Introduction to Software Testing
Chapter 7.1
Engineering Criteria for Technologies

Paul Ammann & Jeff Offutt

http://www.cs.gmu.edu/~offutt/softwaretest/

The Technologies

- Chapters 5-9 emphasize criteria on four models of software
- Emphasis in each chapter was first on the criteria, then on how to construct the models from different software artifacts
- This chapter discusses how to apply the criteria to specific technologies
  - Most of the ideas in this chapter were developed after the year 2000
  - Thus they are still evolving

Chapter 7 Outline

1. **Object-Oriented Software**
2. **Web Applications and Web Services**
3. **Graphical User Interfaces**
4. **Real-Time and Embedded Software**

Section 7.1 Outline

1. **Overview**
2. **Types of Object-Oriented Faults**
   1. **Example**
   2. **The Yo-Yo Graph and Polymorphism**
   3. **Categories of Inheritance Faults**
   4. **Testing Inheritance, Polymorphism and Dynamic Binding**
   5. **Object-Oriented Testing Criteria**

Inheritance

Allows common features of many classes to be defined in one class

A derived class has everything its parent has, plus it can:

- **Enhance derived features (overriding)**
- **Restrict derived features**
- **Add new features (extension)**

Inheritance (2)

Declared type: The type given when an object reference is declared

```
Clock w1; // declared type Clock
```

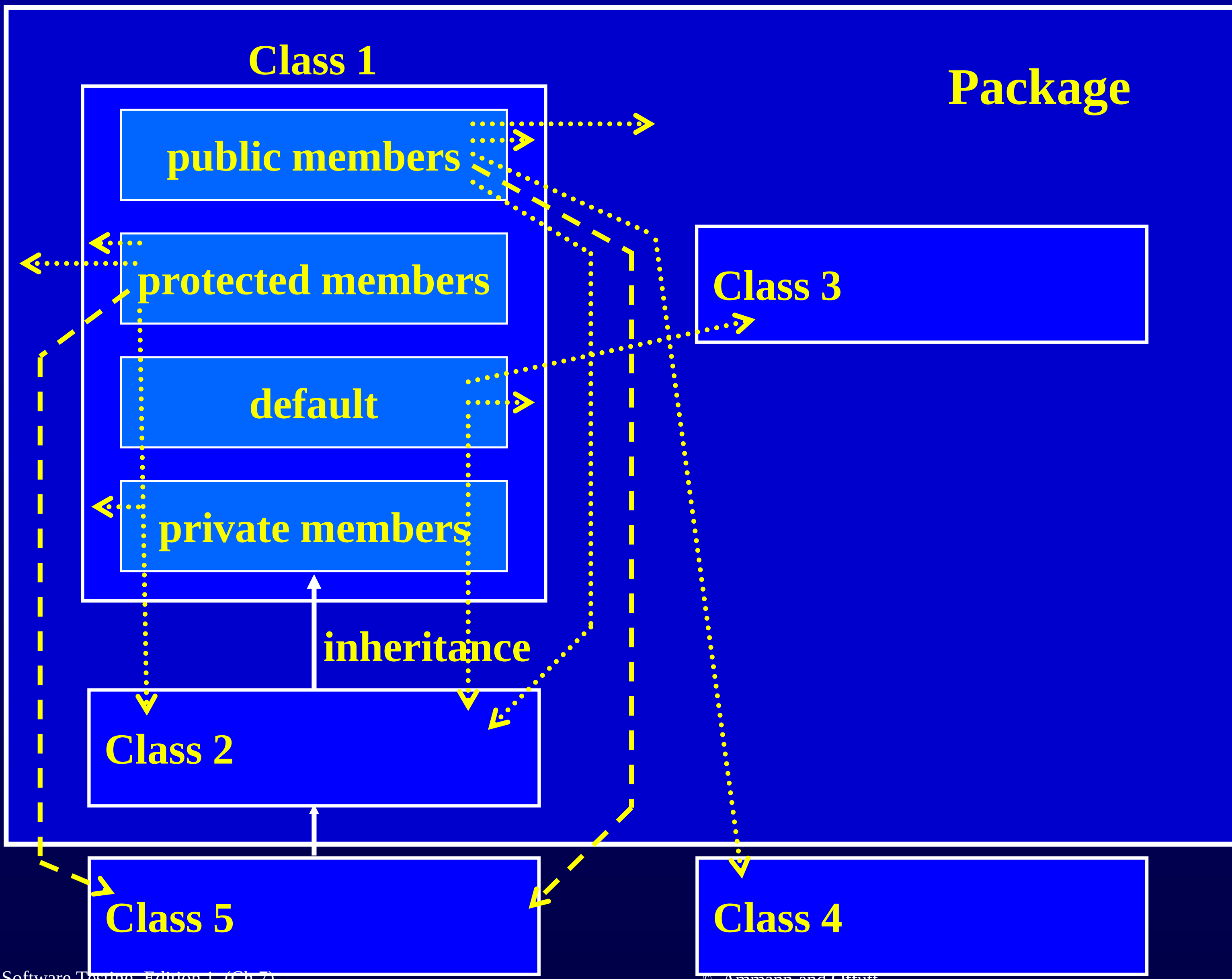
Actual type: The type of the current object

```
w1 = new Watch(); // actual type Watch
```

> In Java, the method that is executed is the lowest version of the method defined between the actual and declared types in the inheritance hierarchy

A
↑
B
↑
C

Access Control (in Java)



Polymorphism

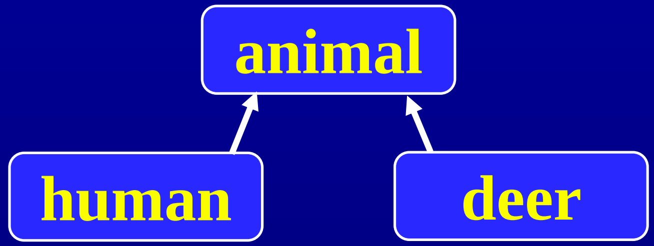
- **The same variable can have different types depending on the program execution**

