COMS W4156 Advanced Software Engineering (ASE)

November 10, 2022

Agenda

1. finish Github Actions demo
2. Integration Testing

Github Actions demo: Prahlad Koratamaddi

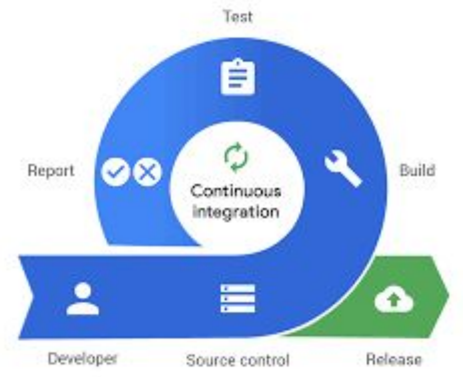
GitHub Actions demo presentation

Read the Docs

Github Actions Documentation

CI/CD Explained

Github Actions as a DevOps Platform



Agenda

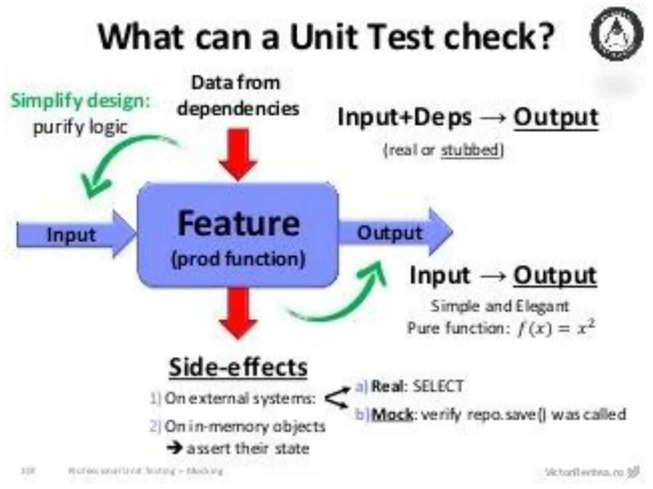
1. finish Github Actions demo
2. Integration Testing

$$Life = \int_{birth}^{death} \frac{happiness}{time} \Delta time$$

Unit Testing

Unit testing tests that individual "units" (methods or functions) do what they are supposed to do

Unit tests aim to test a method *in isolation from the rest of the program*. They call the method from a test runner rather than from the other code in the rest of the program



When a method would normally call other code, use data, or otherwise interact with the environment outside the unit, it instead calls test doubles (mocks and stubs). All dependencies are replaced except for basic libraries like string processing and math
.
Unit tests need to be very fast, so they can run whenever the unit is changed. Real resources are often too slow, but that is not the only reason for using mocks/stubs:
⇒When a unit test fails, we know the bug is in that unit, not in some other unit

Class Testing

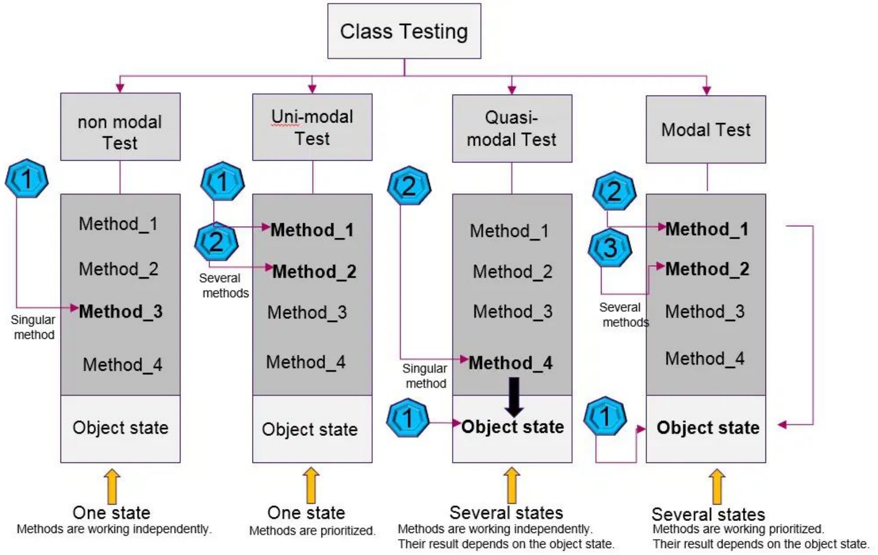
Class testing tests that individual classes do what they are supposed to do

