CSE 403 Software Engineering

More Testing

Back to our four categories of testing

1. Unit Testing
   - Does each module do what it is supposed to do in isolation?
2. **Integration Testing**
   - **Do you get the expected results when the parts are put together?**
3. Validation Testing
   - Does the program satisfy the requirements?
4. System Testing
   - Does the program work as a whole and within the overall environment? (includes full integration, performance, scale, etc.)

Start with plain, “integration”

Integration: combining 2 or more software units and getting the expected results

Why do we care about integration?

- New problems will inevitably surface
    - Many modules are now together that have never been together before
- If done poorly, all problems will present themselves at once
    - This can be hard to diagnose, debug, fix
- There can be a cascade of interdependencies
    - Cannot find and solve problems one-at-a-time

What do you think of phased integration

Phased ("big-bang") integration:

- Design, code, test, debug each class/unit/subsystem separately
- Combine them all
- Hope for the best

This Photo by Unknown Author is licensed under CC BY-NC-ND

In contrast to incremental integration

Incremental integration:

- Repeat
  - Design, code, test, debug a new component
  - Integrate this component with another (a larger part of the system)
  - Test the combination

- Can start with a functional "skeleton" system (e.g., zero feature release)
  - And incrementally “flesh it out”

Is it obvious which is more successful?

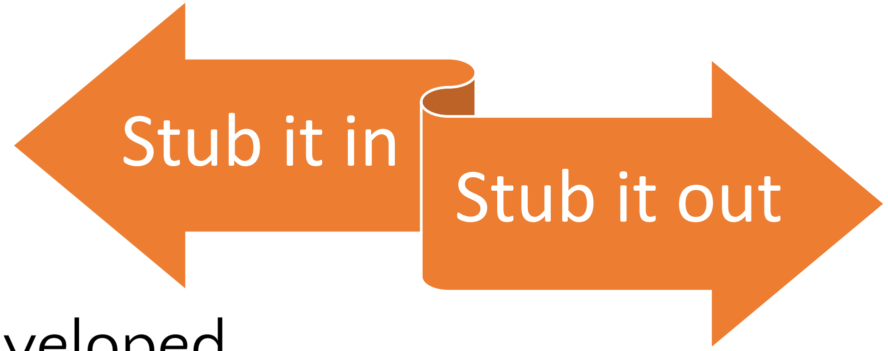
- **Incremental integration** benefits:
  - Errors easier to isolate, find, fix
    - reduces developer bug-fixing load
  - System is always in a (relatively) working state
    - good for customer, developer morale

- But it isn’t without challenges:
  - May need to create "stub" versions of some features that aren’t yet available

What’s a stub?

Stub: a controllable replacement for a software unit

- Useful for simulating difficult-to-control elements, e.g.,
  - network / internet
  - database
  - files



