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 “integration”

Integration: combining 2 or more software units

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

Phased (“big-bang”) integration

- Design, code, test, debug each class/unit/subsystem separately
- Combine them all (“phased” is a misnomer)
- Hope for the best

Incremental integration (CI)

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

- Start with a skeleton system (e.g., zero feature release)
  - Incrementally flesh it out

What are the pros and cons?

Incremental integration: pros and cons

- 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

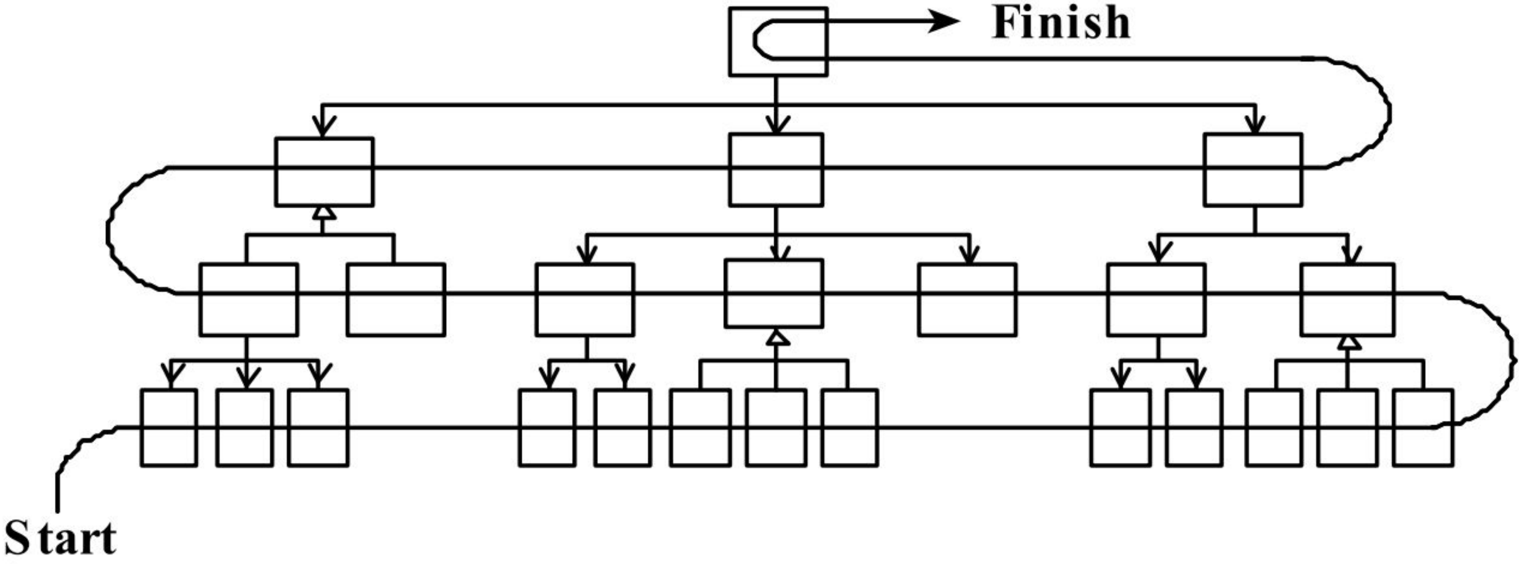
- Incremental integration challenges:
  - Need to create “stub” or “mock” versions of features that aren’t yet available

- Types of incremental integration:
  - Bottom-up, top-down, and sandwich

Bottom-up integration

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

- Must write tests (a.k.a. upper level **stubs**) to drive these layers
- Won't discover high-level / UI design flaws until late