Class tests aim to test a class *in isolation from the rest of the program*. They call a series of units in the same class from a test runner (and from each other) rather than from the other code in the rest of the program

When code in a class would normally call other code, use data, or otherwise interact with the environment outside the class, it instead calls test doubles. All dependencies outside the class are replaced except for basic libraries

⇒When a class test fails, we know the bug is in that class, probably at an interface between units within that class (assuming no errors were found during pure unit testing)

Class Testing is a Special Case of Integration Testing



Although similar to integration testing in that it combines multiple already-tested units, class testing is often conducted together with unit testing or considered part of unit testing, with any test doubles for same class replaced by the actual class code/data

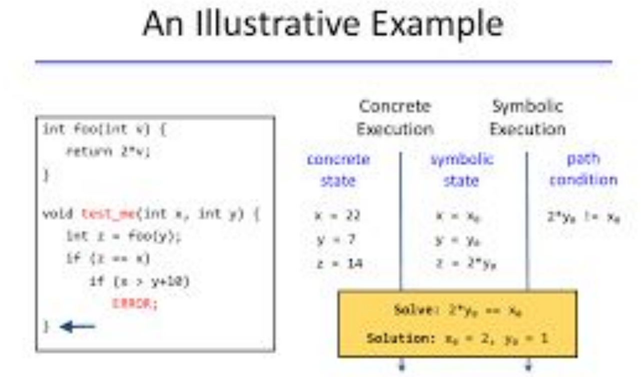
The first phase of class testing is to re-run existing unit tests for the units in the class where the unit under test calls another unit in the same class - but now calling the real unit not a test double, thus exercising the direct dependency

Choosing Inputs for Class Testing

Minimize redundancy: skip any tests that would exercise the same interface (method m calling method n) with inputs equivalent to some previous passing test (m already called n with the same or similar parameters and other inputs)

If some cross-unit calls cannot be reached through existing test cases, peek inside the desired caller to determine the execution path(s) where it calls the callee, and choose inputs to force each such execution path

Work backwards from a call to n through any branches and loops to the beginning of the caller m, to determine the *path conditions* (constraints) on m's own inputs needed to force execution to reach that call to n