Stub it in

Stub it out

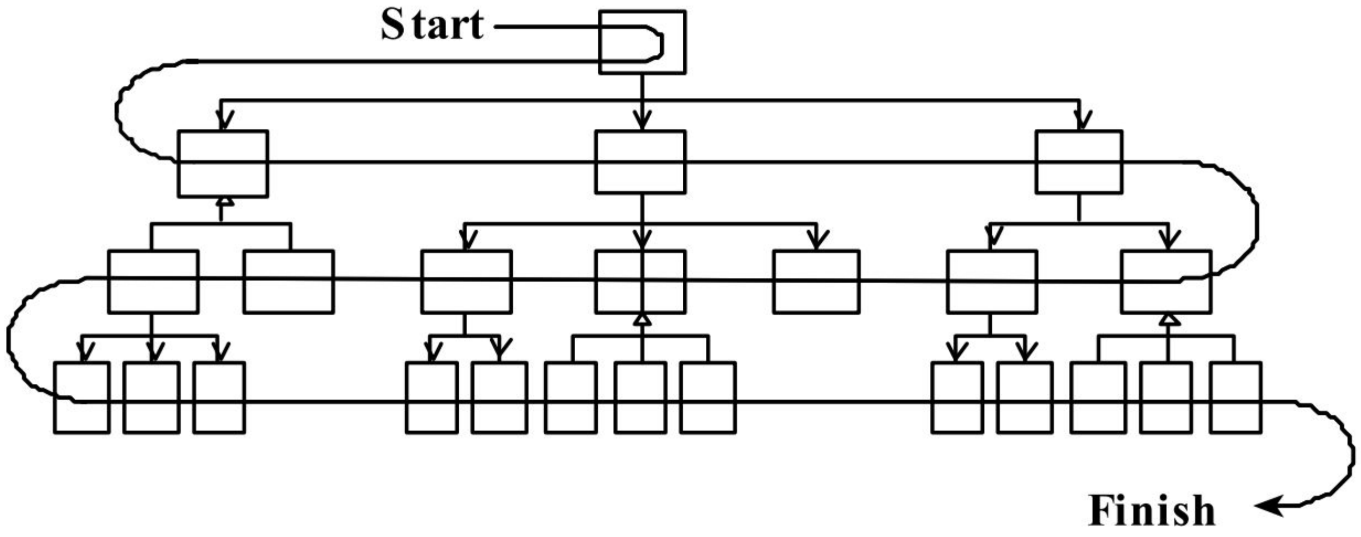
- Useful for simulating components not yet developed

There are different ways to approach integration

Top-down integration:

Start with outer UI layers and work inward

- Must write (lots of) lower level stubs for UI to interact with
- Allows postponing tough design/implementation decisions (
- bad?)



