Integration Testing, Mocking, Testable Design

SWE 261P

material adapted from Marty Stepp

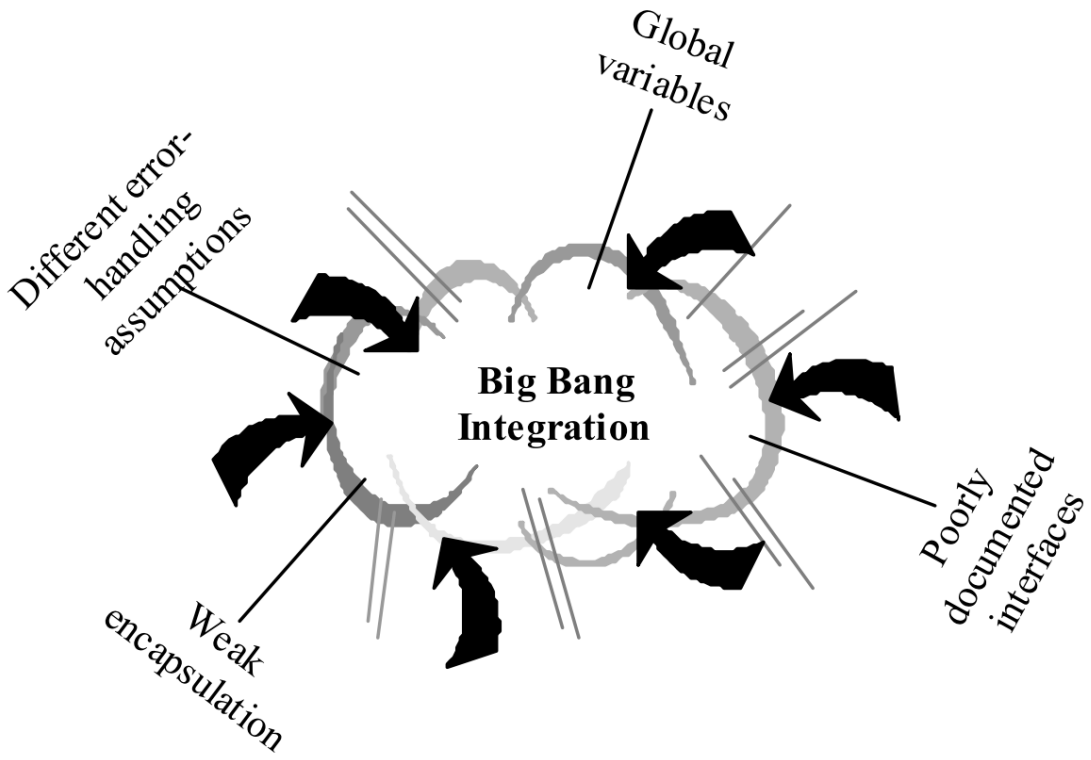
Integration

- **integration**: Combining 2 or more software units
  - often a subset of the overall project (!= system testing)

- Why do software engineers care about integration?
  - new problems will inevitably surface
    - many systems now together that have never been before
  - if done poorly, all problems present themselves at once
    - hard to diagnose, debug, fix
  - cascade of interdependencies
    - cannot find and solve problems one-at-a-time

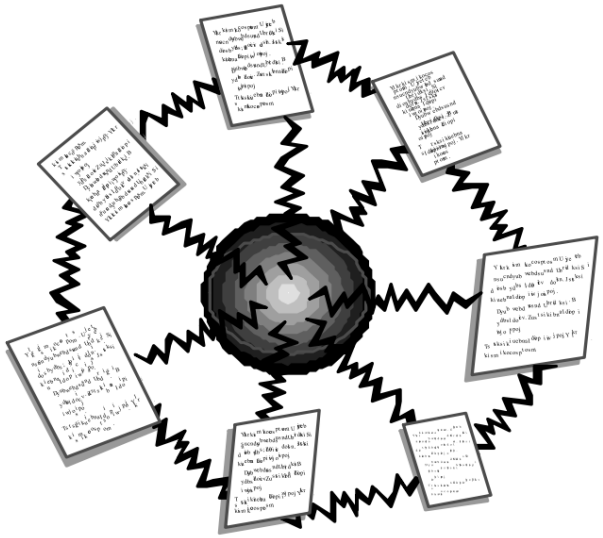
Big-bang integration

- **"big-bang" integration**:
  - design, code, test, debug each class/unit/subsystem separately
  - combine them all
  - pray

