Graph Coverage for Design Elements

CS 3250
Software Testing

[Ammann and Offutt, "Introduction to Software Testing," Ch. 7.4]

Overview

- Use of data abstraction and OO software
  - → Emphasis on modularity and reuse
  - → Complexity in design

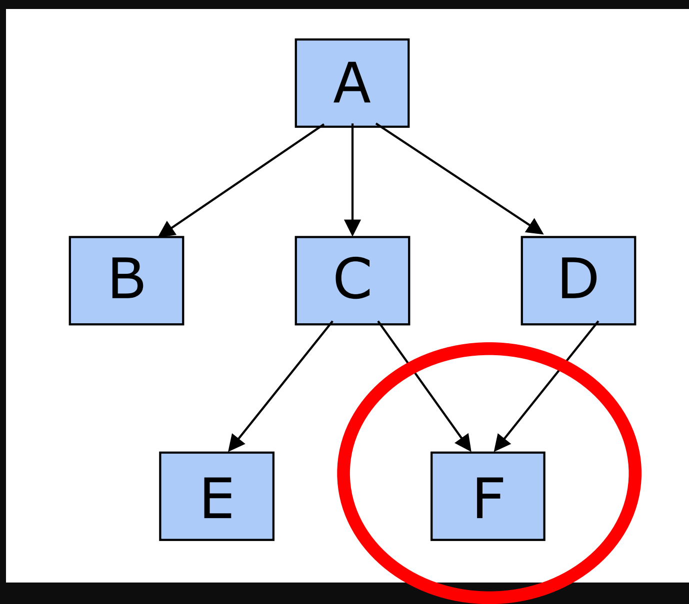
- Testing design of software becomes more important than in the past

- Graphs for the design are based on couplings between software components
  - Couplings = dependency relations between components
    - Faults in one component (unit) may affect the coupled component (unit)

- Most test criteria for design require that connections among components be visited

Call Graph

- The most common graph for structural design testing
  - Nodes represent methods (or units)
  - Edges represent method calls



Node coverage (method coverage)
- Call every method at least once

Edge coverage (call coverage)
- Execute every call at least once

Node `F` must be called at least twice, once from `C` and once from `D`

Node and edge coverage of class call graphs often do not work well because individual methods might not all call each other