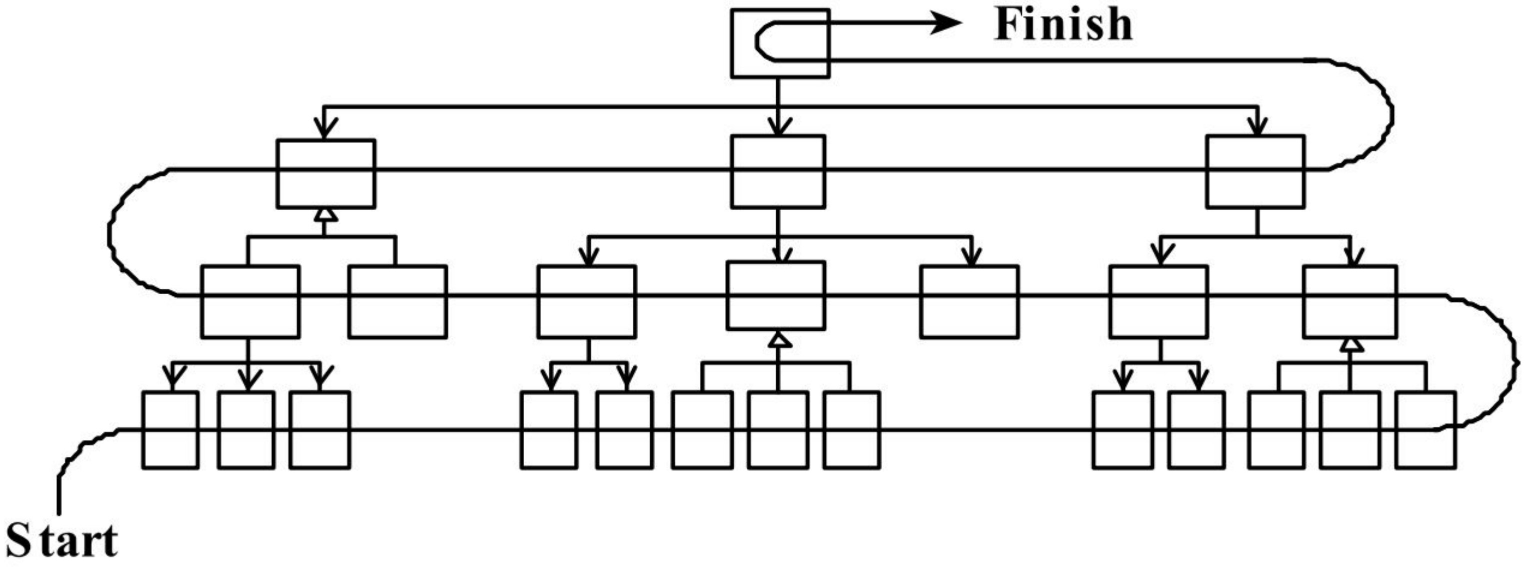
Steve McConnel, Code Complete 2

Or bottom-up

Bottom-up integration:

Start with low-level data/logic layers and work outward

- Must write upper level stubs to drive these layers
- Won't discover high-level / UI design flaws until late



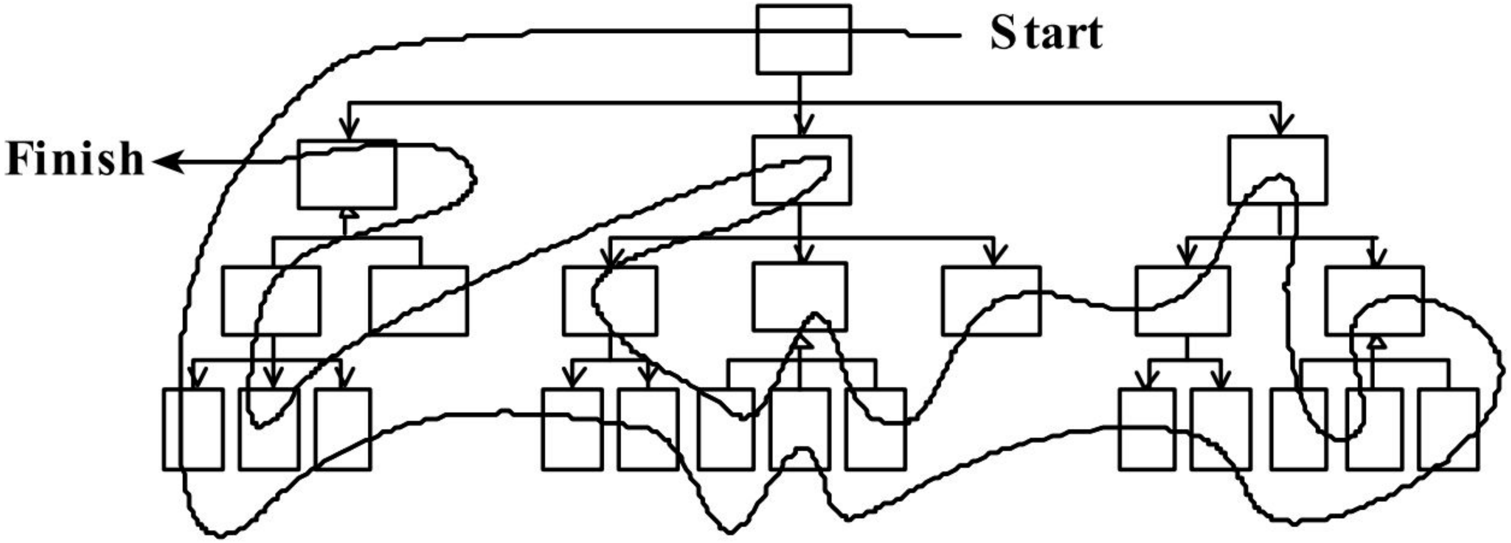
Or “sandwich" integration

“Sandwich" integration by fleshing out a skeleton system:

