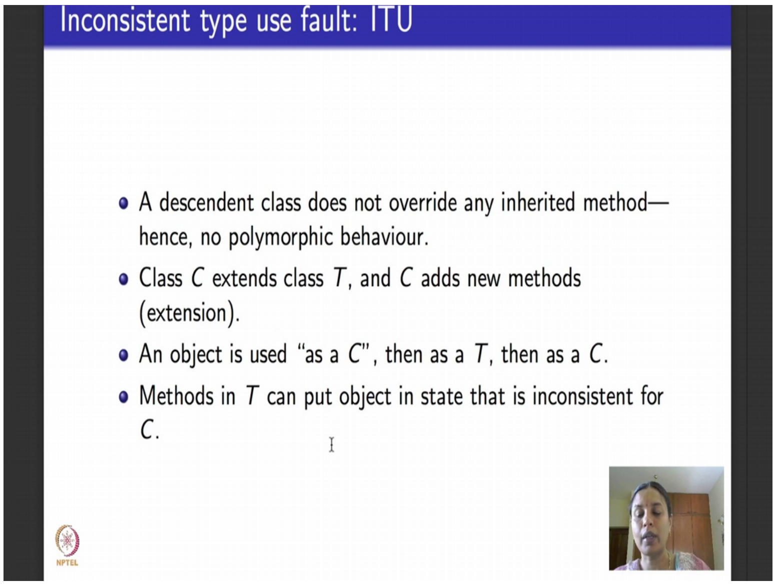
Categories of faults/anomalies

Acronym	Fault/Anomaly
ITU	Inconsistent type use
SDA	State definition anomaly
SDIH	State definition inconsistency
SDI	State defined incorrectly
IISD	Indirect inconsistent state definition
ACB1	Anomalous construction behaviour (1)
ACB2	Anomalous construction behaviour (2)
IC	Incomplete construction
SVA	State visibility anomaly

So, here is a table of the faults and anomalies that are commonly listed for method level interclass and intra class level testing of object oriented applications do not worry too much about remembering these acronyms they are just listed for convenience because its useful to use them in slides subsequently and while talking about it you do not have to memorize them. So, we will understand about the nine faults 9 kinds of faults are anomalies are listed here I will explain about 5 of them in detail with examples illustrating what is the kind of fault are anomaly the remaining 2 3 of them.

I will just give an overview and I will point you to literature at the end of these slides where you could look up for additional information. So, we will look at inconsistent type use state definition can be anomalies or inconsistent or incorrect we look at indirect and inconsistence state definition when it comes to constructors there could be 2 different kinds of anomalies behaviors I have decided to skip these 2 ACB 1 and ACB 2 because we have not really looked at constructors class constructors in details. So, I will skip these 2, but I will tell you in detail about the other things will also do state visibility anomaly.

(Refer Slide Time: 05:22)



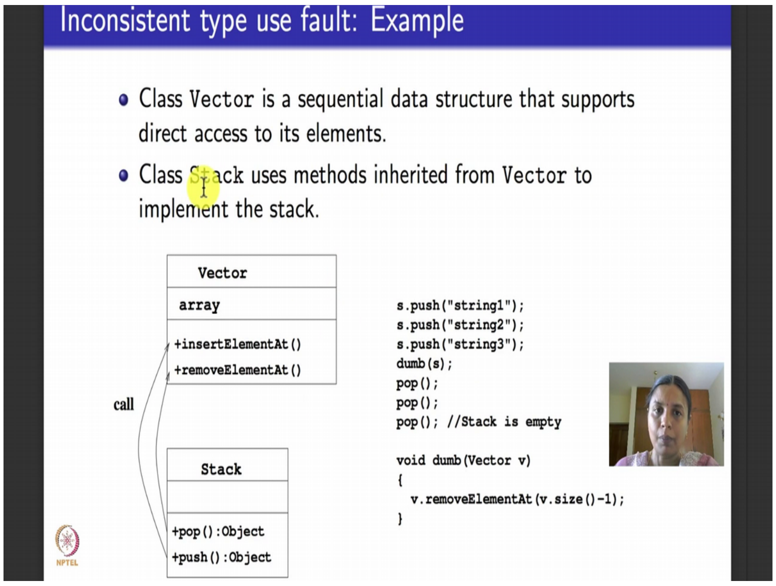
Inconsistent type use fault: ITU

- A descendent class does not override any inherited method—hence, no polymorphic behaviour.
- Class C extends class T, and C adds new methods (extension).
- An object is used "as a C", then as a T, then as a C.
- Methods in T can put object in state that is inconsistent for C.

