

COMS W4156 Advanced Software Engineering (ASE)

October 19, 2021

[shared google doc for discussion during class](#)

Do all Teams know who their IA Mentor is?

Assignments table posted on [piazza](#).

Refresher: What Kinds of Bugs Can Be Found By Testing



The tester enters a number, presses +, enters another number, presses =

- Nothing happens
- Computes the wrong answer

Calculator works correctly for some period of time or some number of computations, and then does nothing

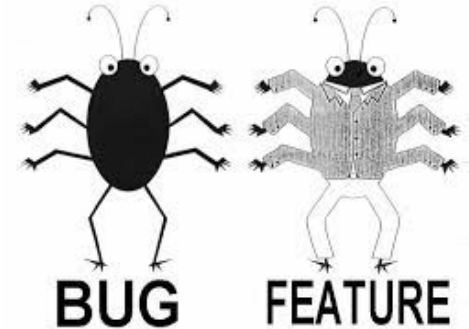
Calculator displays all 0's and won't do anything else

The calculator correctly adds, subtracts, multiplies and divides, but the tester found that if two operators are held down simultaneously, it appears to do square root

The calculator's buttons are too small, the = key is in an odd place, the display is hard to read, ...

Refresher: What Kinds of Bugs Can Be Found By Testing

- Software doesn't do something requirements say it should do
- Software does something requirements say it shouldn't do
- Software does something that requirements don't mention
- Software doesn't do something that requirements don't mention but should
- Software is difficult to understand, hard to use, slow, etc.



Refresher: Is This a Bug or a Feature?

The tester enters a number, presses +, enters another number, presses =

- Nothing happens
- Computes the wrong answer

Probably a bug

Calculator works correctly for some period of time or some number of computations, and then does nothing

Feature - trial period over, now you have to pay for it

Calculator displays all 0's and won't do anything else

Feature - battery low

The calculator correctly adds, subtracts, multiplies and divides, but the tester found that if two operators are held down simultaneously, it appears to do square root

Undocumented feature

The calculator's buttons are too small, the = key is in an odd place, the display is hard to read, ...

The user will consider these bugs even though the developers may not

Refresher: Getting Started with Testing

Test-to-Pass vs. Test-to-Fail

Initial testing makes sure the software minimally works with typical inputs (“smoke test”)

But most testing is trying to find bugs, which means testing with valid inputs designed to trigger “corner cases” and with invalid inputs

Test Granularity

Unit - there should be at least three tests, per parameter, for every non-trivial method

System - there should be at least three tests for every entry point

Why three?

Refresher: How to Choose Test Inputs

Typical valid inputs

Enter age: _____25_____

Atypical valid inputs

Enter age: _____114_____

Invalid inputs

Enter age: _____3.14159_____

Equivalence classes, boundary
conditions and fuzzing covered later in
semester

Refresher: Where do Invalid Inputs come from?

Users

Network

Devices

Databases

Files

...



Example: Choosing Test Inputs



Consider a simple calendar program where the user can enter an arbitrary date and the program returns the corresponding day of the week. The user is prompted to enter the month, the day, and the year in separate fields.

The calendar represents months as integers 1 to 12, days as integers 1 to 31, and years as integers 1 to 9999 (AD).

How should we choose test cases for this program?

Choosing Test Inputs



Is there anything special about the days 29, 30 or 31 that we should test?

Should we test the month, day, or year 0?

What about testing with negative integers?

Should we test with month 13? What about day 32?

Should we test with non-numeric input?

- Note more than one atypical valid input and more than one kind of invalid input

Test to Pass vs. Test to Fail

What would be some good test inputs for “test to pass”? Initial smoke test

What would be some good test inputs for “test to fail”? Most testing is intended to reveal bugs

- How do we know whether the test outputs are correct?



Test Oracles

The entity that knows the correct outputs, and/or can check that the actual outputs are the same as or consistent with the expected outputs, is called the “test oracle”



Imagine the human tester is the test oracle, and knows what the correct output should be for each input. That works for some programs and some inputs...

But... do you know the day of the week for January 1, 0001? How about October 19, 3333?

What would you do if you did not have Google or other web search engines to ask? More on test oracles in a later lecture

More Calendar Test Cases



Now consider another calendar program that represents months as strings, “January”, “February”, and so on.

Focusing only on the months, what would be some good test inputs for “test to pass”?