Connect top-level UI with crucial bottom-level components

- Add middle layers incrementally
- More common and agile approach

Consider starting with a skeleton implementation for your project



Onto integration testing

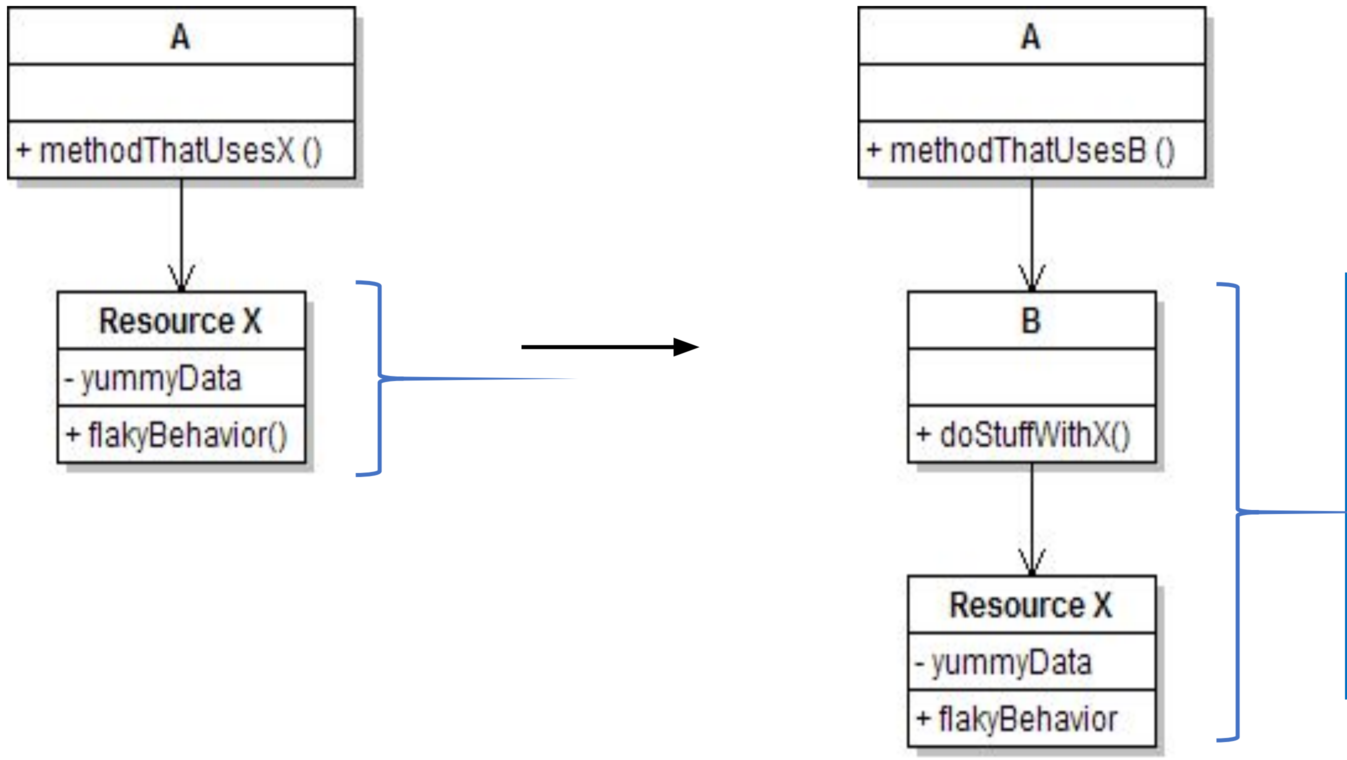
Integration testing: verifying software quality by testing two or more dependent software modules as a group

Can be quite challenging as:

- Combined units can fail in more places and in more complicated ways
- Must use **stubs** to "rig" behavior if not all pieces yet exist OR
    - if you want to simplify problematic components to debug more gradually

How to create a stub, step 1

1. Identify the dependency
   a) This is either a resource or a class/object that is challenging or not yet written
   b) If it isn't an object, wrap it up into one



Goal: Test class A

Create Class B to represent the challenging/missing dependency (as needed)