So, of the first one in the list is inconsistent type use fault ITU. So, what is the cause of this fault the cause of this fault is because a descendent class does not overwrite any inherited method? So, there is no polymorphic method polymorphic behavior because nothing is overridden everything is static. So, you could think; what is the problem. So, will understand it with an example for example, consider a case where class C extends a particular class say T and C also adds new methods an object sometimes is used as a C sometimes is used as T and sometimes is used as a C again now methods in T can put this object in a state.

That is inconsistent for C that is why it is called inconsistent type use fault what do we mean by a state we mean by a state in any programming language including object oriented programming language we mean the values of all the variables. So, this last point you must read it as methods can change the sub values of some of the variables which results in object going into a different kind of state.

(Refer Slide Time: 06:32)



Inconsistent type use fault: Example

- Class Vector is a sequential data structure that supports direct access to its elements.
- Class Stack uses methods inherited from Vector to implement the stack.

```
s.push("string1");
s.push("string2");
s.push("string3");
dumb(s);
pop();
pop();
pop(); //Stack is empty

void dumb(Vector v)
{
  v.removeElementAt(v.size()-1);
}
```

So, here is an example let us state we have these 2 classes there is a class called vector which uses a sequential data structure called array and it supports direct access to all its elements it has 2 methods one method thus inserting an element at a particular point.

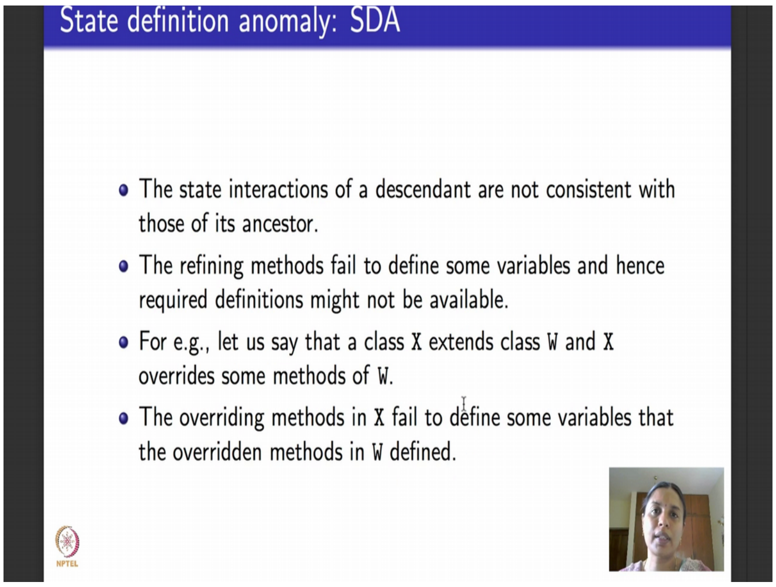
And it also does removing an element at a particular point there is a class stack what is class stack do it does the normal operations of a stack it pushes or it pops an object. So, to pop an object it calls the method remove element from the inherited from the class vector to push an object it calls the method insert element inherited from the class vector. So, here is a abstract piece of code again like all the examples that we saw in the course this course is this code is not complete, but only a snippet of it is given may helping you to understand the code issue.

So, let us say there is there are 3 operations that happen in this code 3 push operations string one string 2 and string 3 are pushed using this method in the class stack and let us say at some point this method dumb is called on s let us go and look at the code of dumb what does dumb do dumb removes an element at v X size v v of size minus one which means in this case it removes the middle element. So, after a call to dumb is main when it returns out of the 3 entities that we are pushed into s this stack s only 2 of them are available now what look at these series of calls of pops first pop after the 2 elements that are available in s.

The one that is available top most gets popped out second pop there is only one left it gets popped up now for the third one there is nothing to pop the stack is already empty why is there is nothing to pop even though there were 3 different pushes that is because there was this call may to a method dumb which also use to remove element which the class stack did not know about. Because it use remove element one of the elements went away the class stack assume that because there were as many pushes is there pops you would have elements for to do 3 pops, but elements where not there.

