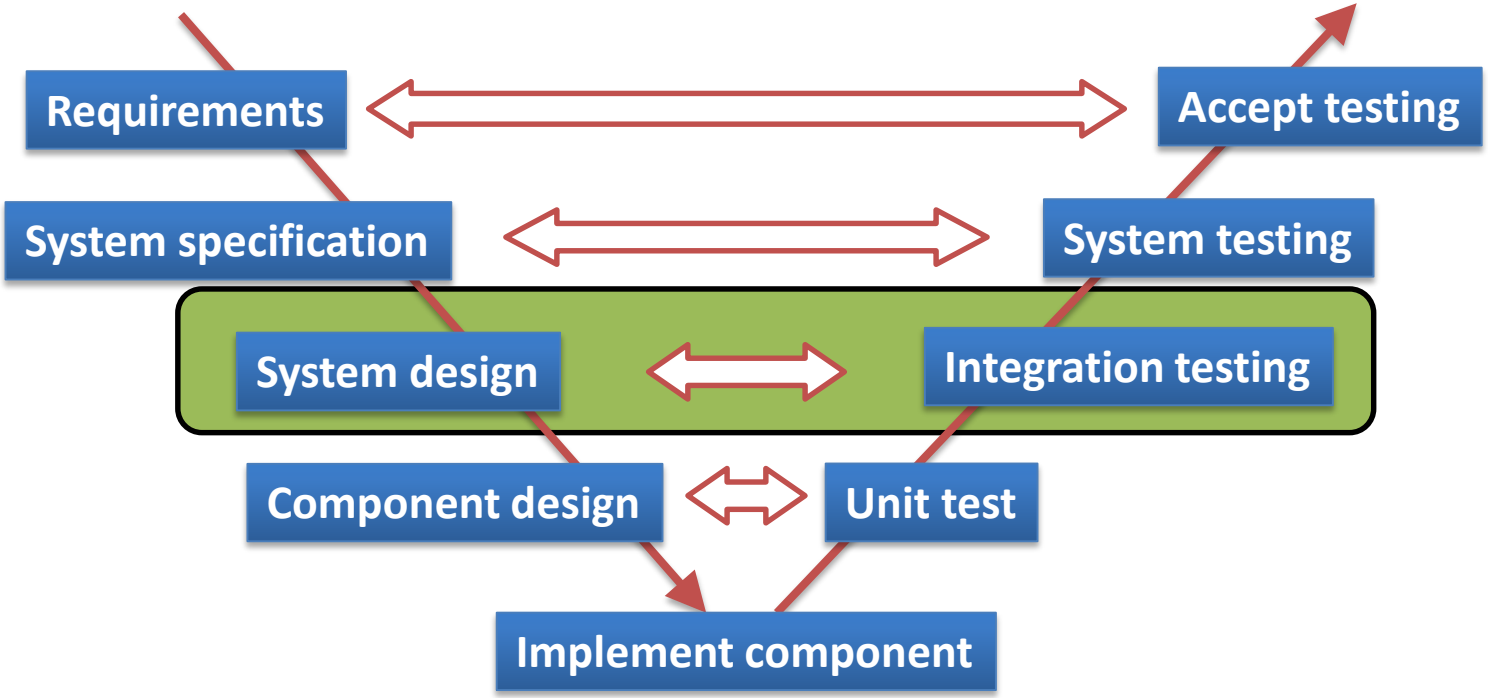
Integration Test Patterns

I4SWT

AARHUS
UNIVERSITY
SCHOOL OF ENGINEERING

Integration test

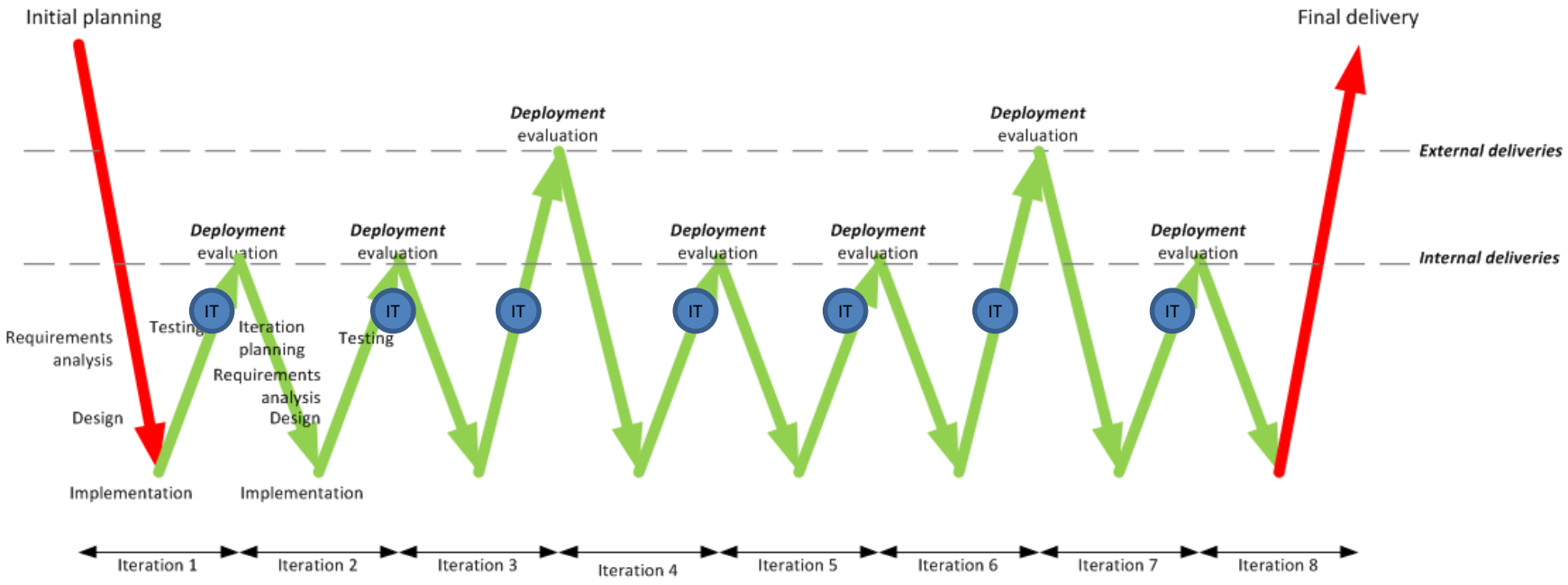


Challenges

- As we progress ”up the V” (and towards the complete system), …
  - Defining a test strategy
  - Defining test cases
  - Evaluating test results
  - Setting up test fixtures
  - Setting up test scenarios
  - Ensuring test coverage
  - Version-controlling tests

- …all become more difficult, expensive, time-consuming – and important!

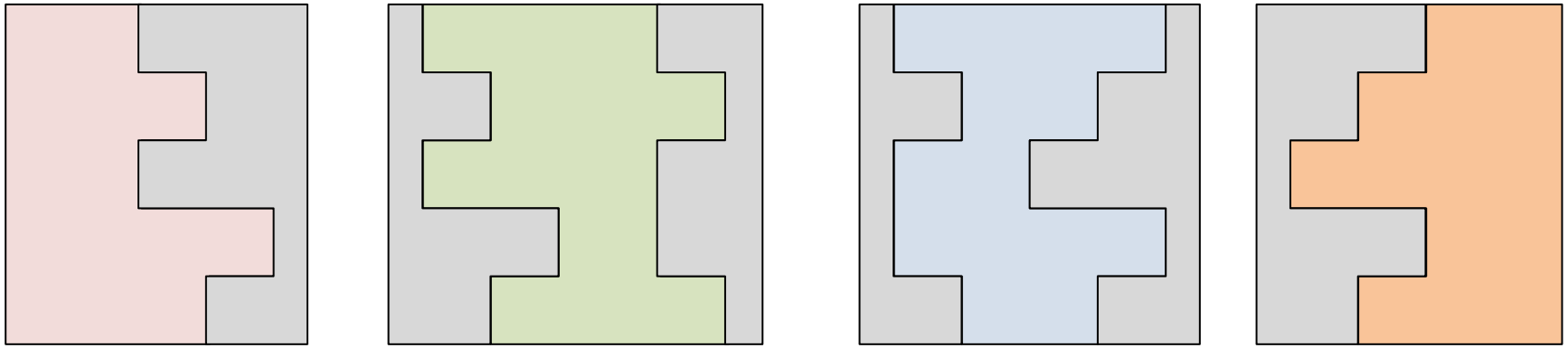
Integration test in an iterative process



Lecture plan

- This lecture
  - Theory of integration test patterns
  - Find the dependency tree
- Next lecture
  - Integration Test Planning
  - Integration Test Case Generation
  - IT and CI

Purpose, aim



Purpose, aim

- The purpose of integration test is to test the **interactions** and **interfaces between** several modules

- The aim is to verify correct interaction of the tested modules
  - Classes
  - Packages
  - Components
  - Subsystems

- Additionally – the interaction **between** the **low level modules** (HW drivers) and the **actual hardware:** that is **hardware-software integration**

- Verification requires 100% *interface coverage* – is hard to measure and can be hard to obtain

Being smart about integration tests

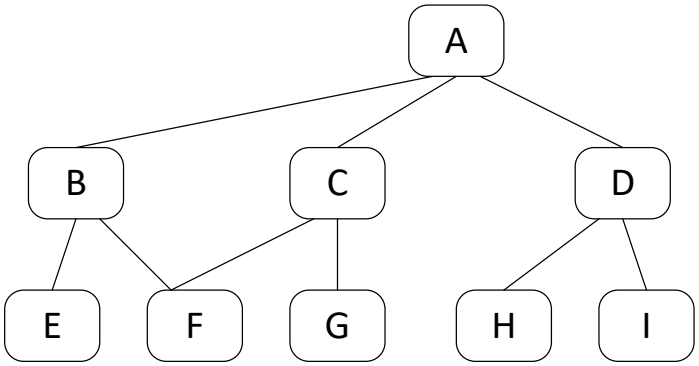
- Integration tests are linked to project ”heartbeat”
  - Integration testing usually requires input from several partners, so test **planning** becomes key.
  - Large-scale component integration (at least) at the end of each iteration.
- Integration tests requires knowledge of system architecture to partition the system into testable chunks
- **A strategy!**
- **And a plan!**

Prereq's for starting integration test

- ✓ Unit testing of all modules is complete
- ✓ System architecture (dependencies) is known
- ✓ Integration test plan is defined
  - Integrated modules
  - System Under Test (SUT) structure?
  - Test fixtures / environment
  - Test cases – SUT stimuli and expected responses

Getting ready – mapping the dependency tree

- Integration test planning is helped along using a *dependency tree*
  - Depicts inter-module dependencies in a tree-like structure
  - *Does not* depict an inheritance hierarchy, layering or the like



A *depends-on* B, C and D
B *depends-on* E and F
C *depends-on* F and G
D *depends-on* H and I

- Some dependencies are obvious from sequence diagrams, object diagrams, state charts, etc.
- Others require inspection (members, parameter types, ..)
- Loops can be broken using stubs

DT design rules

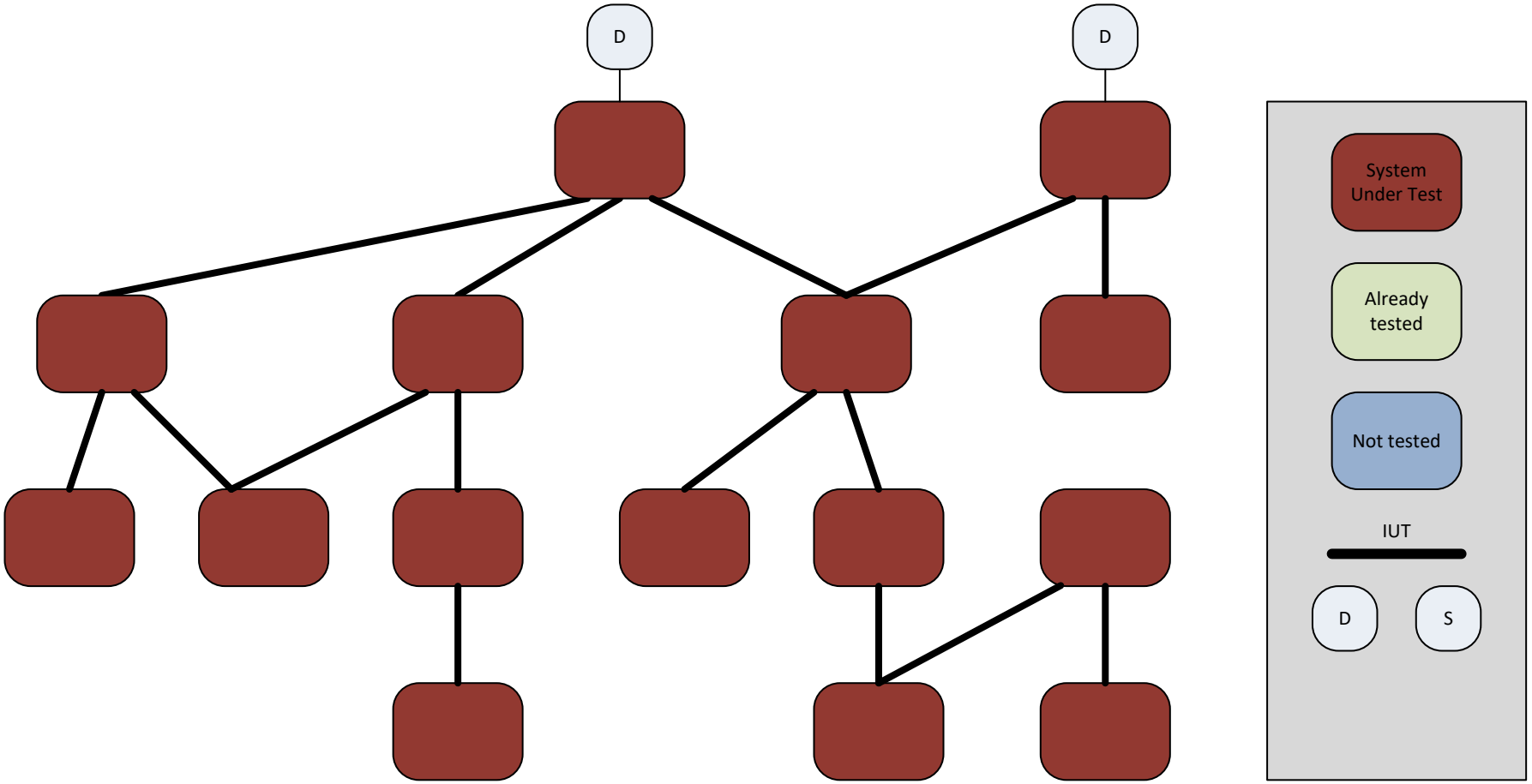
- Dependency tree is a new type of diagram
- DT is not a (official) UML diagram type
- Dependencies always goes from the **bottom** of the dependent module to the **top** of the one it is dependent on!
- That way arrows are **not** needed
- Dependencies **never goes sideways** (horizontal)
- Move modules down to avoid this
- Show loops as loops!