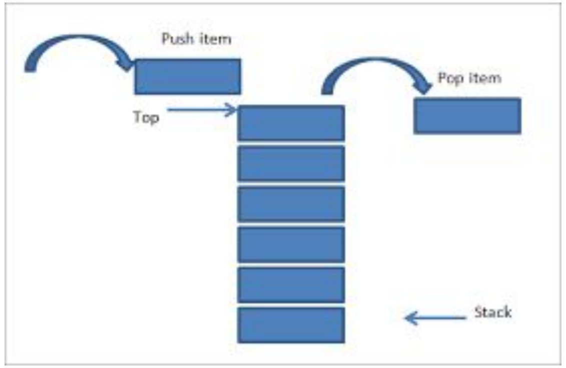
```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Choosing Inputs for Class Testing 2

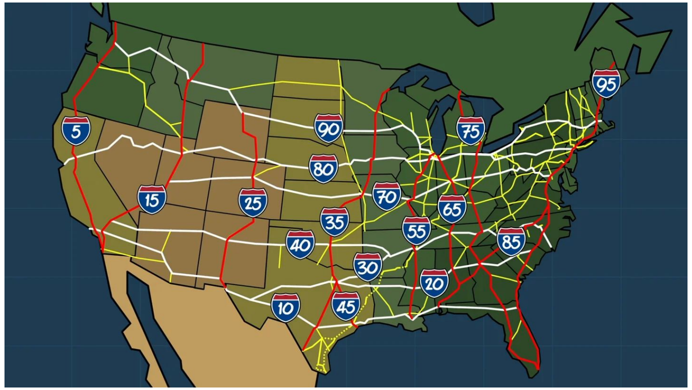
Path conditions do not address cases where two or more units *share data* but neither calls the other - e.g., push() and pop() are dependent on each other because they share the stack data structure, but they do not normally call each other

So also need to exercise all sets of units in the same class that access the same shared data

“State transition testing” calls units in a class in some order that covers all the *states* of that shared data



State Transition Testing



Assumes state shared among the methods of a class, so may not apply to “utility” classes that group methods that share little if any state

In an object-oriented language, the “state” of an object is represented by its fields (or attributes, instance variables, etc.)

And possibly also the classes’ static fields (or class variables, etc.), which are shared among all objects of the class

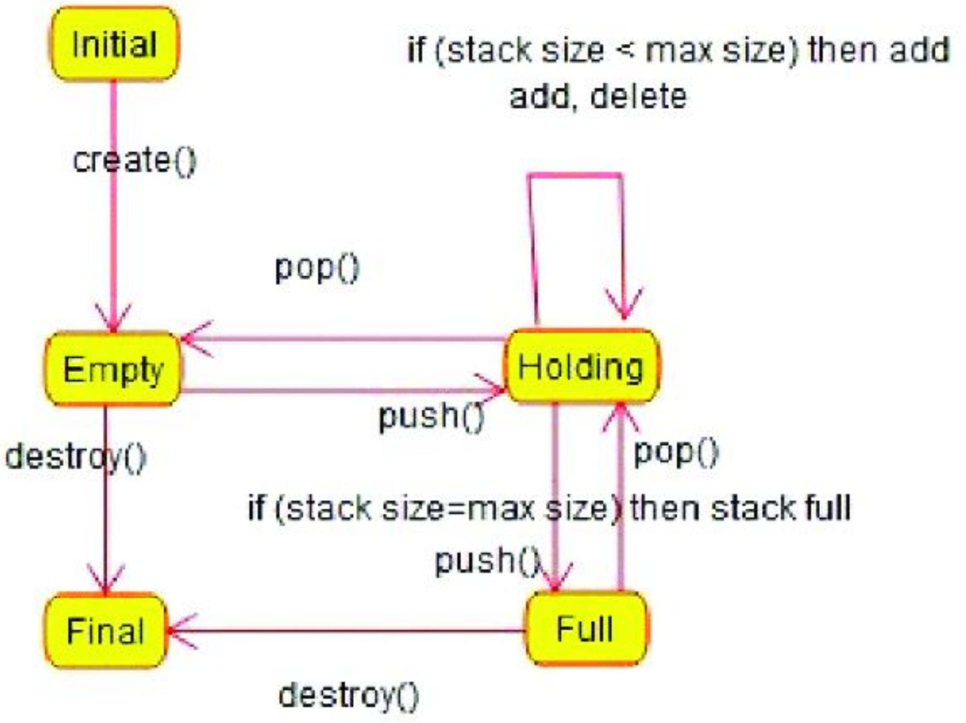
Approach also applies to non-OO languages if state scoped to modular units

Simple Example: Testing a Stack Data Structure

