SYSC 4101 / 5105

Integration Testing
Part II—Integration Orders

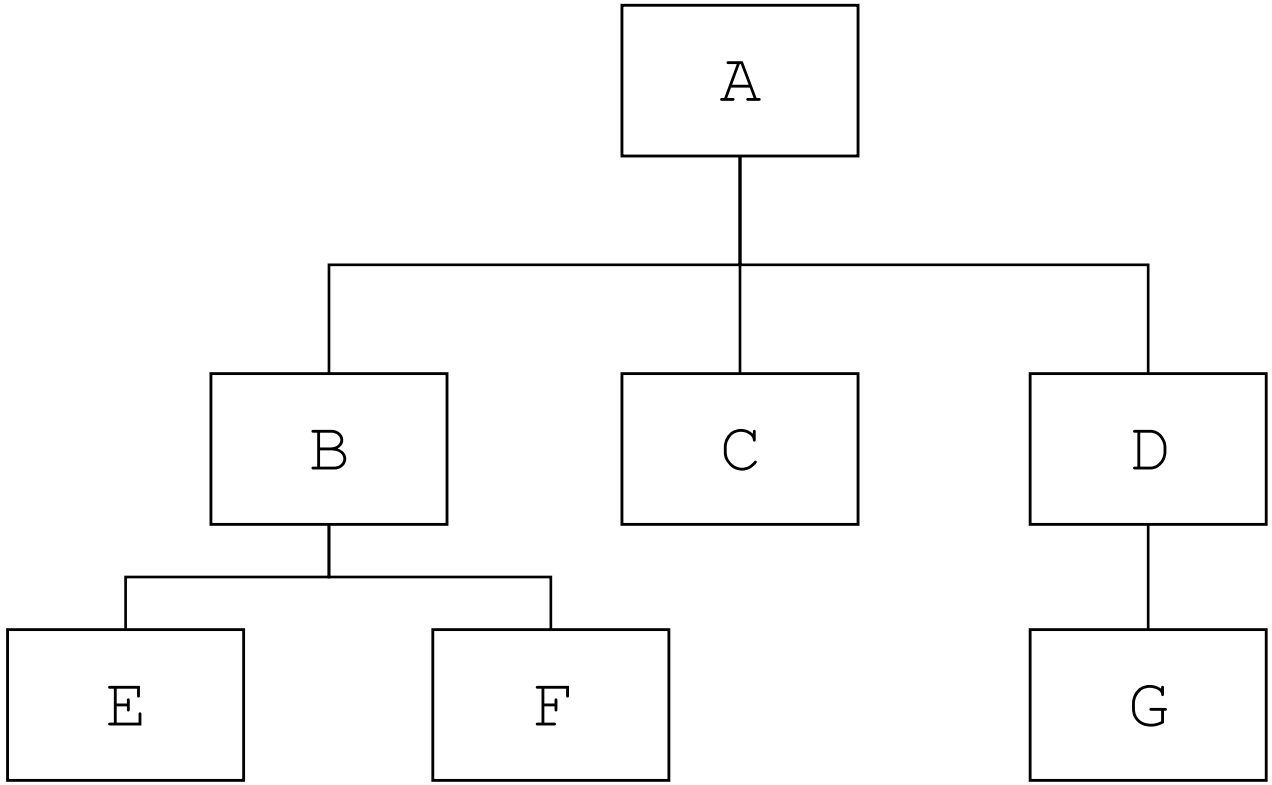
Preliminary Remarks I

- When components have been tested to a satisfactory level, we combine them into working systems
- Components are still likely to be faulty as test stubs and drivers, used during unit testing, are only approximations of the components they simulate
- In addition, possible interface faults
  - e.g., assumptions about parameter semantics
- Order of integration? Drivers and stubs are expensive and the order affects the cost of testing