Integration test patterns

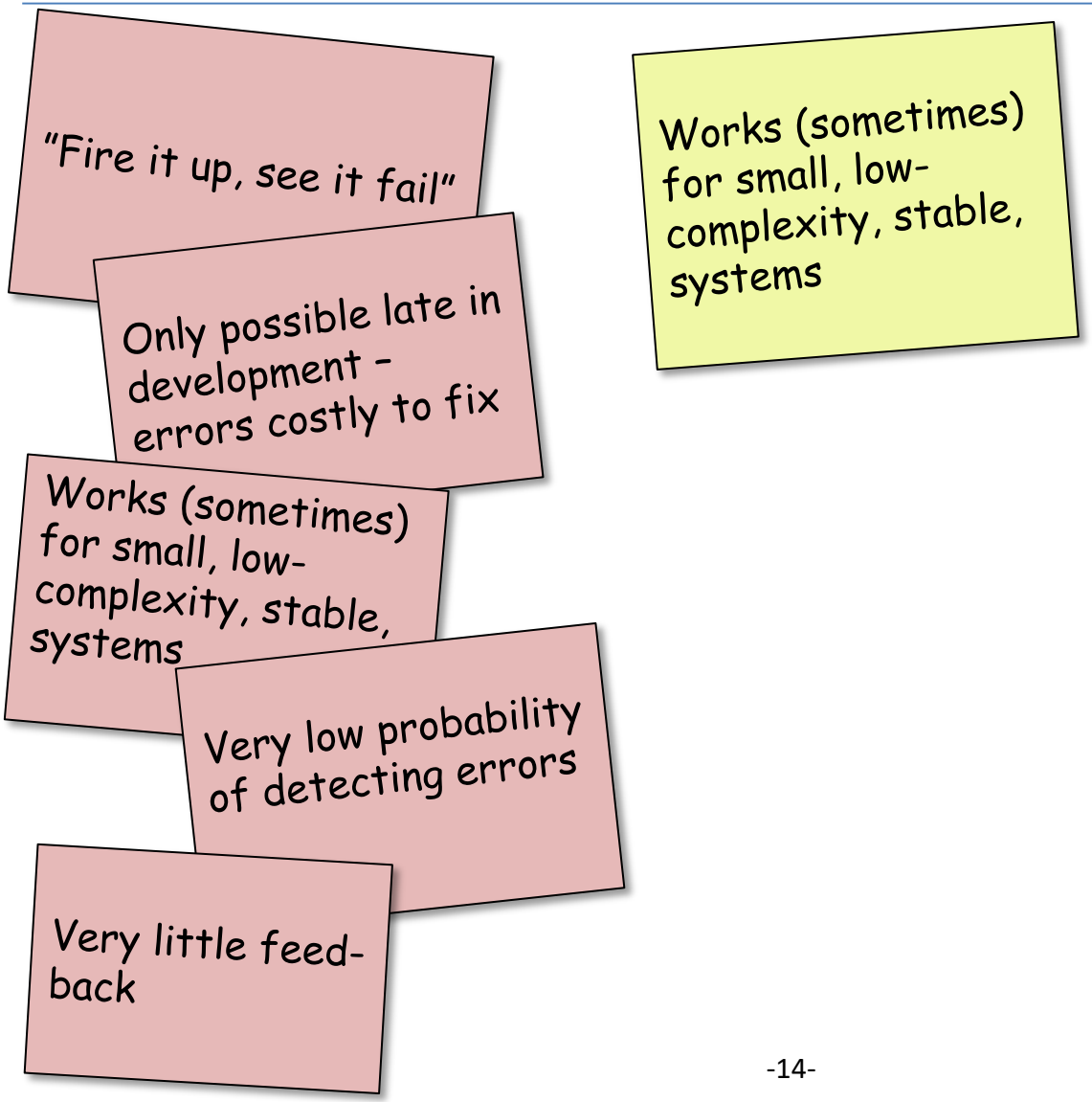
- *Integration test patterns* are used to plan and execute the integration tests.

- This session covers the following patterns
  - Big Bang Integration
  - Bottom-up Integration
  - Top-down Integration
  - Collaboration Integration
  - Sandwich Integration