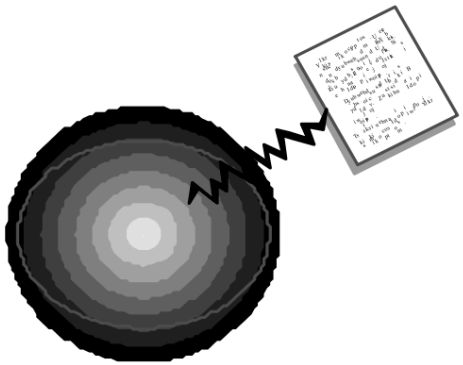


Incremental integration

- **incremental integration**:
  - develop a functional "skeleton" system
  - design, code, test, debug a small new piece
  - integrate this piece with the skeleton
    - test/debug it before adding any other pieces



big bang



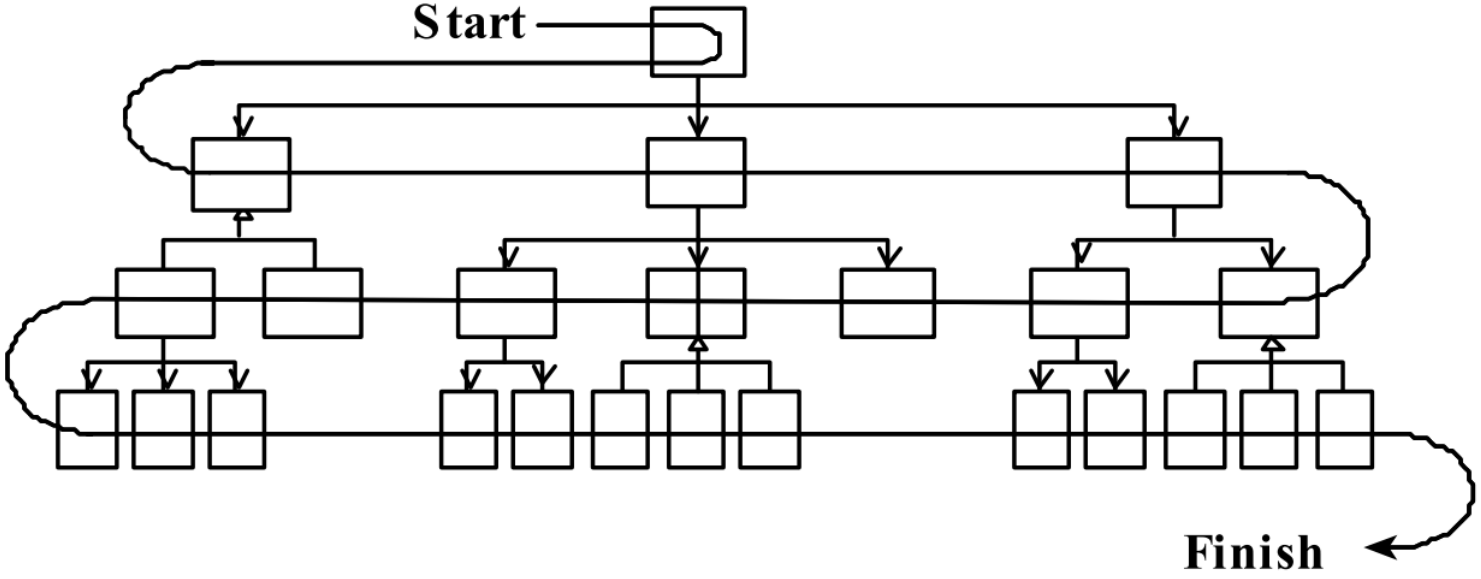
incremental

Benefits of incremental

- Benefits:
  - Errors easier to isolate, find, fix
    - reduces developer bug-fixing load
  - System is always in a (relatively) working state
    - good for customer relations, developer morale
- Drawbacks:
  - May need to create "stub" versions of some features that have not yet been integrated