Preliminary Remarks II

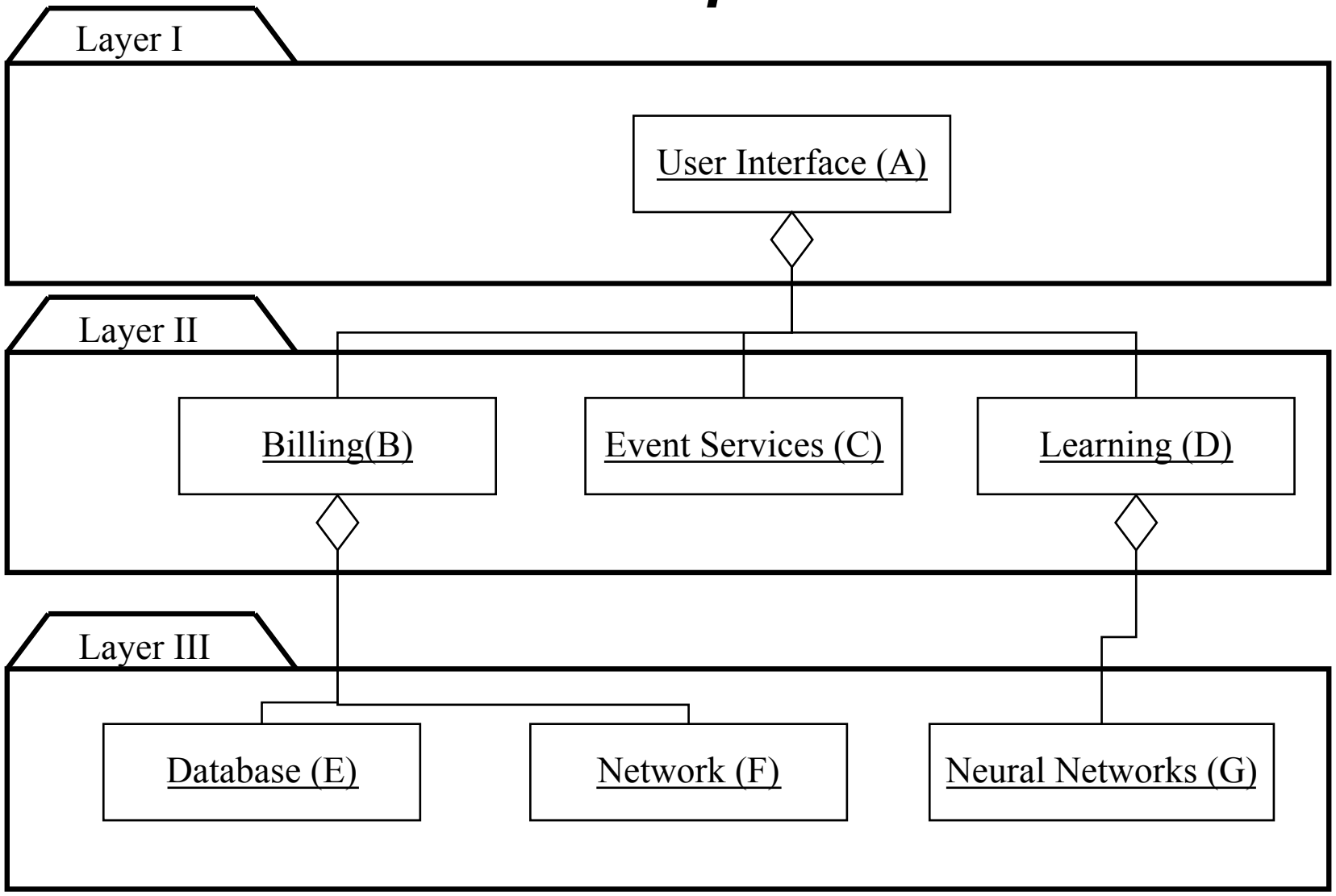
- Integration is planned so that when a failure occurs, we have some idea of what caused it
- Development is not sequential but activities are overlapping: some components may be in coding, unit testing, and integration testing
- Integration strategy affects the order of coding, unit testing, and the cost and thoroughness of testing

Generic Example



Example of a hierarchy of modules, defined by usage dependencies between modules

Example


Layer I
User Interface (A)
Layer II
Billing(B)
Event Services (C)
Learning (D)
Layer III
Database (E)
Network (F)
Neural Networks (G)