Draw “state diagram” to visualize the possible states and their transitions

The idea is to choose combinations of methods and inputs to those methods that exercise all the transitions between states

There may be constraints on some transitions



Simple Example: Testing a Stack Data Structure

Test 1: create()

Test 2: destroy()

Test 3: create() - need new one

Test 4: push(xxx)

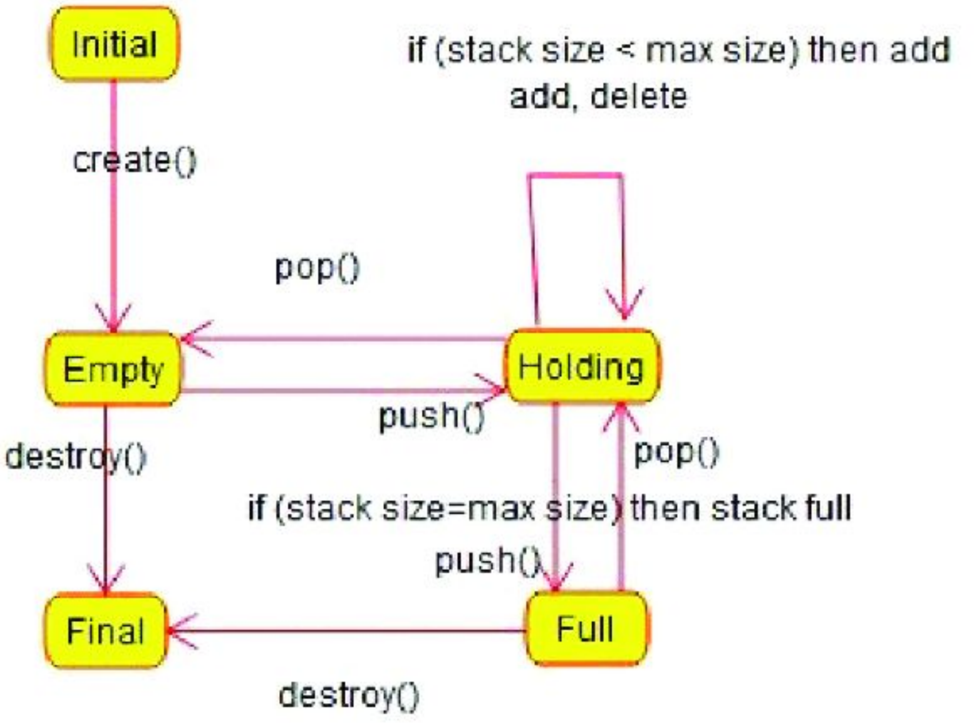
Test 5: pop() - did we get xxx?

Test 6: push(yyy)

Test 7: push(zzz)

Test 8: pop() - did we get zzz?

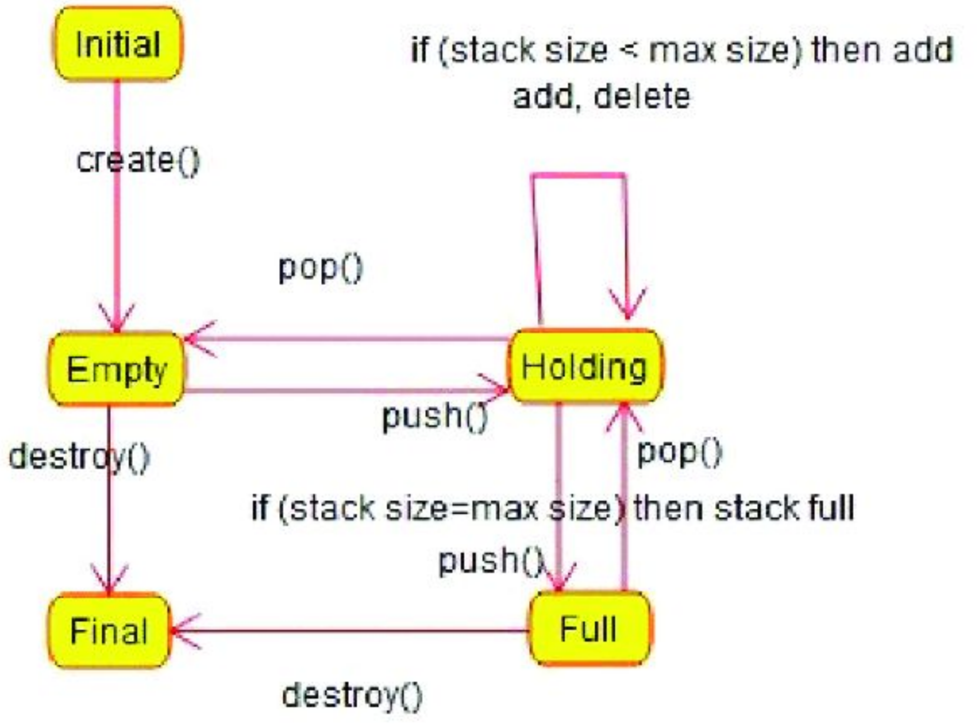
And so on...



Simple Example: Testing a Stack Data Structure

This could go on forever, need "stopping condition"

- Follow each full path without loops
- Do each loop zero, one and two times around
- Also try invalid paths - method sequences disallowed by constraints or not included in diagram

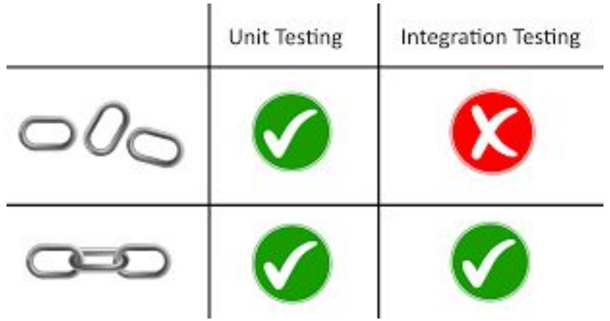


Integration Testing

The goal of *integration testing* is to find errors at the interfaces **between** classes and assemblies consisting of multiple classes

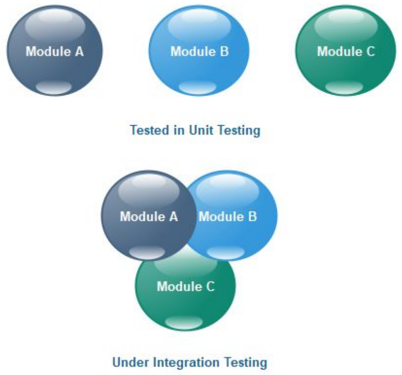