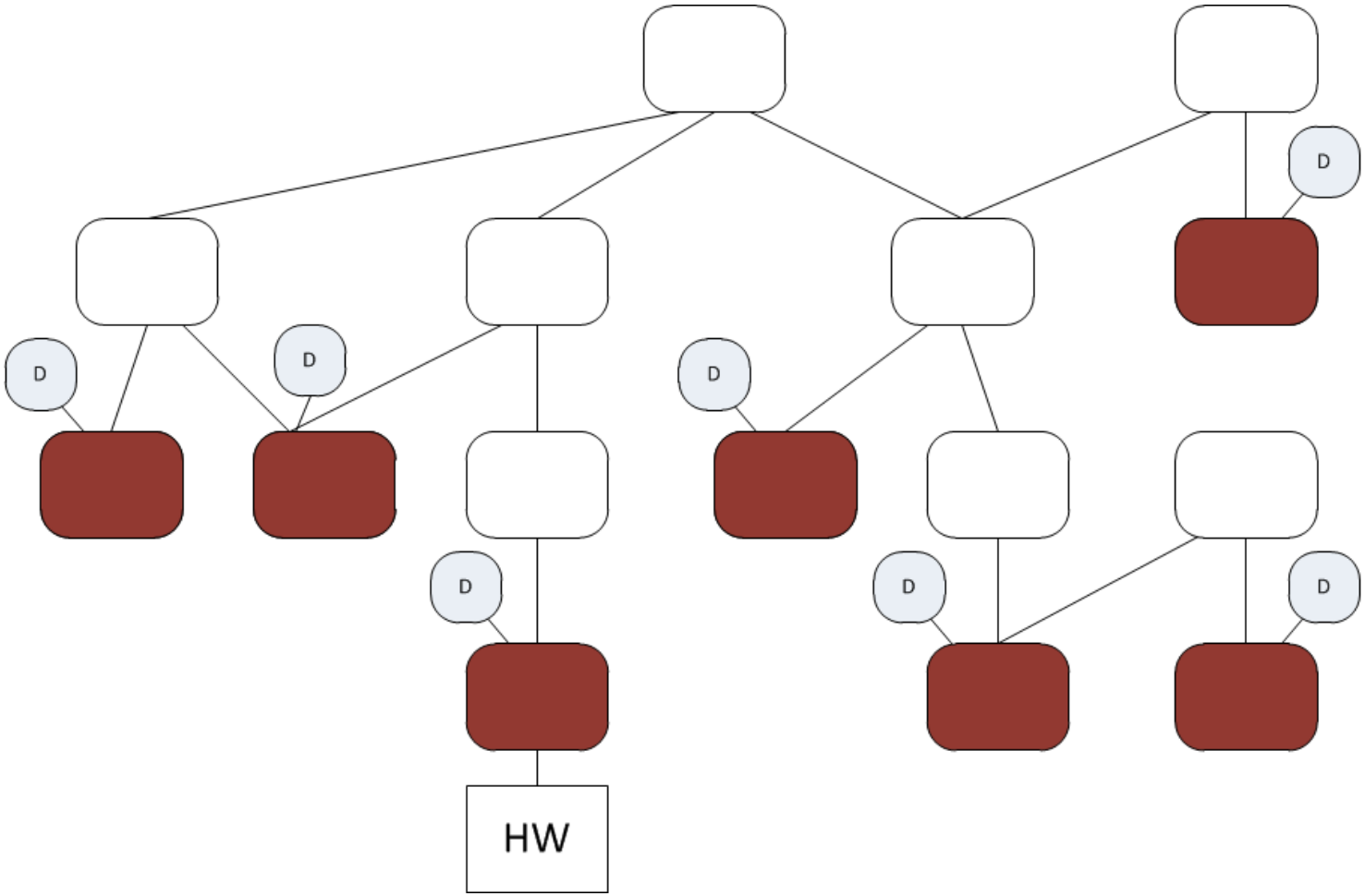
Big Bang Integration

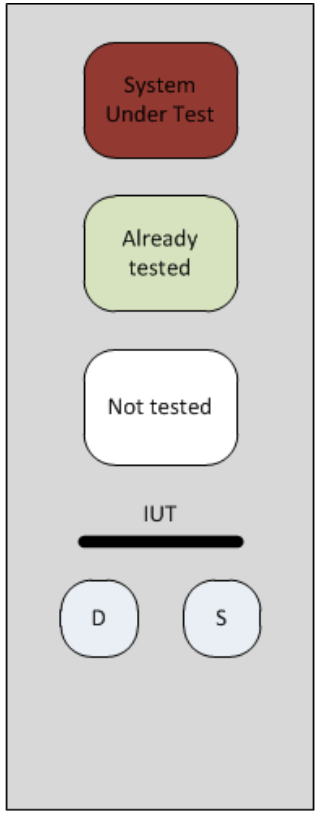


Big Bang Integration

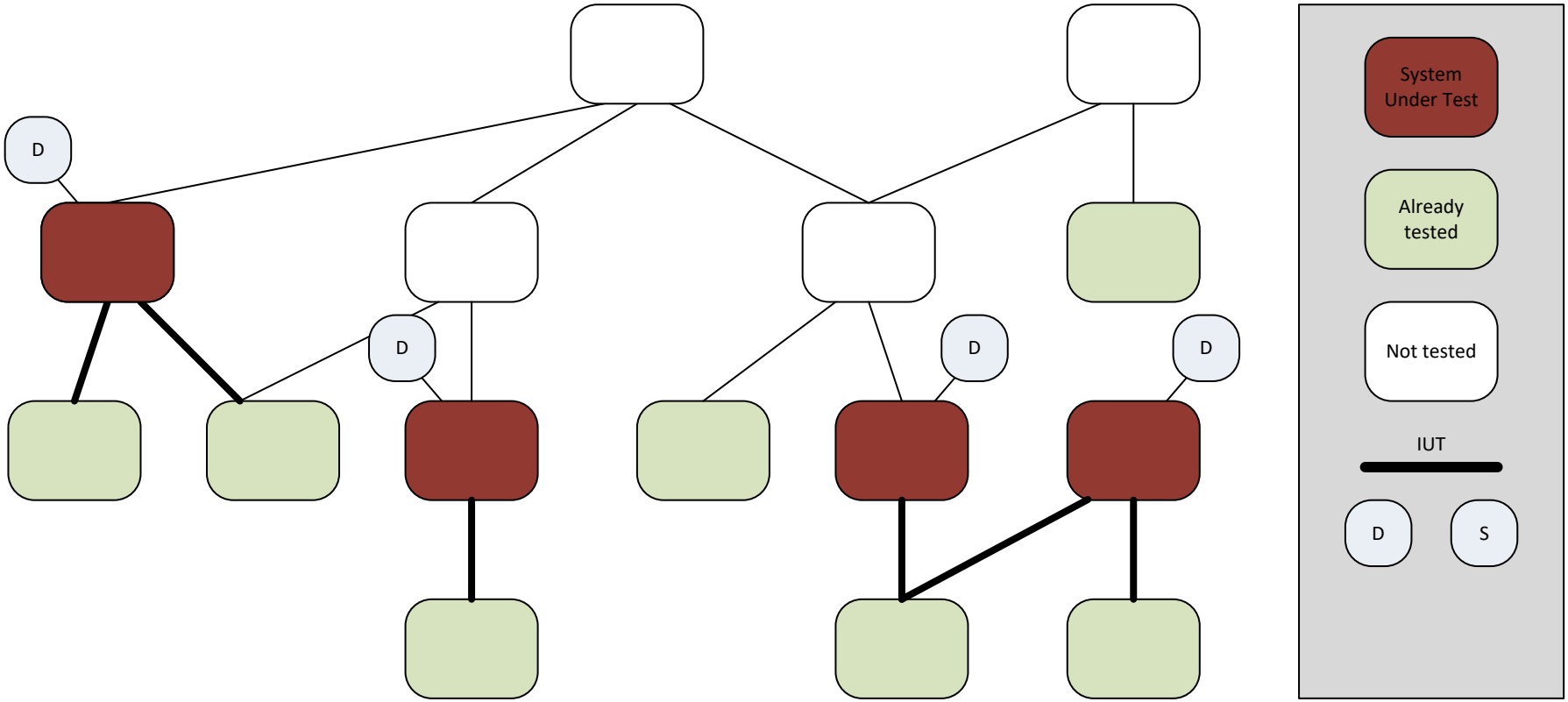


Bottom-up Integration

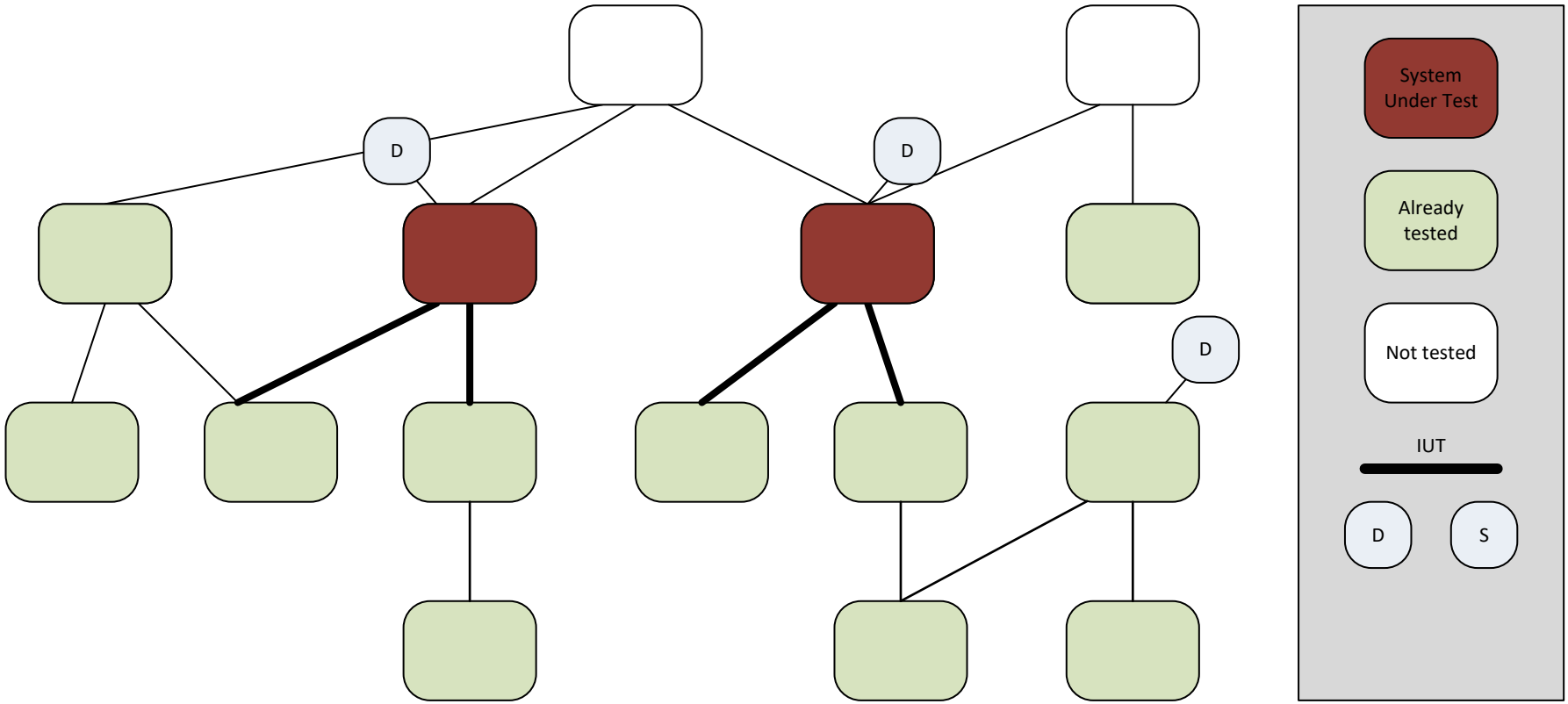




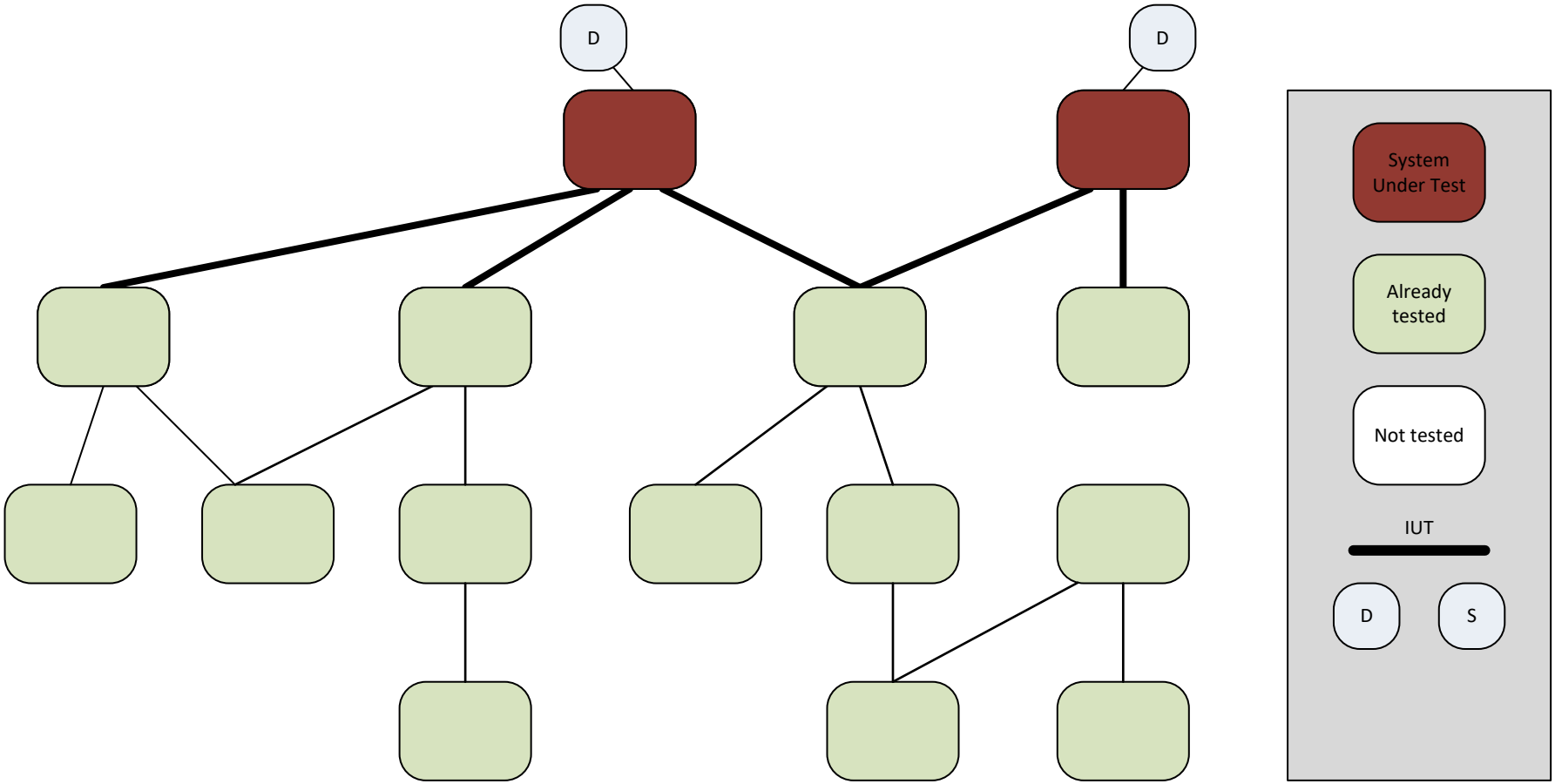
Bottom-up Integration