Because this dumb used, so, this resulted in an inconsistent assumption on the state of s here it assumes that there were 3 elements present whereas, actually only 2 elements were present.

(Refer Slide Time: 09:06)

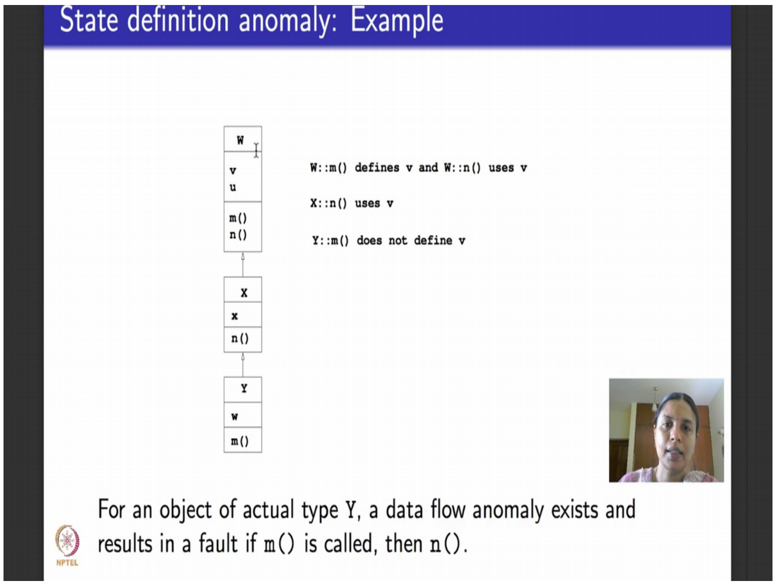


State definition anomaly: SDA

- 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.

So, this is a classical example of how inconsistent type use can occur the next kind of fault that I want to explain is what is called state definition anomaly SDA what happens here the state interactions of a descendent are not consistent with those of its ancestor what do again we mean by state interactions state interactions means operations that involve changing the state which means operations that change one or more variables that are present in the state.

(Refer Slide Time: 09:56)



State definition anomaly: Example

W::m() defines v and W::n() uses v

X::n() uses v

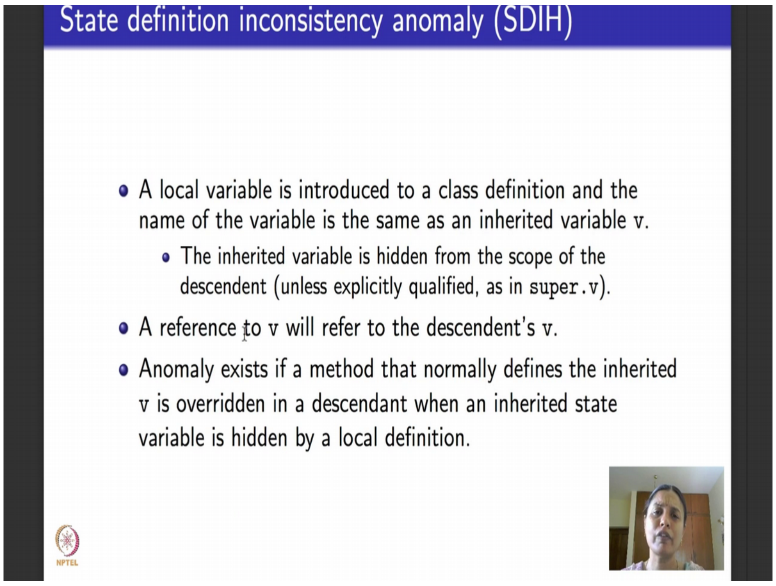
Y::m() does not define v

For an object of actual type Y, a data flow anomaly exists and results in a fault if m() is called, then n().

So, the refining methods fail to define some variables and the required definitions that you need may not be available we will see an example that explains this in detail let us say there is a class X that extents W and along with the extension X also overrides some of the methods of W the overriding methods in X fail to determine some variables that the overridden methods in W actually defined here is an example let us say there are 3 classes W X and Y. So, W defines 2 variables u and v and has 2 methods m and n X you define X has one variables small X and has one method n and y has one variable W small W and has one method m.