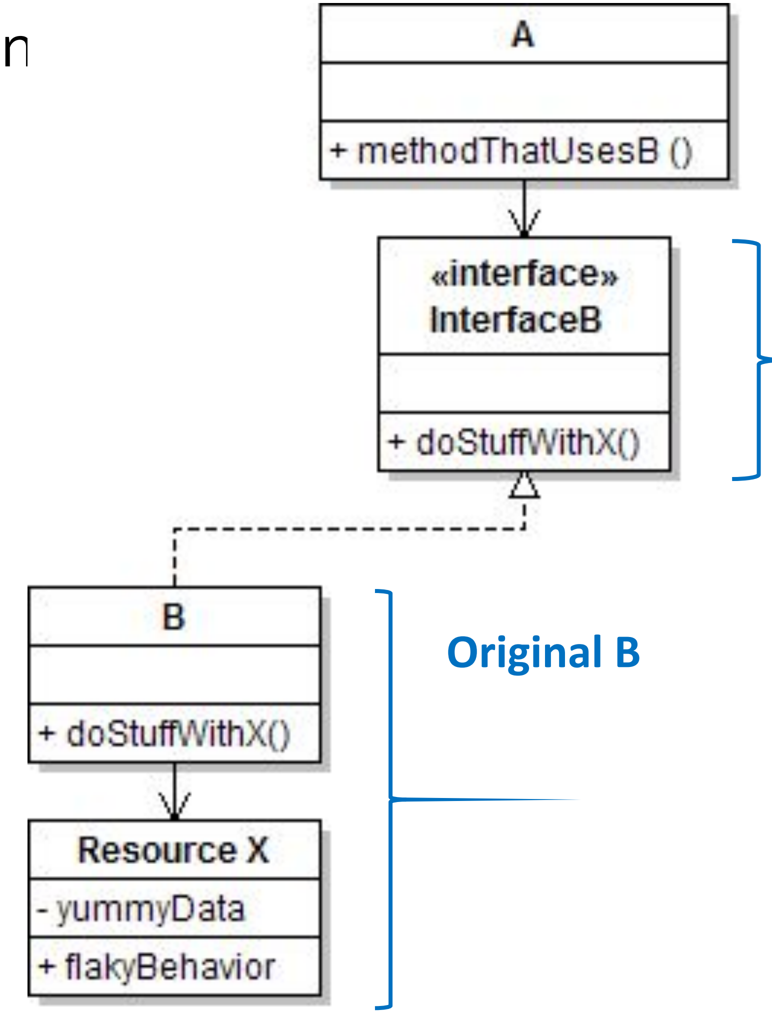
Class A depends on Class B

How to create a stub, step 2

2. Extract the core functionality of the object in an interface

Create a **stub** InterfaceB based on B

Update A's code to work with type InterfaceB, not B



B interface object

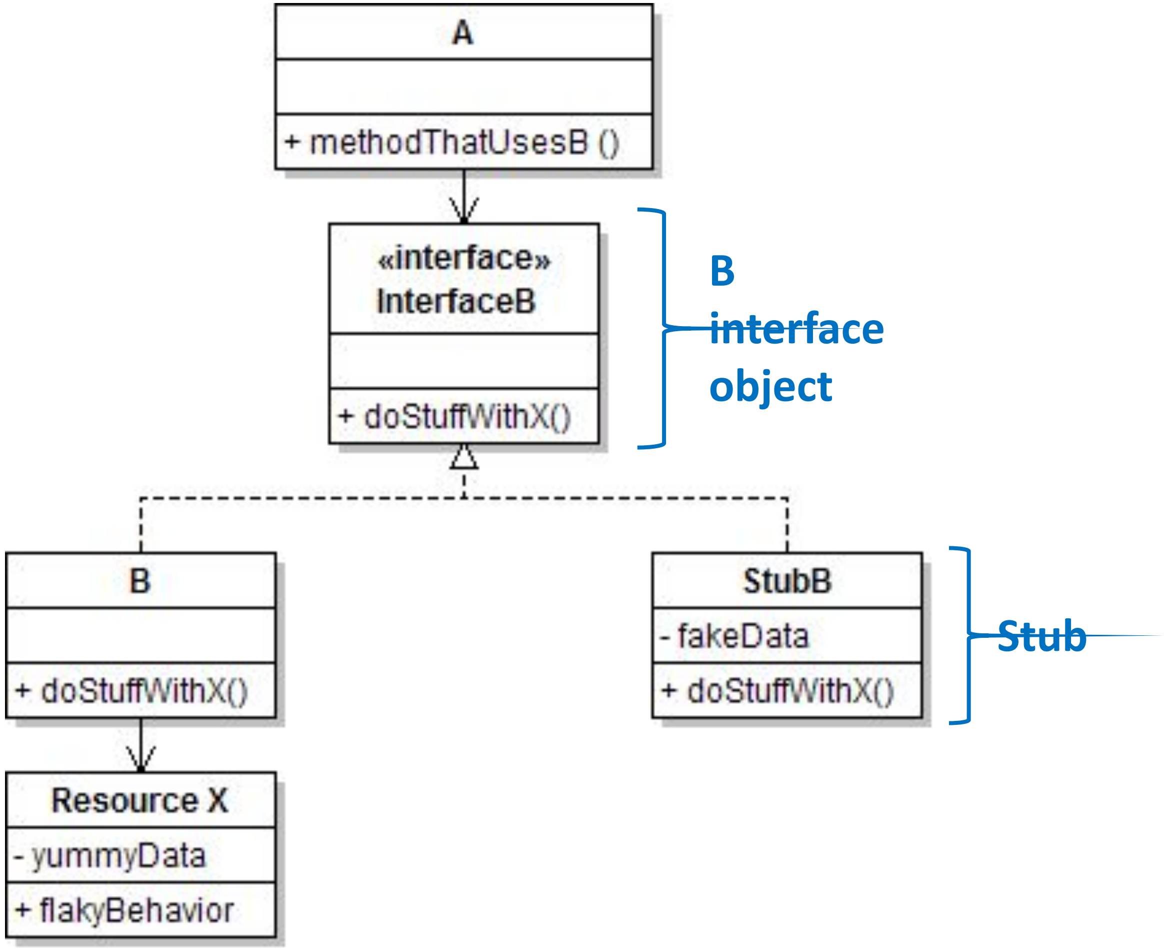
Original B

Create a stub, step 3

3. Write a second "stub" class that also implements the interface,
but returns pre-determined fake data

Now A's dependency on B is dodged and can be tested easily

Can focus on how well A *integrates* with B's expected behavior



Inject the stub, step 4

So cool! Where inject the stub in the code so Class A will reference it?

- At construction
  apple = new A(**new StubB()**);

- Through a getter/setter method
  apple.setResource(**new StubB()**);

- Just before usage, as a parameter
  apple.methodThatUsesB(**new StubB()**);

Think about how to minimize code changes when you no longer depend on the stub

Testing takeaways

- Testing matters!!!
- Test early, test often
  - Bugs become well-hidden beyond the unit in which they occur
- Don't confuse volume with quality of test data
  - Can lose relevant cases in mass of irrelevant ones
  - Look for revealing subdomains (“characteristic tests”)
- Choose test data to cover:
  - Specification (black box testing)
  - Code (white box testing)
- Testing can't generally prove absence of bugs
  - But it can increase quality and confidence