- **If *B* inherits from *A*, then an object of type *B* can be used when an object of type *A* is expected**

- **If both *A* and *B* define the same method *M* (*B* overrides *A*), then the same statement might call either *A*'s version of *M* or *B*'s version**

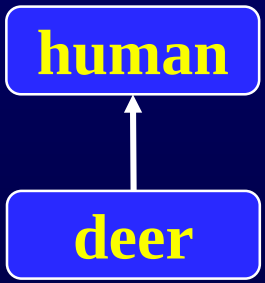
Subtype and Subclass Inheritance

- **Subtype Inheritance : If B inherits from A, any object of type B can be substituted for an object of type A**
  - **A *laptop* "is a" special type of *computer***
  - **Called *substitutability***



- **Subclass Inheritance : Objects of type B may not be substituted for objects of type A**
  - **Objects of B may not be "*type compatible*"**
  - **In Java's collection framework, a *Stack* inherits from a *Vector* … convenient for implementation, but a stack is definitely not a vector**

This gives a deer access to "hands" !



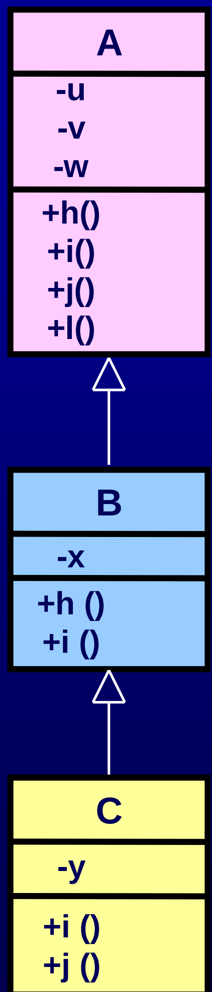
Testing OO Software

1. **Intra-method testing** : Testing individual methods within classes
2. **Inter-method testing** : Multiple methods within a class are tested in concert
3. **Intra-class testing** : Testing a single class, usually using sequences of calls to methods within the class
4. **Inter-class testing** : More than one class is tested at the same time (integration)

Section 7.1 Outline

1. **Overview**
2. **Types of Object-Oriented Faults**
   1. **Example**
   2. **The Yo-Yo Graph and Polymorphism**
   3. **Categories of Inheritance Faults**
   4. **Testing Inheritance, Polymorphism and Dynamic Binding**
   5. **Object-Oriented Testing Criteria**

Example DU Pairs and Anomalies



Consider what happens when an overriding method has a different def-set than the overridden method

Method	Defs	Uses
A::h ()	{A::u, A::w}	
A::i ()		{A::u}
A::j ()	{A::v}	{A::w}
A::l()		{A::v}
B::h()	{B::x}	
B::i()		{B::x}
C::i()		
C::j()	{C::y}	{C::y}

def-use

DU anomaly

A::h() calls j(), B::h() does not

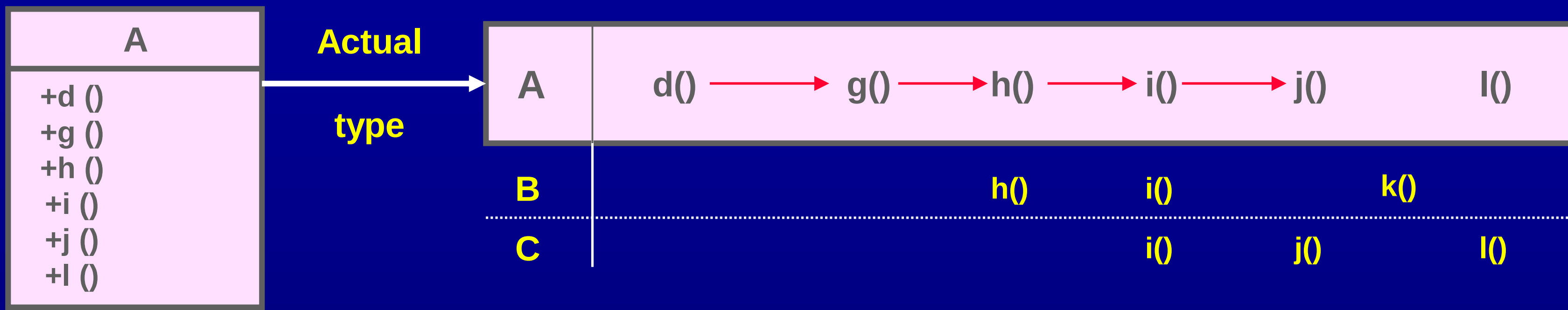
def-use

DU anomaly

Section 7.1 Outline

1. Overview
2. Types of Object-Oriented Faults
    1. Example
    2. The Yo-Yo Graph and Polymorphism
    3. Categories of Inheritance Faults
    4. Testing Inheritance, Polymorphism and Dynamic Binding
    5. Object-Oriented Testing Criteria