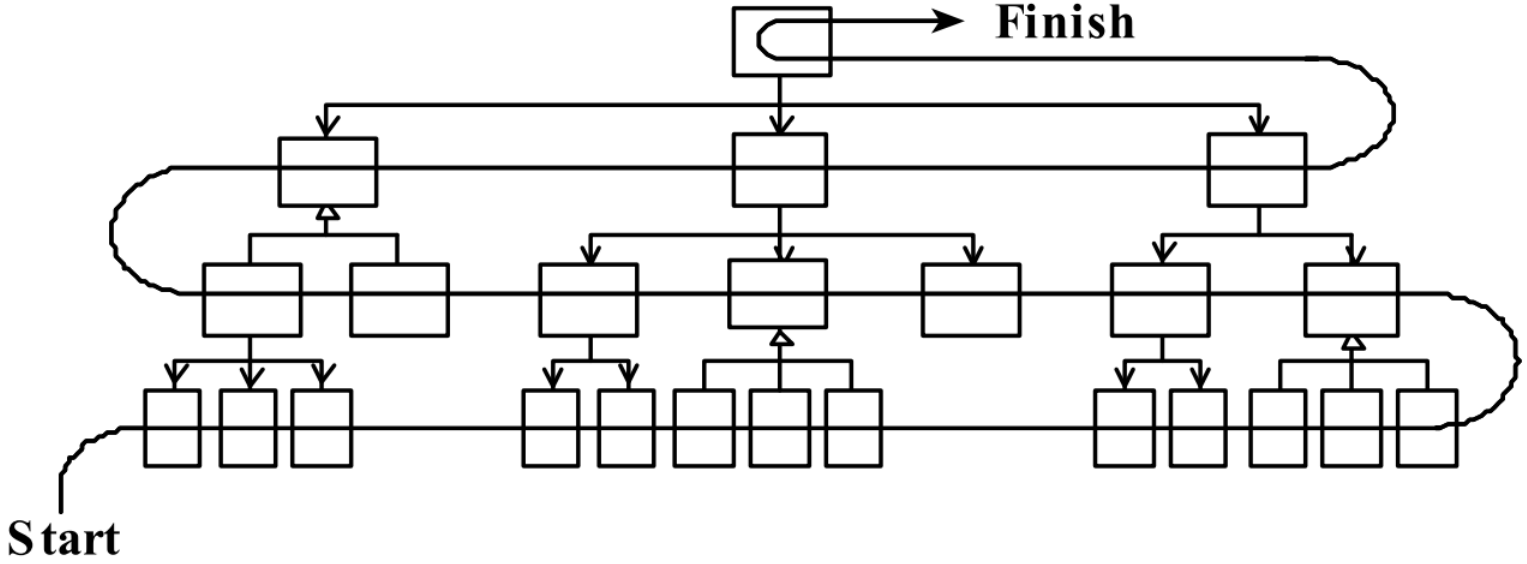
Top-down integration

- **top-down integration**:
  Start with outer UI layers and work inward
  - must write (lots of) stub lower layers for UI to interact with
  - allows postponing tough design/debugging decisions (bad?)