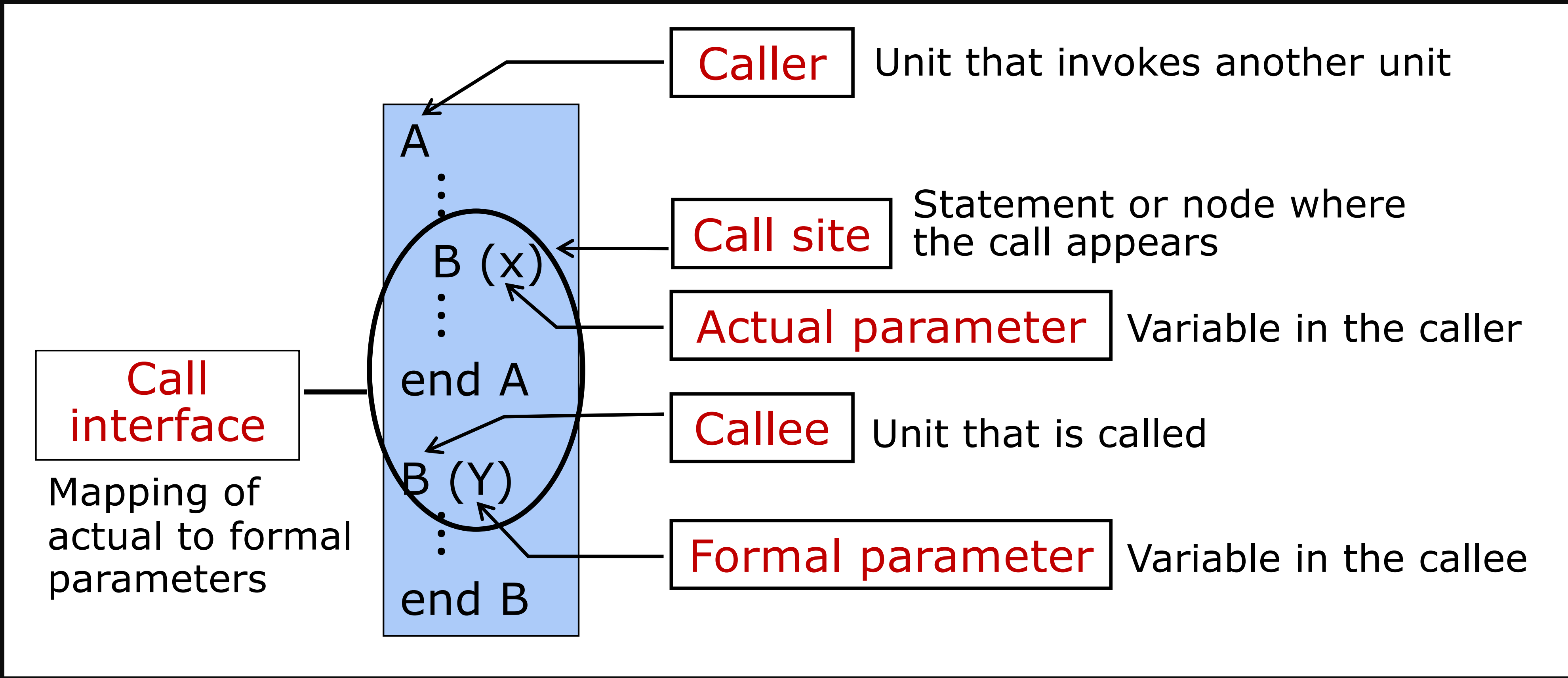
Data Flow for Design Elements

- Control connections among design elements are not effective at finding faults
- Data flow coverage can be applied to call graphs
- Data flow couplings are often more complex than control flow couplings
  - When values are passed, they change names
  - Many different ways to share data
  - Analysis of defs and uses can be difficult
    - Which uses a def can reach

When software gets complicated,
that indicates a source of faults

Call Site Example

The primary issue is where the defs and uses occur



The criteria require execution from definitions of actual parameters through calls to uses of formal parameters

Data Flow Couplings for Call Sites

Types of couplings between caller and callee units

Parameter coupling	Defined by parameter passing from caller to callee
Return value coupling	Defined by return value passing from callee to caller
Shared data coupling	Defined by shared variables between caller and callee
External device coupling	Defined by shared use of a device by caller and callee (e.g., a file)