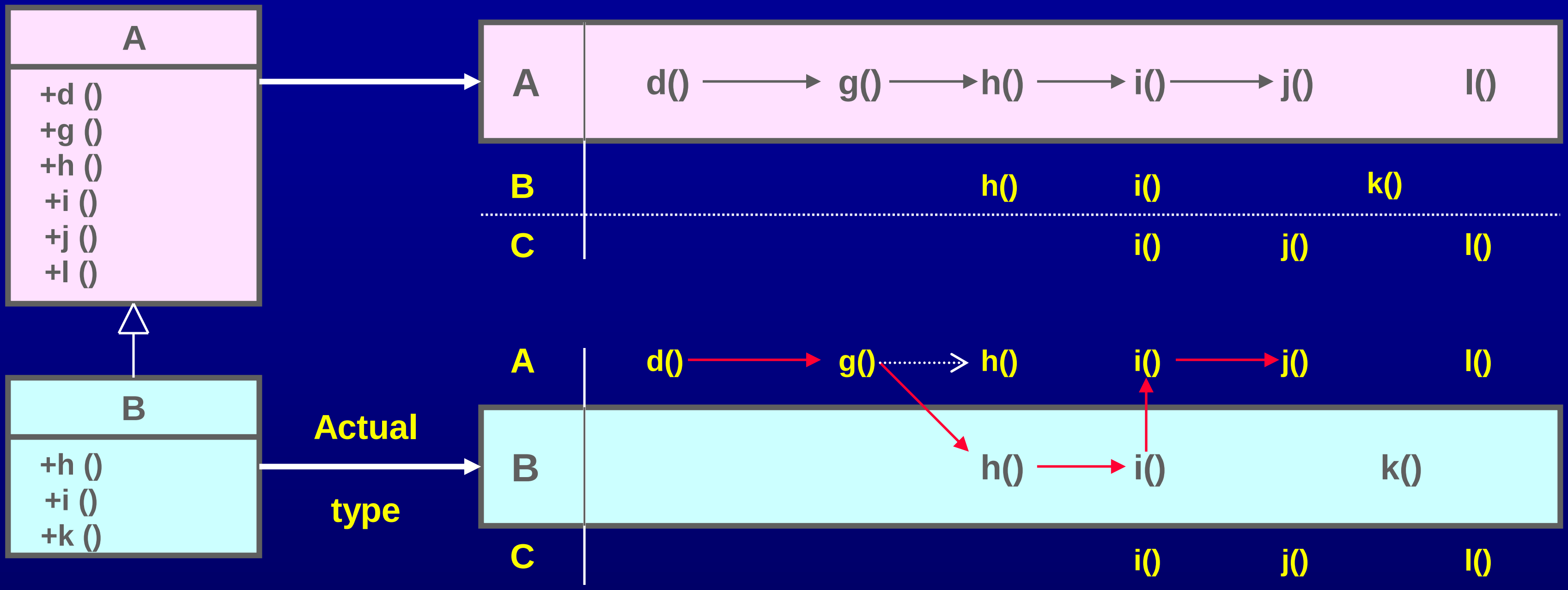
Polymorphism Headaches (Yo-Yo)



Object is of actual type A
A::d ()

d() calls g(), which calls h(), which calls i(), which calls j()

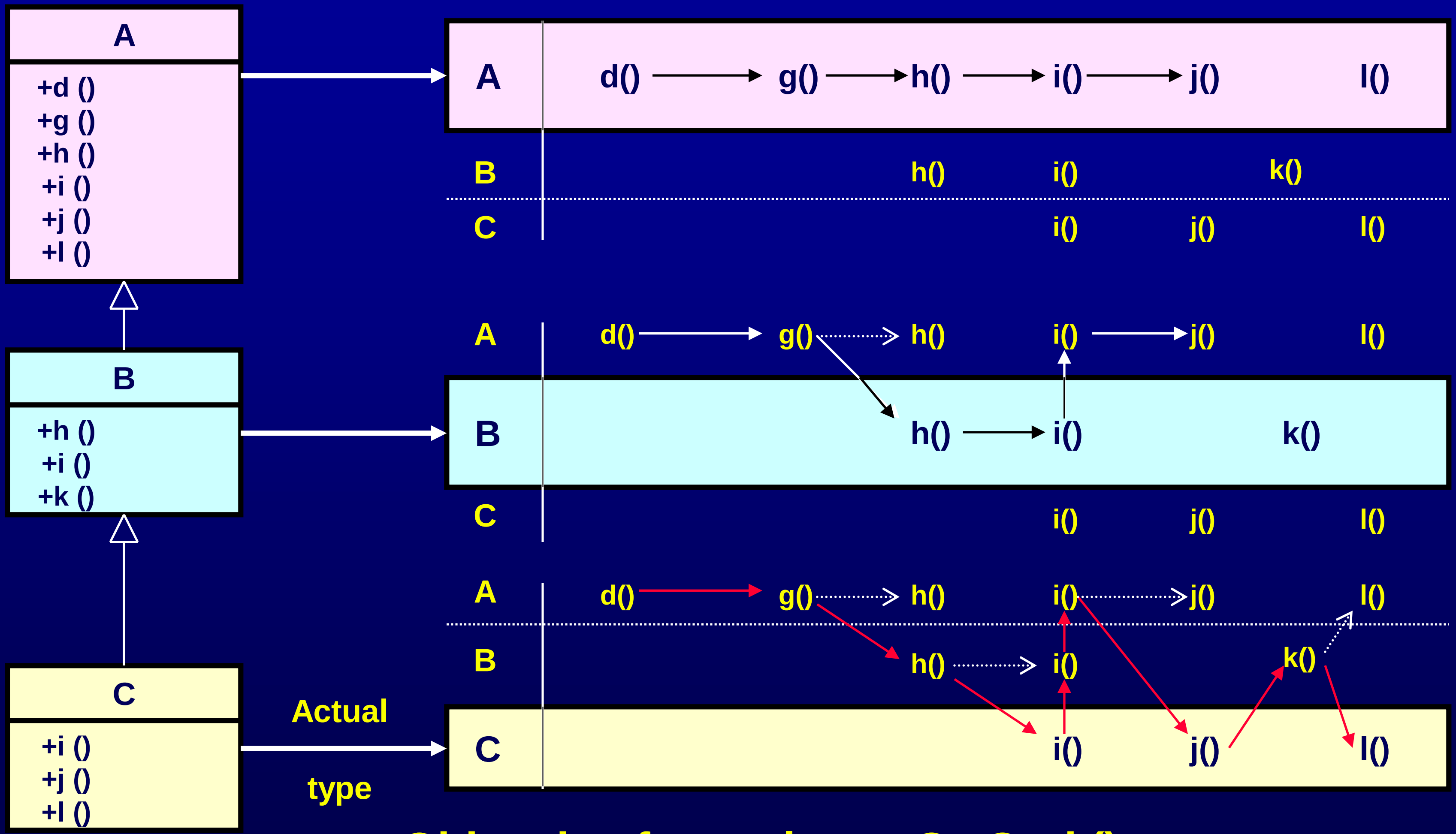
Polymorphism Headaches (Yo-Yo)