Lab

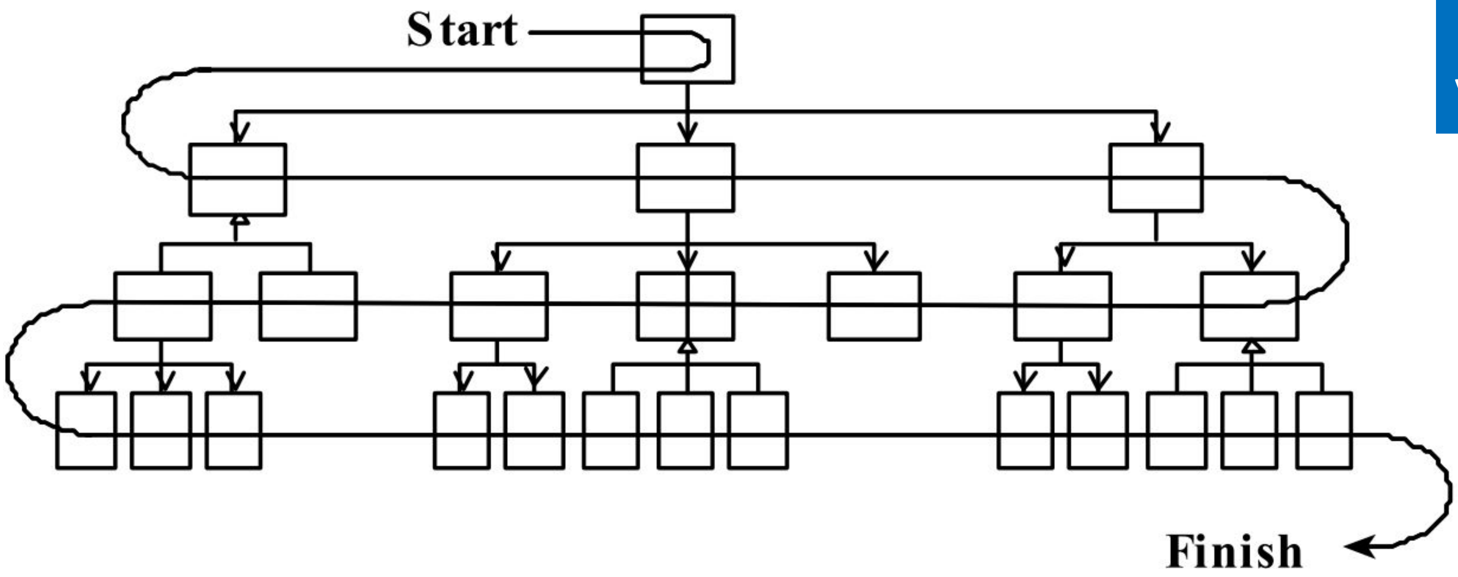
Lab it-01

Top-down integration

Start with outer UI layers and work inward

- Must write (lots of) **mocks** (a.k.a. lower level stubs) for UI to interact with
- Allows postponing tough design/implementation decisions (good or bad?)