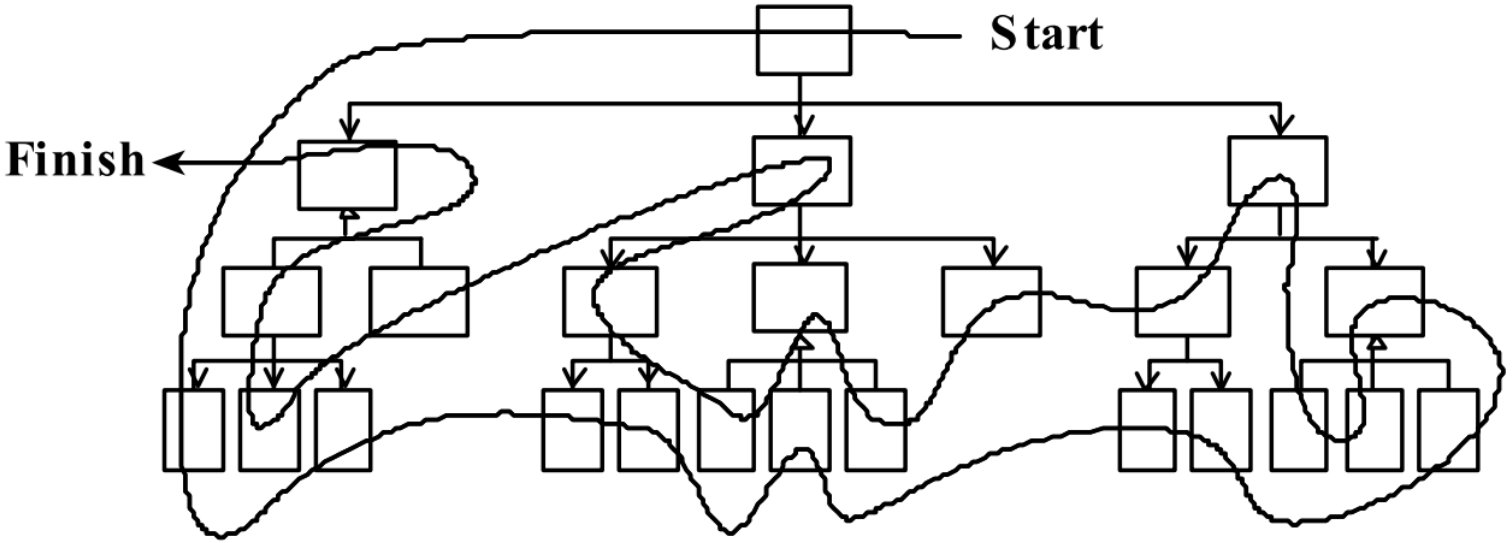
Bottom-up integration

- **bottom-up integration**:
  Start with low-level data/logic layers and work outward
  - must write test drivers to run these layers
  - won't discover high-level / UI design flaws until late



"Sandwich" integration

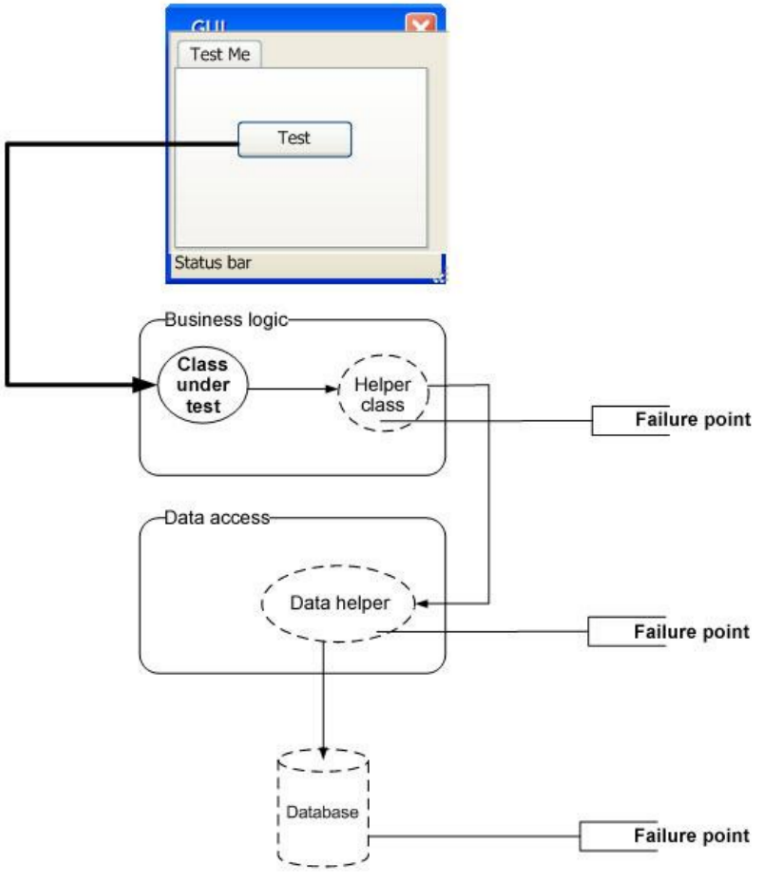
- **"sandwich" integration**:
  Connect top-level UI with crucial bottom-level classes
  - add middle layers later as needed
  - more practical than top-down or bottom-up?



Integration testing

- **integration testing**: Verifying software quality by testing two or more dependent software modules as a group.