Inter-Procedural DU Pairs

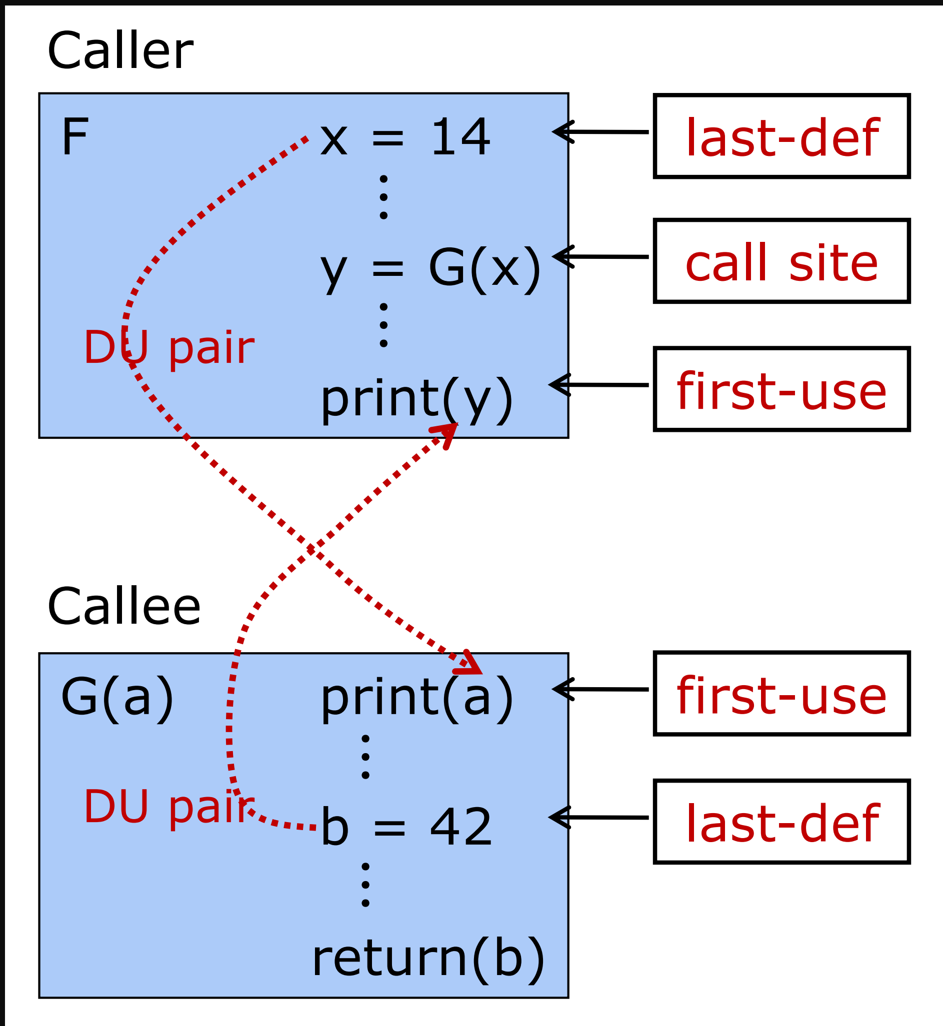
- To achieve confidence in the interfaces between integrated program units, variables defined in caller unit must be appropriately used in callee

- For a variable *x* that expresses a coupling between caller and callee

  - Last-def (of x) — Set of locations (or nodes) that last define *x* (def-clear) in one of the units (caller or callee)

  - First-use (of x) — Set of locations (or nodes) that first use *x* in the other unit (def-clear and use-clear path from the call site to the nodes)

