Software Engineering 301:
Software Analysis and Design

# Testing at different granularities

# Agenda

- Introduction
- Unit testing
- Integration testing
- System testing

# Example scales of testing

- Unit testing
  - test individual “units” of the system (in OO, this basically means classes) in isolation
  - but how can we test a unit that depends on other units?
- Integration testing
  - start putting some units together and see if they work as a group
- System testing
  - test the entire system

# Why test at multiple scales?

- If my unit tests all work, why bother with integration or system tests?

1. Problems may involve incompatible components that work fine in isolation
2. Test cases may not have detected problem

# Why test at multiple scales?

- If system testing passes, why bother with integration or unit testing?
- Again, test cases may have been good enough
- Software evolution!
  - maybe the system doesn't currently use the code with the problem in it
  - later changes may cause this problem code to be used and only then faults will occur

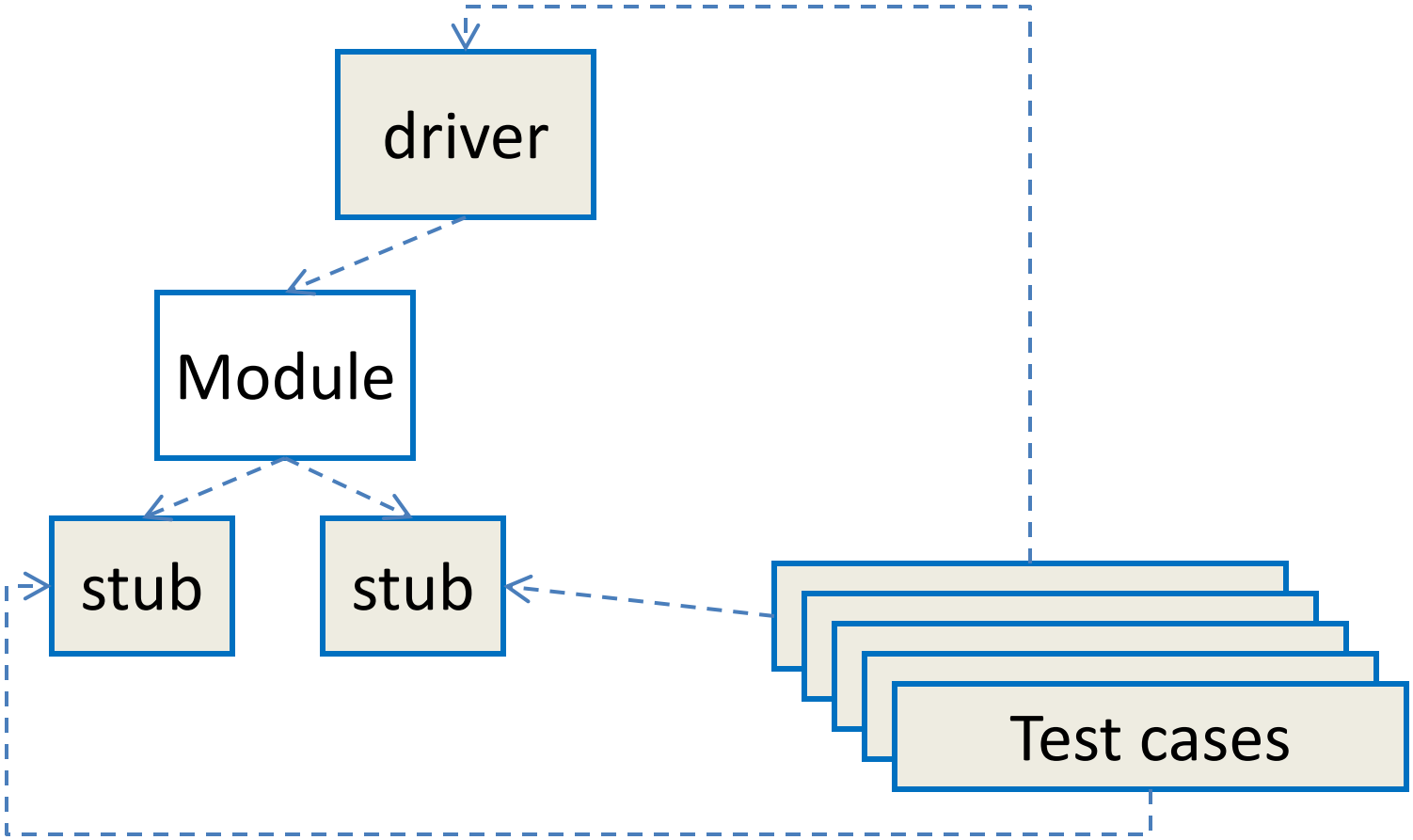
# Agenda

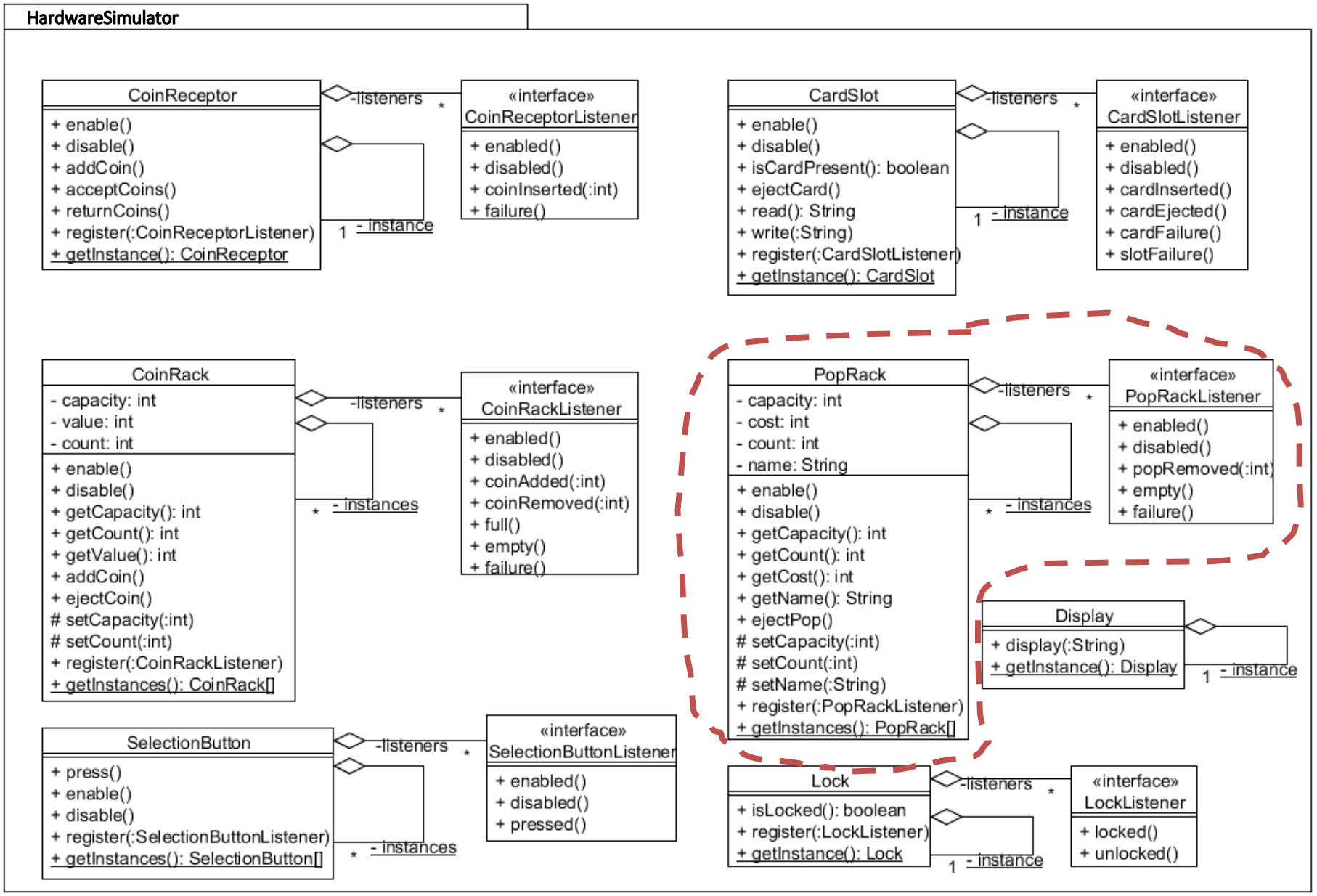
- ~~Introduction~~
- Unit testing
- Integration testing
- System testing