This may be necessary for your beta release



Steve McConnel, Code Complete 2

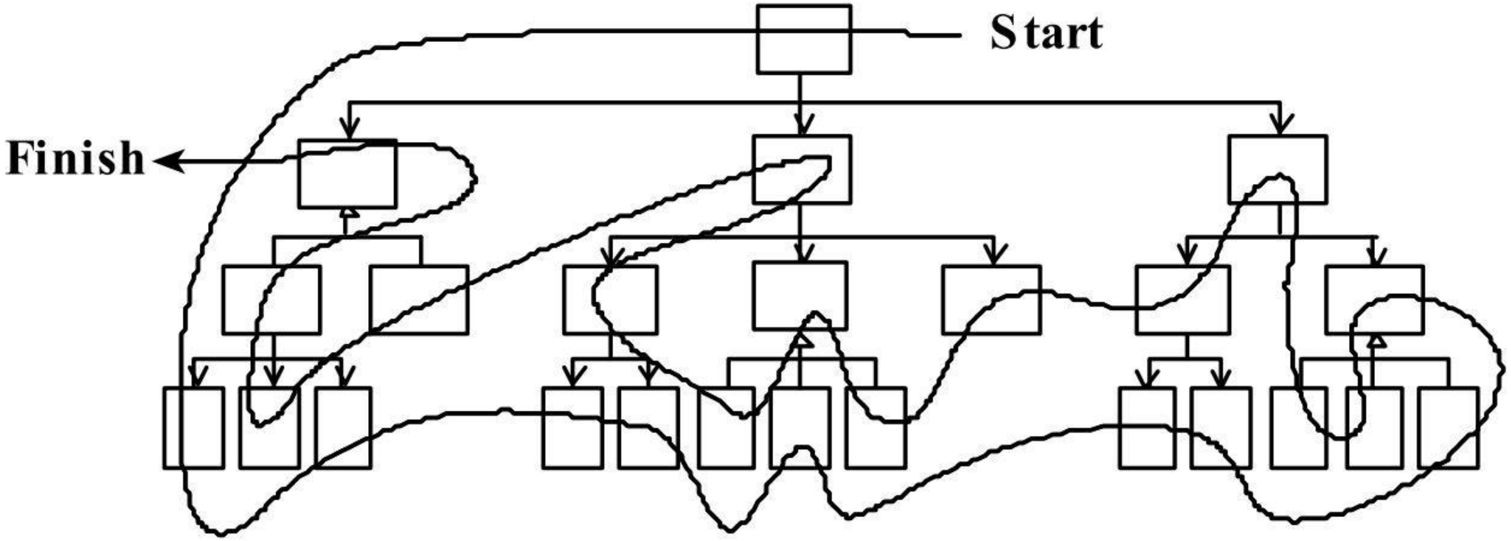
“Sandwich" integration

Sandwich integration fleshes out a skeleton system:

Connect top-level UI with important bottom-level components

- Add middle layers incrementally
- Pragmatic, agile approach
- Need to make decisions in a principled way!

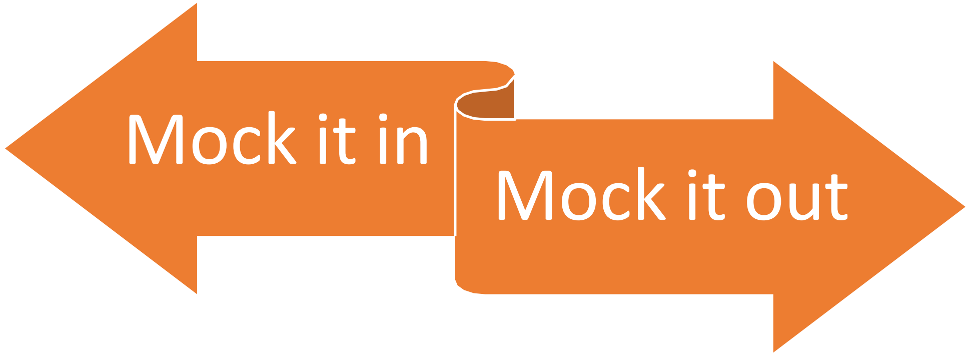
Consider starting your project with a skeleton implementation



What’s a mock?

Mock: a controllable replacement for a software unit

- Simulates components not yet developed
- Simulates difficult-to-control elements
  - network / internet
  - database
  - files
  - physical components
  - expensive components