Object is of actual type B

B::d ()

Polymorphism Headaches (Yo-Yo)



Object is of actual type C, C::d ()

Section 7.1 Outline

1. Overview
2. Types of Object-Oriented Faults
   1. Example
   2. The Yo-Yo Graph and Polymorphism
   3. Categories of Inheritance Faults
   4. Testing Inheritance, Polymorphism and Dynamic Binding
   5. Object-Oriented Testing Criteria

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

Object-oriented Faults

- **Only consider faults that arise as a direct result of OO language features:**
  - **inheritance**
  - **polymorphism**
  - **constructors**
  - **visibility**
- **Language independent (as much as possible)**

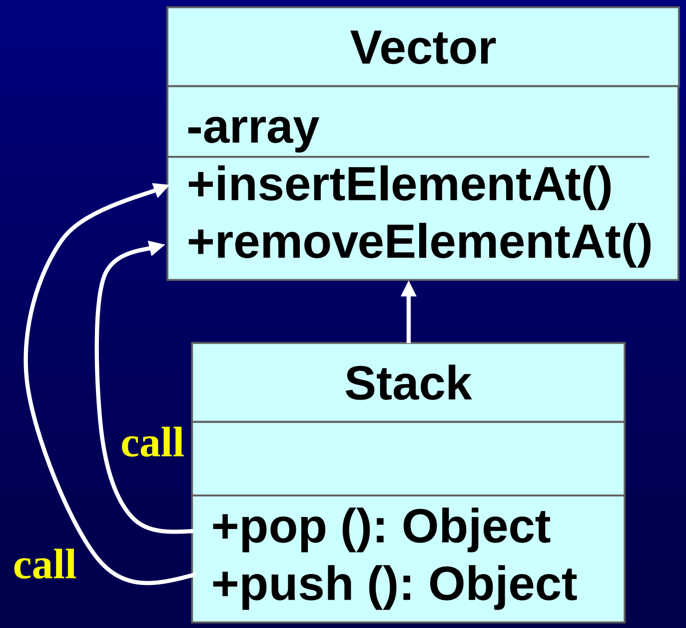
OO Faults and Anomalies

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 Behavior (1)
ACB2	Anomalous Construction Behavior (2)
IC	Incomplete Construction
SVA	State Visibility Anomaly

Examples shown

Inconsistent Type Use (ITU)

- No **overriding** (no polymorphism)
- *C* extends *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 <u>inconsistent</u> for *C*

