Overview

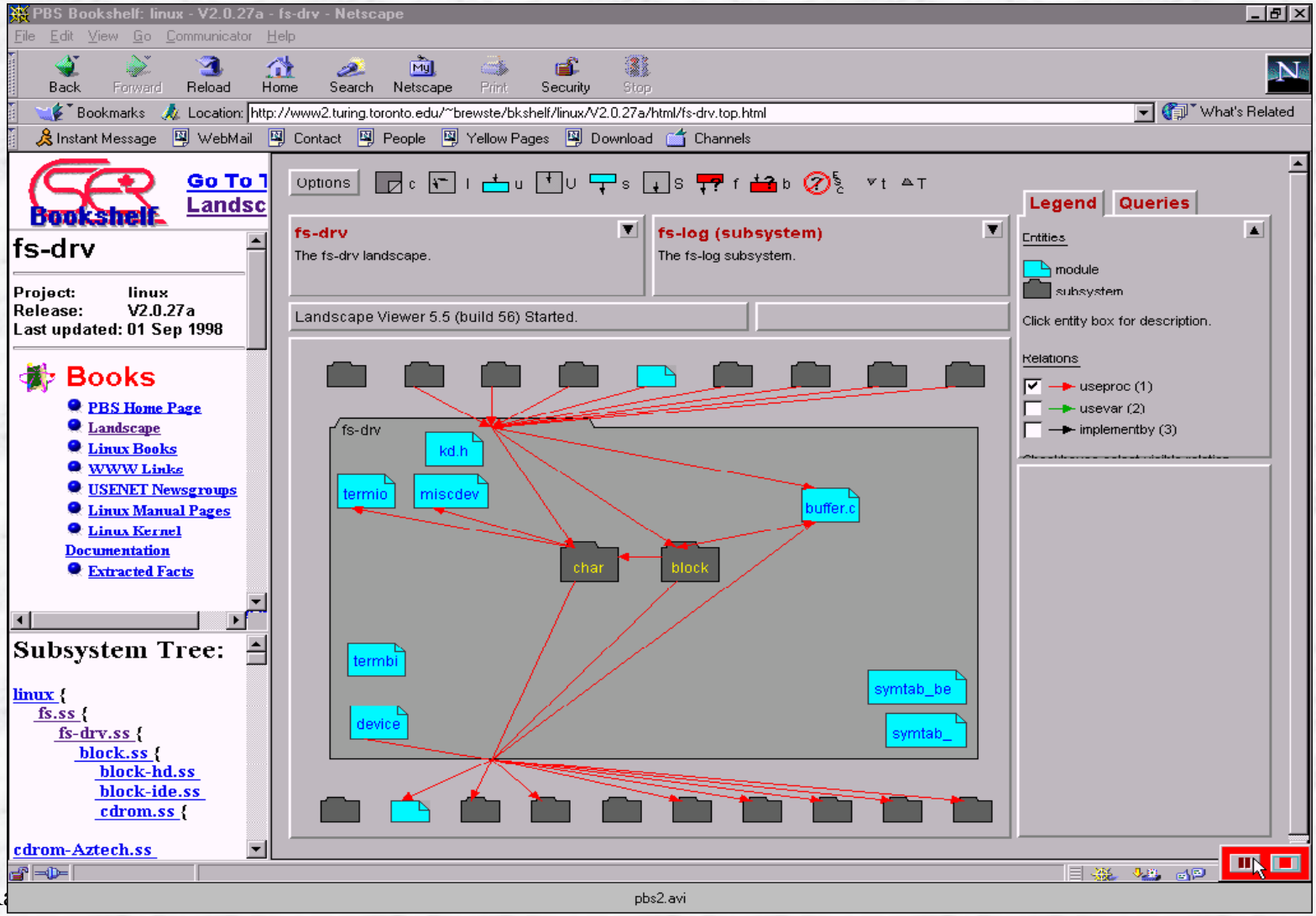
- Integration Testing
  - Decomposition Based Integration
  - Call Graph Based Integration
  - Path Based Integration
  - Discussion

Ref: "*Object-Oriented Software Engineering – Using UML, Patterns and Java*", Bernd Bruegge, Allen H. Dutoit and *Software Testing A Craftsman's Approach* 2nd edition, Paul C. Jorgensen