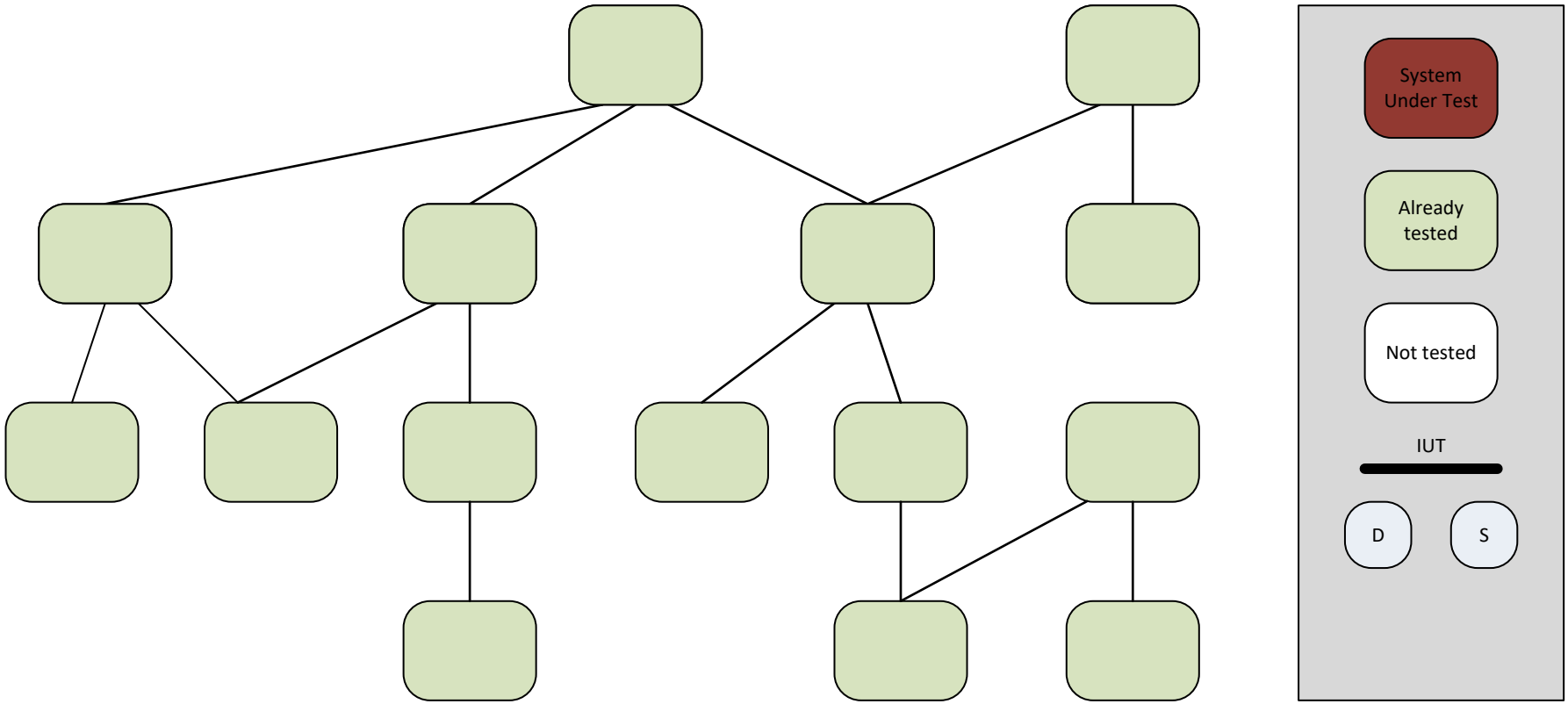
Bottom-up Integration



Bottom-up Integration



Bottom-up Integration



Bottom-up Integration

Requires many drivers at different levels

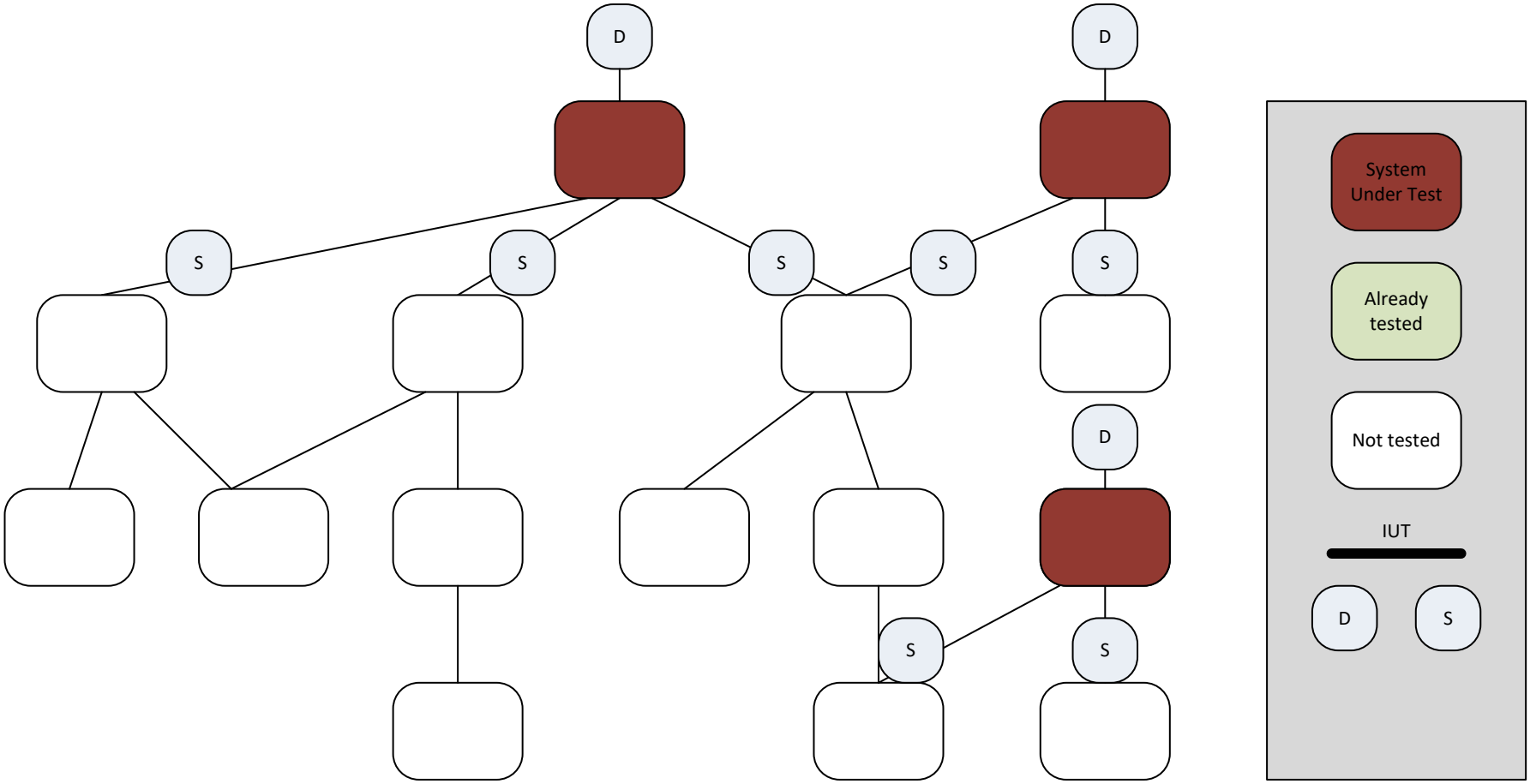
Postpones test of critical control com-ponent interfaces