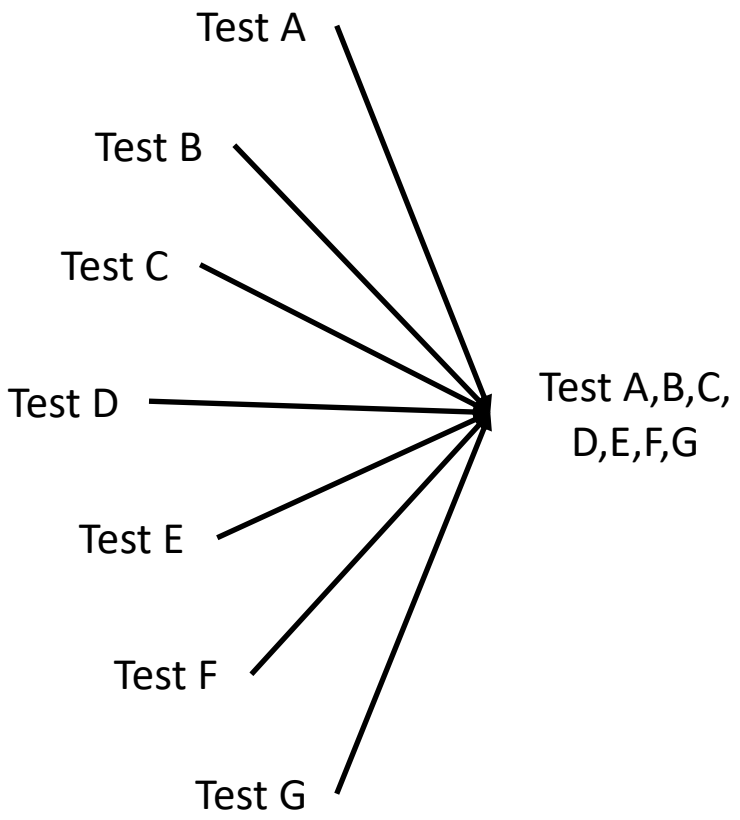
Big Bang Integration

- All components are brought together at once into a system and tested as entire system

☹ Difficult to find the root cause of any failure

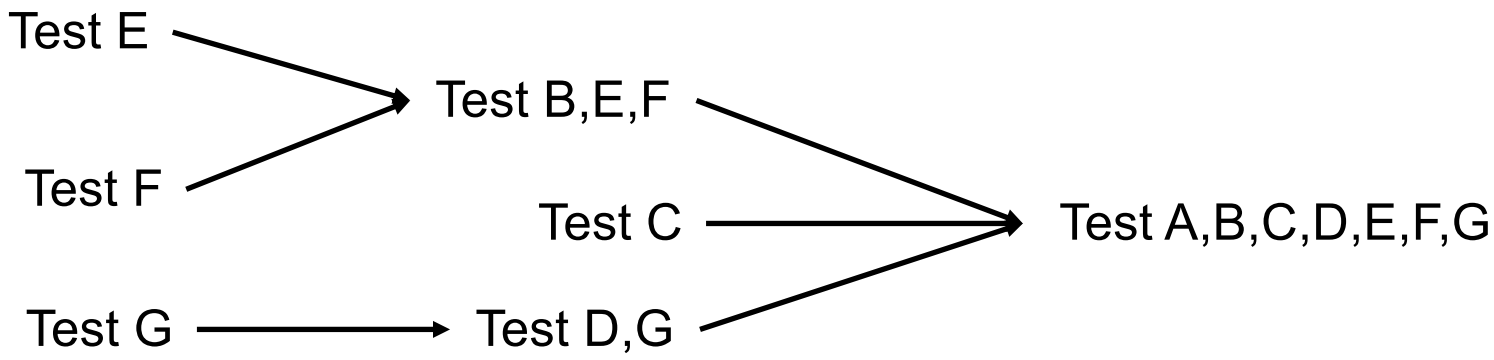
☹ Wait for all components to be ready

☹ Not recommended



Bottom-Up Integration I

- Each component at the lowest of the hierarchy are tested first (e.g., E, F, and G)
- Context: Design is based on functional decomposition
- If a problem happens when testing B (with E and F), the heuristic says that its cause is either in B, or the interface between between B and E or F, since E and F have passed their tests prior to integration.



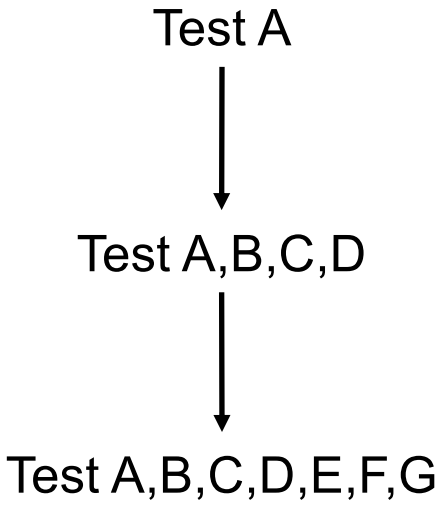
Bottom-Up Integration II

- ☺ Need to develop component *drivers* but no *stubs*
- ☺ Strategy helps us isolate faults more easily, in particular interface faults
- ☺ Convenient when many general purpose utility routines, invoked often by others, at the lowest levels.

- ☹ Problem: top level components may be more important (major system activities) but last to be tested, e.g., user interface components
- ☹ Faults in top level sometimes reflect faults in design, e.g., subsystem decomposition – have to be corrected early

Top-Down Integration I

- Reverse of Bottom-up
- When components that are not yet tested are needed, we use “stubs”, which simulate the activity of missing components