Integration testing considers two or more components that communicate or depend on each other in some way (or where one component communicates with or depends on another component, does not need to be reciprocal)

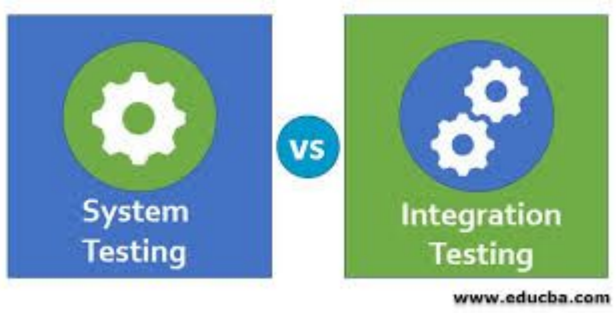
Integration tests are often automated via the same testing tools as unit testing, with constructed inputs, assertions that check outputs, setup, teardown, etc.



	Unit Testing	Integration Testing
	✓	✗
	✓	✓

Choosing Integration Testing Inputs



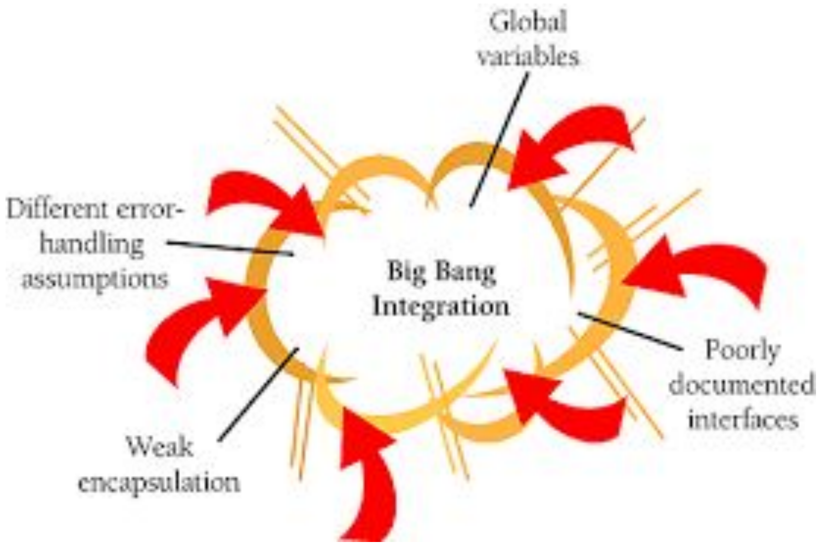


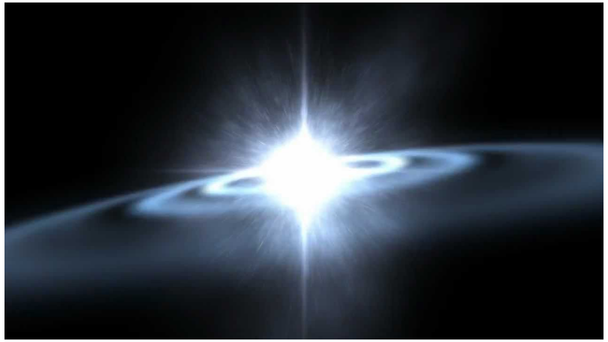
Similar to class testing except 1. some integration order strategy is usually chosen in advance (explained next) and 2. source code may not be available for all components to be integrated (e.g., third-party libraries, external resources)

Given two components (or previously integrated assemblies of components): replace test doubles with real code from the other component being integrated, avoid redundant tests, choose additional test inputs to reach replaced test doubles as needed

Integration Testing Strategies

“Big bang” approach to integration testing: Unit and class test all independently developed classes in isolation, then test the whole system. All done!