Reflects very "engineering-like" mindset

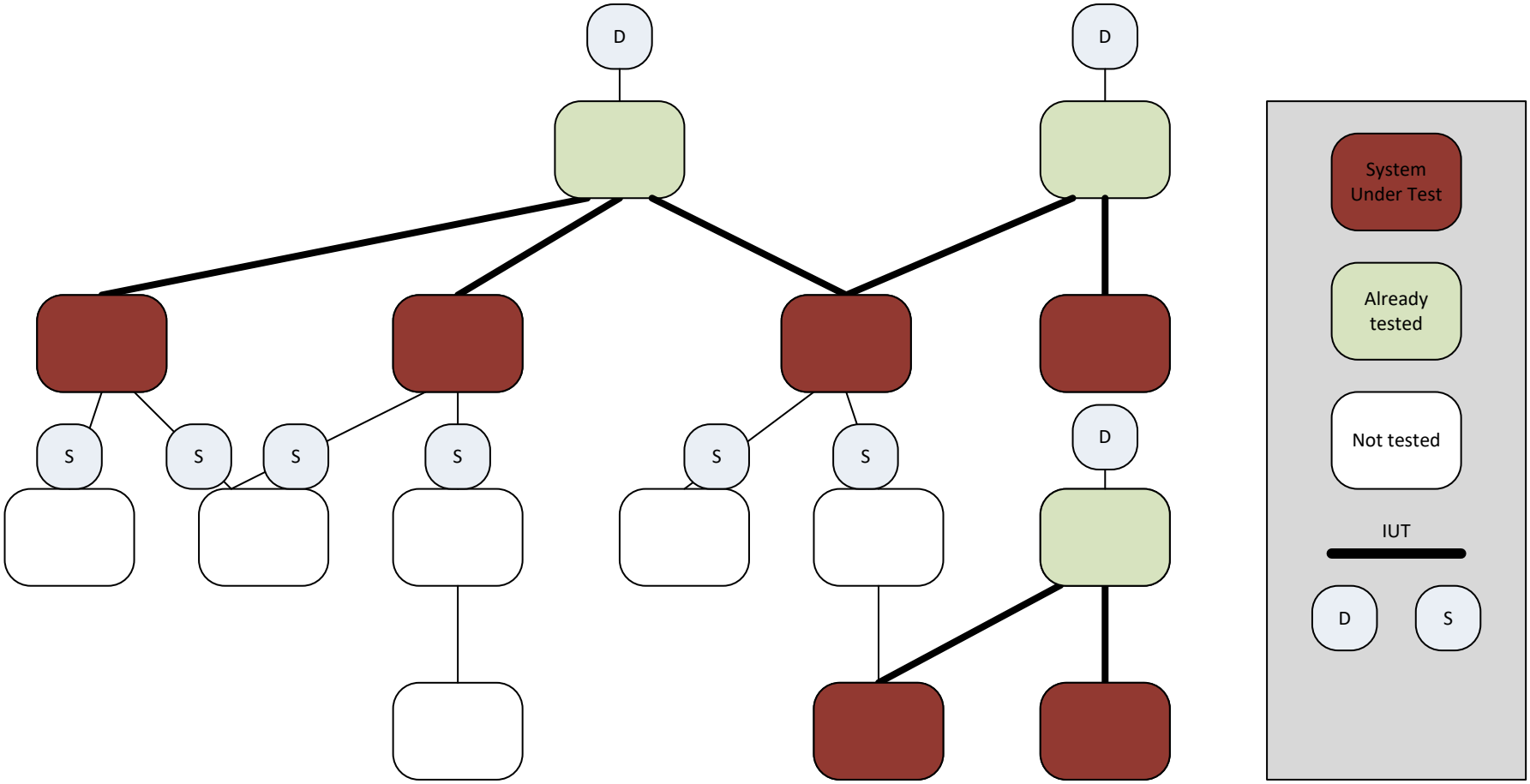
No (few) stubs to develop

Easy to cover interfaces at all levels

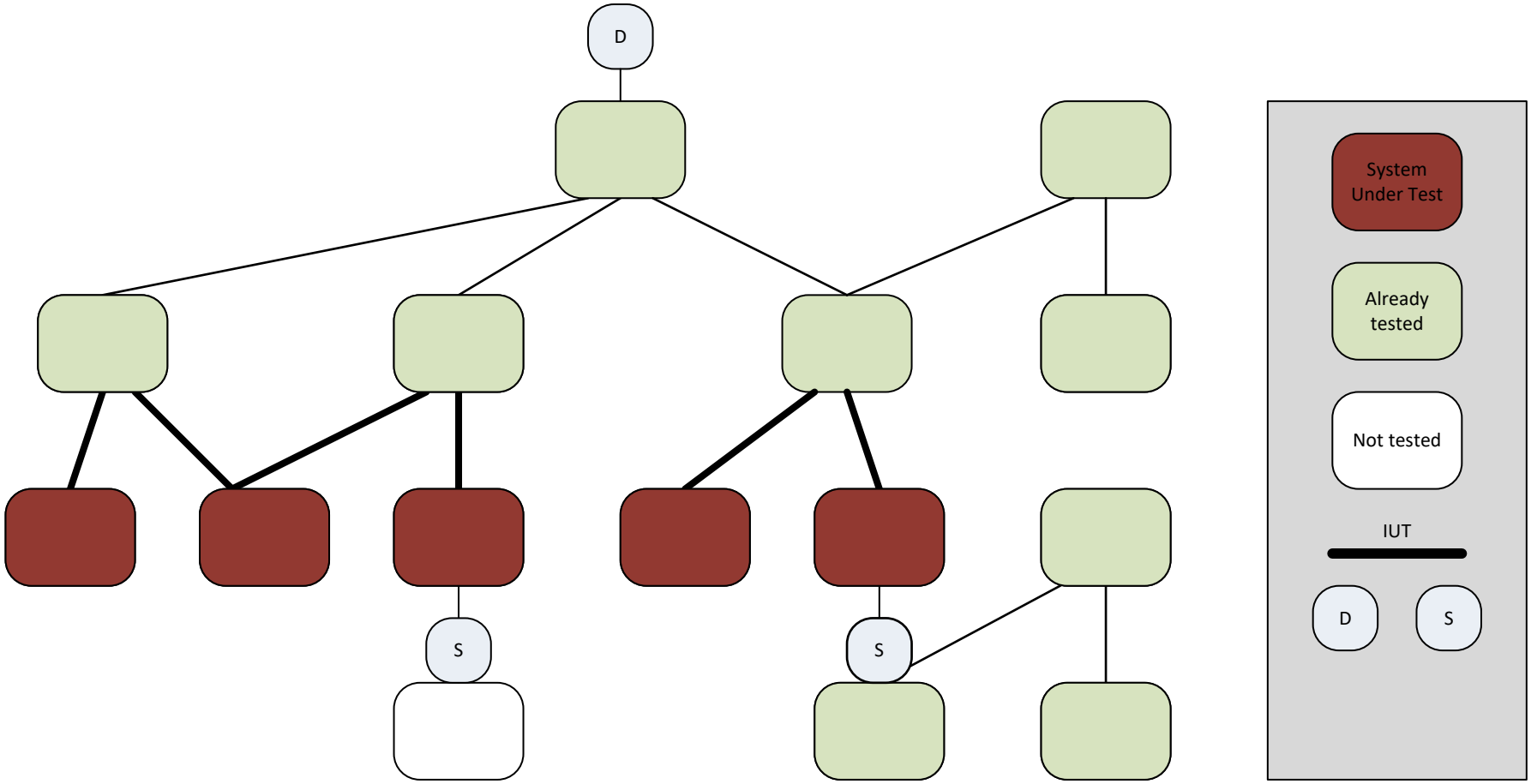
Top-down Integration



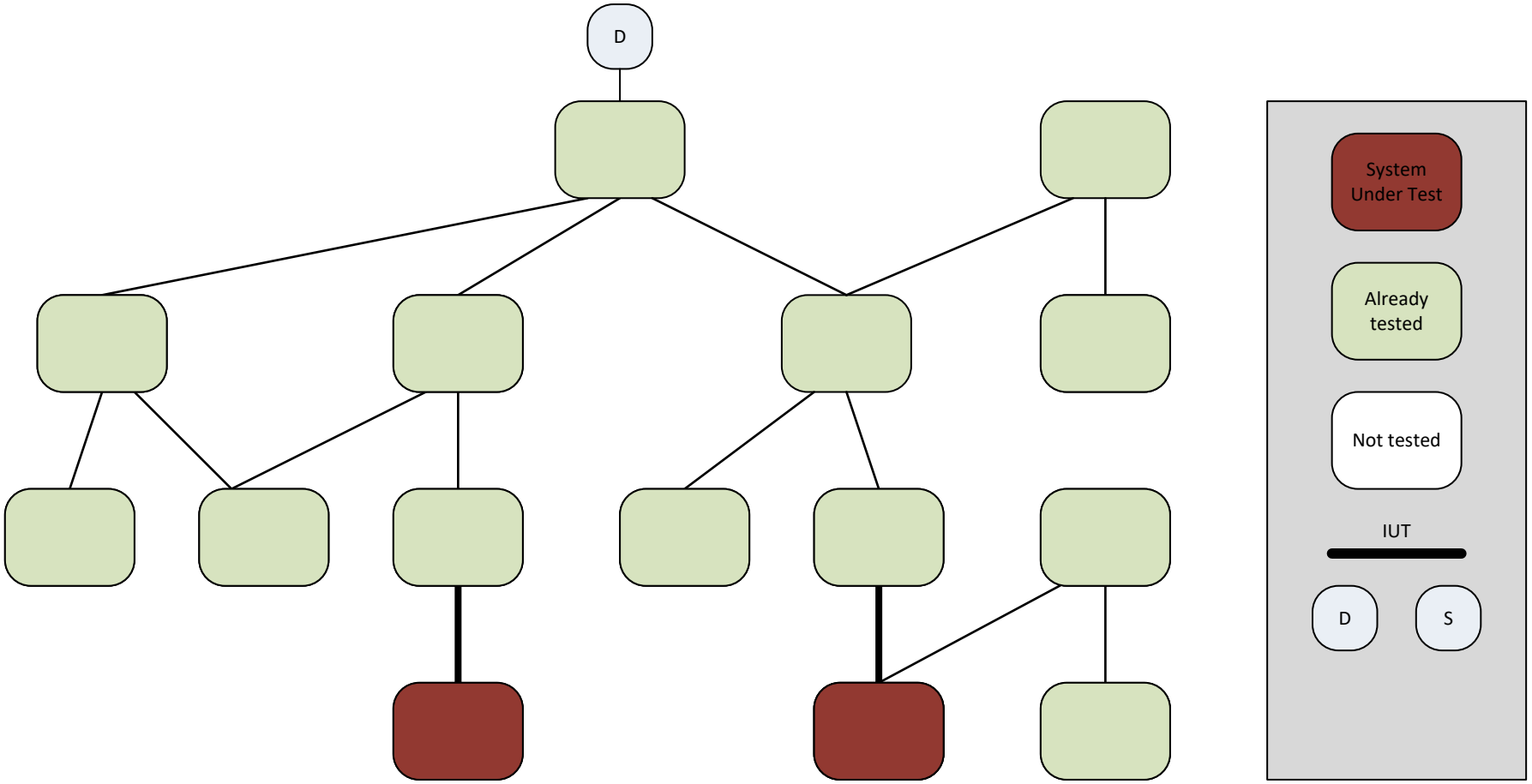
Top-down Integration



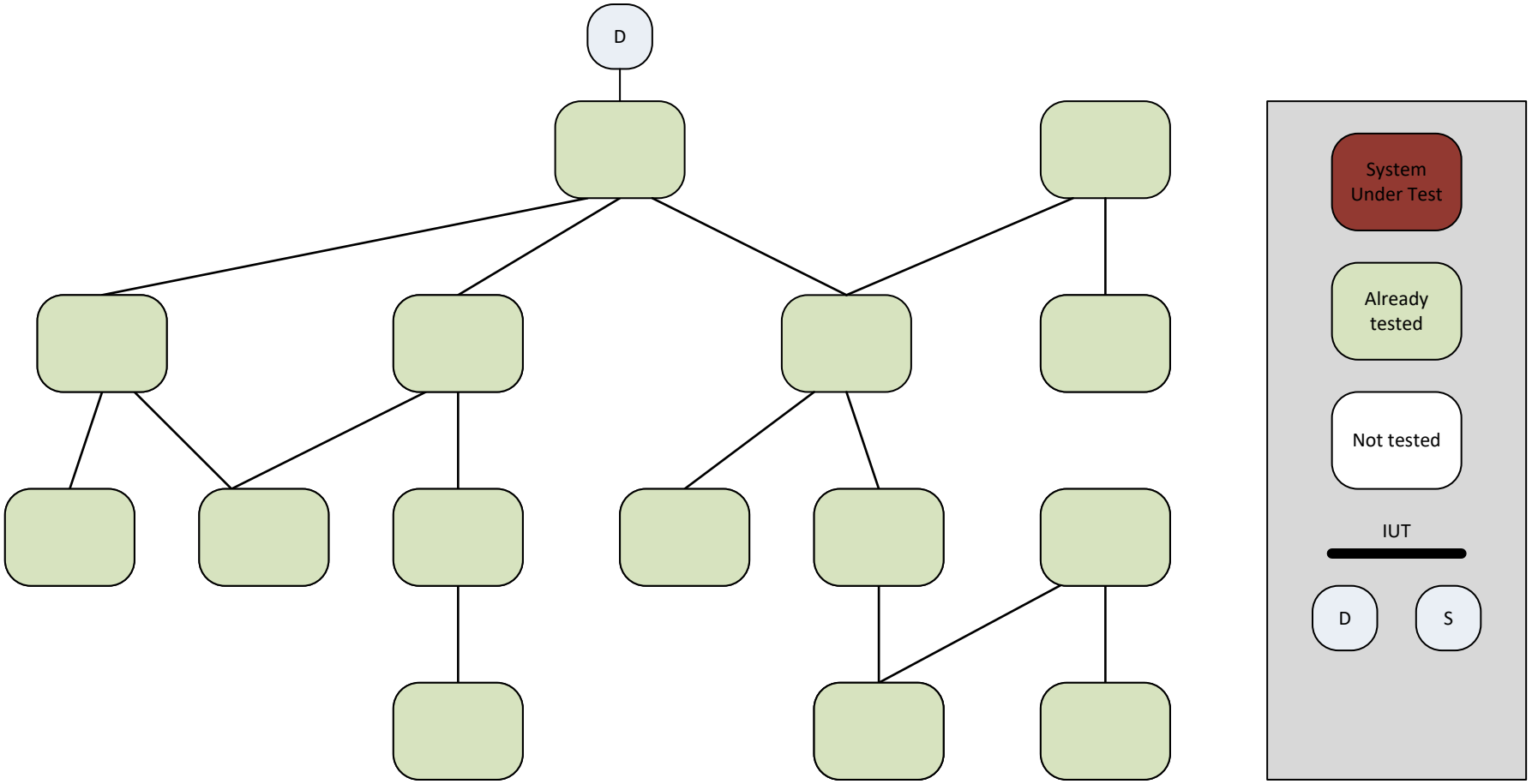
Top-down Integration



Top-down Integration



Top-down Integration



Top-down Integration

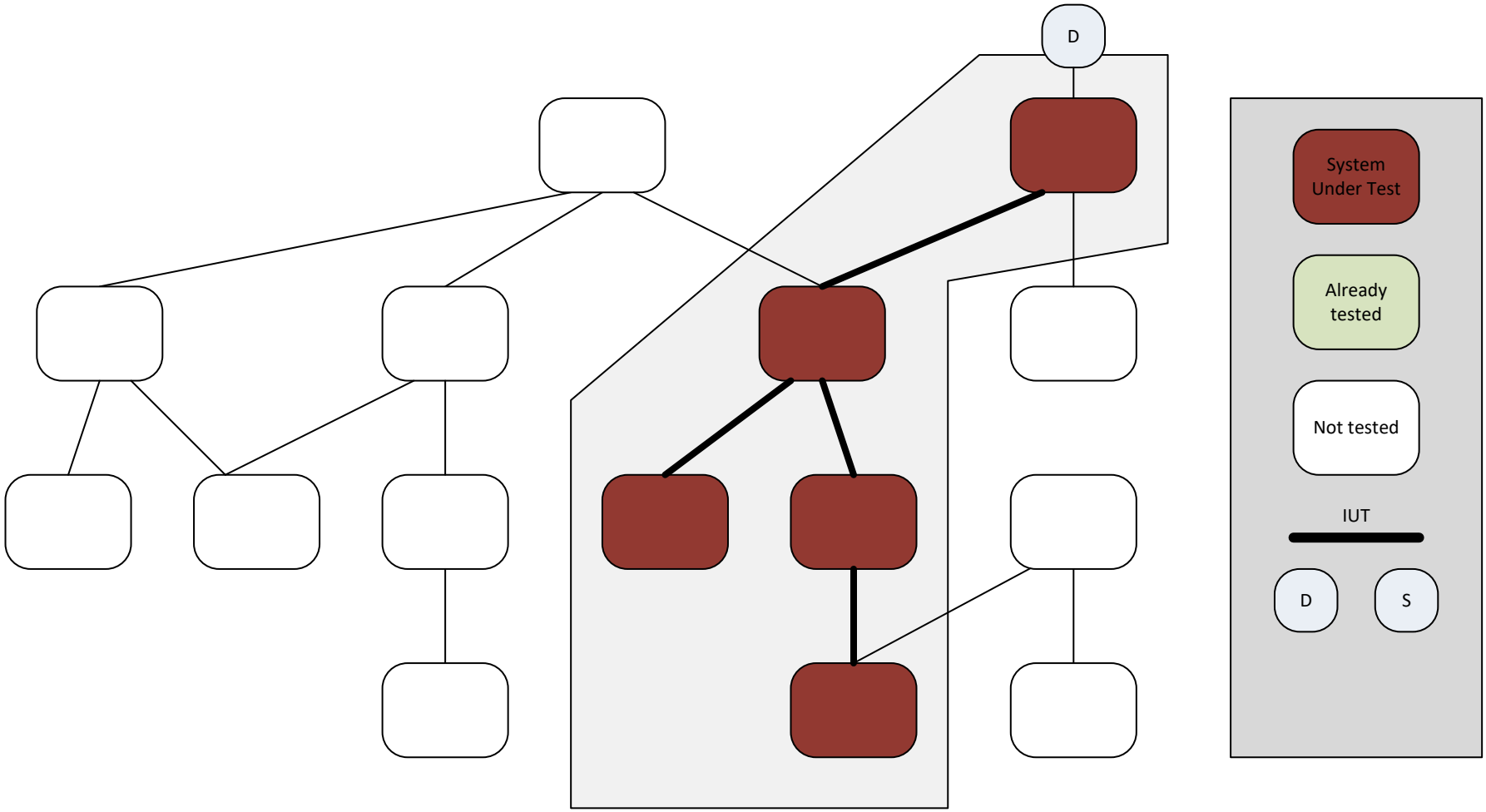
Hard to exercise low-level interfaces from the top

Needs lots of stubs (OK with isolation framework)

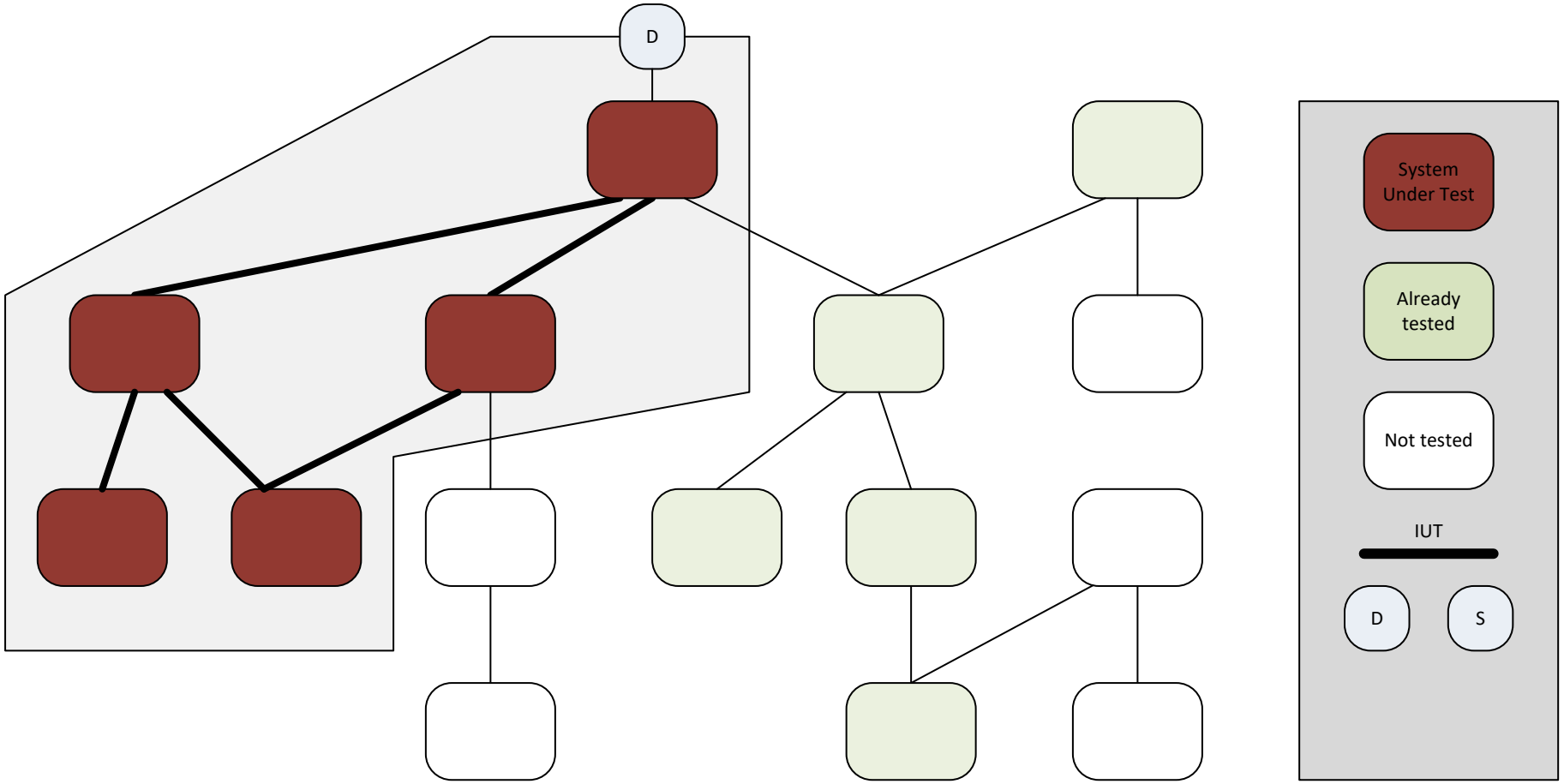
Early feedback on controller compo-nents