What would be some good test inputs for “test to fail”?

More Calendar Test Cases



Say the same calendar program that represents months as strings, “January”, “February”, and so on, also allows *unambiguous* abbreviations, e.g., “F”, “Feb”, but gives an error message for *ambiguous* abbreviations, e.g., “J”, “Ju”

Again focusing on the months, what would change about testing this program?

For the initial calendar with integer months, 0 and 13 are invalid inputs. What are some examples of *invalid* inputs for the calendar that represents months as strings?

Invalid Input Formats

String inputs with valid vs. invalid *formatting*

- Printing vs. non-printing characters
- Upper vs. lower case
- Space, tab, newline characters (whitespace)
- Any characters with special meaning to the application (or to the implementation, e.g., programming language, SQL)
- Null/empty string
- Any string above maximum buffer size (the buffer is the data storage for reading the string input)

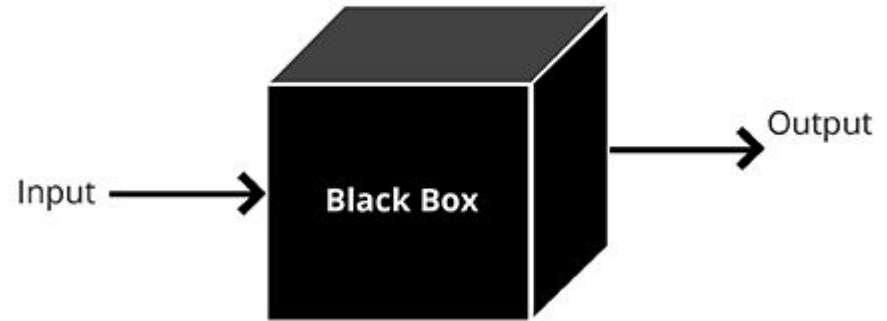
➤ The user will not know these rules unless the software tells them

System Testing

What we have been doing is
blackbox system testing

For system testing, the box is the full
program - we choose external inputs
and check the externally visible
outputs

BLACK BOX TESTING APPROACH



Consider what the box is supposed to
do, not how it does it. Don't look
inside the box.

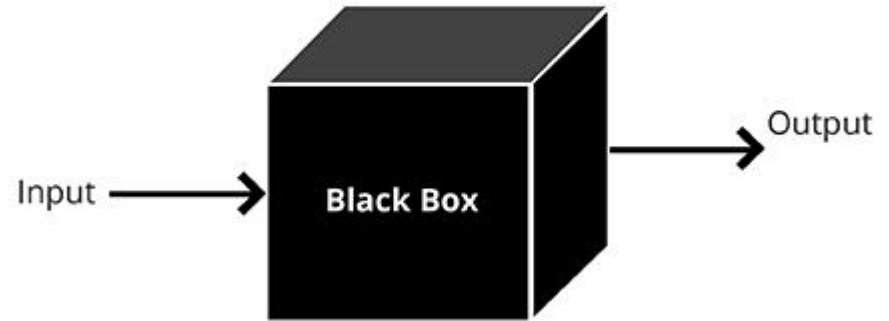
Component Testing

The box might be a component in a larger system, used as a library or service (or a separate process in a distributed system)

Then choose inputs according to how the box is used within the system

We have been doing blackbox system testing with the calendar program, but essentially the same tests might be appropriate for a calendar component - except the data would probably be provided in a different way, e.g., in network packets

BLACK BOX TESTING APPROACH



Consider what the box is supposed to do, not how it does it. Don't look inside the box.

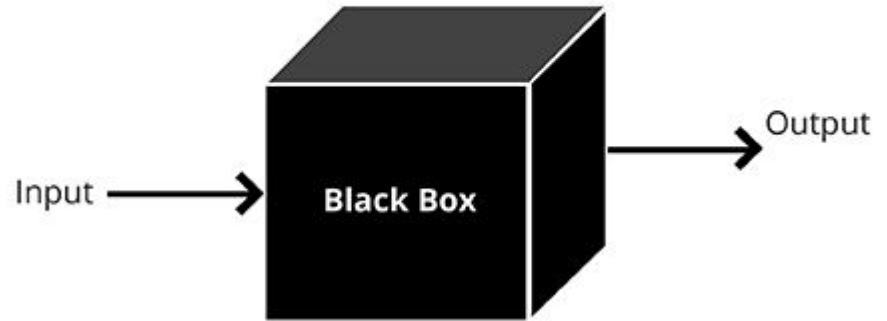
Unit Testing

For unit testing, the box is a method (or function, procedure, subroutine)

Choose inputs to the methods and check the outputs of the method

The method might contain one line of code or might invoke a complicated graph of method calls

BLACK BOX TESTING APPROACH



Consider what the box is supposed to do, not how it does it. Don't look inside the box.