# Drivers and stubs

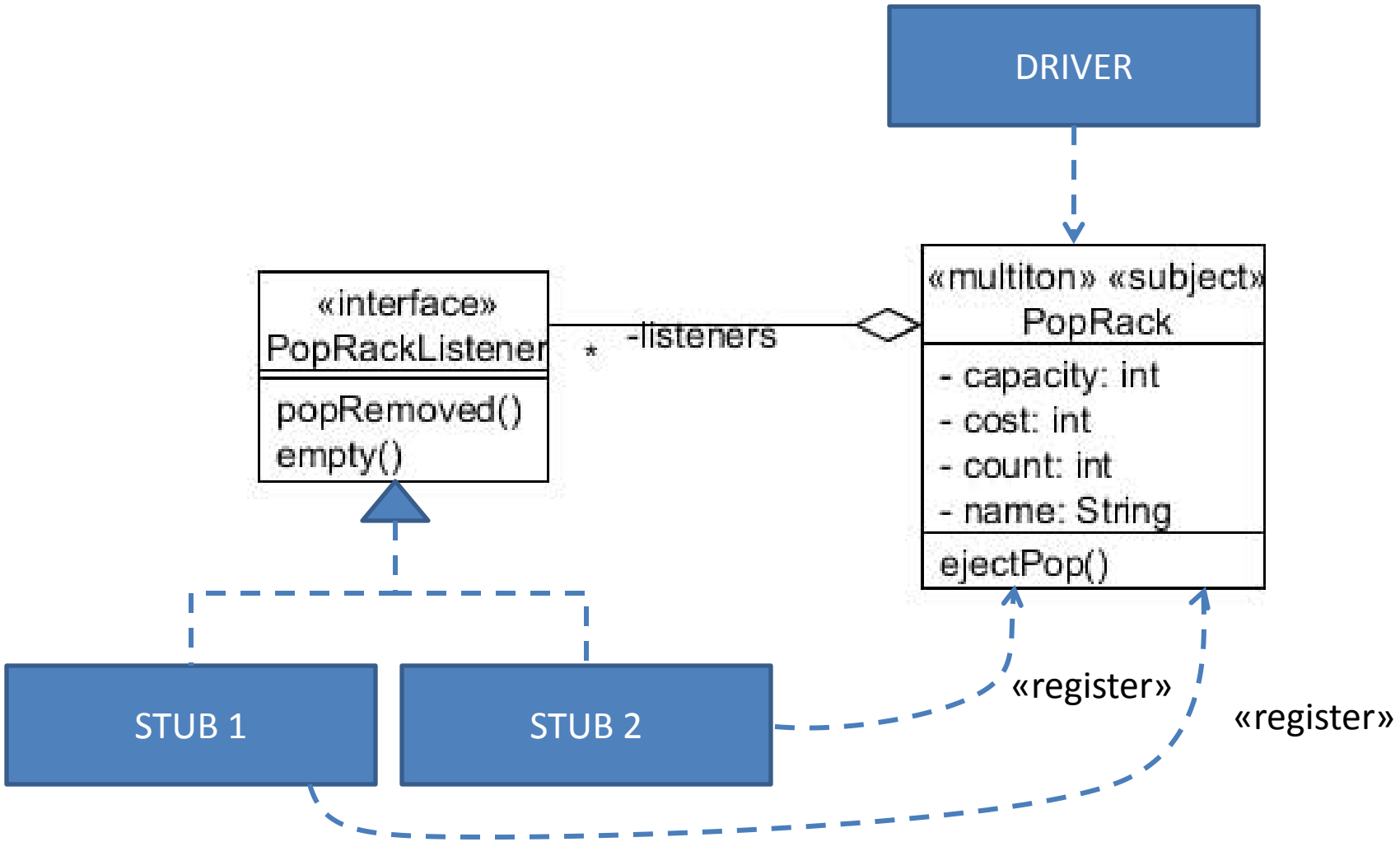
- A driver is a piece of testing code that makes calls to a module that is to be tested
  - It “drives” the test
- A module that is to be tested typically depends on other modules
- A stub simulates the presence of those modules
  - Low or no functionality, typically
    - Set a flag to determine which test case is being run, return appropriate results
  - Stubs can be expensive to create