Facilitates concurrent HW and SW development

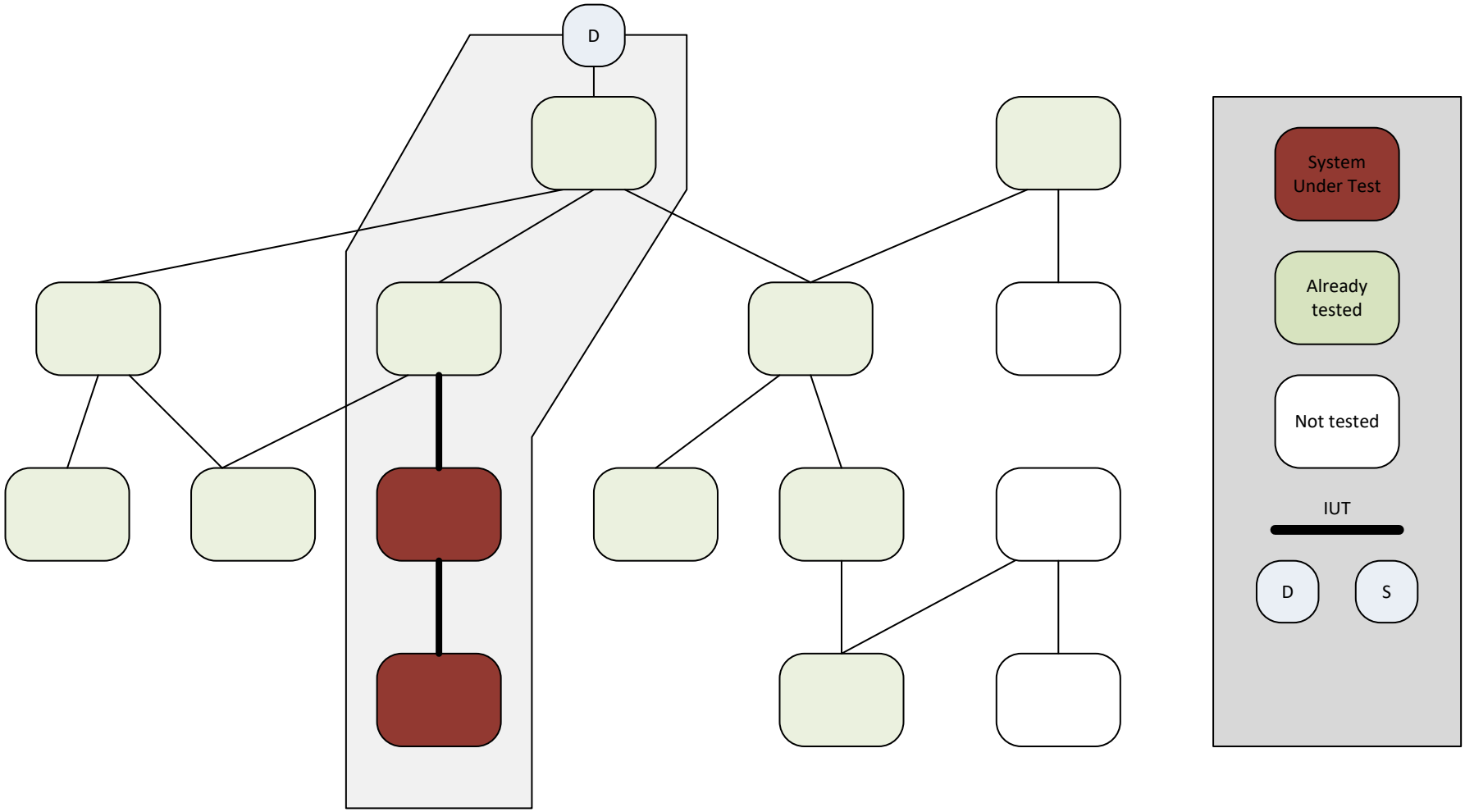
Collaboration Integration



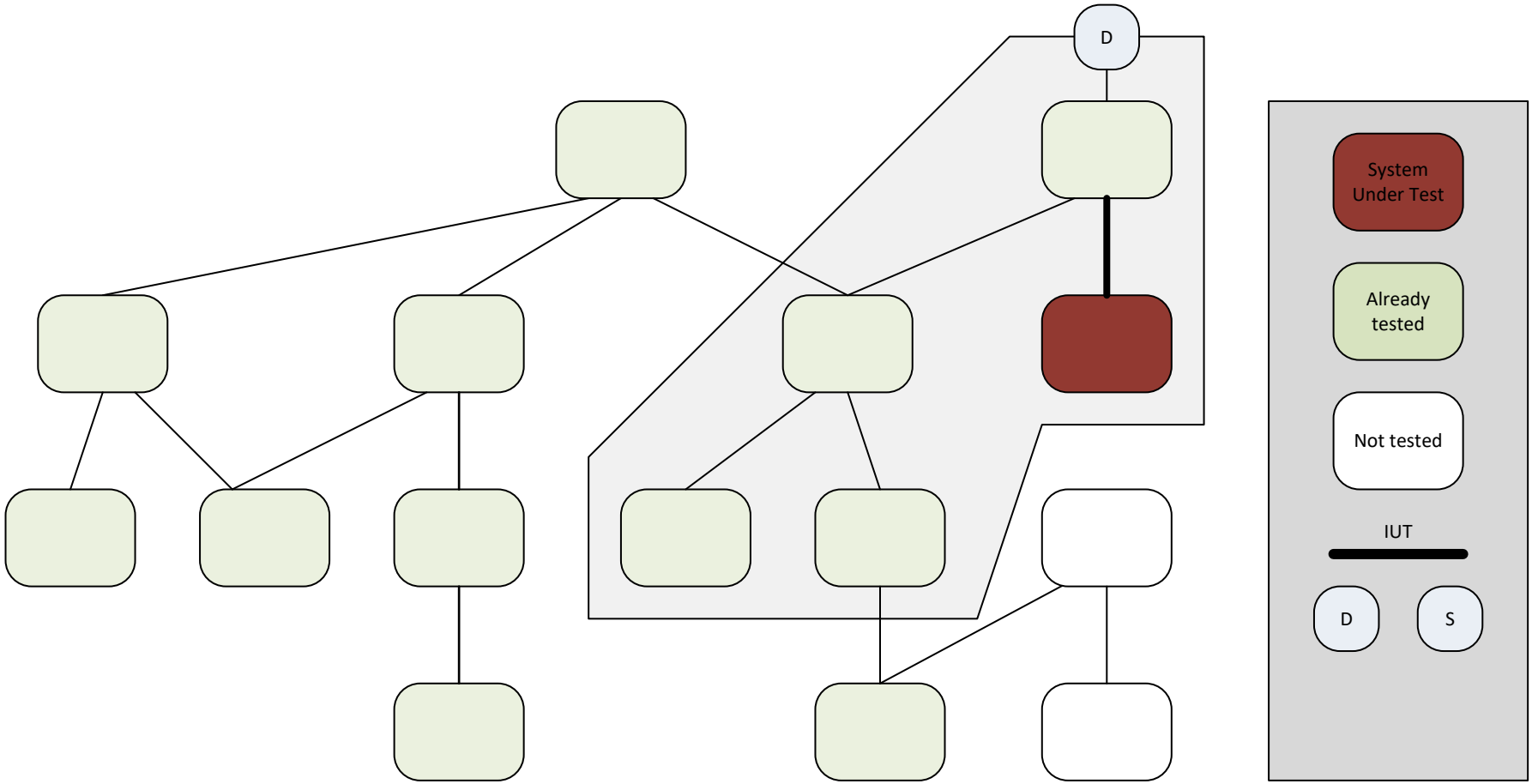
Collaboration Integration



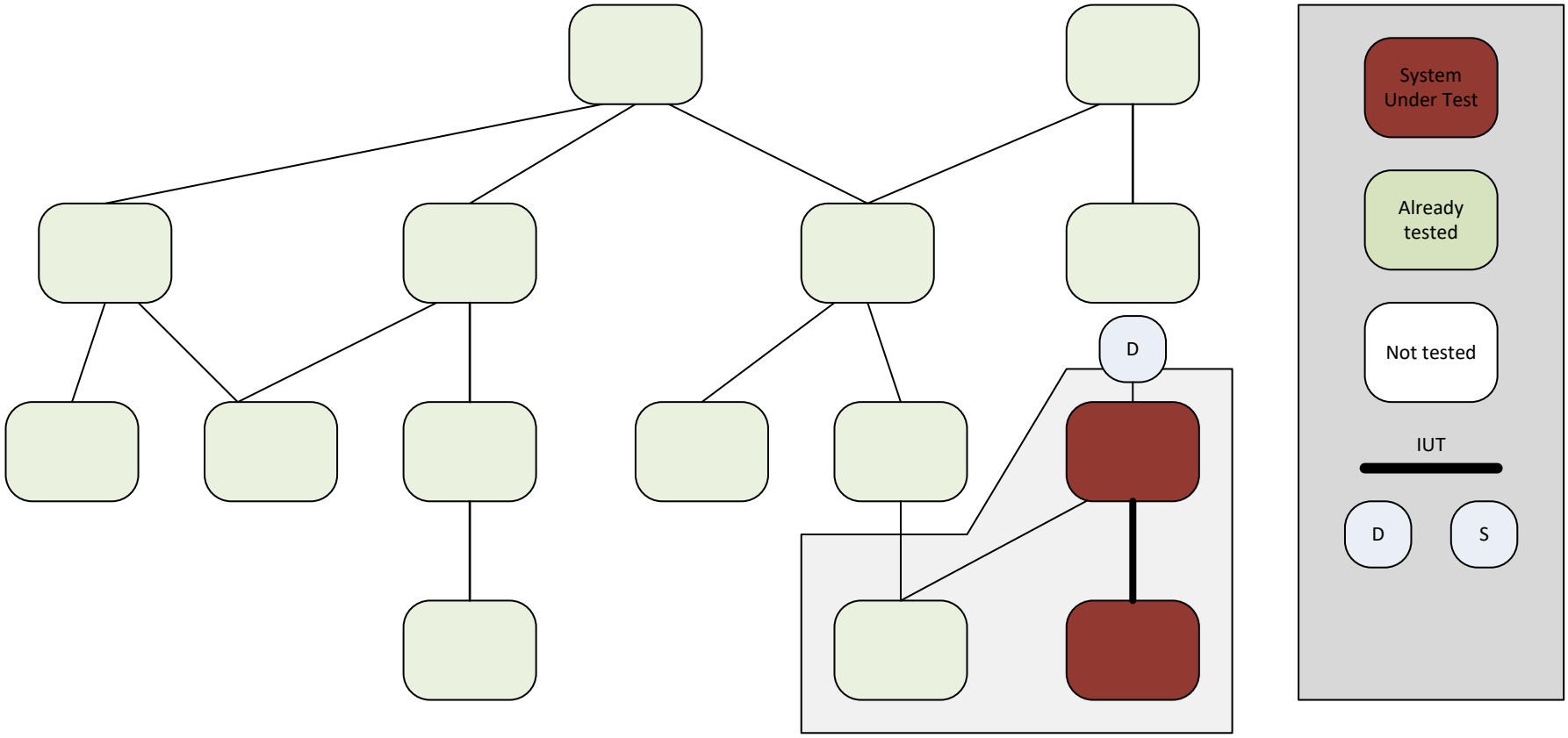
Collaboration Integration



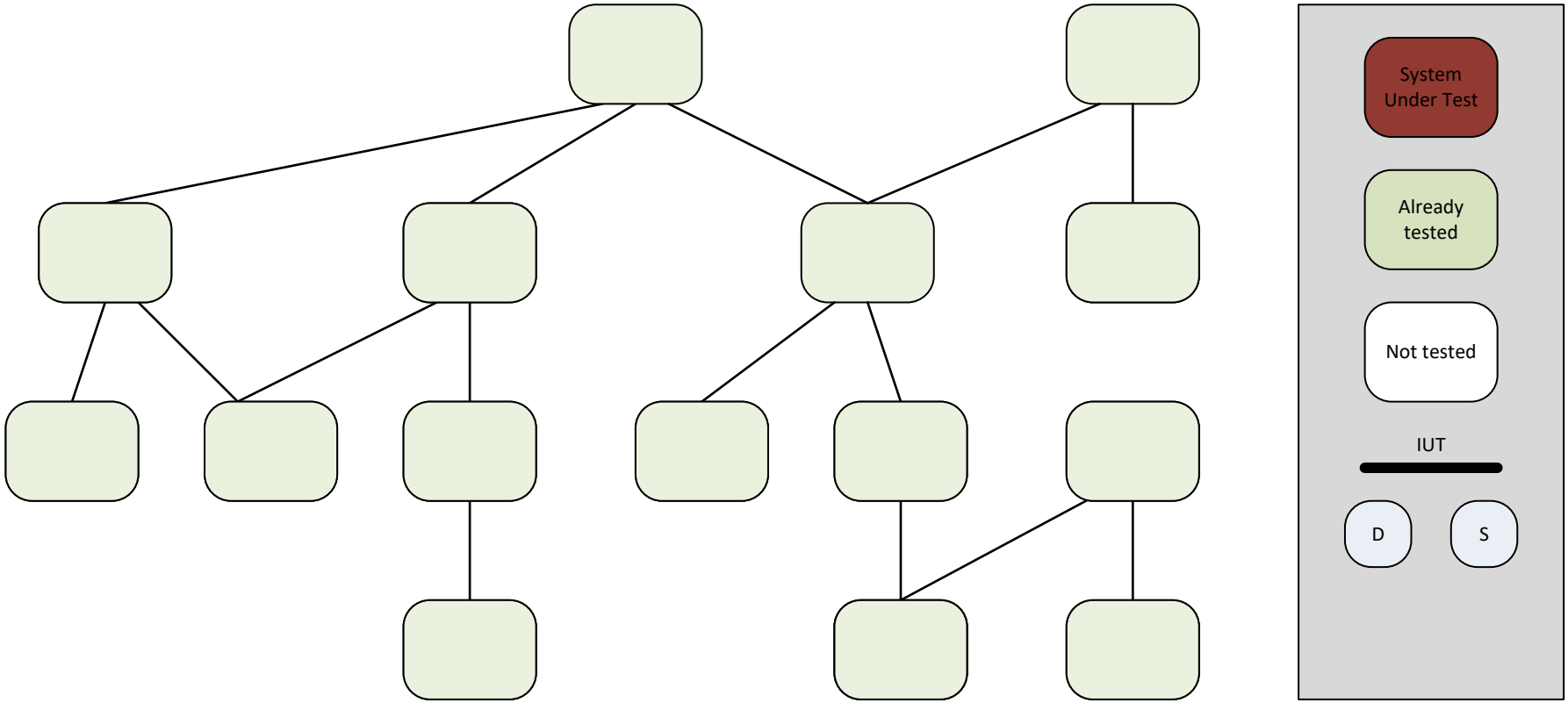
Collaboration Integration



Collaboration Integration



Collaboration Integration



Collaboration Integration

Hard to exercise low-level interfaces

Participants not exercised separa-tely – mini Big-Bang!

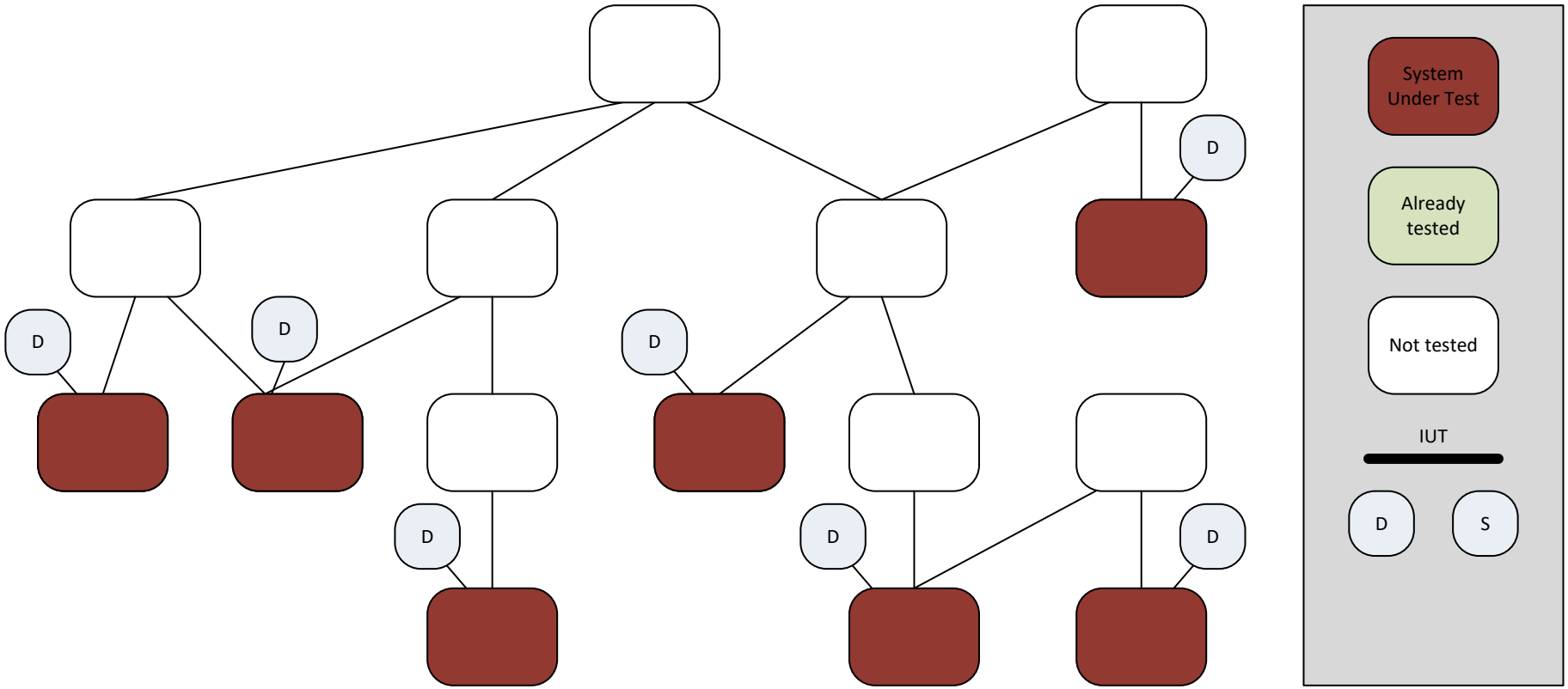
Needs lots of stubs (OK with isolation framework)

Intuitive for users (may follow use cases)

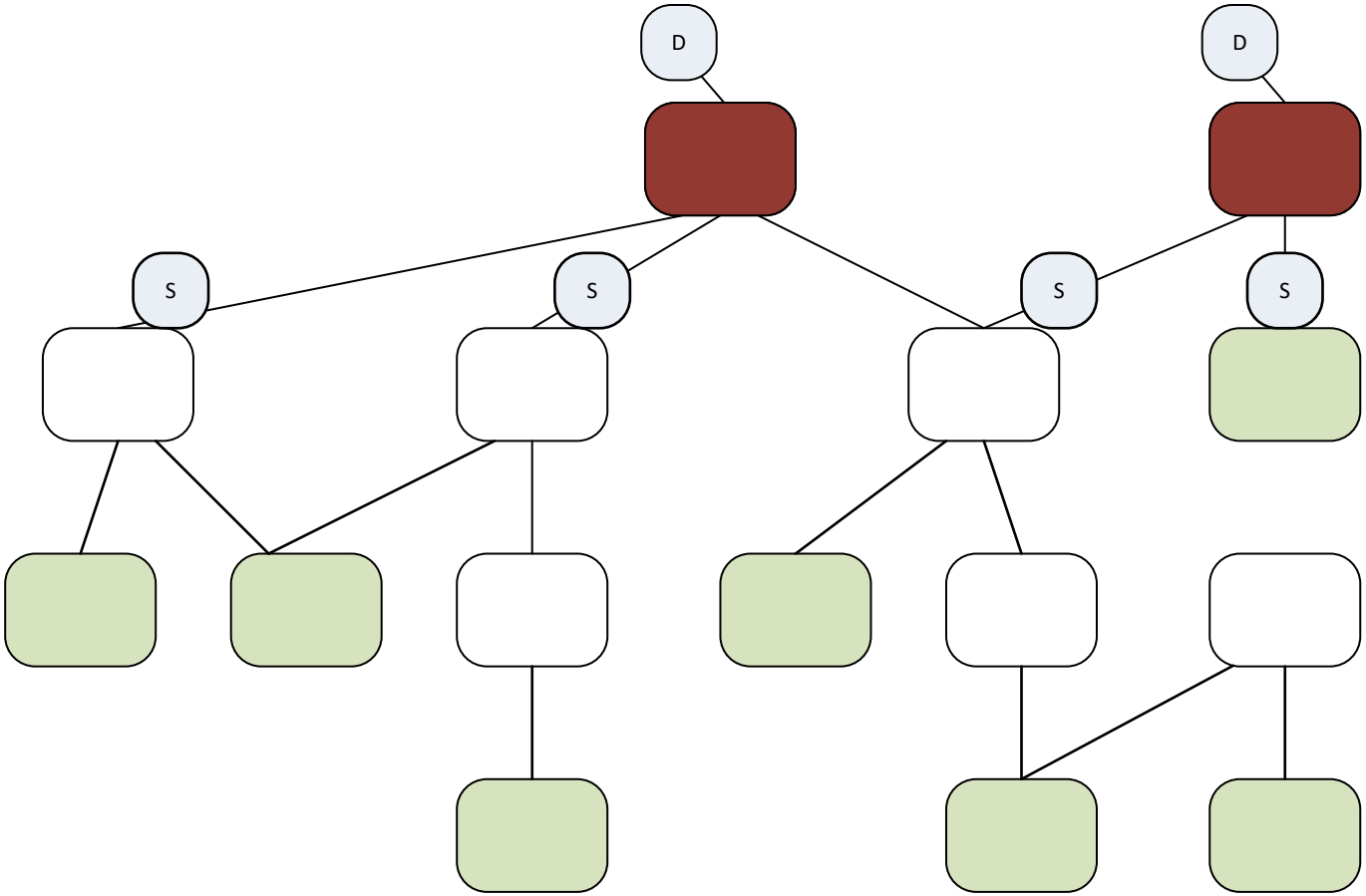
Especially useful for higher-level system tests (component, subsystem)