Inter-Procedural DU Pairs Example



Parameter coupling

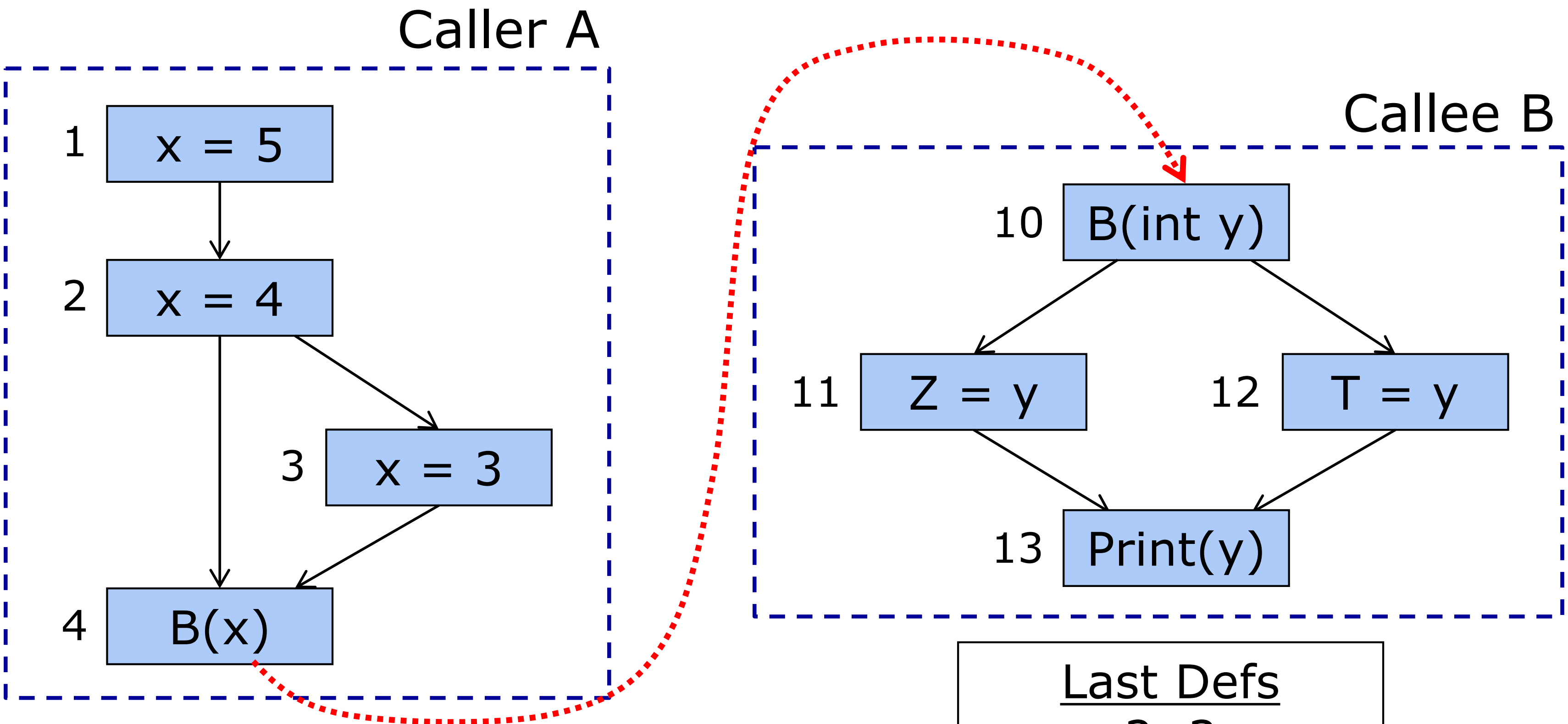
- last-def of x: set of locations in caller that last define a call param x just before the call site
- first-use of x: set of locations in callee that first use a param a after the entry point

Return value coupling

- last-def of b: set of locations in callee that last define return result
- first-use of b: set of locations in caller that first use the result of the call after the call site

last-defs and first-uses define coupling du-pairs

Inter-Procedural DU Pairs Example



Coupling DU Pairs

(A, x, 2)—(B, y, 11)
(A, x, 2)—(B, y, 12)
(A, x, 3)—(B, y, 11)
(A, x, 3)—(B, y, 12)

Last Defs

2, 3

First Uses

11, 12

coupling du-path = path from a last-def to a first-use

Coupling DU-Paths and Coverage Criteria

- A coupling du-path for *x* is a path from a last-def of *x* to a first-use of *x*

- Data flow coverage criteria for coupling du-paths:
  - All-Coupling-Defs Coverage (~All-Defs Coverage)
    - For each last-def of x, cover at least one first-use
  - All-Coupling-Uses Coverage (~All-Uses Coverage)
    - For each last-def of x, cover every first-uses
  - All-Coupling-DU-Paths Coverage (~All-DU-Paths Coverage)
    - For each last-def of x, cover all paths to every first-uses