# Unit testing environment





# Where to put the stubs & drivers?



# Drivers and stubs more generally

- I can draw a circle around any parts, and treat this as the thing under test
  - If any dependencies (incl. associations, etc.) cross my circle outwards, these dependencies have to be stubbed out somehow
  - I need to be able to call the methods provided by the interfaces within the circle

# Stub alternatives

- Stubs are a kind of entity sometimes called a test double (like a stunt double in movies)
- There are a variety of subtly different kinds of test doubles
  - Dummy objects
    - used for parameters that will be ignored
  - Fake objects
    - working implementation but not suitable for production
  - Stub objects
    - canned answers for specific expected calls, usually based on expectations of state changes
  - Mock objects
    - expectations as to specific message sequences that should be received

# Agenda

- ~~Introduction~~
- ~~Unit testing~~
- Integration testing
- System testing

# Why integration testing?

- If all units work individually, why doubt that they work together?

  1. Unit test cases may not have discovered some faults

  2. Assumptions about mutual interfaces may differ

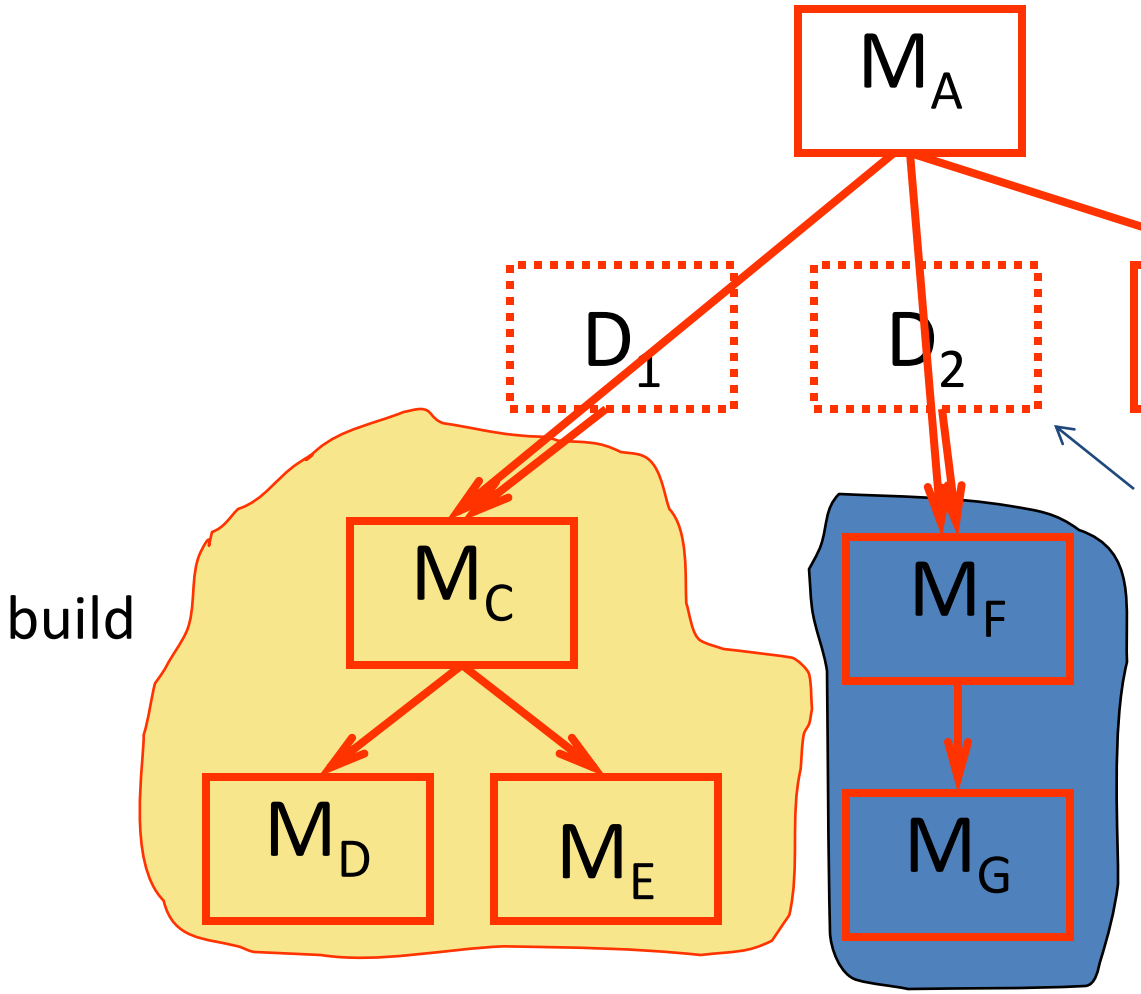
# Integration testing

- How to go about it?
- Different strategies:
  - Big bang integration
  - Bottom-up integration
  - Top-down integration
  - Sandwich integration

# Big bang integration

- Assumes all components have been tested individually
- Put them altogether, run some tests … and we're done! Simple, right?
- Remember that the point of testing is to find failures so their underlying faults can be repaired
  - Is the fault in the interface description or in component implementations?
  - Which component contains the fault?
  - Hard to answer these questions with big bang integration