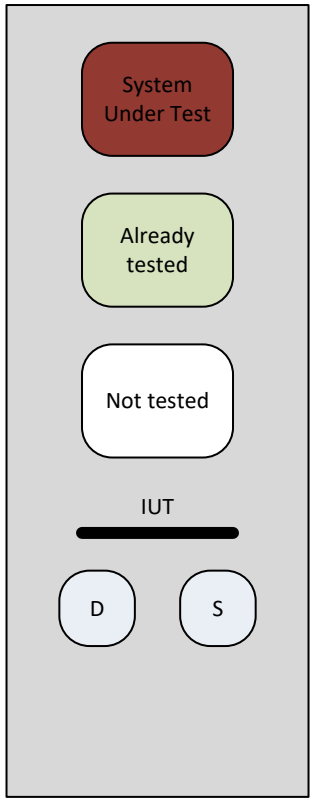
Models iterative development with UCs as unit

Sandwich Integration

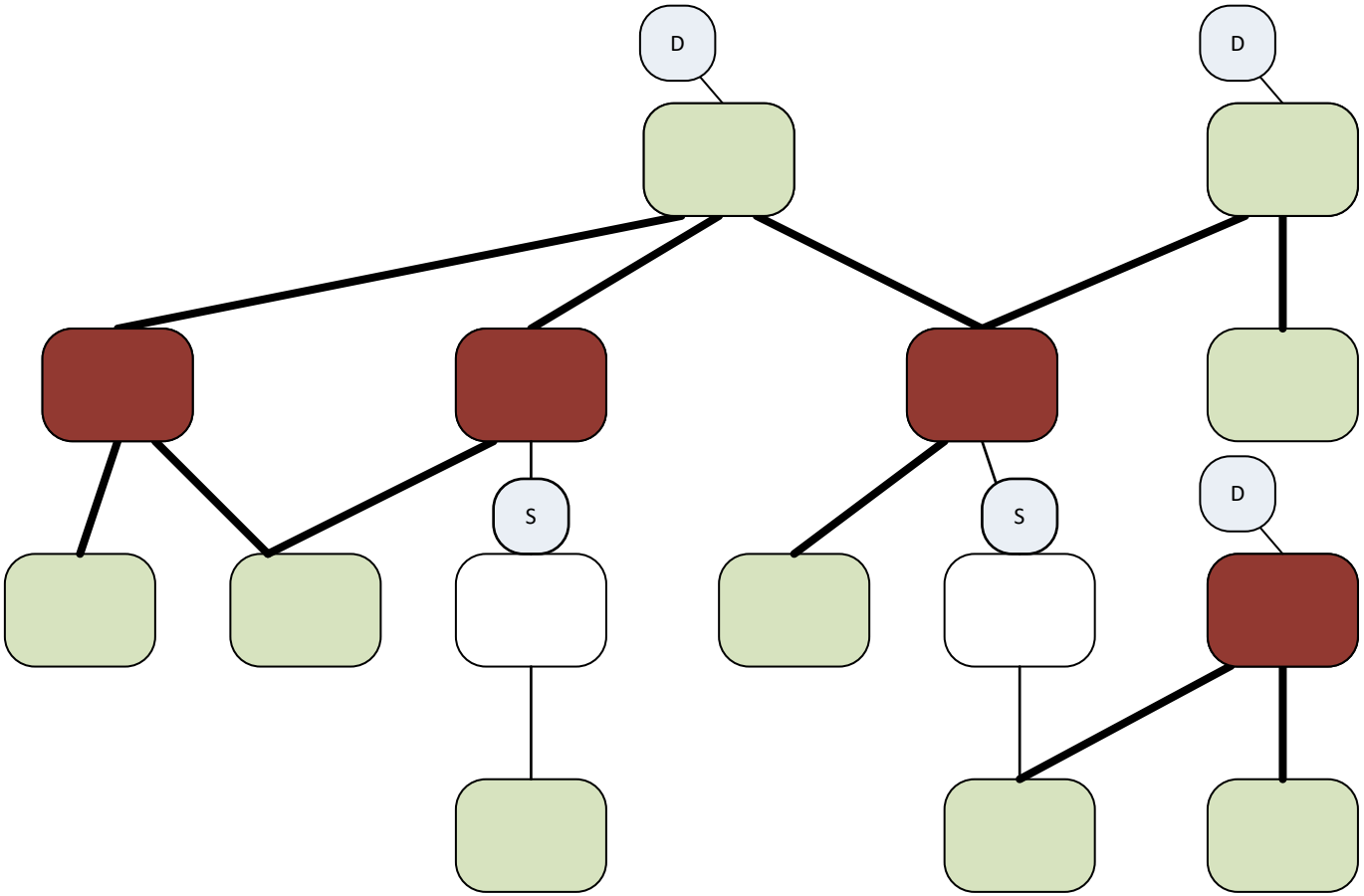


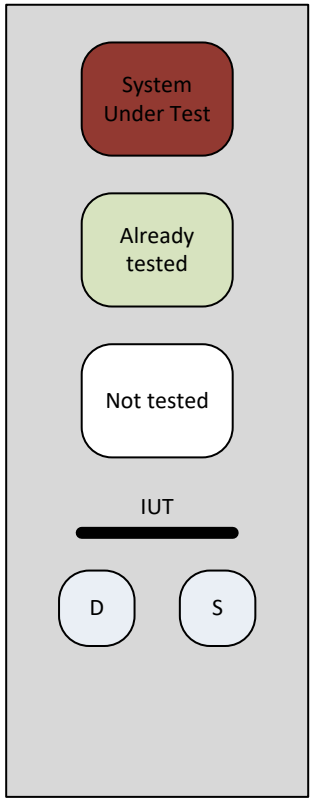
Sandwich Integration



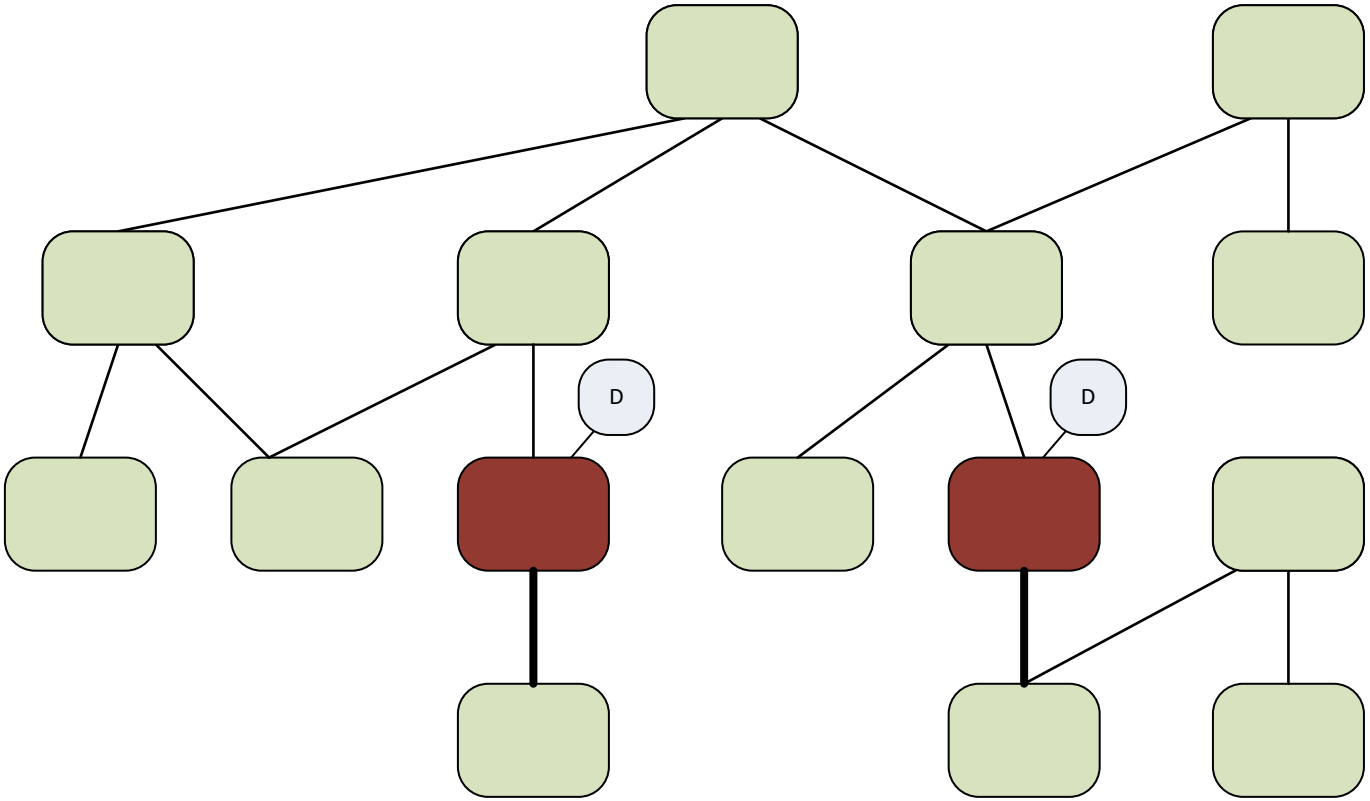


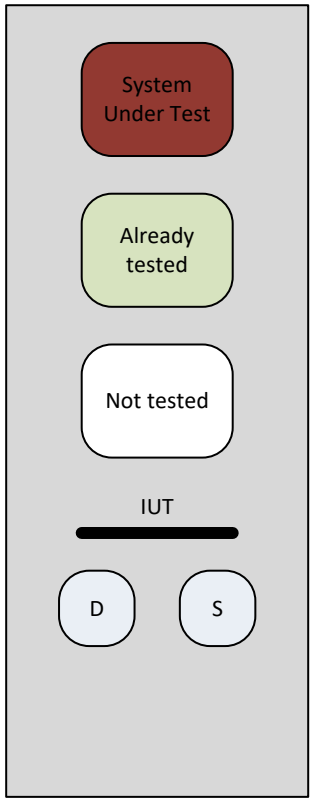
Sandwich Integration



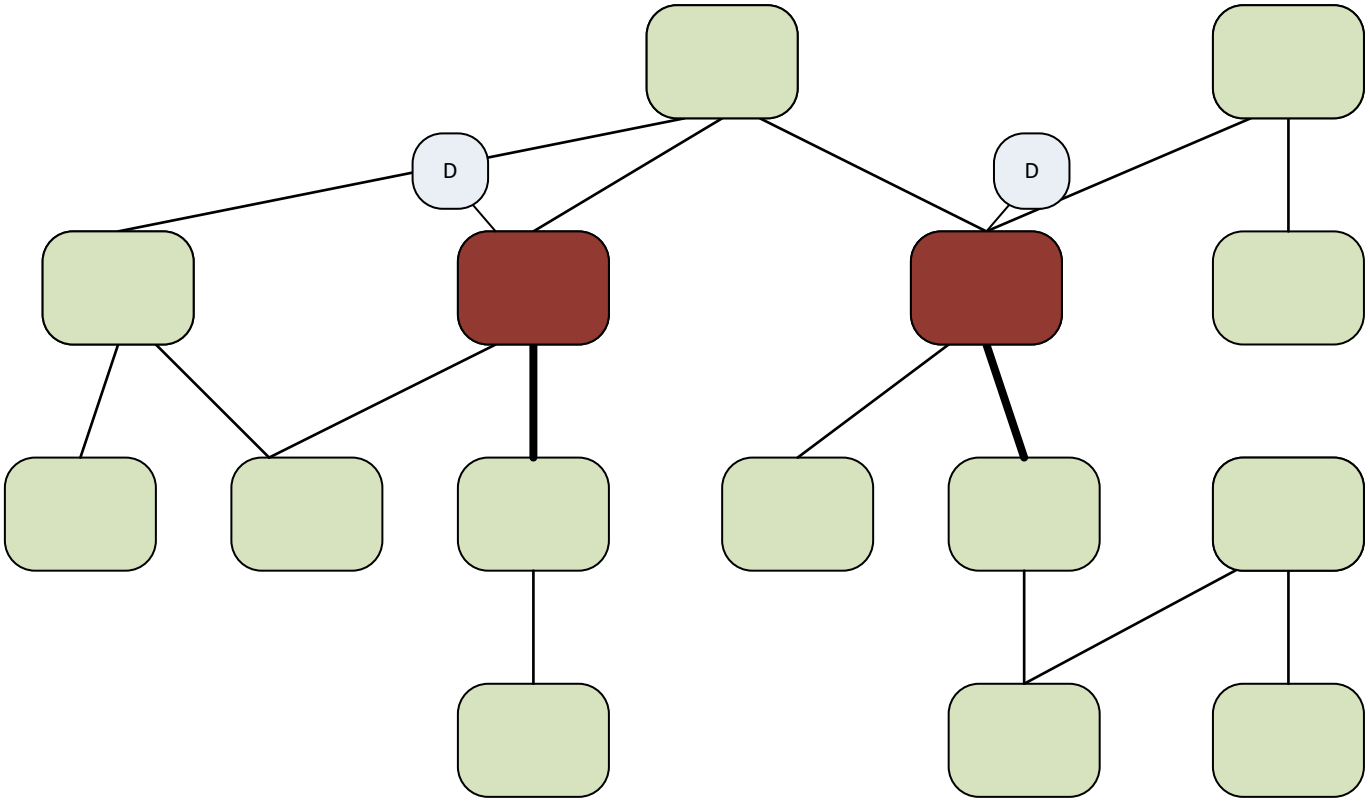


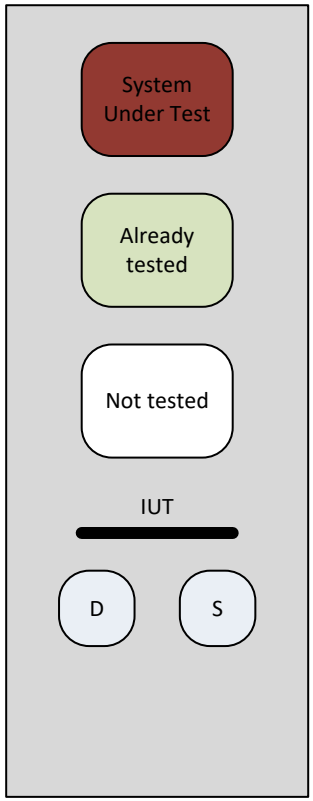
Sandwich Integration



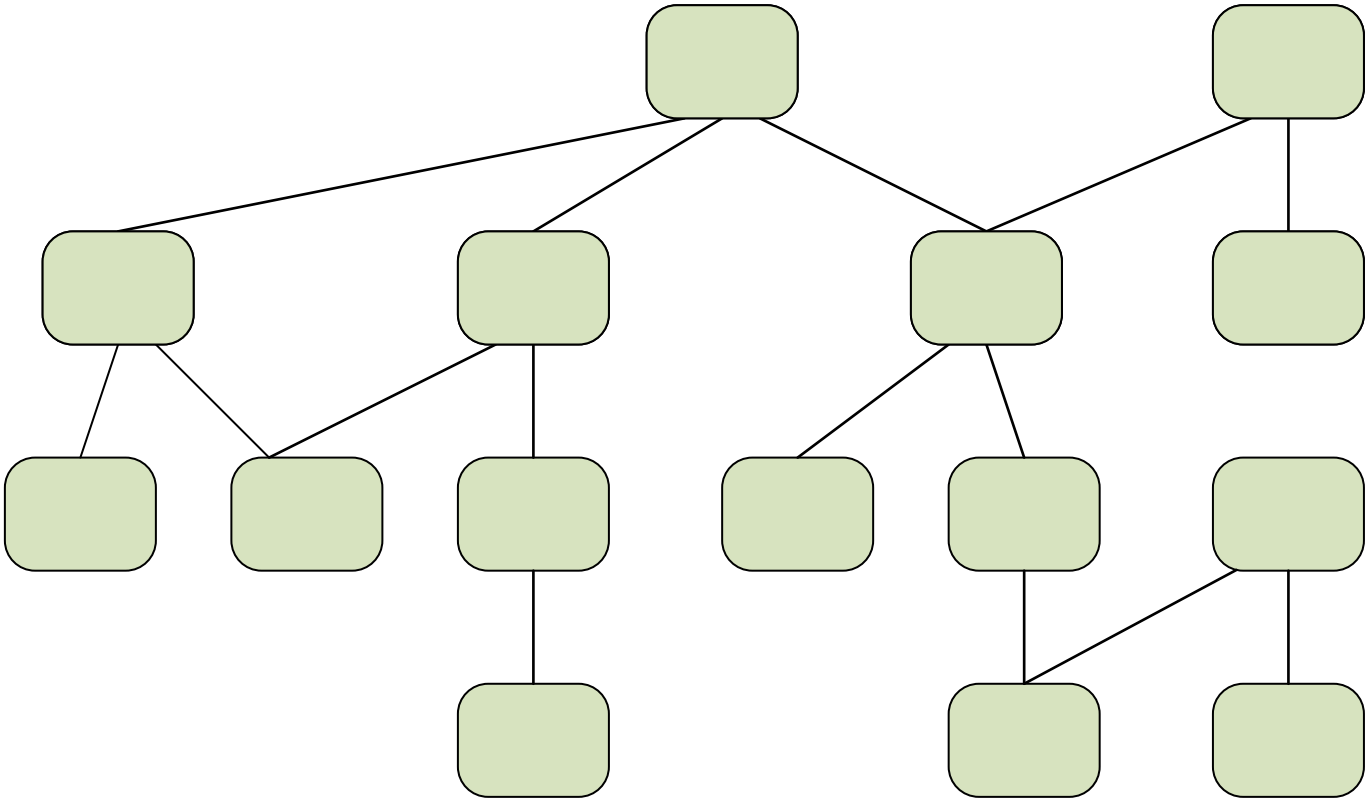


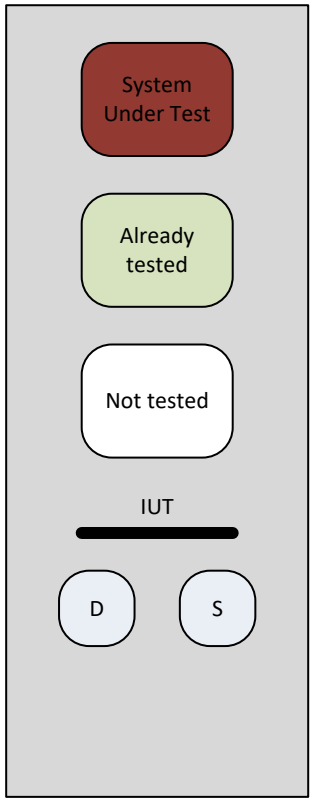
Sandwich Integration



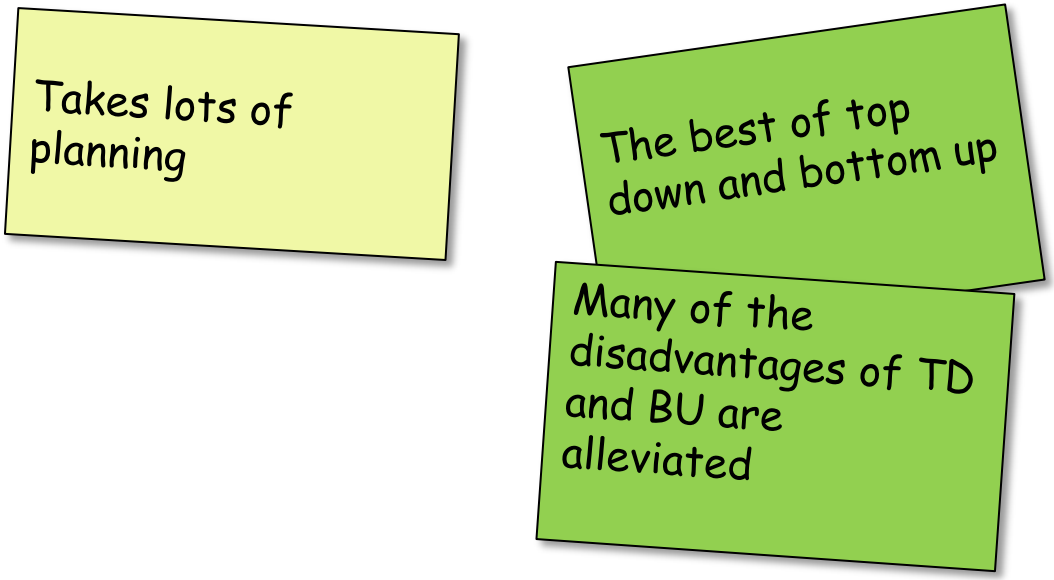


Sandwich Integration





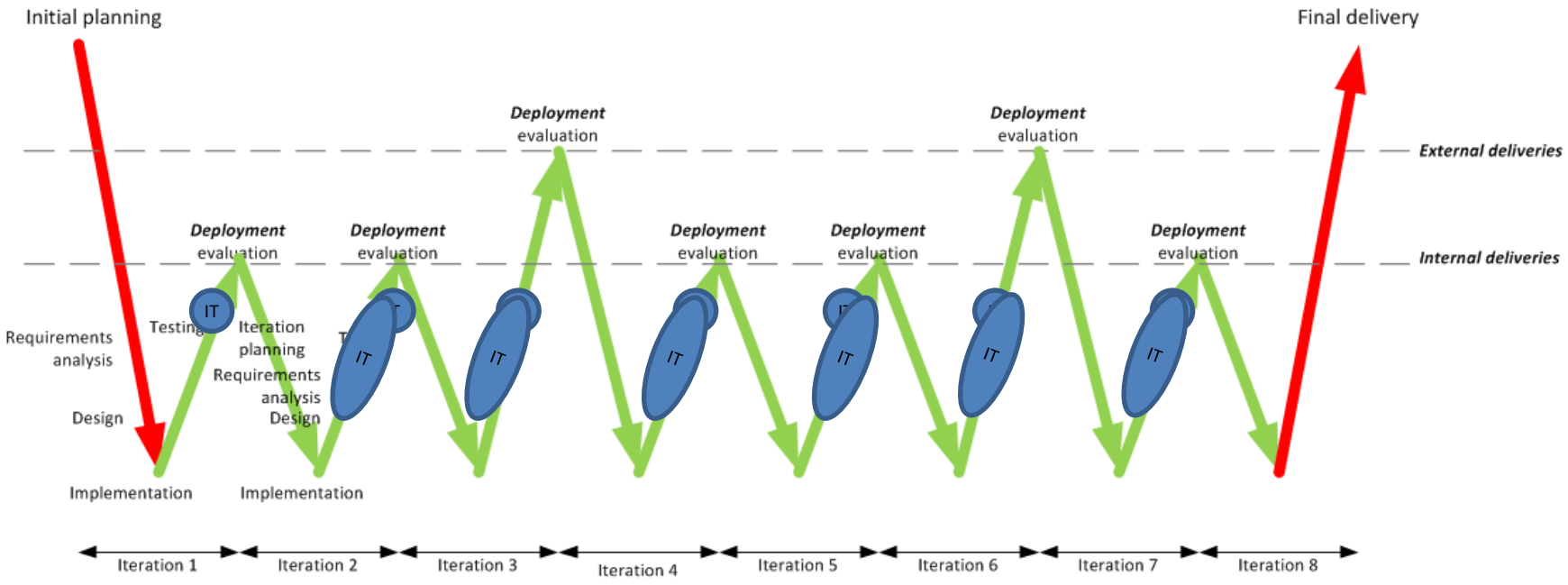
Sandwich Integration



Best practices

- Automate, automate, automate
  - Higher complexity → lower probability
  - Use High Frequency integration: Nightly CI builds for each test scenario to check integration (common code repo)
  - Use Unit Test and isolation frameworks as much as possible
- Easier to code stubs than drivers
- Use a little bit of all patterns (except Big Bang)

Integration test with Continuous Integration



Dependency Tree – What?

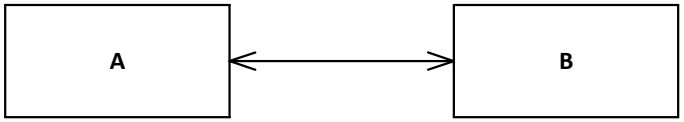
- The Dependency Tree must reflect the dependencies in the REAL code!
- It does NOT show interfaces!
- Search for usage of base classes
  - They may be used implicitly, e.g. as parameters
  - Include all derived classes!
- Search for usage of interfaces
  - They may be used, e.g. as parameters
  - Replace them with all implementations!

Step 1 – Find the Dependency tree!

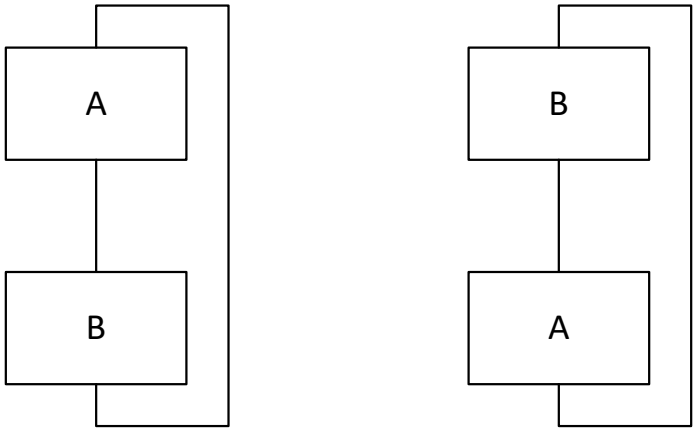
- Class diagrams (static information)
  - Remember polymorphic classes
  - Remember interfaces and base classes
- Sequence diagrams (behavioral information)
- Test cases for units
  - Which interfaces were defined
  - Which IFs were faked
- Call trees? Manually or automated tool?