Top-Down Integration II

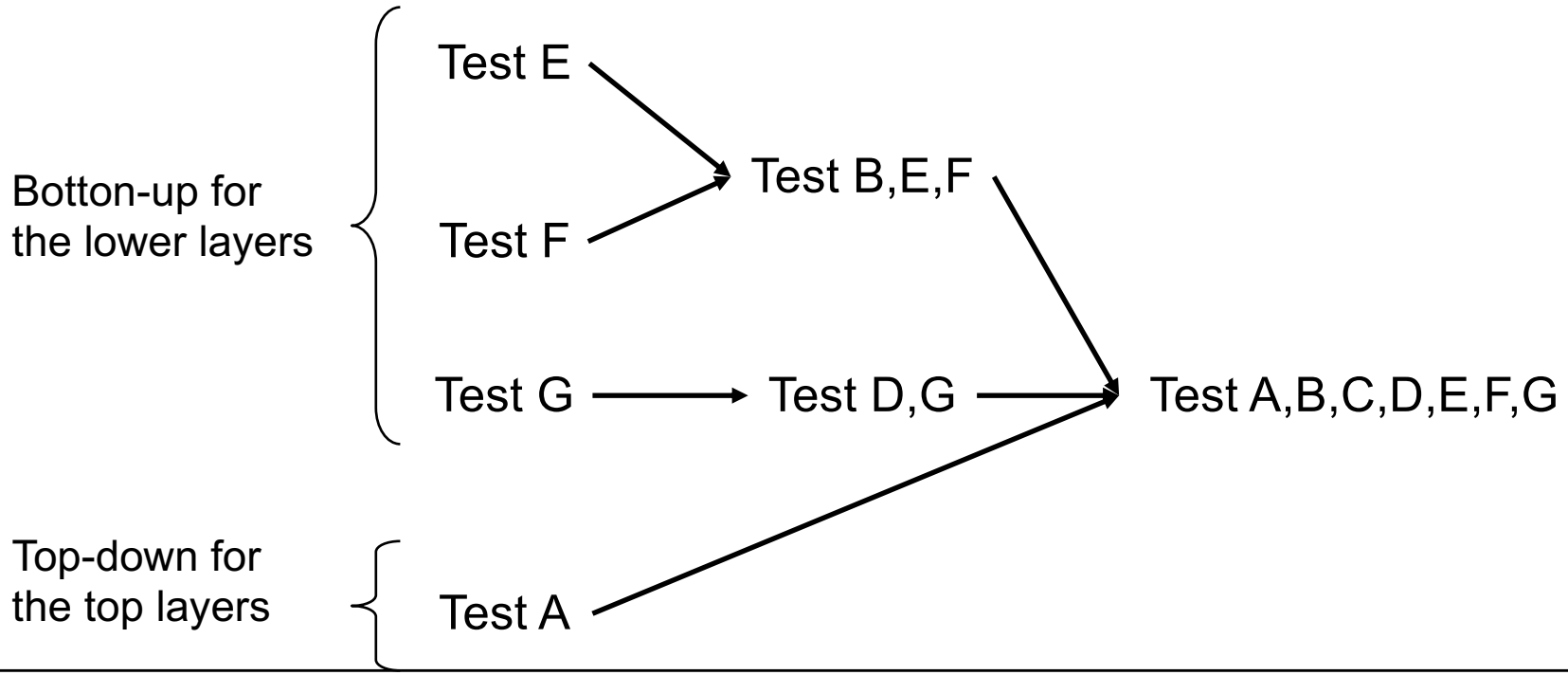
☺ "Controlling" components are tested first, e.g., user interface

☹ Fewer drivers needed (some reuse likely), but stubs are time consuming and error prone

☹ Writing stubs may turn out to be complex and their correctness may affect the validity of the test

Sandwich Integration I

1. Choose a target layer (the middle one in our example)
2. Top down approach used in top layers
3. Bottom-up approach in lower layers



Sandwich Integration II

- ☺ Allows to test general-purpose utilities' correctness at the beginning
- ☺ No need for stubs for utilities
- ☺ Allows integration testing of top components to begin early
- ☺ Tests "control" component early (top layer)
- ☺ No need for test drivers of top layer

- ☹ But does not test thoroughly the individual components of target layer
  - B and D, as well as C, may not be tested as throughly

Integration Strategies - Overview

Criteria for comparison:

- Time to "working" system:
  - System you can demonstrate (e.g., to future users)
- Ability to test paths:
  - Ensure path coverage of components.
  - Top-down is hard because it is difficult to control the inputs of lower level components through higher level ones. Sandwich alleviates a bit the problem as the top-down hierarchy is more shallow.
- Ability to plan and control:
  - Top-down is hard because of the inherent instability of lower level components, which may invalidate the integration testing of higher components if their spec change. Recall they may not exist yet when integration testing of higher level components start. Sandwich inherits the "hard" score for the same reason.