# Bottom-up integration



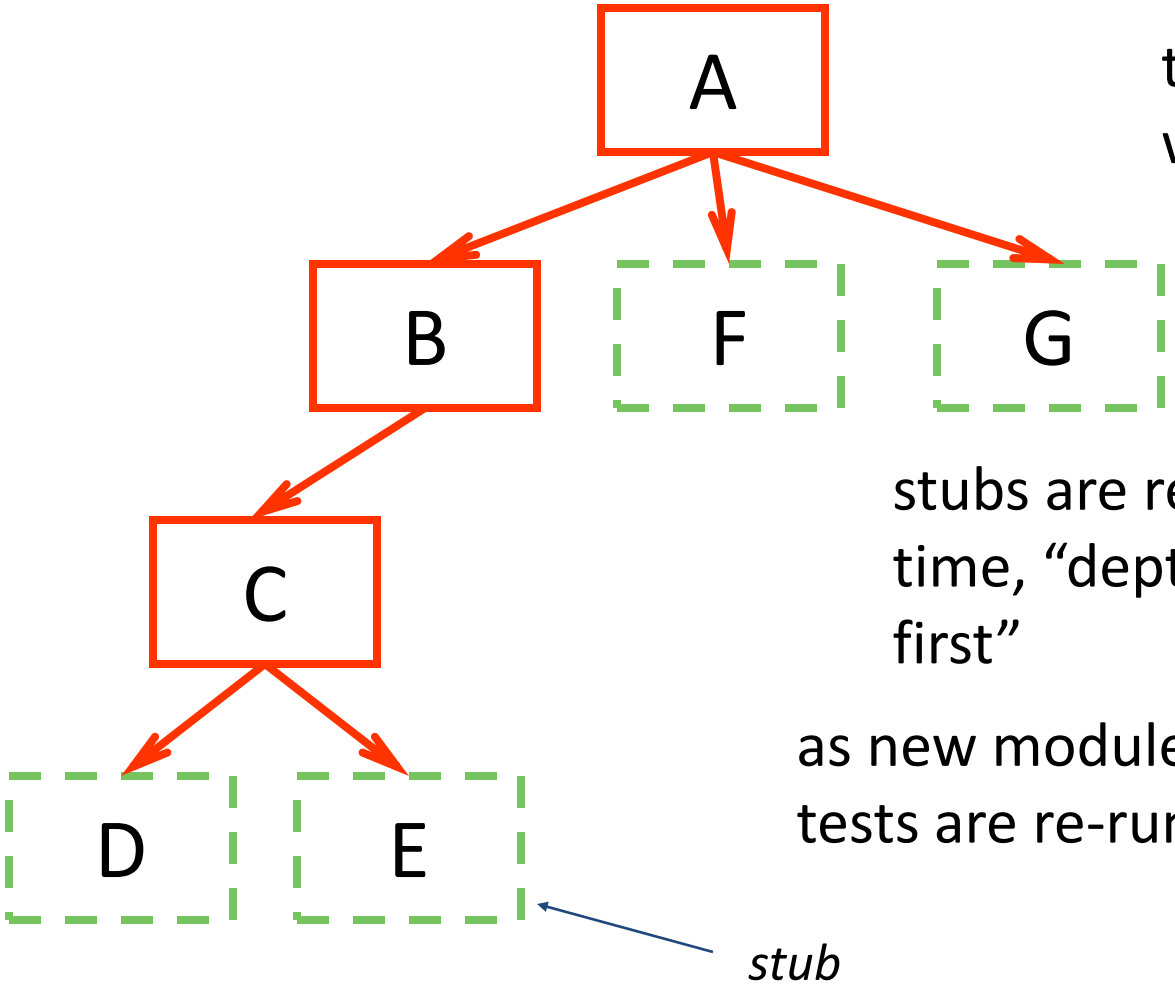

drivers are replaced one at a time

Low-level modules are grouped into “builds” (or “clusters”) and integrated

# Bottom-up integration

- Issues:
  + avoids need for expensive stub creation
  + replacement of driver with higher-level component can allow interface violations to be tracked down quickly