Blackbox Unit Testing Inputs and Outputs

“Inputs” to methods may include parameters, global variables (e.g., static fields in classes), data entered by the user, data received from network/database/files/devices, return values and side-effects from API calls visible inside the code

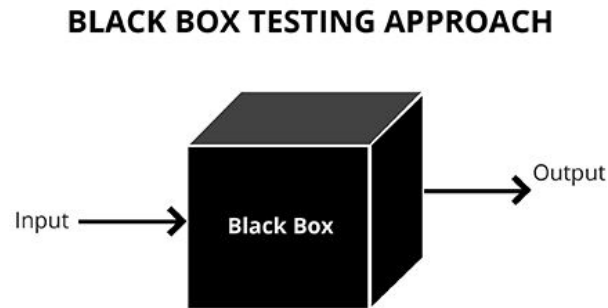
“Outputs” from methods may include return values from the code, global variables changed by the code, data displayed to the user, data sent to network/database/files/devices, parameters to APIs that become visible outside the code

Others?

Blackbox Unit Testing Inputs and Outputs

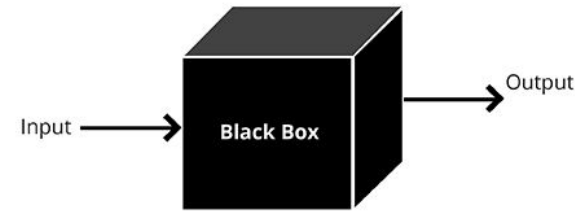
Where do these inputs come from and where do the outputs go during unit testing?

Test runners, test assertions, drivers, stubs, mocks, test doubles discussed in later lectures

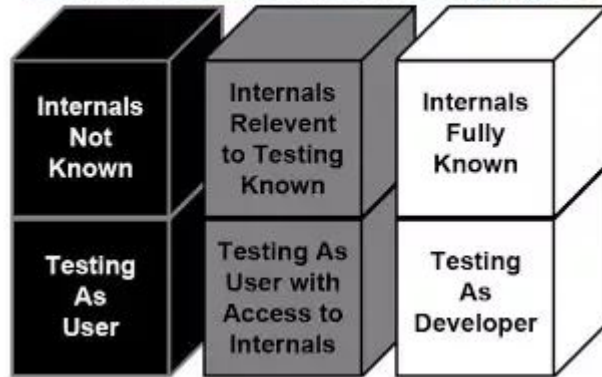


Refresher: Approaches to Testing

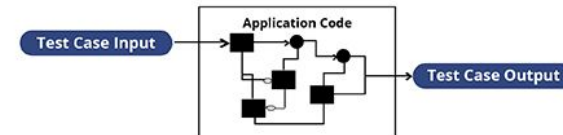
BLACK BOX TESTING APPROACH



Differences Between Box Testing Types



WHITE BOX TESTING APPROACH



Refresher: Blackbox matches Testing Intuition, so...

Why greybox?

Some bugs are not immediately visible externally, e.g., in-memory side-effects, resource leaks, corrupted data

Check logs, databases, file system, network traffic

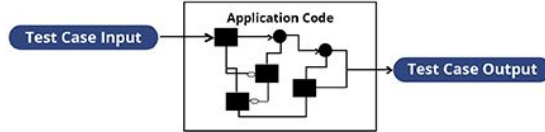
Why whitebox?

If you never ran this part of the code, how can you have any confidence that it works?

Helps with choosing inputs intended to reach specific parts of the code

Whitebox Unit Testing

WHITE BOX TESTING APPROACH



Leverage full knowledge of what is inside the box

Whitebox testing is necessary to achieve coverage - track what code has and has not already been exercised by previous tests, then choose test inputs to force execution of previously unexercised code

If the test suite never executed a method, a statement or a branch, we should not have any confidence in what that code will do when it is eventually executed during operation

- If it will never be executed during operation, why is it there?