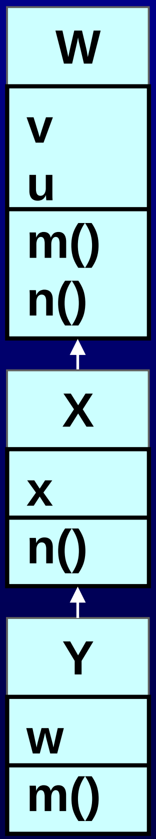



```
s.push ("Steffi");
s.push ("Joyce");
s.push ("Andrew");
dumb (s);
s.pop();
s.pop();
s.pop();  // Stack is empty!

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

State Definition Anomaly (SDA)

- *X* extends *W*, and *X* overrides some methods
- The overriding methods in *X* fail to define some variables that the overridden methods in *W* defined

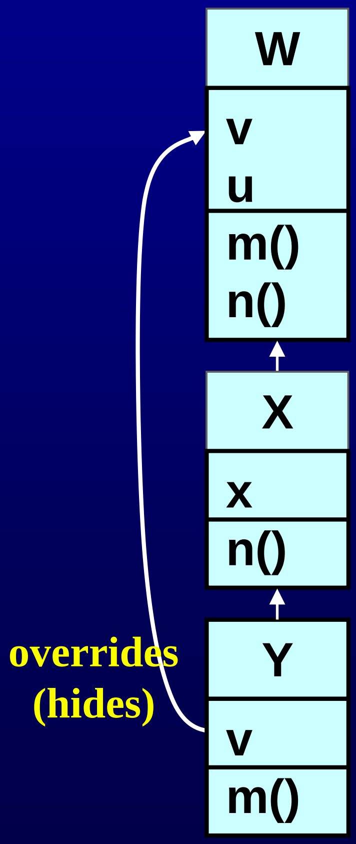


- *W::m ()* defines *v* and *W::n()* uses *v*
- *X::n ()* uses *v*
- *Y::m ()* does not define *v*

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

State Definition Inconsistency (SDIH)

- **Hiding** a variable, possibly accidentally
- If the descendant's version of the variable is defined, the **ancestor's** version may not be

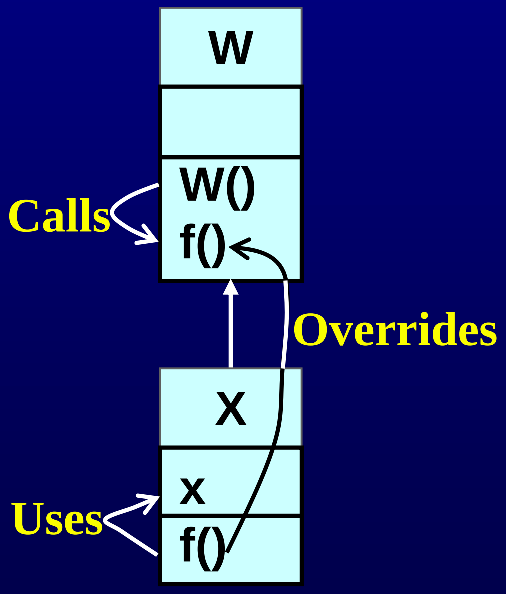


- *Y* overrides *W*'s version of *v*
- *Y::m()* defines *Y::v*
- *X::n()* uses *v* … getting *W*'s version of *v*