- challenges:
  - Combined units can fail in more places and in more complicated ways.
  - How to test a partial system where not all parts exist?
  - How to "rig" the behavior of unit A so as to produce a given behavior from unit B?

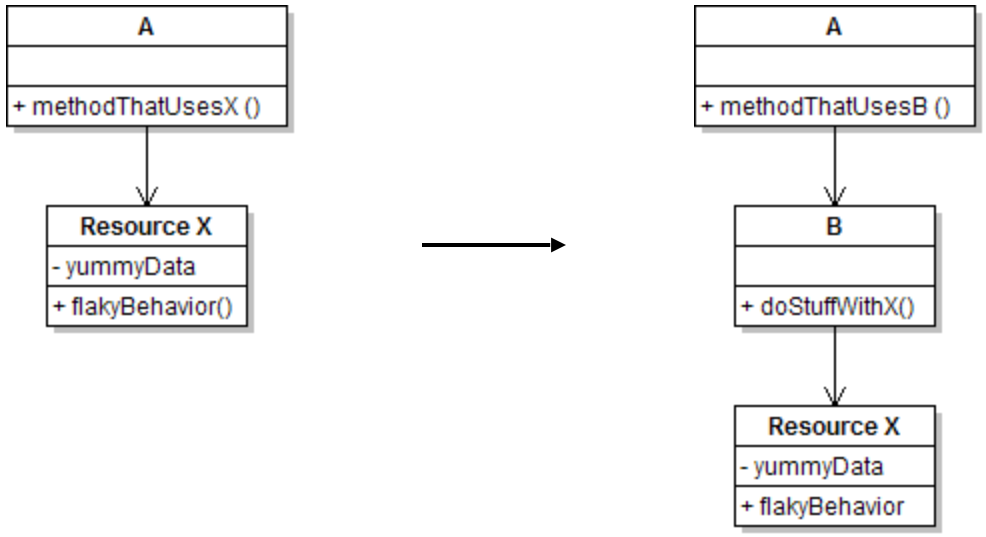


Stubs

- **stub**: A controllable replacement for an existing software unit to which your code under test has a dependency.
  - useful for simulating difficult-to-control elements:
    - network / internet
    - database
    - time/date-sensitive code
    - files
    - threads
    - memory
  - also useful to avoid side effects of real systems

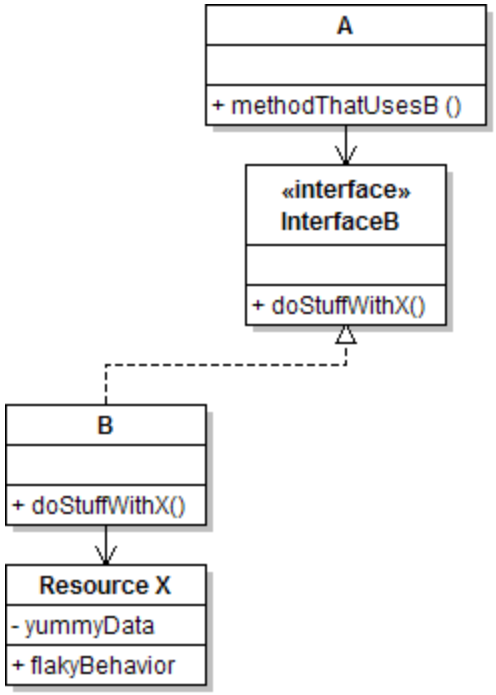
Create a stub, step 1

- Identify the external dependency.
  - This is either a resource or a class/object.
  - If it isn't an object, wrap it up into one.
    - (Suppose that Class A depends on troublesome Class B.)