Refresher: Coverage

At minimum, unit testing and system testing collectively exercise every statement - much easier to force with unit than system testing

Better: Exercise every branch

Stricter coverage discussed later in semester

Coverage may check missing conditional cases but can never detect entirely missing code

Beware the missing else!

```
if (condition) { do something }  
else { do something else }  
// some other code is here  
// some other code should be here but isn't
```

```
if (condition) { do something }  
// some other code is here  
// some other code should be here but isn't
```


Greybox Testing









During component and system testing, check any error status, exceptions, logs, network and device I/O, etc. produced by one box that are visible to other boxes or to human users (or administrators) and external systems

Consider leftover side-effects, e.g., temporary files not deleted, files or database connections left open

Integration Testing is Special Case of Greybox Testing

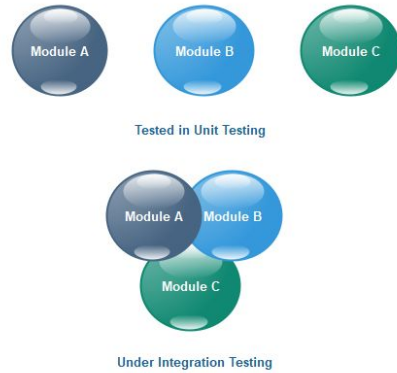
The goal of integration testing is to find errors at the interfaces **between** units and assemblies consisting of multiple units

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

	Unit Testing	Integration Testing
		
		



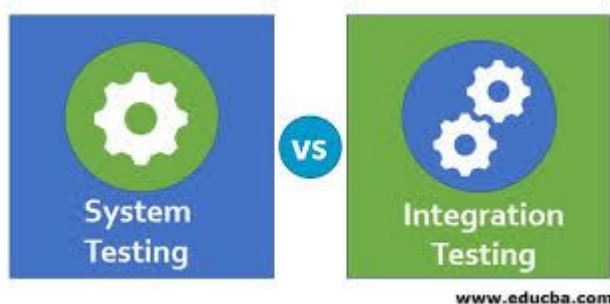
Choosing Integration Testing Inputs



Choose inputs aimed to exercise the communication paths or dependencies among boxes (more on this in a later lecture)

Implicitly combines blackbox and whitebox, since we need to know what inputs to one box will lead to exercising the other box(es)

It's greybox because we look at intermediate inputs and outputs

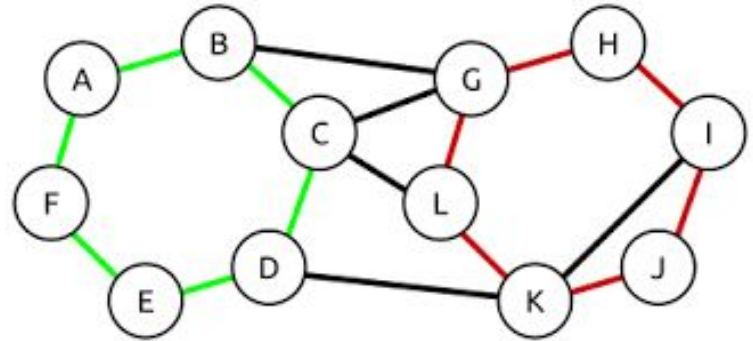


Class Testing

Class (module) testing is a special case of integration testing that is often conducted together with unit testing or considered part of unit testing

The communication paths and dependencies of interest are only between units within the class (module)

Complicated when paths and dependencies between units in the same class *go through other code not in the class*, need stubs/mocks specialized to this case



Integration Bugs

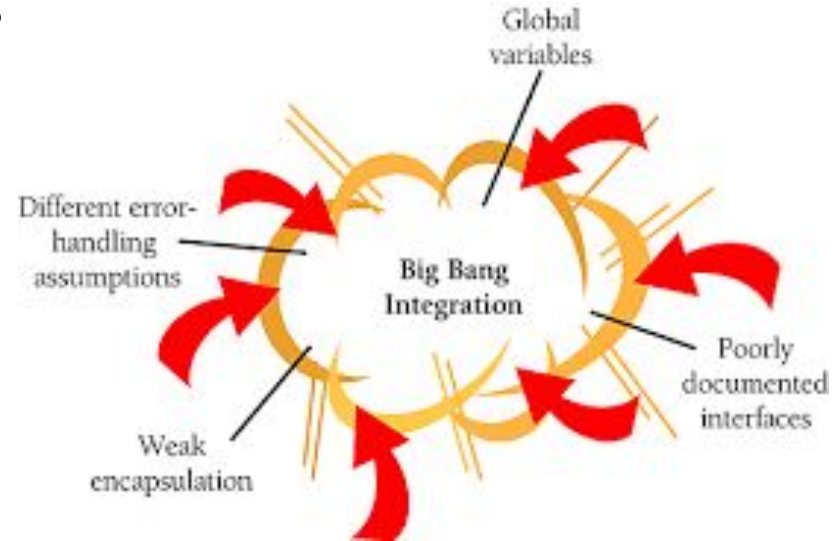
A common cause of integration bugs is inconsistencies between the viewpoints of different modules or units