  - Reveals "tool classes", objects passed around
- Dependency tree generator tool?

Dep. Tree trap – bidirectional association

Classes with bidirectional association
(class diagram)



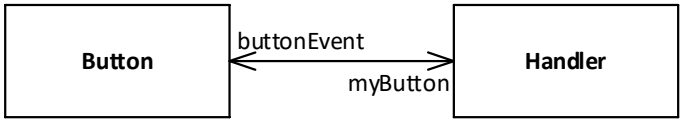
Which depends on which?
(dependency diagram)



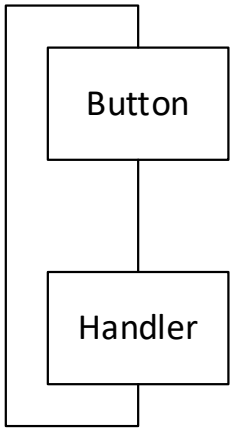
- Choose the one that fits the flow of the system best, or gives the best integration plan
- In some steps, you may need both the real and a fake version of one of the classes
- Don't panic – just do it!

Dep. Tree trap – events

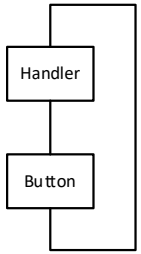
Classes with event connection (class diagram)



Which depends on which? (dependency diagram)



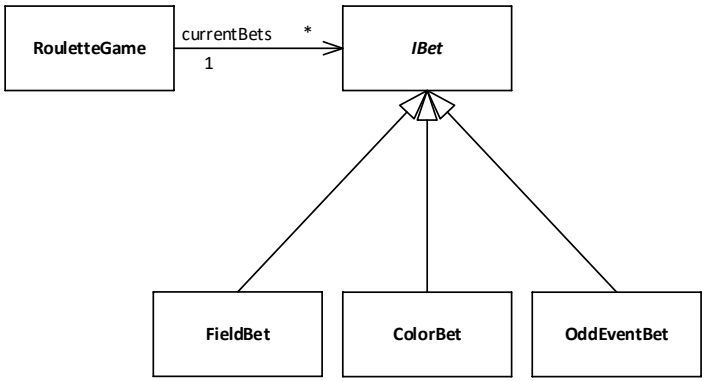
Typical choice for a Button – for other classes with events it may be different



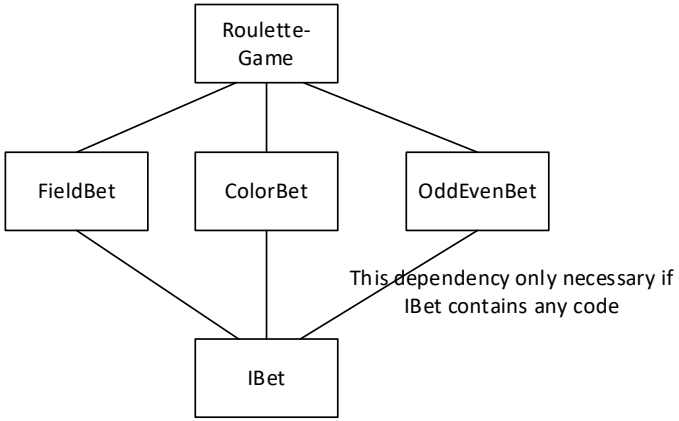
- A C# event is a dynamic bidirectional association
- Look at the flow of actions through the system
- Buttons and timers are examples
- Timers are of one of the types of classes it is a good idea to fake as long as possible
- Often one of the directions is only for setting up the **event connection;** that is often the least important one

Dep. Tree trap – inheritance – dependency inversion

Class diagram with inheritance hierarchy



Dependency Tree



- Integration planning concerns the integration of the real code.
- Dependency tree must show the dependency between the actual code as implemented!
- Not between abstract interfaces!
- Often this is reflected in the inversion of the inheritance hierarchy from the class diagram to the dependency tree

Exercise

- Find the dependency tree for the Microwave Oven System.
- Look for dependencies using the list on the previous slide
- Remember – it is the actual code/modules/classes that must be integrated!