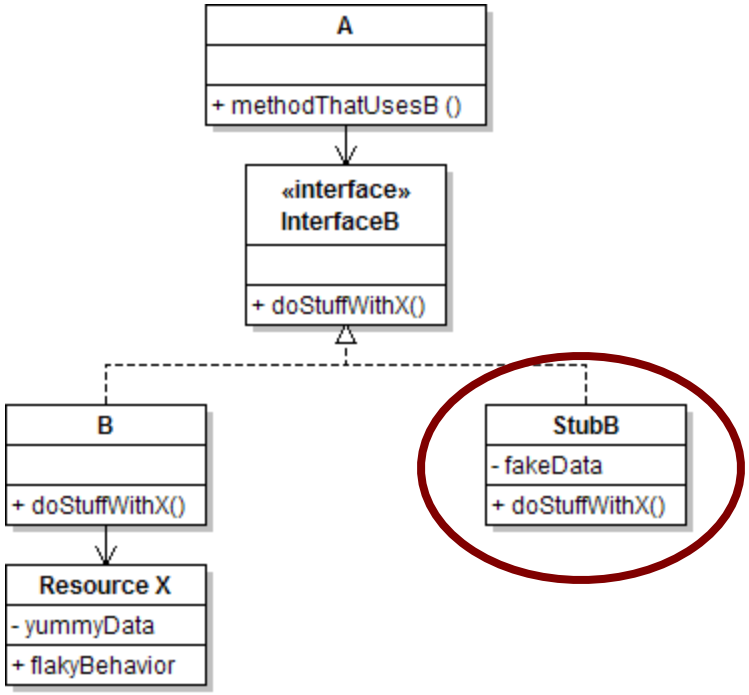
Create a stub, step 2

- Extract the core functionality of the object into an interface.
  - Create an InterfaceB based on B
  - Change all of A's code to work with type InterfaceB, not B



Create a stub, step 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 external behavior.



Injecting a stub

- **seams**: Places to inject the stub so Class A will talk to it.
  - at construction

```
A aardvark = new A(new StubB());
```

  - through a getter/setter method

```
A apple = new A(...);
aardvark.setResource(new StubB());
```

  - just before usage, as a parameter

```
aardvark.methodThatUsesB(new StubB());
```

- All of these are examples of **"dependency injection"**

  - You should not have to change A's code everywhere (beyond using your interface) in order to use your Stub B. (a "testable design")

Testable Design

(subtopic)

Note: The following slides are merely *guidelines* to keep in mind to increase *testability*. These guidelines often come with tradeoffs, and other goals may overrule testability.

material adapted from Software Testing Magazine

Avoid Complex private Methods

- Private methods cannot be tested
- Simple private methods are fine because they likely do not need much testing
- But complex logic in private methods can be a source for bugs that cannot be found by direct testing

Avoid static Methods

- static methods operate on the class rather than the object
- They have their place, for example, in utility classes, for example, Math.cosine()
- But, for functionality that has side effects or that has randomness (anything that we may want to stub out), static makes that difficult or impossible

Be Careful hardcoding in "new"

- This goes to the dependency injection point made earlier in the stubbing slide
- If we hardcode in a "new" then that object cannot be stubbed
- Instead (when appropriate), allow the object reference to be created outside the method and passed in (dependency injection)

Avoid Logic in Constructors

- Constructors are difficult to bypass because a subclass's constructor always triggers at least one of the superclass's constructors
- Better to have a more simple constructor and have the functionality placed in another method
- Basically, make sure that any code that happens in a constructor is not something that we might want to substitute in a test case. If it were moved to a method, it can be overridden.

Avoid Singleton Pattern

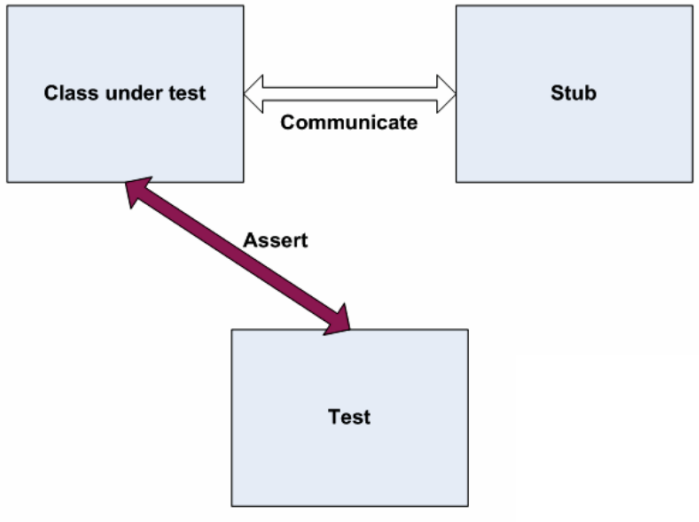
- Singleton pattern ensures that there is only one object instance of a class
- There is appropriate use of this pattern, however, make sure that it is something that does not need to be swapped out for testing