The parts work fine in isolation but do not work together



Integration Testing Approaches

“Big bang” approach to integration testing: Unit test all independently developed units in isolation (usually classes or modules, not literally testing only individual methods), then test the system. All done!



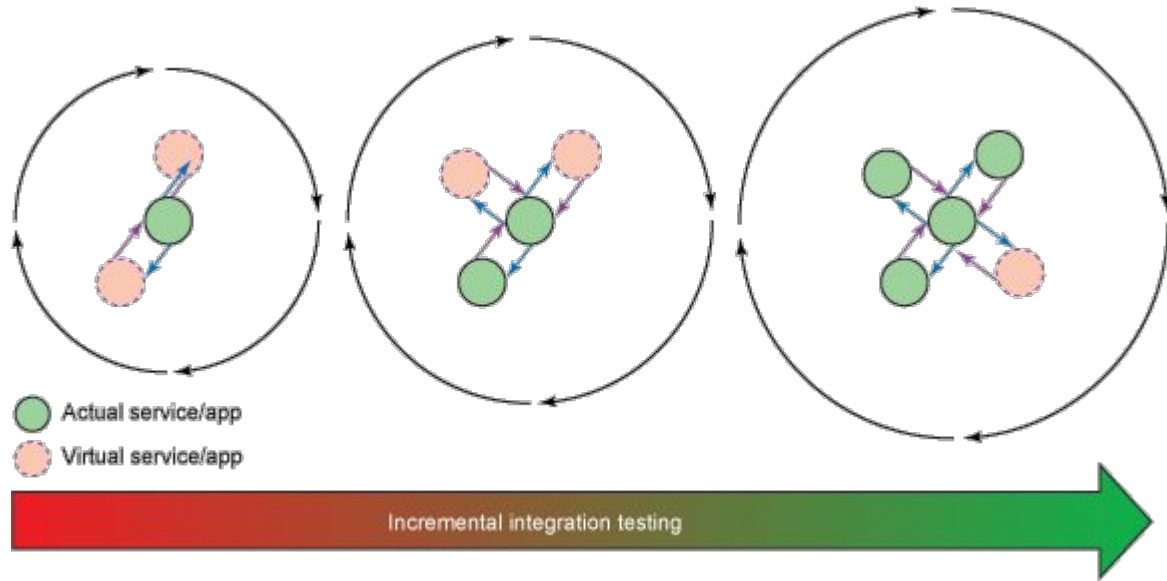
Not so fast: Bugs detected during “big bang” could be anywhere!