Functional Decomposition

- Functional Decomposition is defined as a method in which a problem is broken up into several independent task units, or functions, which can be run either sequentially and in a synchronous call-reply manner or simultaneously on different processors
- This method is used during planning, analysis and design, and creates a functional hierarchy for the software
- A model by which we represent functional decomposition in a system is the *call graph* or *call tree*
- A *call graph* is a directed labeled graph where nodes represent functional units and edges represent call events or specific requests of resources
- If we exclude back edges then we have a *call tree – decomposition tree*

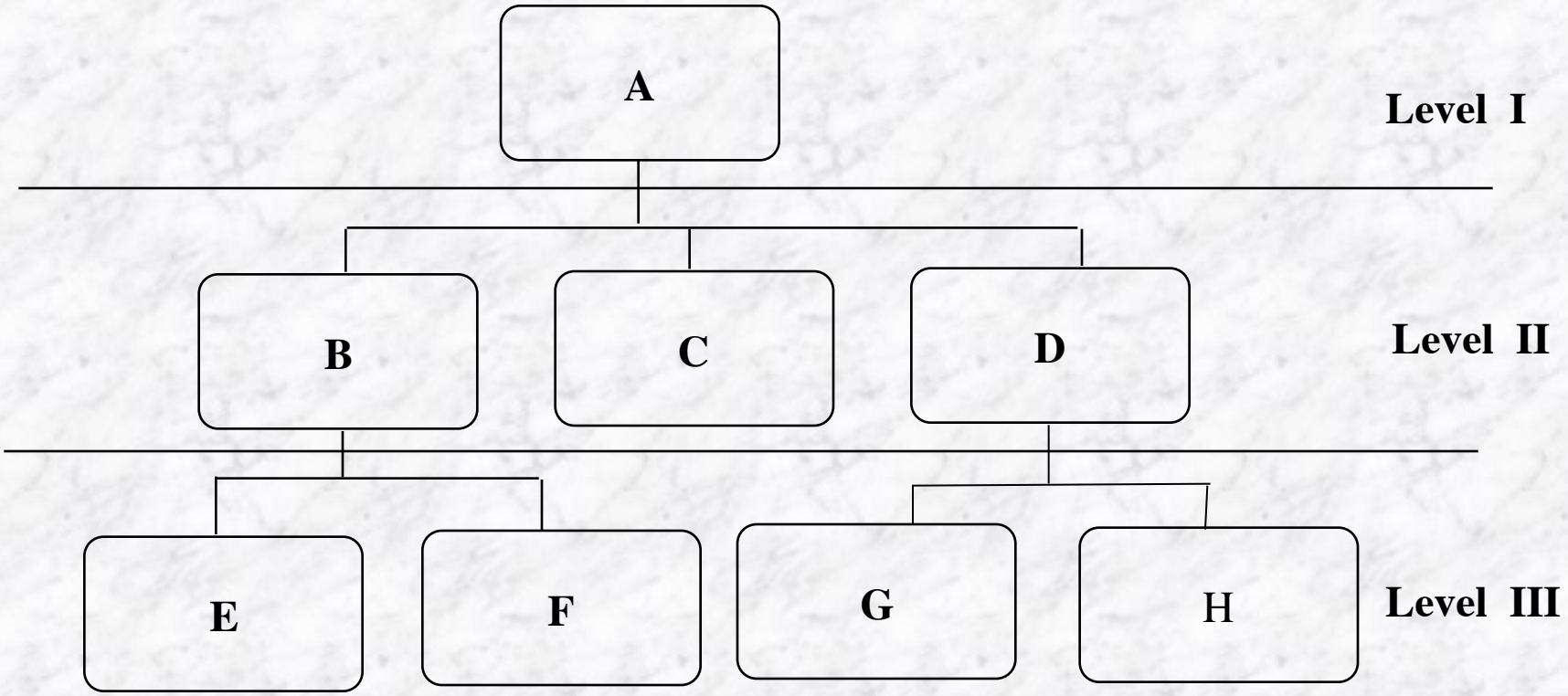
Example Call Graph



Integration Testing Strategy

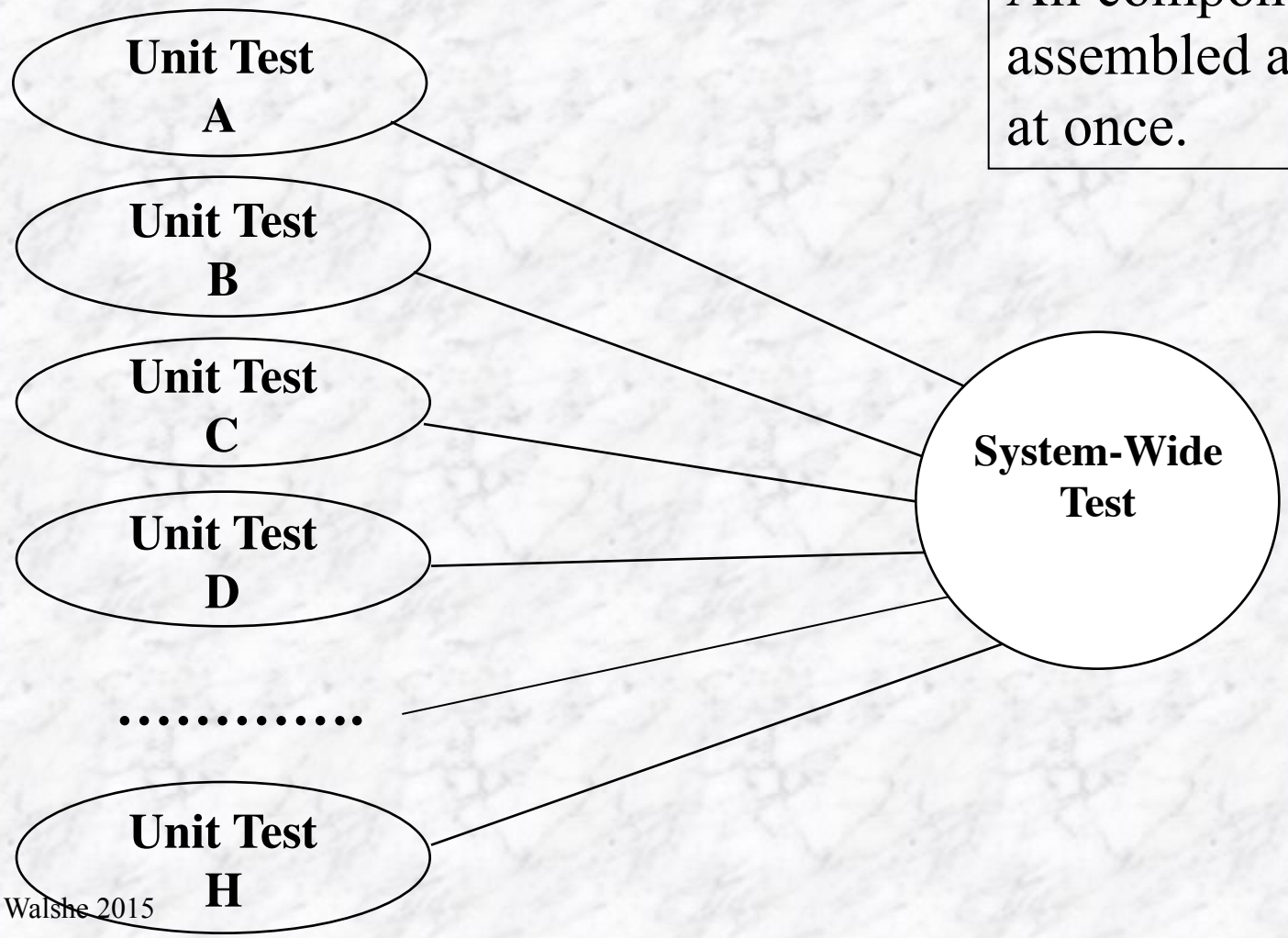
- The entire system is viewed as a collection of subsystems (sets of classes) determined during the system and object design
- The order in which the subsystems are selected for testing and integration determines the testing strategy
  - Big bang integration (Nonincremental)
  - Bottom up integration
  - Top down integration
  - Sandwich testing
  - Variations of the above
- For the selection use the system decomposition from the System Design