Mock it in

Mock it out

Integration testing

Integration testing: testing 2+ modules together

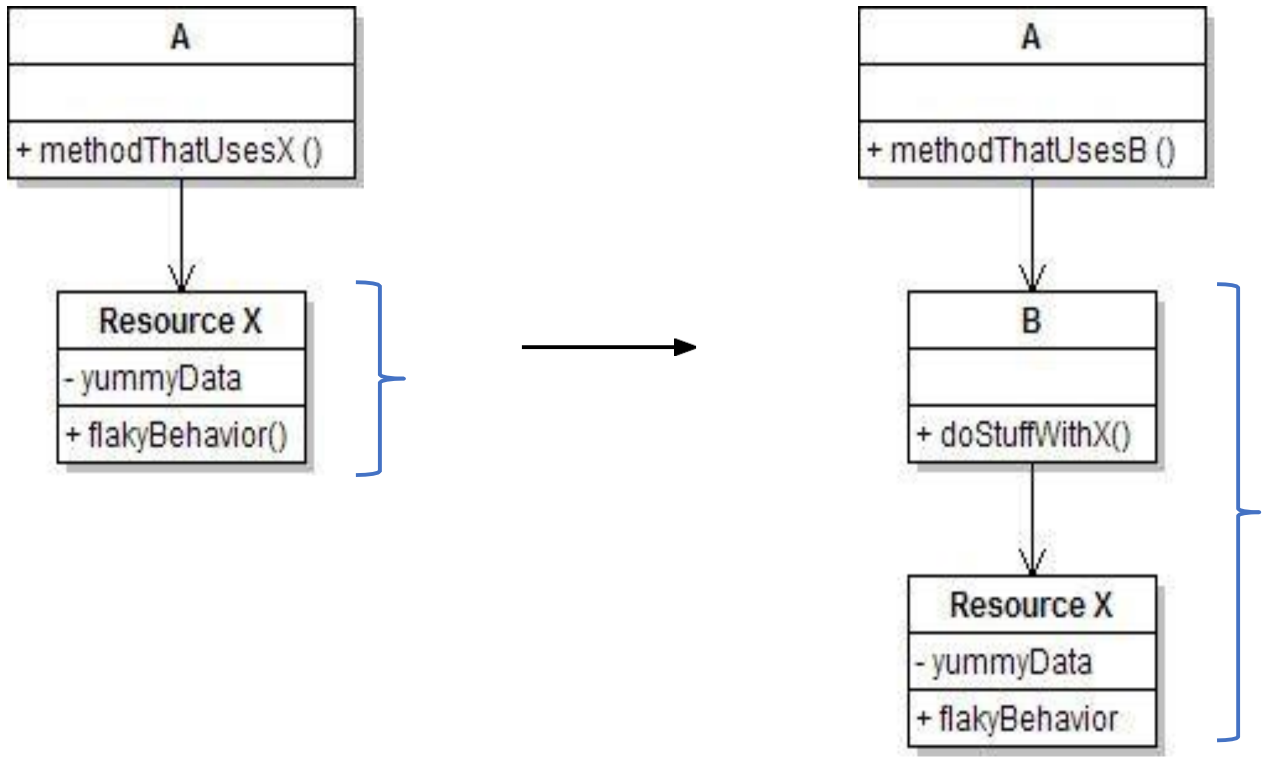
Challenging!

- Combined units can fail in more places and in more complicated ways
- Must use **mocks** or **stubs**
  - to simulate behavior if not all pieces yet exist, OR
  - to reduce scope of debugging

Suppose class A depends on class B, which is not yet written.
To test class A, we need a *mock* implementation of B.