  - most important components tested last, i.e., the system interface components
  - problems in interface components can lead to widespread changes, requiring testing to be largely redone

# Top-down integration




top module is tested with stubs

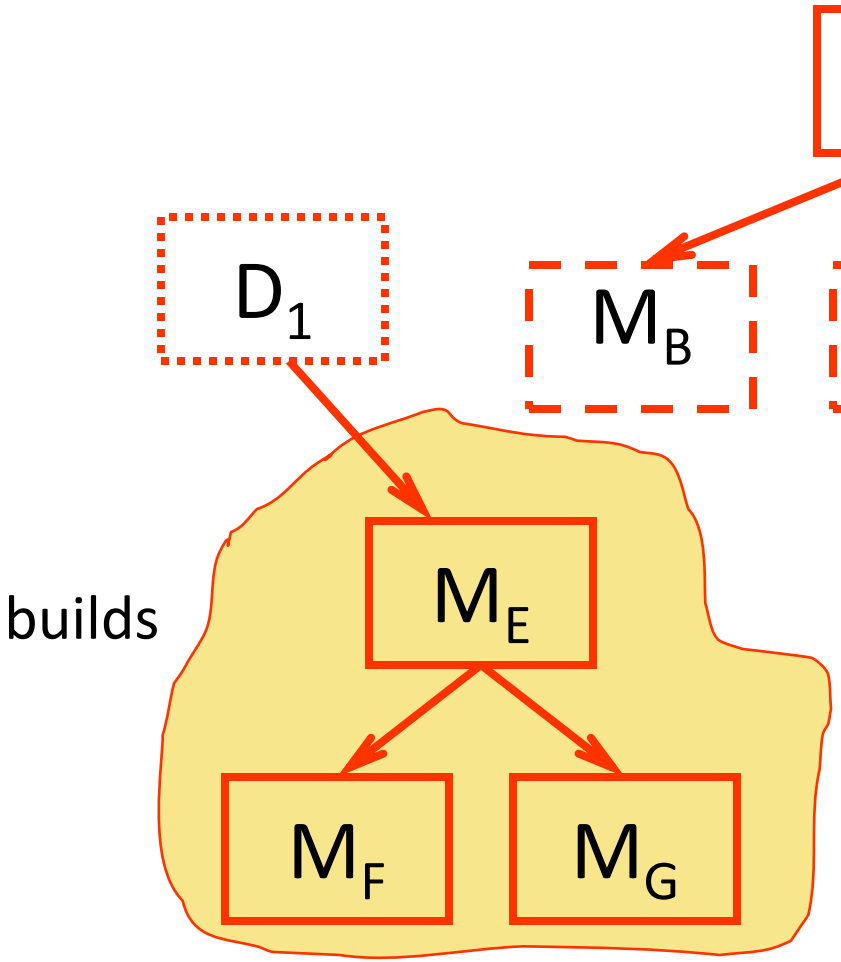
stubs are replaced one at a time, “depth first” or “breadth first”