Global variables
Different error-handling assumptions
Big Bang Integration
Poorly documented interfaces
Weak encapsulation

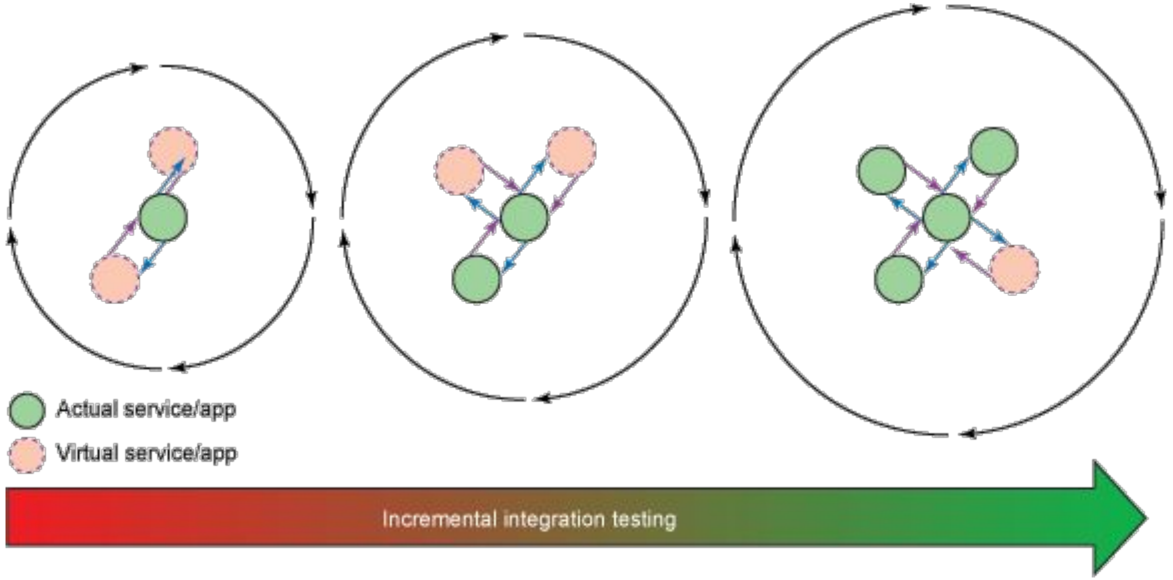


Not so fast: Bugs detected during “big bang” could be anywhere!

Imagine Big Bang Testing for Automobiles

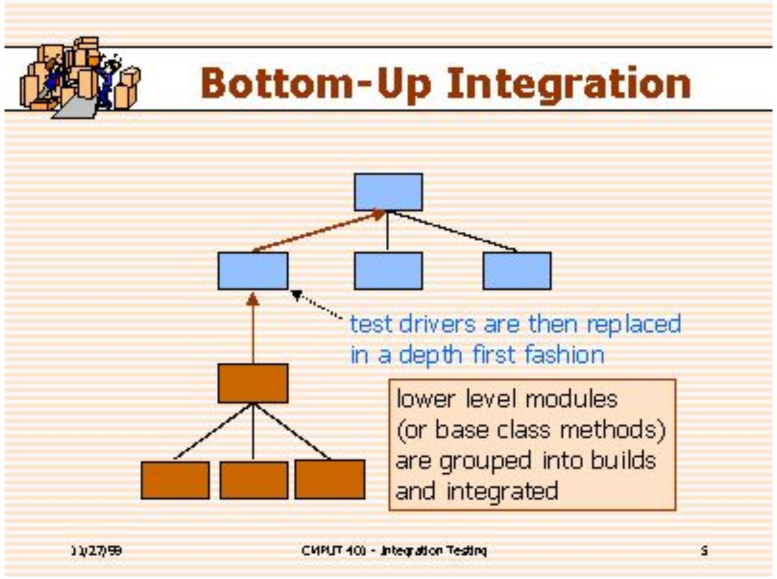
Incremental Integration Helps Localize Bugs

At each step, pick two components (or assemblies of components) that have both already been tested individually and at least one is dependent in some way on the other. Continue using test doubles for all other components



Since only a small number of components (ideally two) are integrated at a time, that limits where we need to look to find the “root cause” (the underlying coding mistake or interface mismatch) and fix the bug

Integration Testing Order

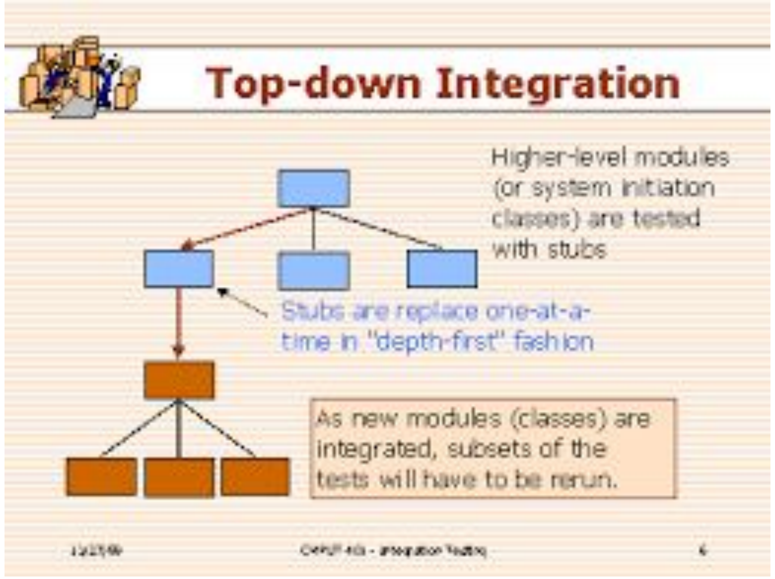


Bottom-Up Integration

test drivers are then replaced in a depth first fashion

lower level modules (or base class methods) are grouped into builds and integrated

11/27/99 CMPUT 401 - Integration Testing 5



Top-down Integration

Higher-level modules (or system initiation classes) are tested with stubs