So, W this method m and W defines v and this method n in W uses v let us say some piece of code I have not given the actual piece of code and let us say the method X and n also uses v and let us say the method y in m does not define v at all. So, now, for an object of actual type y there will be a data flow anomaly if a fault in m if m is called and then n why because if m is called and then m y m does not define v and W m is the one that actually defines v y m does not define v and X n tries to use v, but there will be no v available for X n to use. So, this results in a variable not being defined which means a state a part of the state not being defined and hence is called state definition anomaly.

(Refer Slide Time: 11:15)



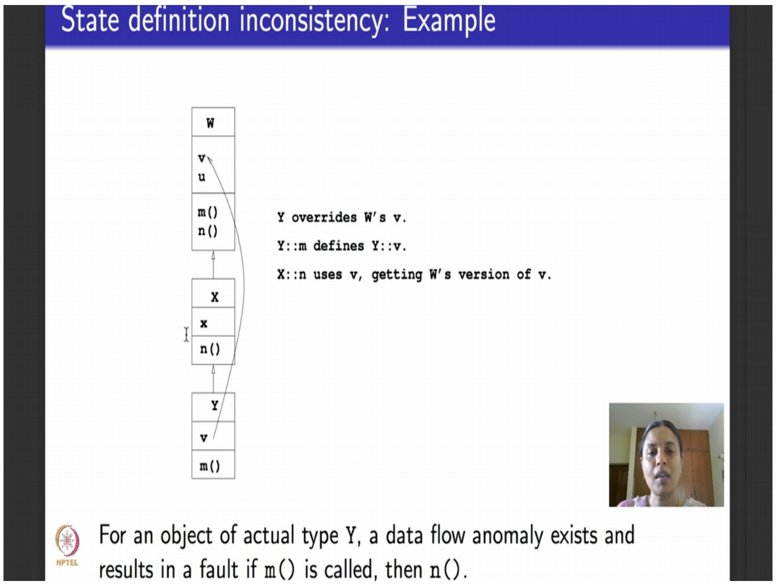
State definition inconsistency anomaly (SDIH)

- A local variable is introduced to a class definition and the name of the variable is the same as an inherited variable v.
  - The inherited variable is hidden from the scope of the descendent (unless explicitly qualified, as in `super.v`).
- A reference to v will refer to the descendent's v.
- Anomaly exists if a method that normally defines the inherited v is overridden in a descendant when an inherited state variable is hidden by a local definition.

The next kind of fault is state definition inconsistency fault SDIH here what happens a local variable is introduced in to a class definition and the name of the variable is the same as the name of the inherited variables. So, there is some amount of inconsistency that can happen because 2 different variables share the same name the inherited variable is hidden from this scope of descendant.

Of course you if you explicitly use this keywords super and refer to the variable v then it is not hidden, but otherwise you can assume that the inherited variable is hidden from the scope of the descendant now a reference to v will actually referred to the descendants v because that is where it is rewritten and anomaly can exist if a method that normally defines the inherited variable is overridden in a descendant when an inherited state variable is hidden by a local definition; so, will see again in an example.

(Refer Slide Time: 12:12)



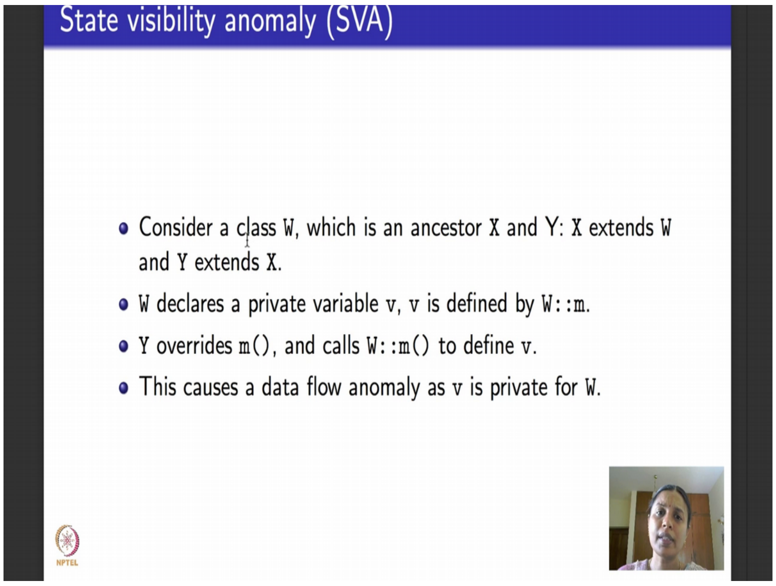
State definition inconsistency: Example

```
Y overrides W's v.
Y::m defines Y::v.
X::n uses v, getting W's version of v.
```

For an object of actual type Y, a data flow anomaly exists and results in a fault if m() is called, then n().