as new modules are integrated, tests are re-run

# Top-down integration

- Issues:
  + starts by testing system interface components
  + same test suite can be used to test ever deeper into the system
  - test stubs are expensive to create, easier to use the real components
    - degeneration into big bang testing

# Sandwich integration testing




Low-level modules are grouped into builds and tested with drivers

Progressively integrate more into the middle of the “sandwich”

# Agenda

- ~~Introduction~~
- ~~Unit testing~~
- ~~Integration testing~~
- System testing

# System testing

- Functional testing
- Performance testing
- Usability testing
- Pilot testing
- Acceptance testing
- Installation testing

# Functional testing

- Look for discrepancies between requirements for system and reality
  - work from use case model
    - perform scenarios that try out each use case and each of its extensions
  - use case model is a black-box description of the system
    - can perform equivalence or boundary analysis to select test cases
  - *"monkey testing"* can also test for the unexpected
- Test-driven development
  - functional testing is a given
- For assignment 3, I'm really asking you to do this for the system test plan

# Performance testing

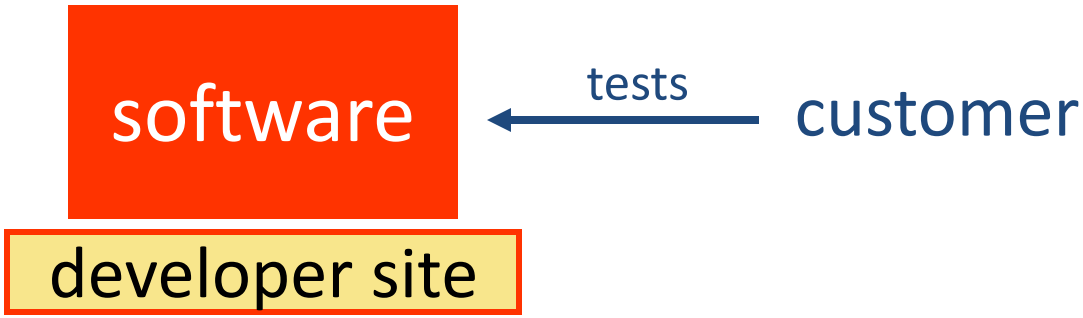
- Functionality is important but not sufficient
  - stress testing (handling requests)
    - often pushes the system until it does fail, even beyond the required performance
    - an unusual example: frequently dropped connections
  - volume testing (data volume)
  - security testing
  - timing testing
  - recovery testing

# Usability testing

- Needed functionality may be hard to use
  - watch users and record what they do
- How and when to test?
  - Scenario tests
    - how quickly are users able to understand how to perform them?
  - Prototype tests
    - paper prototypes, mock-ups
    - “Wizard of Oz” prototypes
  - Product tests
    - functional versions

# Pilot tests

*Alpha test*




*Beta test*

# Acceptance testing

- Is the product good enough?
- Often done for legal sign-off
- Run customer-defined benchmarks
- Compare against existing systems
- Generates report of remaining problems that must be addressed
- After acceptance, *installation tests* must be performed to ensure that the system works in the deployment environment
  - success means that the system is ready

# Next time

- Test case selection