Stubs are replace one-at-a-time in "depth-first" fashion

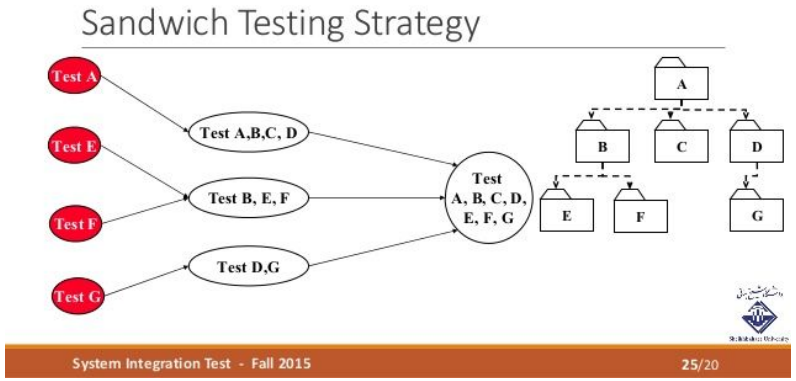
As new modules (classes) are integrated, subsets of the tests will have to be rerun.

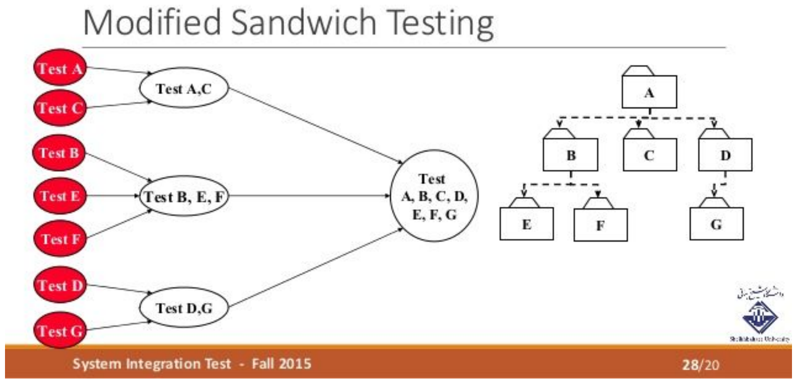
11/27/99 CMPUT 401 - Integration Testing 6

Bottom up starts integration process from lowest layer, top down starts integration process from top

Hybrid (“Sandwich”) Integration Testing

Integrate “natural” assemblies or layers, meet in middle - e.g., integrate front end and back end separately





Other Integration Strategies

History: Prioritize testing interfaces where the most bugs, or the most challenging bugs, have been found previously

Risk: Prioritize testing the *most risky* interfaces, e.g., a new component being introduced to an existing system, replacing an existing third-party component with an alternative component from a different third party, replacing an existing third-party component with a new version of that component

Expense: Prioritize testing the interfaces where debugging will be the most difficult and/or expensive

"The Key Is Not To Prioritize What's On Your Schedule, But To Schedule Your Priorities."
- STEPHEN R. COVEY