Integration Strategies - Overview

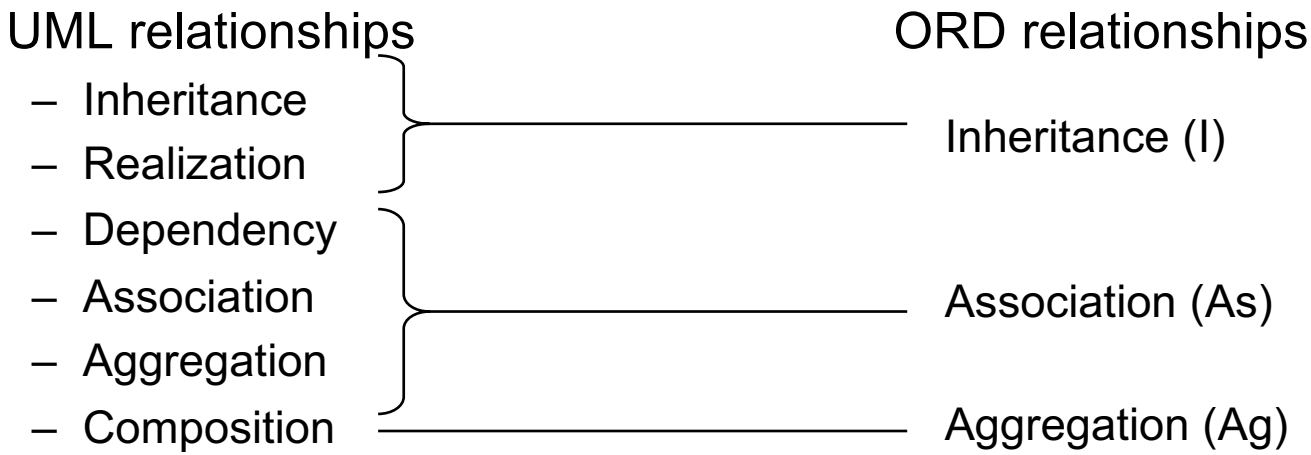
	Bottom-up	Top-down	Big-Bang	Sandwich
Integration	Early	Early	Late	Early
Time to working system	Late	Early (thanks to stubs)	Late	Early (thanks to stubs)
Drivers	Many	Few	Many	Medium
Stubs	No	Many	Many	Few
Ability to test paths	Easy	Hard	Easy	Medium
Ability to plan and control	Easy	Hard	Easy	Medium

Integration Strategies and OO Software

- It is not always feasible to construct a stub that is simpler than the actual piece of code it simulates
- Stub generation cannot be automated because it requires an understanding of the semantics of the simulated functions
- The fault potential for some stubs may be higher than that of the real function
- Minimizing the number of stubs to be developed should result in drastic savings

- Class dependency graphs usually do not form simple hierarchies but complex networks
  - Not trees (with a bottom and an up) but graphs (cycles)
- Class dependency graphs have specific features
  - Different kinds of relationships (e.g., association vs. composition vs. generalization)
  - Polymorphism
- Specific Graph: Object Relation Graph (ORD)

Mapping: UML Class Diagram → ORD



Rational: working with three kinds of relations is an abstraction

- I is for subtyping/subclassing
- Ag is for associations that entail the most coupling (highest level of dependencies)
- As is for associations that entail the lowest coupling