For an object of actual type *Y*, a data flow inconsistency may exist and result in a fault if *m()* is called, then *n()*

Anomalous Construction Behavior (ACB1)

- **Constructor** of *W* calls a method *f()*
- A **child** of *W*, *X*, overrides *f()*
- *X::f()* uses **variables** that should be defined by *X*'s constructor

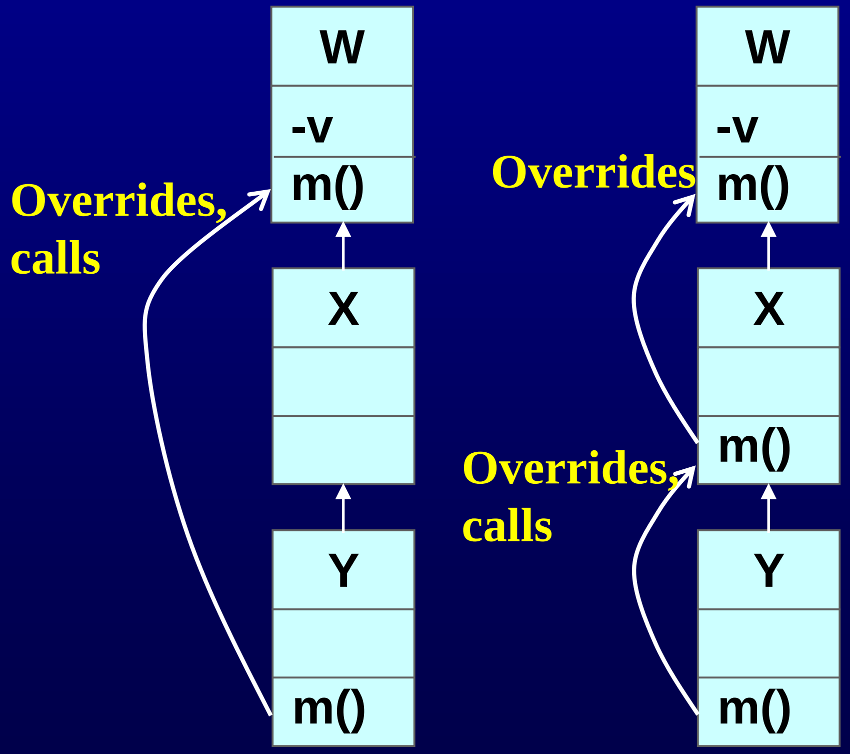


When an object of type *X* is constructed, *W()* is run before *X()*.

When *W()* calls *X::f()*, *x* is used, but has <u>not</u> yet been given a value!

State Visibility Anomaly (SVA)

- A private variable *v* is declared in ancestor *W*, and *v* is defined by *W::m()*
- *X* extends *W* and *Y* extends *X*
- *Y* overrides *m()*, and calls *W::m()* to define *v*



X::m() is added later

Y:m() can no longer call *W::m()*!

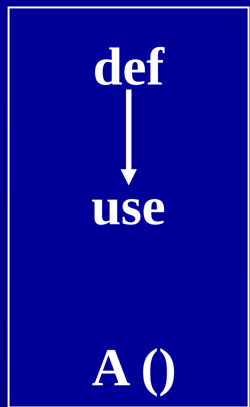
Section 7.1 Outline

1. **Overview**
2. **Types of Object-Oriented Faults**
   1. **Example**
   2. **The Yo-Yo Graph and Polymorphism**
   3. **Categories of Inheritance Faults**