Three Level Call Hierarchy



Big-Bang Integration Testing

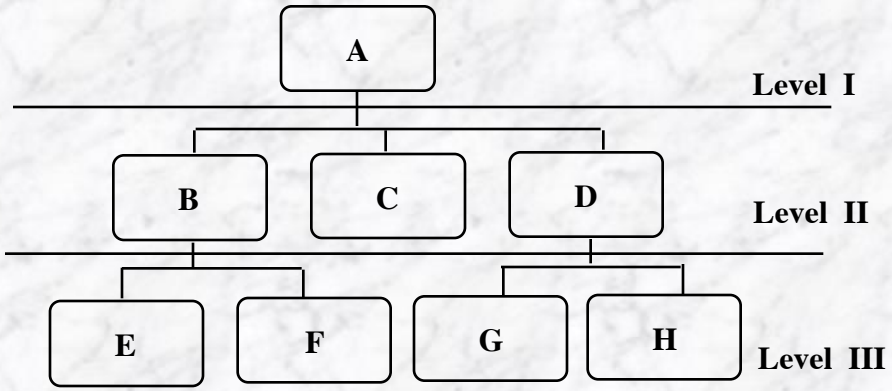
All components are assembled and tested at once.

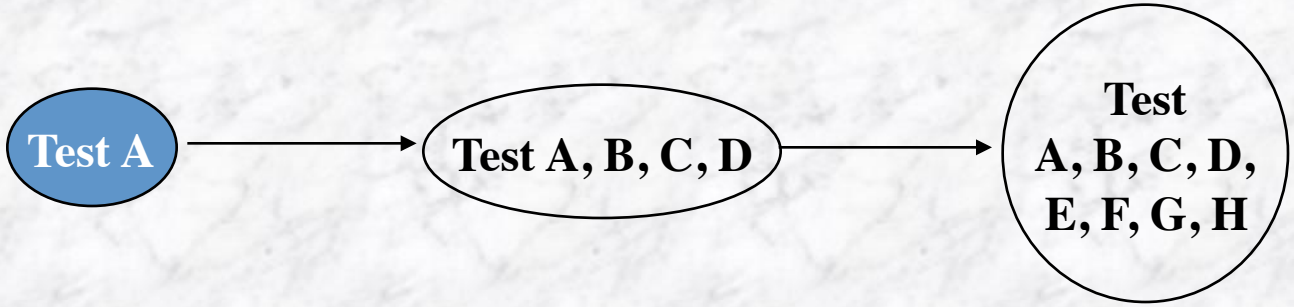


Top-Down Integration Testing Strategy

- Top-down integration strategy focuses on testing the top layer or the controlling subsystem first (i.e. the *main*, or the root of the call tree)
- The general process in top-down integration strategy is to gradually add more subsystems that are referenced/required by the already tested subsystems when testing the application
- Do this until all subsystems are incorporated into the test
- Special program is needed to do the testing, *Test stub:*
  - A program or a method that simulates the input-output functionality of a missing subsystem by answering to the decomposition sequence of the calling subsystem and returning back simulated or "canned" data.

Top-down Integration Testing Strategy





Level I Unit(s)