Mapping: Source Code → ORD

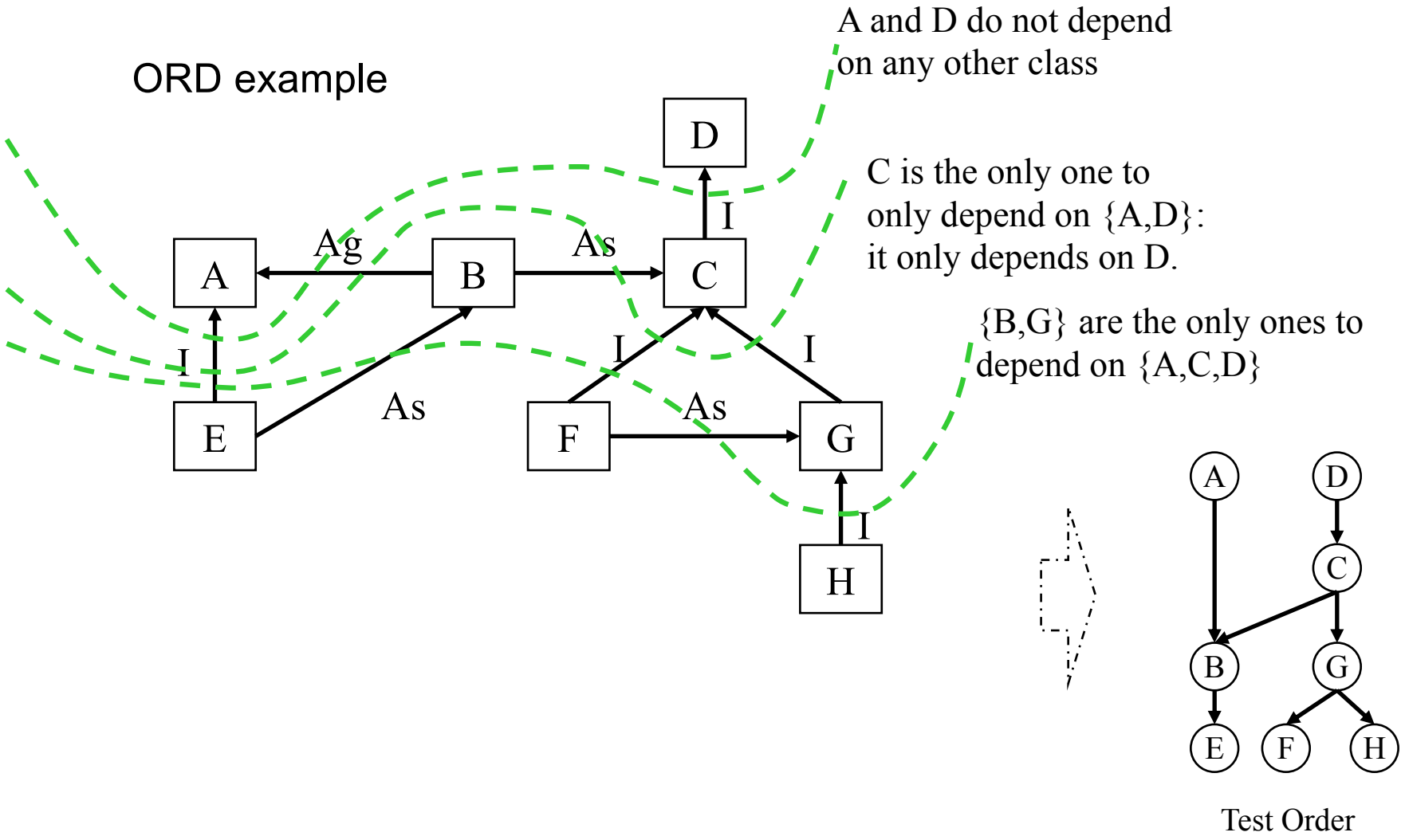
We have to find Inheritance, Association and Aggregation (as used in the ORD) from the source code.

- Inheritance: there are specific constructs in OO programming languages
- Association and Aggregation:
  - Each time there is an attribute there is an association
    - Manual analysis is required to identify aggregation, i.e. association with tight coupling.
    - Use of collection data structures (e.g., class Vector)?
  - Each time there is a method level collaboration (and no attribute with the same type) there is an association in the ORD.

Strategies

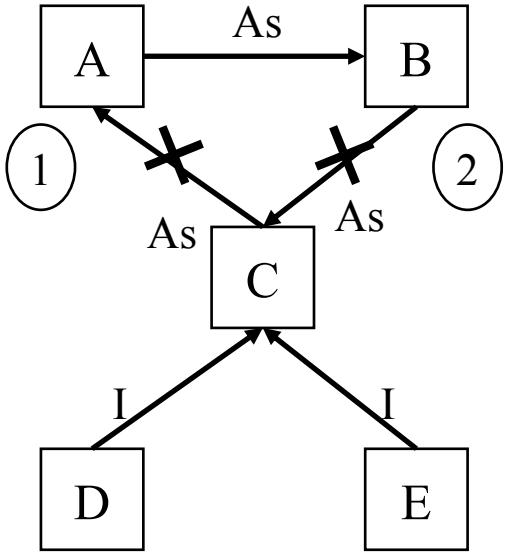
- There is not cycle in the ORD
  - Bottom-up
    - Start with classes that do not depend on other classes
    - Continue with classes that only depend on classes already integrated
- There are cycles in the ORD
  - Find a way to break cycles
    - Temporarily remove relation(s), thus leading to stubs
  - How to break?
    - Based on topology of graph
    - Based on coupling measurement entailed by relations
  - Then, apply bottom-up strategy
- Test Order
  - Provide the tester with a road map for conducting integration testing
  - Desirable order is the one that minimizes the number of stubs