   4. **Testing Inheritance, Polymorphism and Dynamic Binding**
   5. **Object-Oriented Testing Criteria**

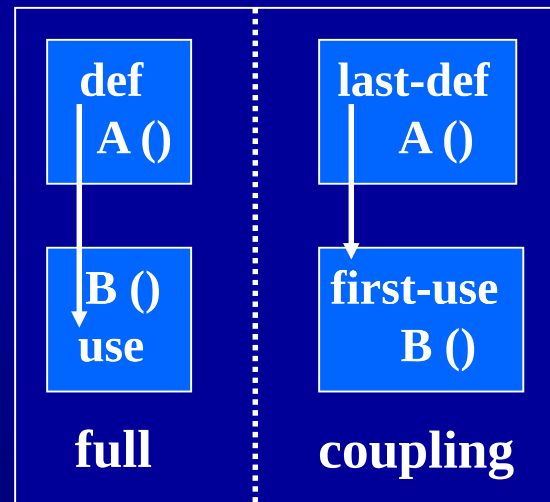
Coupling Sequences

- **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

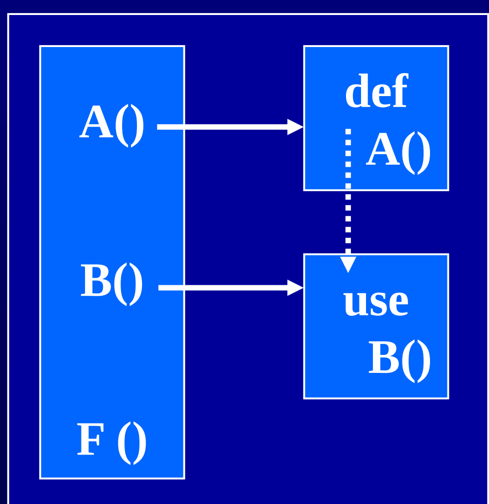
Types of Def-Use Pairs



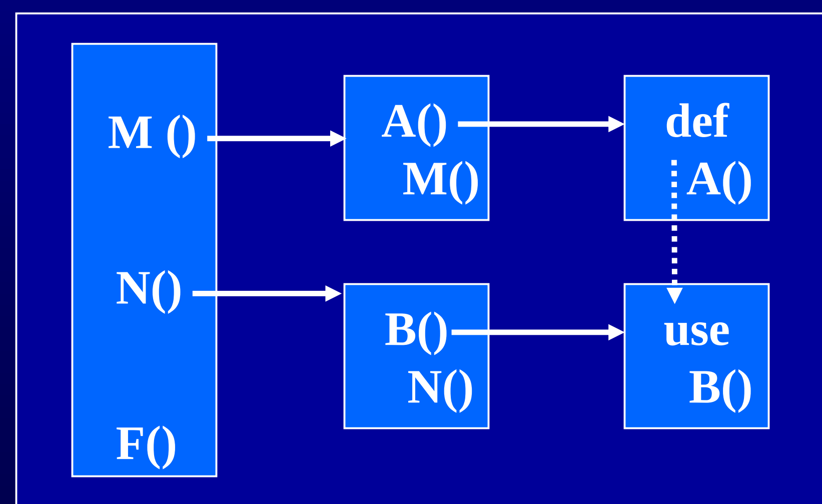
intra-procedural data flow
(within the same unit)



inter-procedural
data flow



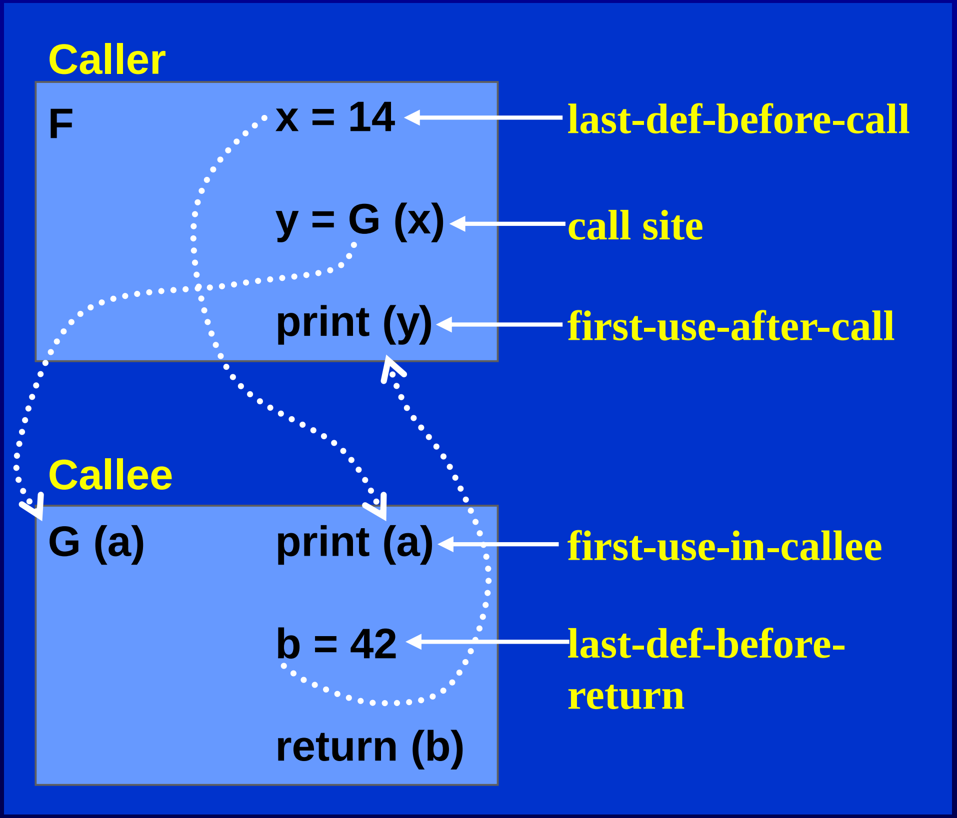
object-oriented <u>direct</u>
coupling data flow



object-oriented <u>indirect</u>
coupling data flow

Coupling-Based Testing (from Ch 2)

- Test data and control connections
- Derived from previous work for procedural programs
- Based on insight that integration occurs through couplings among software artifacts



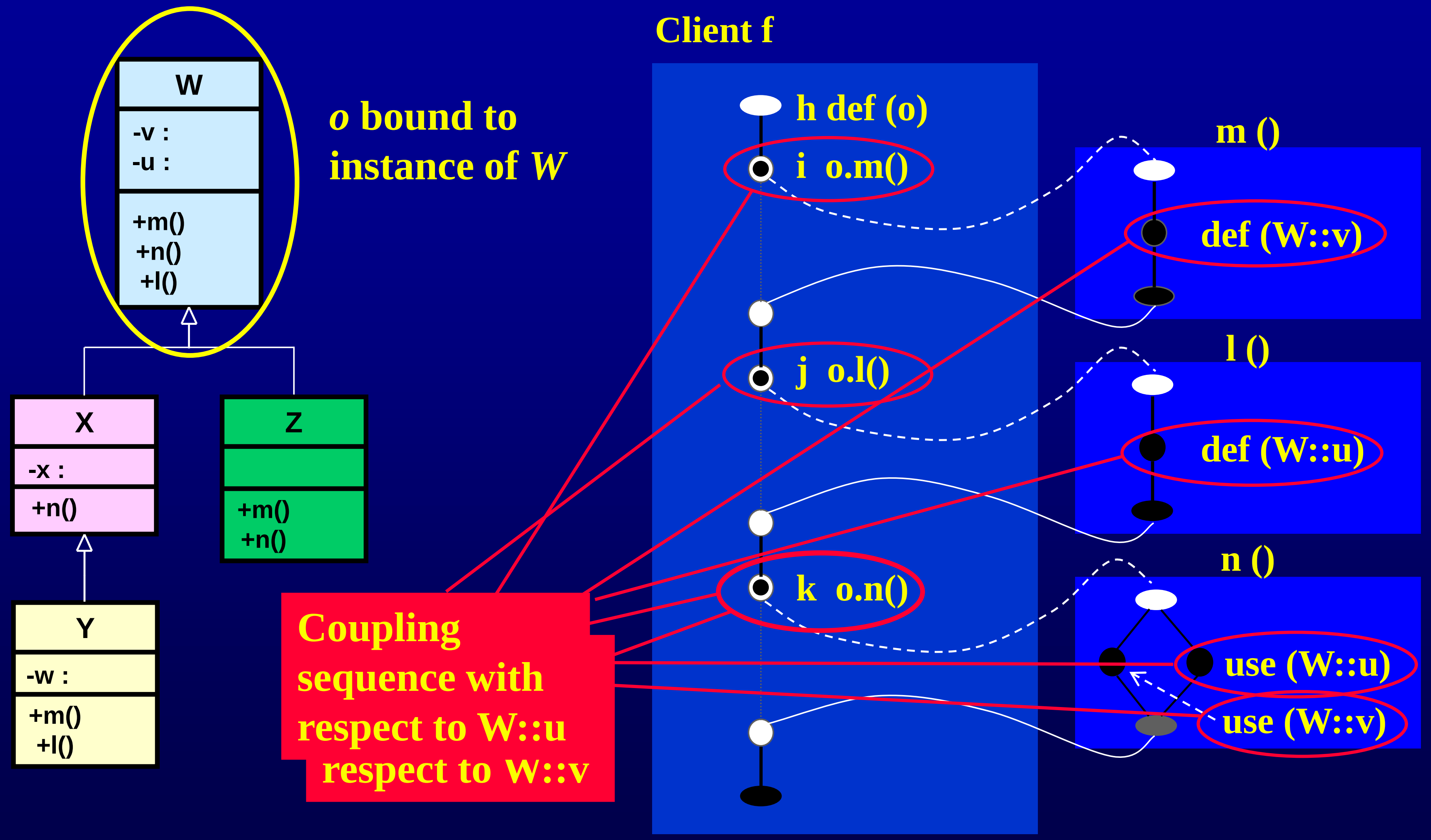
Polymorphic Call Set

Set of methods that can potentially execute as result of a method call through a particular instance context

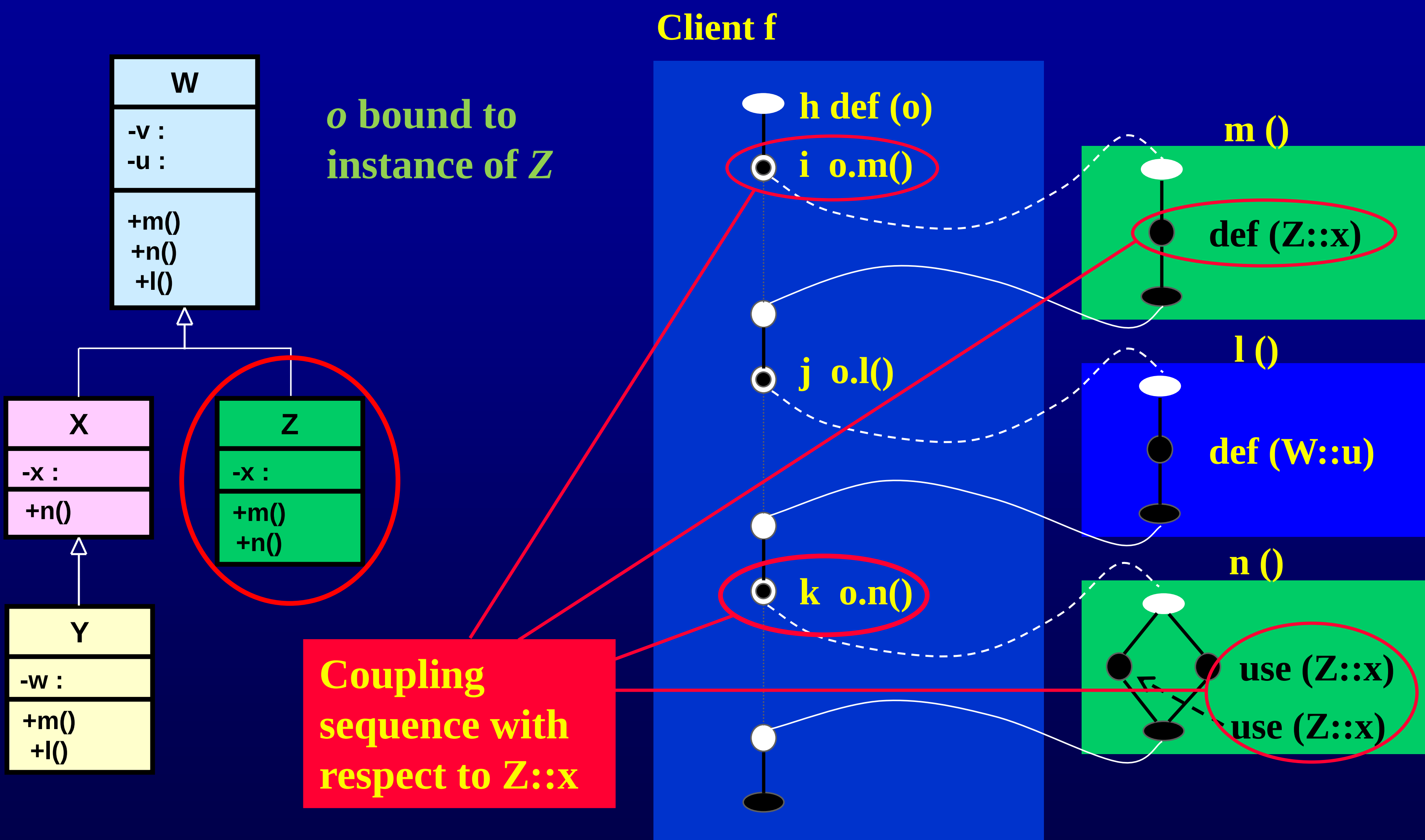
pcs (o.m) = {*W::m, Y::m, X::m*}