How to create a mock, step 1

1. Identify the dependency
   a) A resource/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)

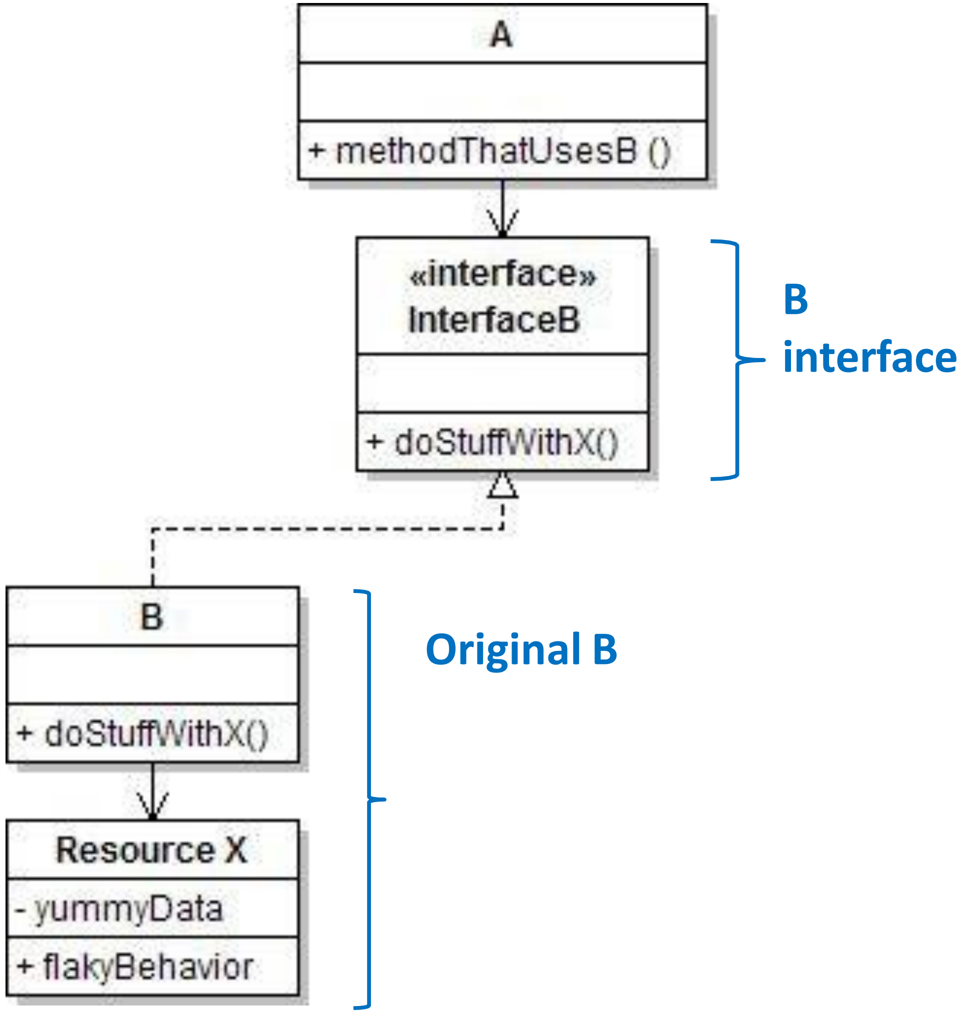
Class A depends on Class B

How to create a mock, step 2

2. Create an interface that expresses the core functionality of the object
    - Class A no longer knows about Class B, only InterfaceB
    - Every B object also has type InterfaceB