Example Quadratic

Shared variables

```java
1  // Program to compute the quadratic root for two numbers
2  import java.lang.Math;
3  public class Quadratic
4  {
5     private static double Root1, Root2;
6     public static void main (String[] argv)
7     {
8        int X, Y, Z;
9        boolean ok;
10       if (argv.length == 3)
11       {
12          X = Integer.parseInt (argv[0]);
13          Y = Integer.parseInt (argv[1]);
14          Z = Integer.parseInt (argv[2]);
15       }
16       else
17       {
18          System.out.println ("Inputs not
19          X = 8;
20          Y = 10;
21          Z = -33;
22       }
```

last-def

Call site

```java
23       ok = Root (X, Y, Z);
24       if (ok)
25          System.out.println
26             ("Quadratic: Root 1 = " + Root1 + ", Root 2 = " + Root2);
27       else
28          System.out.println ("No solution.");
29    }
30
31    // Finds the quadratic root, A must be non-zero
32    private static boolean Root (int A, int B, int C)
33    {
34       double D;
35       boolean Result;
36       D = (double)(B*B) - (double)(4.0*A*C);
37       if (D < 0.0)
38       {
39          Result = false;
40          return (Result);
41       }
42       Root1 = (double) ((-B + Math.sqrt(D)) / (2.0*A));
43       Root2 = (double) ((-B - Math.sqrt(D)) / (2.0*A));
44       Result = true;
45       return (Result);
46    } // End method Root
47 } // End class Quadratic
```

first-use

first-use

last-def

(Root(), Root1, 42) – (main(), Root1, 26)

(Root(), Root2, 43) – (main(), Root2, 26)

(Root(), Result, 39) – (main(), ok, 24)

(Root(), Result, 44) – (main(), ok, 24)

Example: Quadratic Coupling DU-Pairs

Pairs of locations: method name, variable name, statement

(main(), X, 12) – (Root(), A, 36)
(main(), Y, 13) – (Root(), B, 36)
(main(), Z, 14) – (Root(), C, 36)
(main(), X, 19) – (Root(), A, 36)
(main(), Y, 20) – (Root(), B, 36)
(main(), Z, 21) – (Root(), C, 36)

(Root(), Root1, 42) – (main(), Root1, 26)
(Root(), Root2, 43) – (main(), Root2, 26)
(Root(), Result, 39) – (main(), ok, 24)
(Root(), Result, 44) – (main(), ok, 24)

Summary

- Call graphs are common and very useful ways to design integration tests
- Inter-procedural data flow is relatively easy to compute and results in effective integration tests
- The ideas of coupling data flow for OO software and web applications are preliminary and have not been used much in practice