```
 public void f ( W o )
{
      …
j     o.m();
      …
l     o.l();
      …
k      o.n();
  }
```

Example Coupling Sequence



Example Coupling Sequence (2)



Section 7.1 Outline

1. Overview
2. Types of Object-Oriented Faults
    1. Example
    2. The Yo-Yo Graph and Polymorphism
    3. Categories of Inheritance Faults
    4. Testing Inheritance, Polymorphism and Dynamic Binding
    5. **Object-Oriented Testing Criteria**

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**

All-Coupling-Sequences

All-Coupling-Sequences (ACS) : For every coupling sequence $S_{j,k}$ 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**
- **Does not consider inheritance and polymorphism**
- **Should be covered during integration testing**

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
  - Only classes that override the antecedent or consequent methods are considered
- At least one test for every type the object can bind to
  - Not consider the state interactions that can occur when multiple coupling variables may be involved. Thus, some definitions or uses of coupling variables may not be covered during testing
- Test with every possible type substitution that can occur in a given coupling context (coupling sequence should be tested with every type substitution)

All-Coupling-Defs-Uses

<u>All-Coupling-Defs-Uses (ACDU)</u> : For every <u>coupling variable</u> *v* in each coupling $S_{j,k}$ of *t*, there is a coupling path induced by $S_{j,k}$ such that *p* is a sub-path of the execution trace of *f(t)* for at last one test case *t*

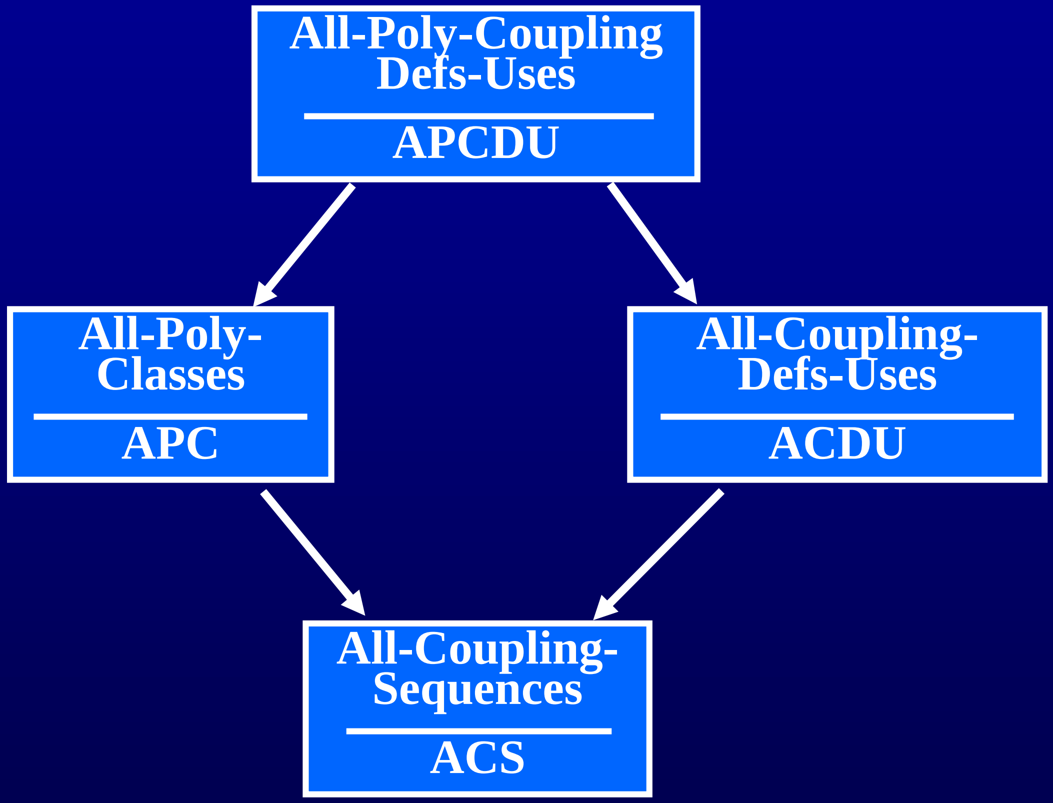
- **Every last definition of a <u>coupling variable</u> reaches every first use**
- **Does not consider inheritance and polymorphism**

All-Poly-Coupling-Defs-and-Uses

All-Poly-Coupling-Defs-and-Uses (APCDU) : 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 (APC and ACDU)
- Handles inheritance and polymorphism
- Takes definitions and uses of variables into account

OO Coverage Criteria Subsumption



Conclusions

- 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