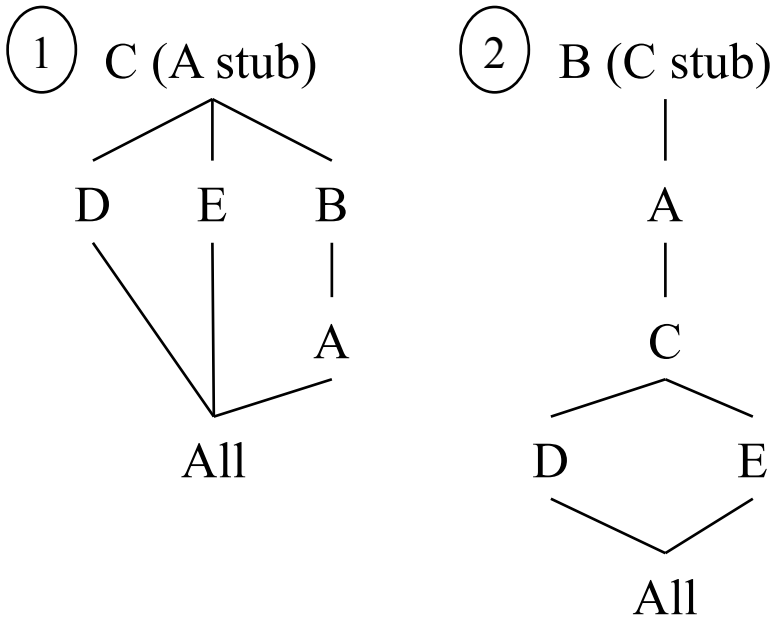
Bottom Up from ORD



Cycles

- Depending on where you break, the resulting test order is very different





All = broken cycle is restored and all classes are tested (the stub is replaced by the class)

Breaking Cycles

Automated technique to break cycles

We assign each class in the class diagram (or ORD)

1. A *major* level number

   Major level numbers are assigned based on inheritance and aggregation dependencies only.

   No cycles are possible then, and topological sorting can be used.

2. A *minor* level number

   Then, minor level numbers are assigned, within each major level, based on association dependencies only.

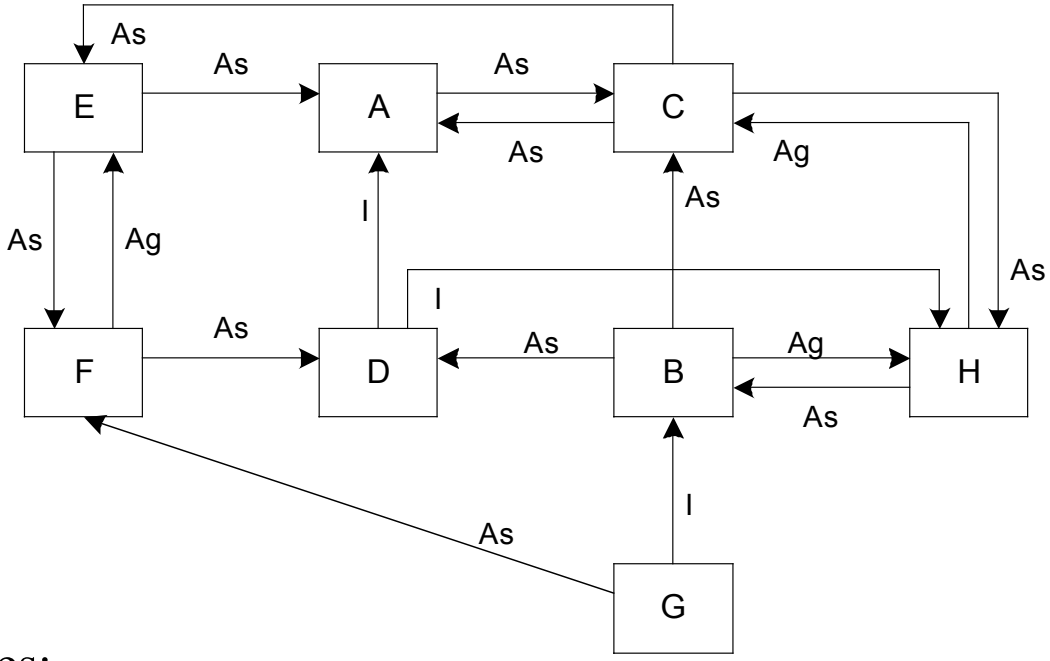
   Cycles may appear here and a weight function, *weight(d_i)* for each association d_i, is used to select what association to delete (and stub):

   weight(d_i) = *the number of the incoming dependencies of the head node of d_i*
   +
   the number of outgoing dependencies of the tail node of d_i.

   The rationale is that the higher the weight the more likely breaking a dependency will break a larger number of cycles.