Imagine Big Bang Testing for Automobiles



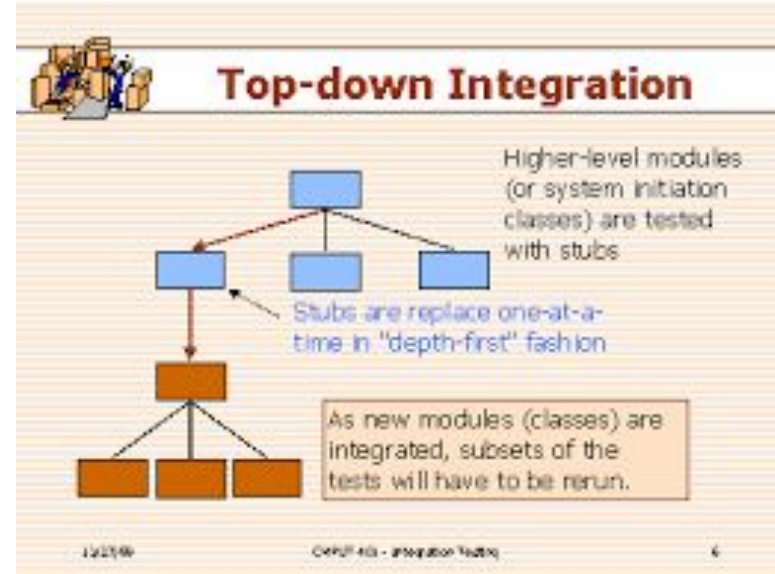
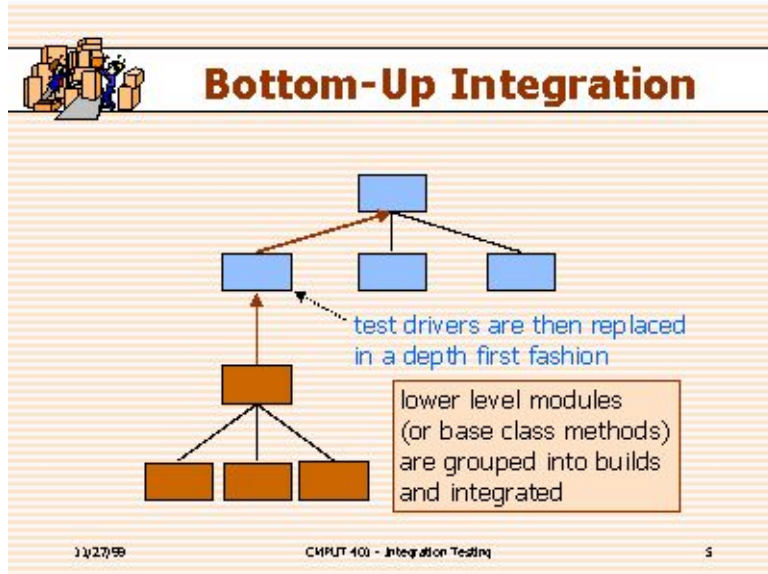
Incremental Integration Helps Localize Bugs

Pick two units that have both already been unit-tested, one unit and an assembly of multiple units (that have already been unit-tested individually and then integrated), or two assemblies



Since we integrate only a small number of units and assemblies 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

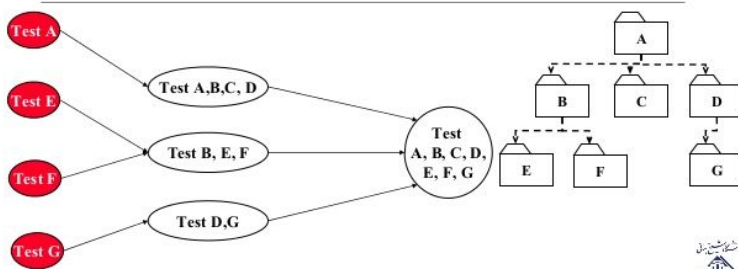


Bottom up starts integration process from lowest layer, top down starts integration process from top

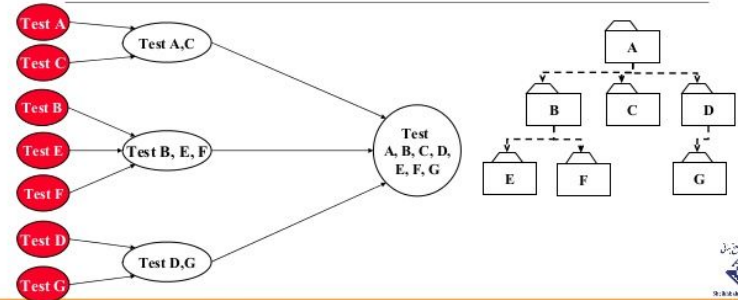
Hybrid (“Sandwich”) Integration Testing

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

Sandwich Testing Strategy



Modified Sandwich Testing



Simple Testing Plan

Blackbox only, no integration testing

1. Test all individual units (e.g., methods), with at least one typical valid input, one atypical valid input, one invalid input
2. Big bang system testing, with at least one typical valid input, one atypical valid input, one invalid input for each entry point
3. Hope for the best

More sophisticated testing plans coming soon!

Team Project

You need to contact your IA mentor and schedule a team meeting

[Assignment T2: Revised Project Proposal](#) due October 25

You must meet with your IA mentor *before* submitting the revised proposal

You can and should start coding as soon as possible after the meeting