Create a **stub** InterfaceB based on B

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



B interface

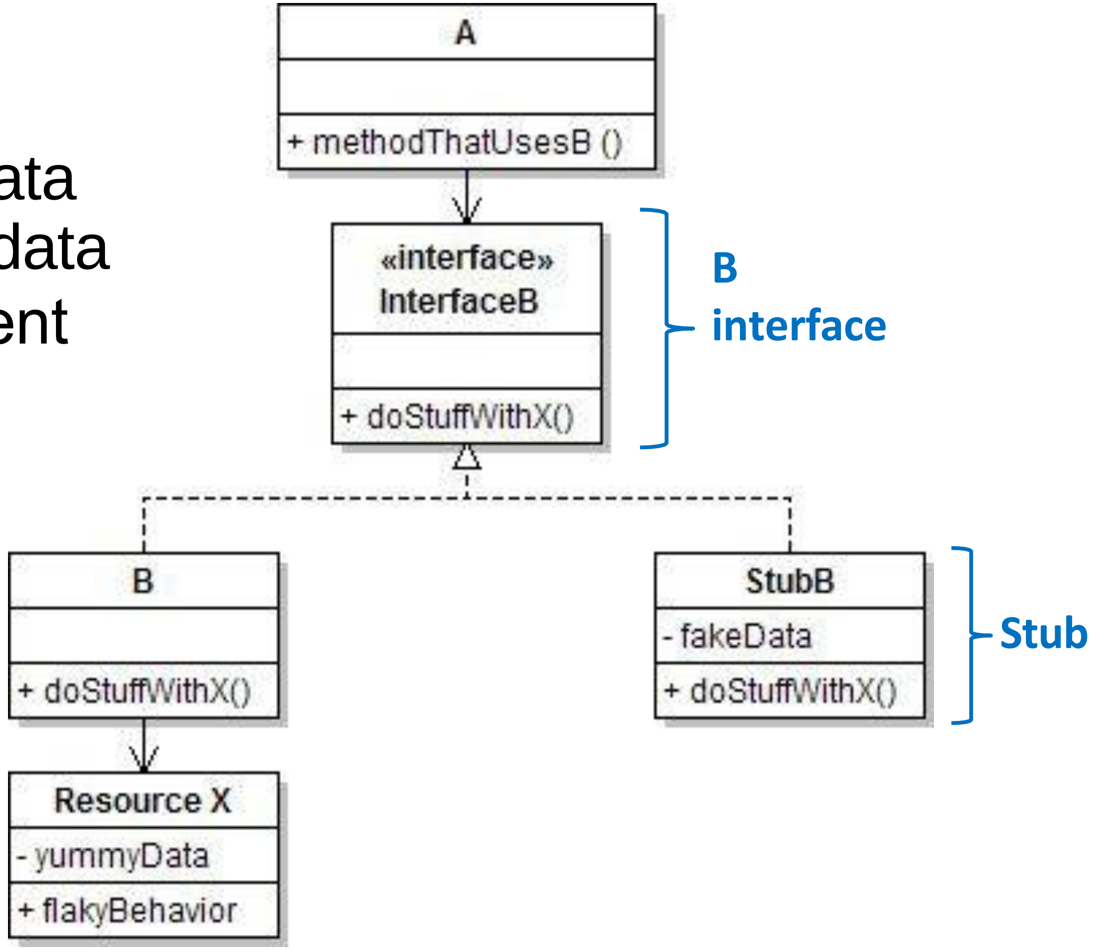
Original B

How to create a mock, step 3

3. Write a second "stub" class that also implements the interface
   - It may return pre-determined fake data
     - Crashes if called on unexpected data
   - It may be a simple-to-verify, inefficient implementation

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



Lab

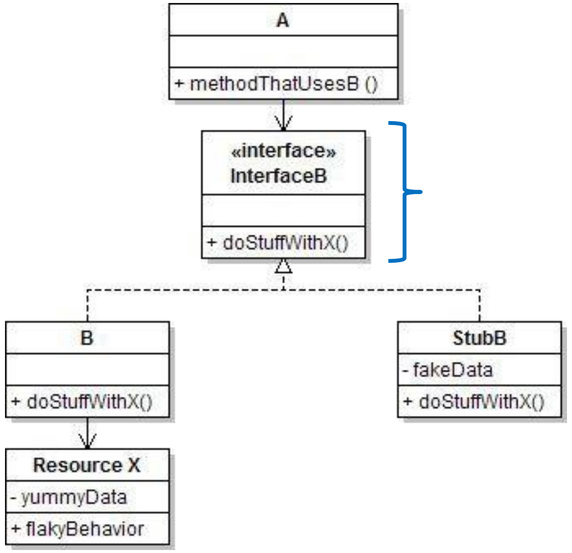
Lab it-02

Use the mock, step 4

Goal: At test time, Class A uses the stub; in the field, Class A uses the real implementation

Good design: minimize code changes between using and not using the stub

What does the client of the mock look like? (Hint: how did A use B?)



Use the mock, step 4

Goal: At test time, Class A uses the stub; in the field, Class A uses the real implementation
Good design: minimize code changes between using and not using the stub

What does the client of the mock look like? (Hint: how did A use B?)