Level I and II Units

All Units – all levels

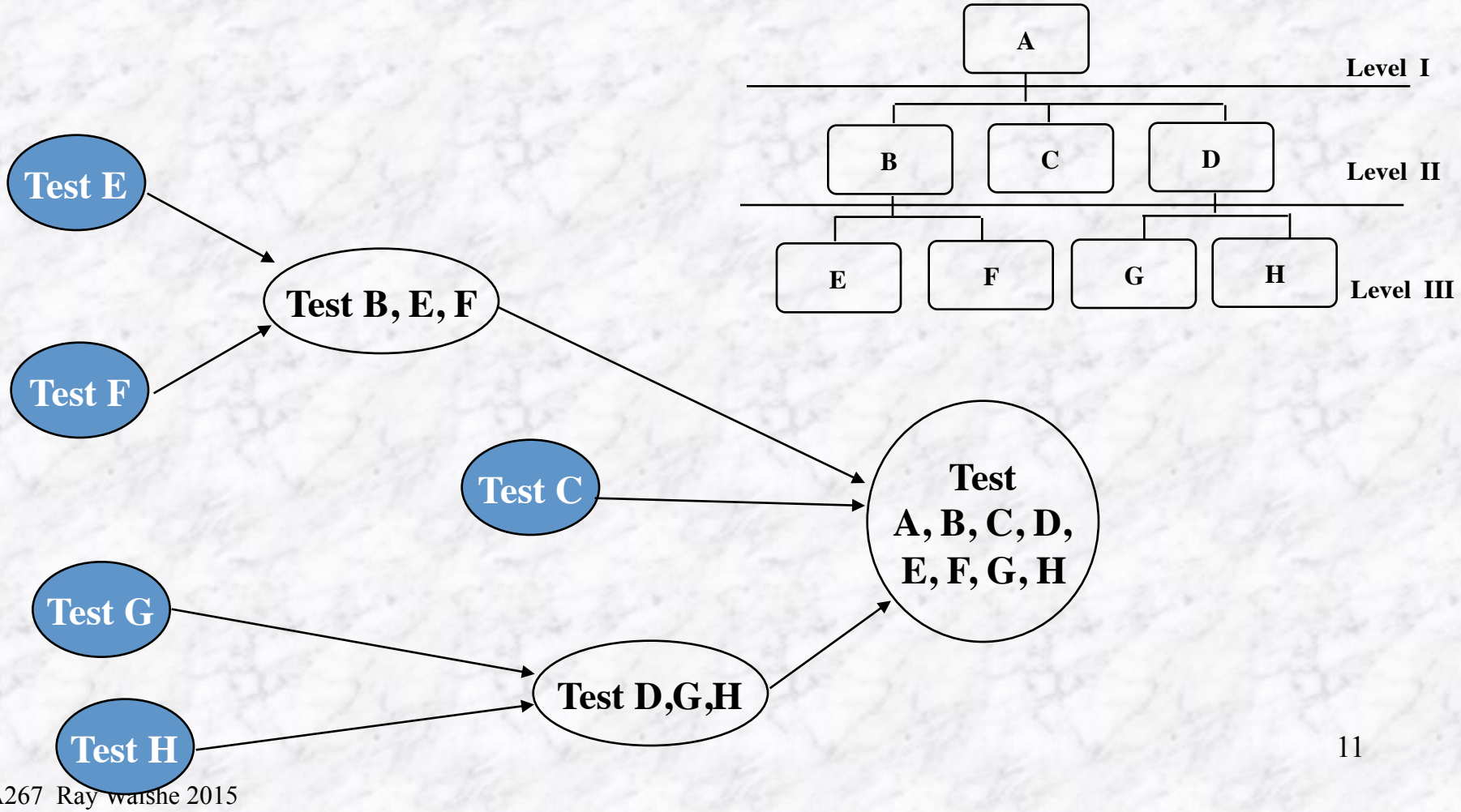
Advantages and Disadvantages of Top-Down Integration Testing

- Test cases can be defined in terms of the functionality of the system (functional requirements). Structural techniques can also be used for the units in the top levels
- Writing stubs can be difficult especially when parameter passing is complex. Stubs must allow all possible conditions to be tested
- Possibly a very large number of stubs may be required, especially if the lowest level of the system contains many functional units
- One solution to avoid too many stubs: *Modified top-down testing strategy (Bruege)*
  - Test each layer of the system decomposition individually before merging the layers
  - Disadvantage of modified top-down testing: Both, stubs and drivers are needed

Bottom-Up Integration Testing Strategy

- Bottom-Up integration strategy focuses on testing the units at the lowest levels first (i.e. the units at the leafs of the decomposition tree)
- The general process in bottom-up integration strategy is to gradually include the subsystems that reference/require the previously tested subsystems
- This is done repeatedly until all subsystems are included in the testing
- Special program called *Test Driver* is needed to do the testing,
  - The Test Driver is a "fake" routine that requires a subsystem and passes a test case to it

Example Bottom-Up Strategy



Advantages and Disadvantages of Bottom-Up Integration Testing

- Not optimal strategy for functionally decomposed systems:
  - Tests the most important subsystem (UI) last
- Useful for integrating the following systems
  - Object-oriented systems
  - Real-time systems
  - Systems with strict performance requirements