So, consider the same kind of situation we have 3 classes W X and y W has 2 variables u and v 2 methods m and n X has a variables small X and a method n Y has a variable small v and a method m now the catch is the Y the class Y overrides the class y overrides this w s v. So, which means what these 2 v is the same, but this v overrides this v now the method m of y defines this variable y now X in this method n uses v, but whose version will it get it will get ws version is that clear. So, now, for an actual type of object actual type y there can be a data flow analysis if m is called an then n is called because X is actually expecting ws version where as y is already overridden it. So, it with v that is sent is basically not the expected v.

(Refer Slide Time: 13:20)

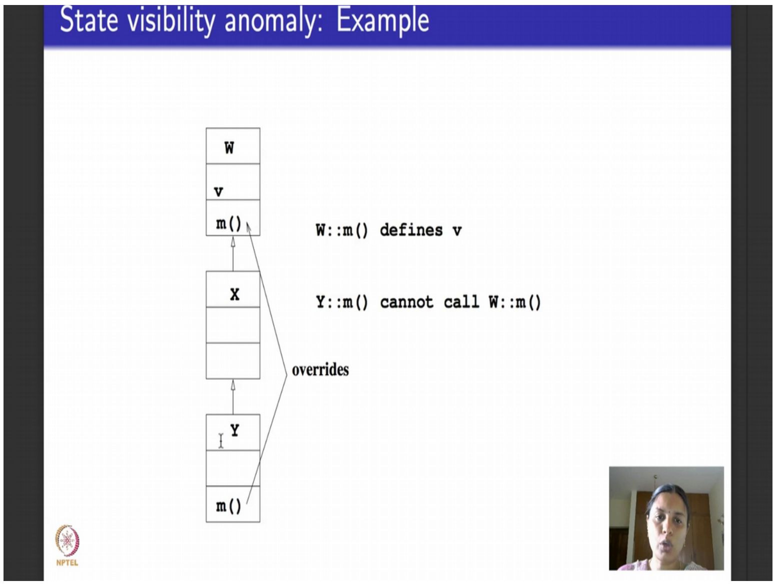


State visibility anomaly (SVA)

- Consider a class W, which is an ancestor X and Y: X extends W and Y extends X.
- W declares a private variable v, v is defined by W::m.
- Y overrides m(), and calls W::m() to define v.
- This causes a data flow anomaly as v is private for W.

So, this state part corresponding to that v is inconsistent. So, the next fault state visibility anomaly consider a class W which is an ancestor of X and y X extends W y also extends X; now say W declares a private variable v in a method m y overrides that method m calls W m to define v now this causes data flow anomaly because the call for y is not well defined why because v is private for W right. So, which means the visibility or the access level of v is not consistent the way y and W want to use it that is what is wrong.

(Refer Slide Time: 13:57)



State visibility anomaly: Example

W::m() defines v

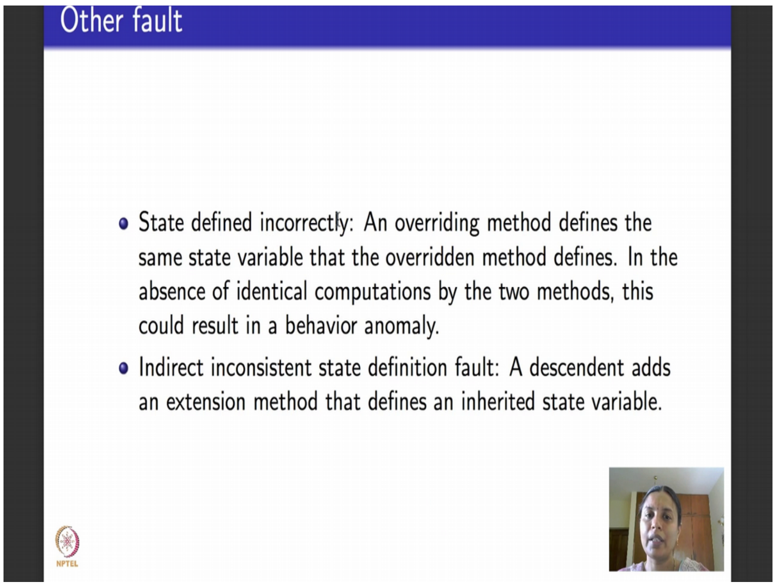
Y::m() cannot call W::m()