- At construction

  ```
  a = new A( new MockB() );
  ```

- Through a getter/setter method

  ```
  a.setResource( new MockB() );
  ```

- Just before usage, as a parameter

  ```
  a.methodThatUsesB( new MockB() );
  ```

Unacceptable hard-coding

Also known as “**dependency injection**”: the dependency is provided (“injected”), rather than the client creating it

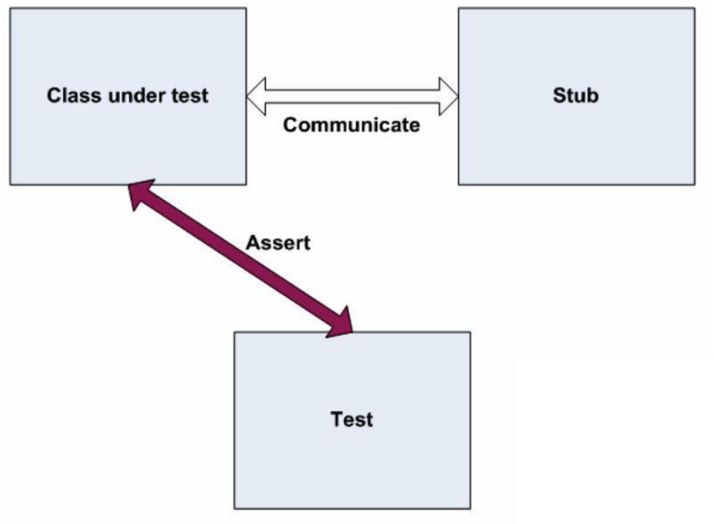
Lab

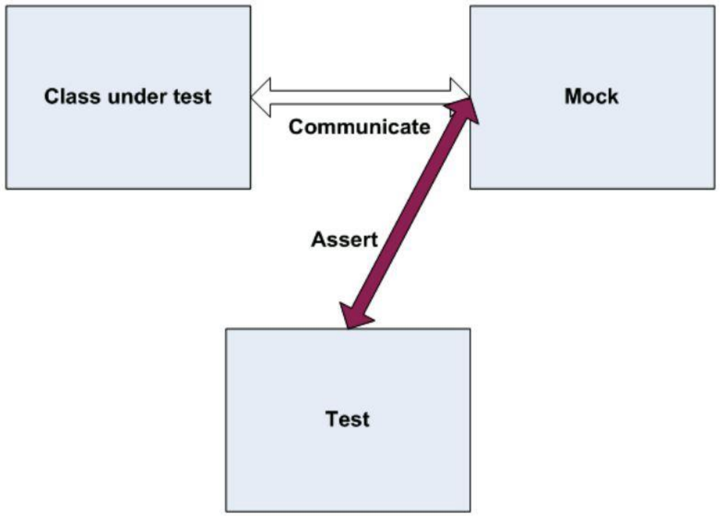
Lab it-03

"Mock" objects

mock object: a fake object that models the behavior of the desired service.

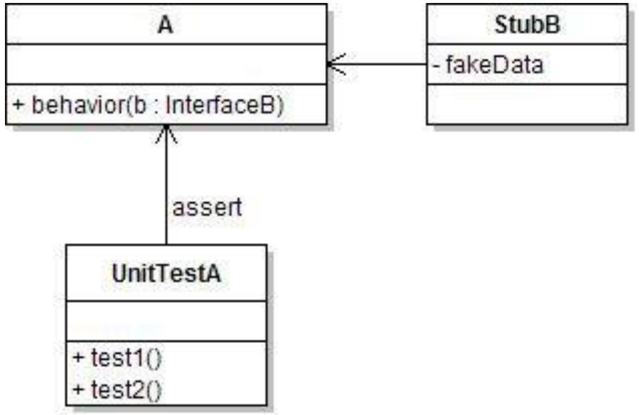
- useful for **interaction testing** (as opposed to **state testing**)