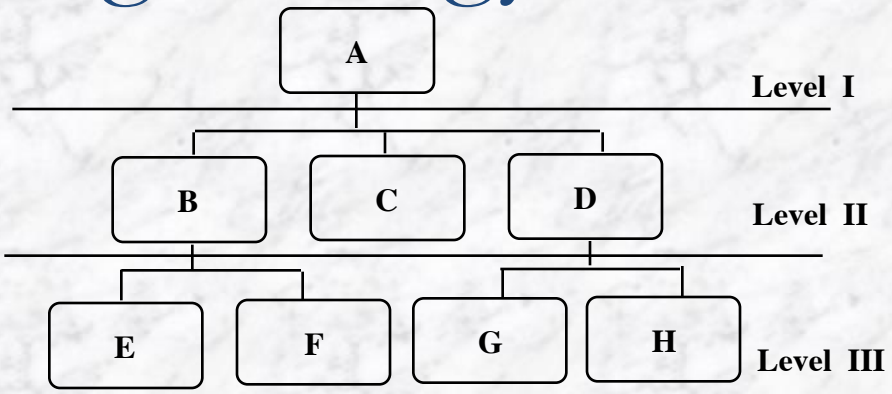
Sandwich Testing Strategy

- Combines top-down strategy with bottom-up strategy

- *The system is viewed as having three layers*
  - A target layer in the middle
  - A layer above the target
  - A layer below the target
  - Testing converges at the target layer

- How do you select the target layer if there are more than 3 layers?
  - Heuristic: Try to minimize the number of stubs and drivers

Sandwich Testing Strategy



Bottom Level Tests

Test E

Test F

Test G

Test H

Test B, E, F

Test D,G,H

Test A,B,C, D

Test A

Top Level Tests

Test A, B, C, D, E, F, G, H

Advantages and Disadvantages of Sandwich Integration Testing

- Top and Bottom Layer Tests can be done in parallel
- Does not test the individual subsystems thoroughly before integration
- Solution: Modified sandwich testing strategy (Bruege)

Steps in Integration-Testing

1. Based on the integration strategy, *select a component* to be tested. Unit test all the classes in the component.
2. Put selected component together; do any *preliminary fix-up* necessary to make the integration test operational (drivers, stubs)
3. Do *functional testing:* Define test cases that exercise all uses cases with the selected component
4. Do *structural testing:* Define test cases that exercise the selected component
5. Execute *performance tests*
6. *Keep records* of the test cases and testing activities.
7. Repeat steps 1 to 7 until the full system is tested.

The primary *goal of integration testing is to identify errors* in the (current) component configuration.

Ref: "*Object-Oriented Software Engineering – Using UML, Patterns and Java*", Bernd Bruegge, Allen H. Dutoit

Which Integration Strategy should you use?

- Factors to consider
  - Amount of test harness (stubs &drivers)
  - Location of critical parts in the system
  - Availability of hardware
  - Availability of components
  - Scheduling concerns
- Bottom up approach
  - good for object oriented design methodologies
  - Test driver interfaces must match component interfaces
  - ...
  - ...Top-level components are usually important and cannot be neglected up to the end of testing
  - Detection of design errors postponed until end of testing
- Top down approach
  - Test cases can be defined in terms of functions examined
  - Need to maintain correctness of test stubs
  - Writing stubs can be difficult

Ref: "*Object-Oriented Software Engineering – Using UML, Patterns and Java*", Bernd Bruegge, Allen H. Dutoit

Call Graph Based Integration

- The basic idea is to use the call graph instead of the decomposition tree
- The call graph is a directed, labeled graph
- Two types of call graph based integration testing
  - Pair-wise Integration Testing
  - Neighborhood Integration Testing

Pair-Wise Integration Testing

- The idea behind Pair-Wise integration testing is to eliminate the need for developing stubs/drivers
- The objective is to use actual code instead of stubs/drivers
- In order no to deteriorate the process to a big-bang strategy, we restrict a testing session to just a pair of units in the call graph
- The result is that we have one integration test session for each edge in the call graph

Neighborhood Integration Testing

- We define the neighborhood of a node in a graph to be the set of nodes that are one edge away from the given node
- In a directed graph means all the immediate predecessor nodes and all the immediate successor nodes of a given node
- The number of neighborhoods for a given graph can be computed as:

InteriorNodes = nodes – (SourceNodes + SinkNodes)
Neighborhoods = InteriorNodes + SourceNodes
Or
Neighborhoods = nodes – SinkNodes

- Neighborhood Integration Testing reduces the number of test sessions

Advantages and Disadvantages of Call-Graph Integration Testing

- Call graph based integration techniques move towards a behavioral basis
- Aim to eliminate / reduce the need for drivers/ stubs
- Closer to a build sequence
- Neighborhoods can be combined to create "villages"
- Suffer from the fault isolation problem especially for large neighborhoods
- Nodes can appear in several neighborhoods