Major and minor numbers are used to derive the integration test order

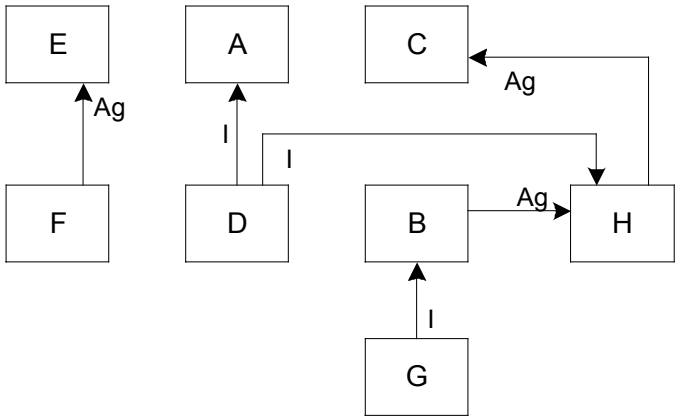
Example Cyclic ORD



ORD with cycles:

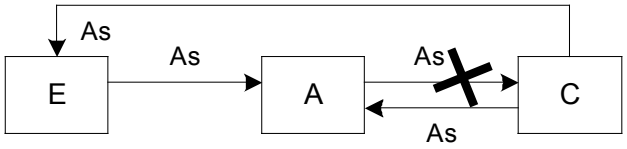
- E-F-E, B-H-B, A-C-A, C-H-C
- E-A-C-E, C-H-B-C, D-H-B-D
- …
- D-H-B-C-E-F-D-A

Major and Minor Level Numbers (I)



Class	Major Level
E, A, C	1
F, H	2
D, B	3
G	4

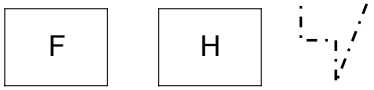
In major level 1



weight(E,A)=2
weight(A,C)=4
weight(C,A)=2
weight(C,E)=2

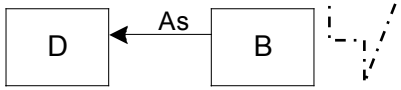
Class	Minor Level
A	1
E	2
C	3

In major level 2



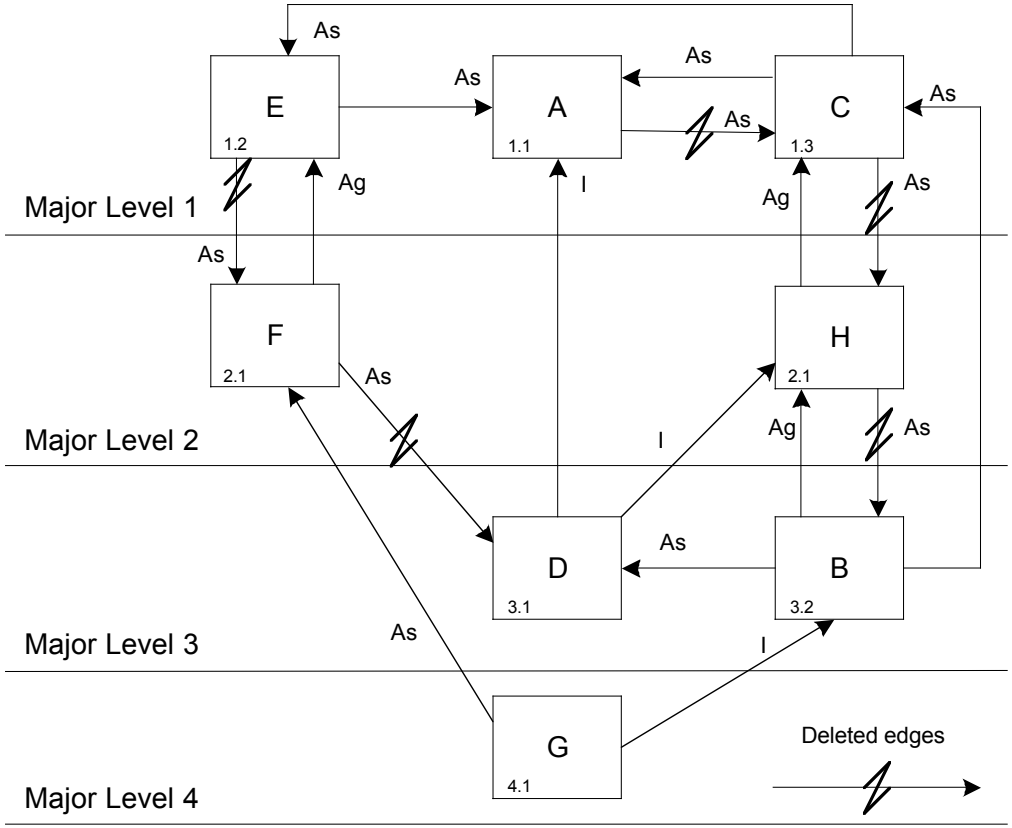
Class	Minor Level
F	1
H	1

In major level 3



Class	Minor Level
D	1
B	2

Major and Minor Level Numbers (II)



- Five classes need to be stubbed (B, C, D, F, H)
  - 5 stubs
- Result sub-optimal:
  - Breaking F-D is not necesary