Appendix – Mock objects for integration testing

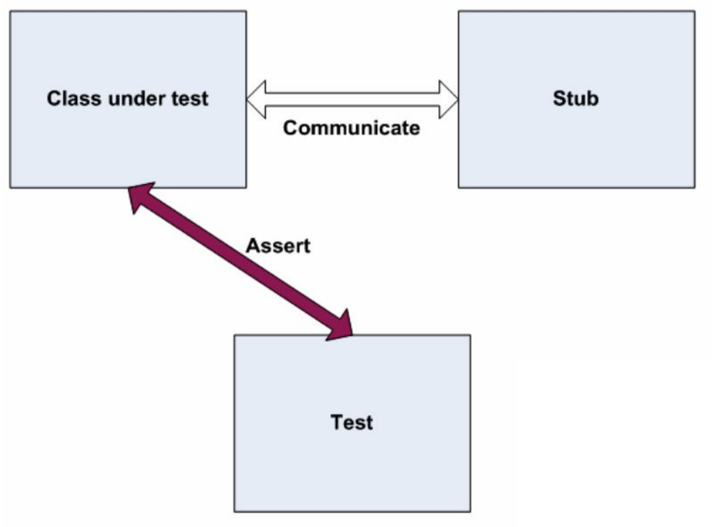
Mock objects
Mock vs stub objects

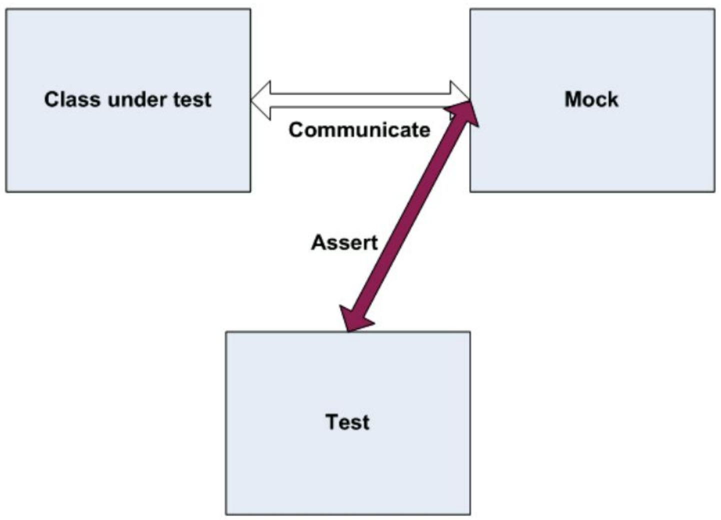
Thanks to Marty Stepp, previous UW CSE 403 instructor, for providing this and an earlier version of the integration testing material

"Mock" objects

mock object: a fake object that decides whether a unit test has passed or failed by watching interactions between objects

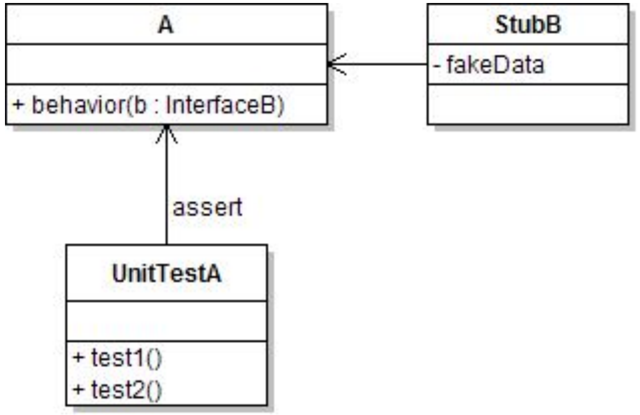
- useful for **interaction testing** (as opposed to **state testing**)