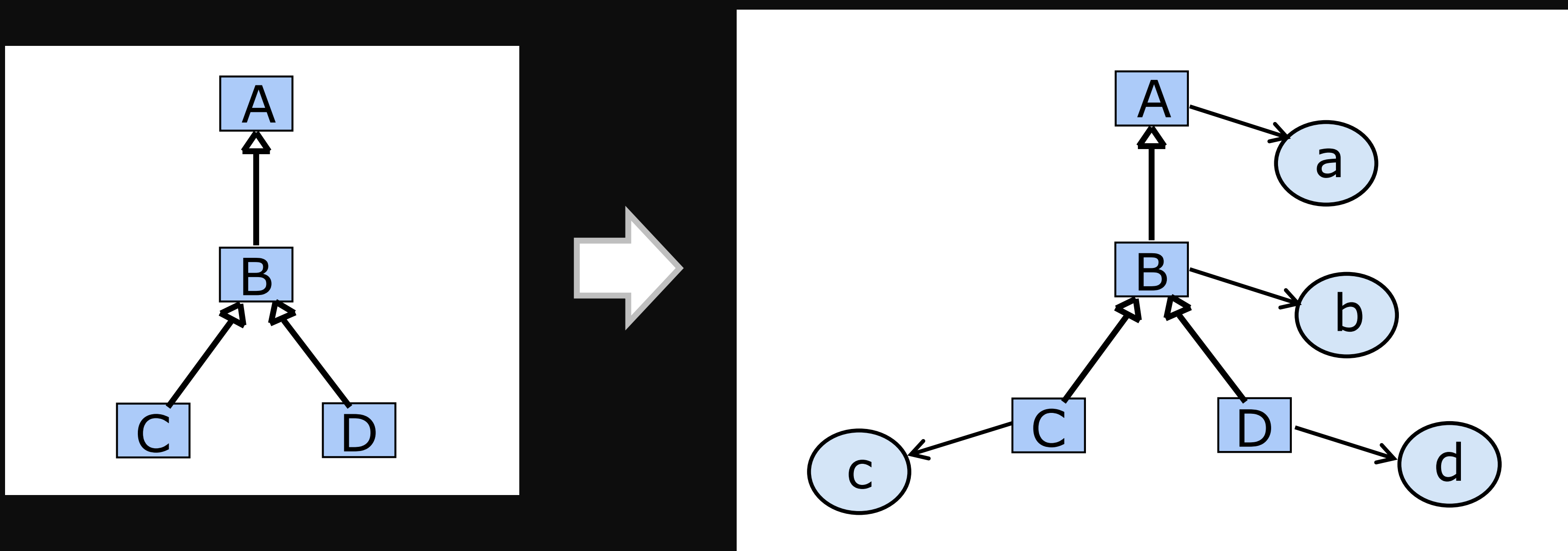
Extra Slides

If you may be interested in
graph coverage for inheritance

(will not be tested)

Inheritance and Polymorphism

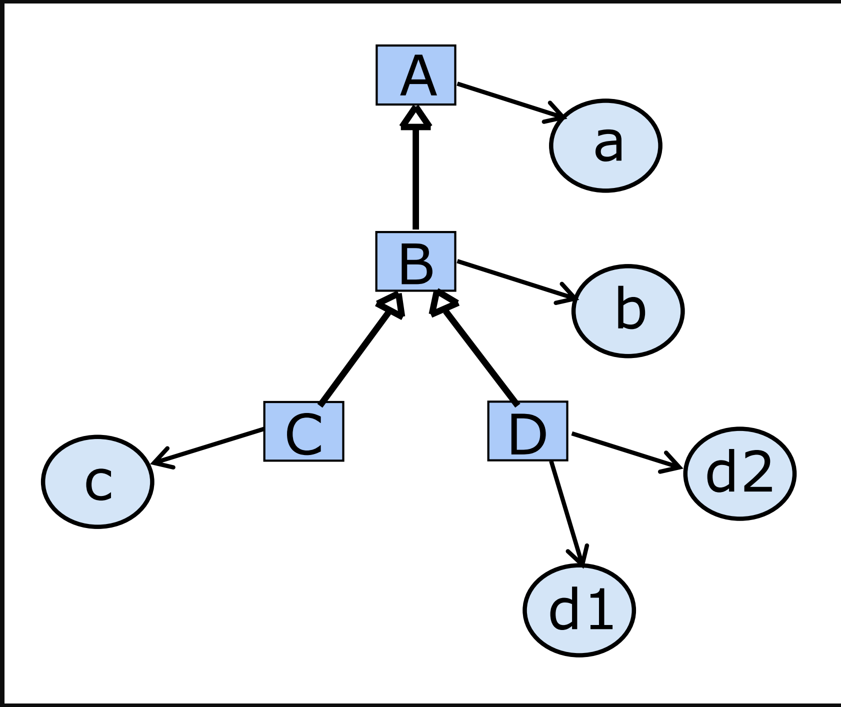
- The most obvious graph for testing these OO features is the inheritance hierarchy
- Classes are not executable → the graph is not directly testable. To test the inheritance hierarchy graph, we need to instantiate objects for the classes



Ideas of graph coverage for inheritance and polymorphism are preliminary and have not been widely used [noted by Offutt and Ammann]

Coverage on Inheritance Graph

- Node coverage: create at least one object for each class
  - Weak because there is no execution
- Thus, we create an object for each class and then apply call coverage (execute every call at least once)



OO call coverage

- Cover each node in the call graph of an object instantiated for each class in the inheritance hierarchy graph

All object call coverage

- Cover each node in the call graph of every object instantiated for each class in the inheritance hierarchy graph