So, this is what is explained in the previous slide 3 classes W X and Y; Y has a private variable v and defined by the method W has a private variable v sorry define by the method m and y has a method m that overrides this method. So, y m cannot call W m.

Because of W is I mean the variable v is private to w. So, that results in the visibility of v not being there. So, that is what is called state visibility anomaly there are several other faults that were listed in the table that I presented to you earlier for a second I will go back to that table. So, these where the faults why have I colored them these 4 green because I gave you examples for these 4 that is why these 4 were colored green we will also tell you what is state defined in consistently an indirect inconsistent state definition is as I told you we will not discuss the 3 constructive faults.

Because I dint introduced a notion of class constructor in detail to you. So, I will skip it in my lectures, but I will give you references where you can learn more from it. So, the 2 pending things without examples are state defined incorrectly and indirect inconsistent state definition.

(Refer Slide Time: 15:16)



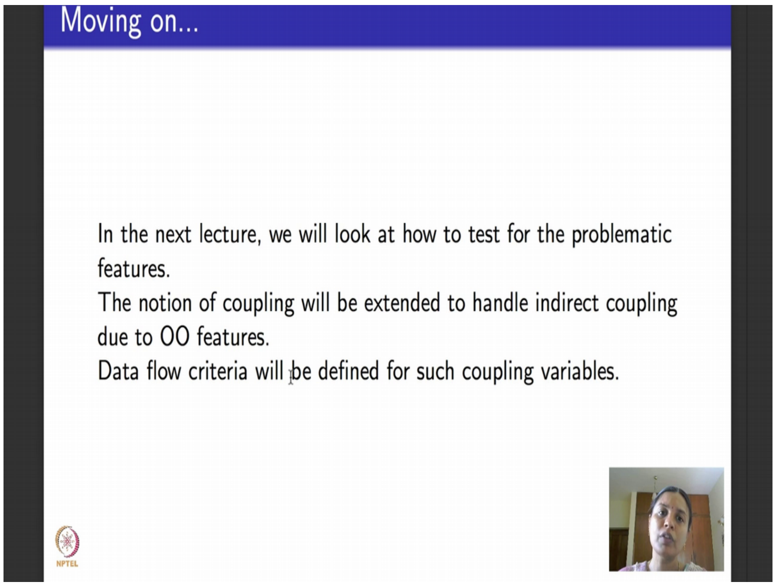
Other fault

- State defined incorrectly: An overriding method defines the same state variable that the overridden method defines. In the absence of identical computations by the two methods, this could result in a behavior anomaly.
- Indirect inconsistent state definition fault: A descendent adds an extension method that defines an inherited state variable.

So, we will look at that state defined incorrectly what happens here an overriding method defines the same state variable that the overridden method also defines so; obviously, 2 variables defined by 2 different methods one could say values one the other could say values the other state will be defined incorrectly in the absence of identical computations.

If the 2 methods 2 identical computations fine, but let us say 2 methods 2 different things they could highly likely that they will come up with different values for the same variable v. So, the state gets defined incorrectly the next is indirect inconsistent state definition fault this is when happens when a descendant adds an extension method that defines an inherited state variable the reference that I will be pointing you to you will find examples for these also feel free to look at them in you can pin in the forum.

(Refer Slide Time: 16:06)



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.

If you do not understand anything or you have doubts moving on what are we going to do in the next lecture in next lecture will test will look at how to test for these problematic features such that we detect the faults that we understood in this lecture.

If you remember when we did integration testing data flow using graphs and all that we did data flow testing and we introduced the notion of coupling variable we will extended to object oriented coupling and we will define data flow criteria over these coupling variables. So, that is it for today.

Thank you.