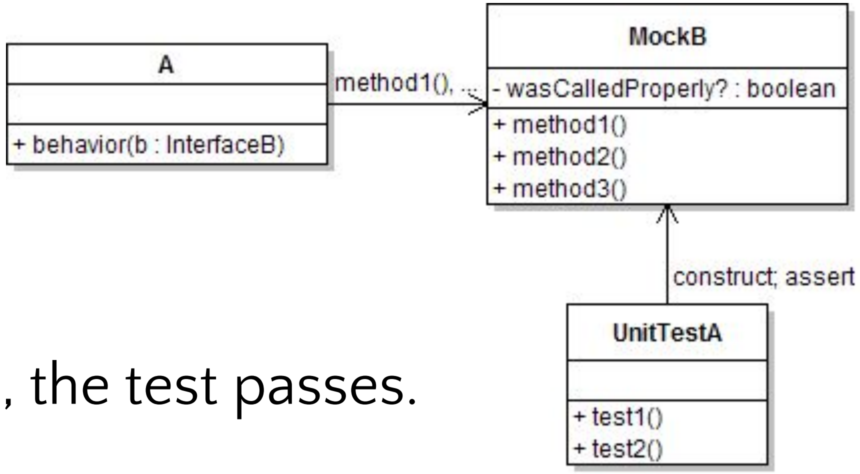


Stubs vs. mocks

- A **stub** gives out data that goes to the object/class under test.
- The unit test directly asserts against class under test, to make sure it gives the right result when fed this data.



- A **mock** waits to be called by the class under test (A).
  - Maybe it has several methods it expects that A should call.



- It makes sure that it was contacted in exactly the right way.
  - If A interacts with B the way it should, the test passes.

Mock object frameworks

- Stubs are often best created by hand/IDE.
  Mocks are tedious to create manually.

- Mock object frameworks help with the process.
  - android-mock, EasyMock, jMock (Java)
  - FlexMock / Mocha (Ruby)
  - SimpleTest / PHPUnit (PHP)
  - ...

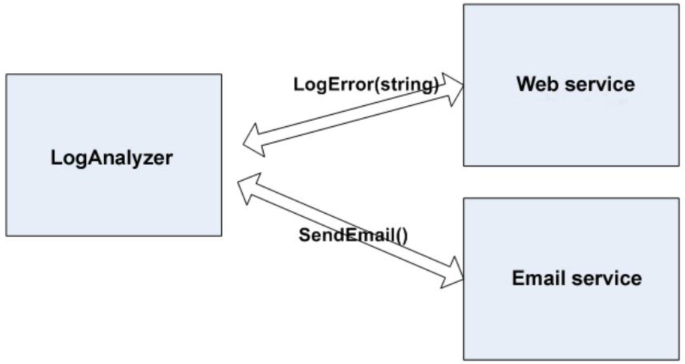
jMock

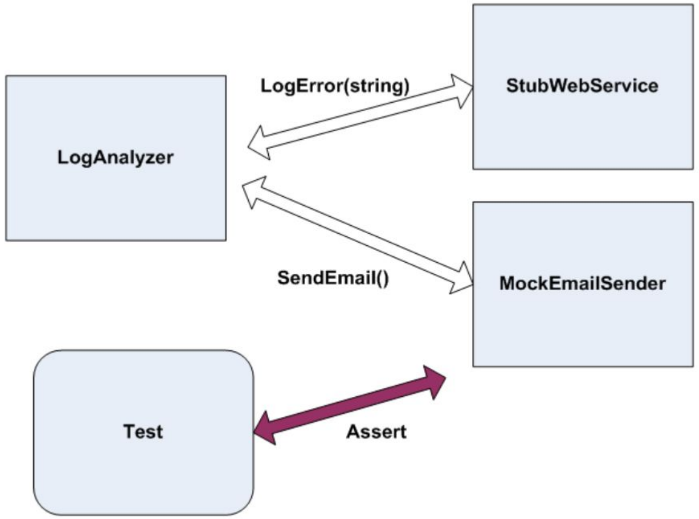
- Frameworks provide the following:
  - auto-generation of mock objects that implement a given interface
  - logging of what calls are performed on the mock objects
  - methods/primitives for declaring and asserting your expectations

Using stubs/mocks together

- Suppose a log analyzer reads from a web service.
  If the web fails to log an error, the analyzer must send email.
  - How to test to ensure that this behavior is occurring?

- Set up a *stub* for the web service that intentionally fails.
- Set up a *mock* for the email service that checks to see whether the an end an em


LogAnalyzer
LogError(string)
Web service
SendEmail()
Email service


LogAnalyzer
LogError(string)
StubWebService
SendEmail()
MockEmailSender
Test
Assert