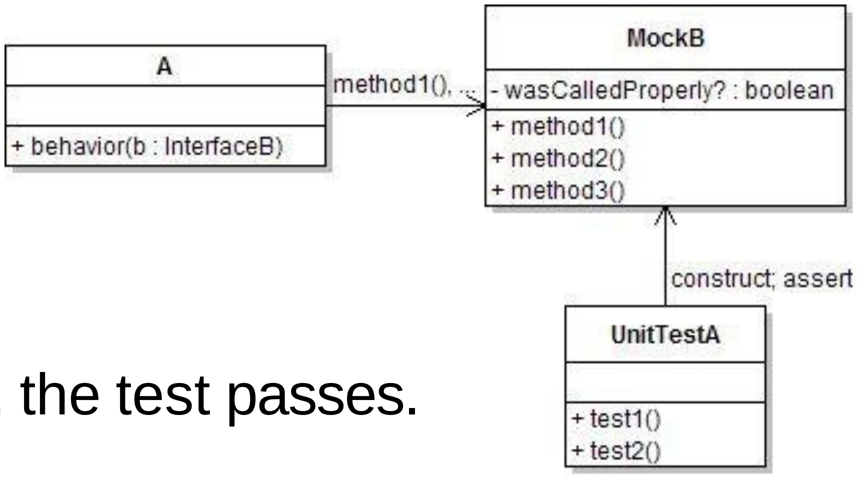


Stubs vs. mocks

- 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.



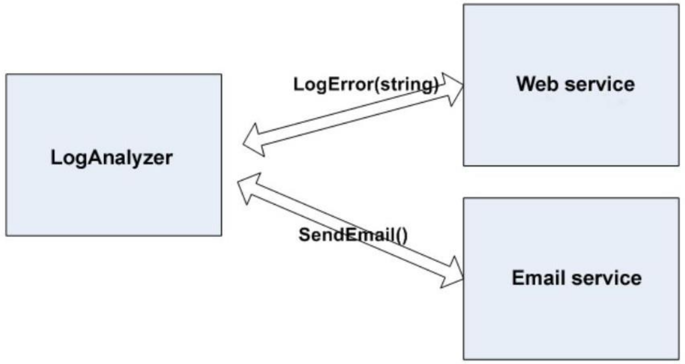
- A **mock** waits to be called by the class under test (A).
  - Maybe it has several methods it expects that A should call.
- It makes sure that it was contacted in exactly the right way.
  - If A interacts with B the way it should, the test passes.

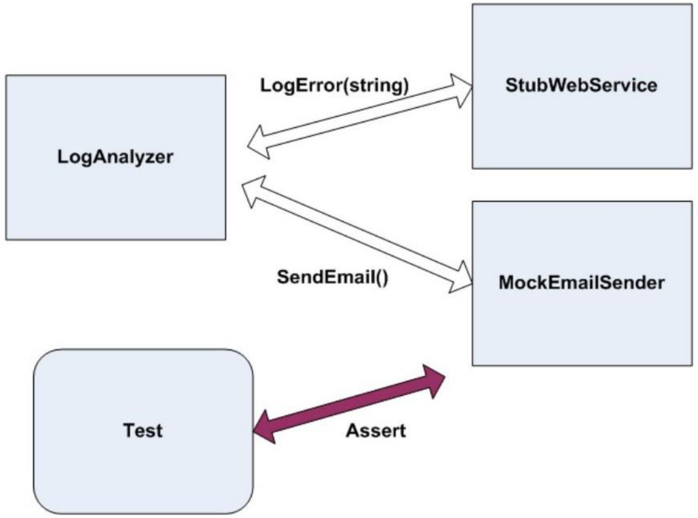


Using stubs/mocks together

- 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 analyzer contacts it to send an email message.





Lab

Lab it-04

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)
  - ...

- 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

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 prove absence of bugs
  - It can increase quality and confidence