Integration Testing with Test Doubles

Integration testing should fake dependencies outside the code under test just like class and unit testing

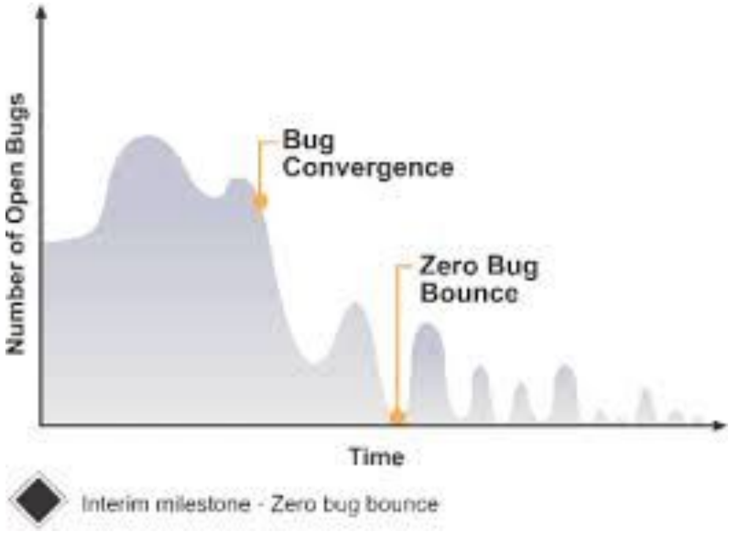
When an integration test fails, we want to know the bug is in one of the components being integrated into the assembly, probably at the interface being integrated, not some other part of the system

When integration testing specifically tests interfaces with external resources, the real resources are used after initial testing with test doubles

How Do We Know When We Are Done Testing?

- You’re *never* done if you keep changing the code (regression testing)
- This is why many software organizations “freeze” a planned release build a week or more in advance, with no changes allowed except bug fixes (with each set of repairs followed by full re-testing)

Ideally achieve *zero bug bounce*: Rate of finding new bugs is stable and low (approaching zero), reached point of diminishing returns



Upcoming Assignments

Second Iteration due November 28

Second Iteration demo due December 5

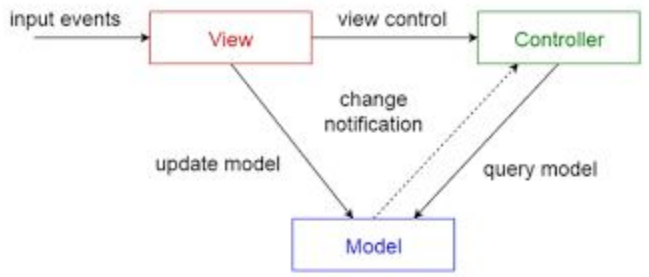
(a)

(b)

Next Week

Software Architecture

Design Patterns



Ask Me Anything