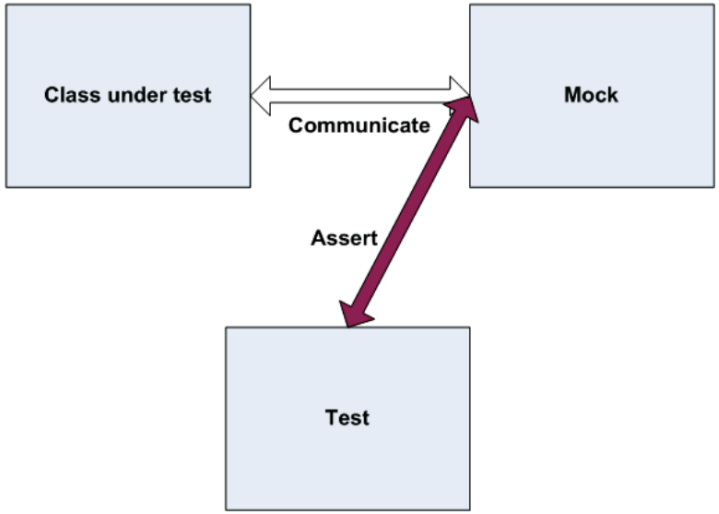
```
public class Clock {
    private static final Clock singletonInstance = new Clock();
    // private constructor prevents instantiation from other classes
    private Clock() { }
    public static Clock getInstance() {
      return singletonInstance;
    }
}
```

Mocking

"Mock" objects

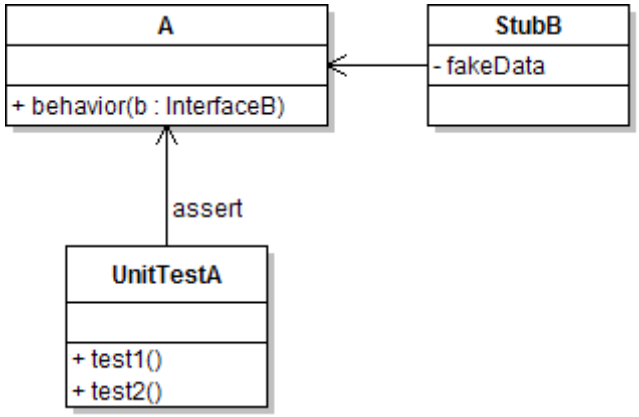
- **mock object**: A fake object that decides whether a unit test has passed or failed by watching interactions between objects.
  - useful for **interaction testing** (as opposed to **state testing**)



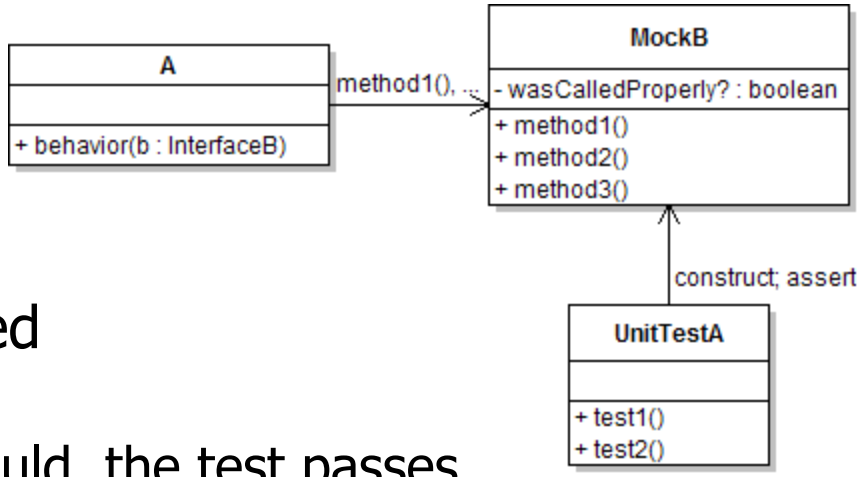


Stubs vs. mocks

- 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, Mockito, 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

Mockito Example

https://www.vogella.com/tutorials/Mockito/article.html

```java
import static org.mockito.Mockito.*;

@Test
public void testVerify()  {
    // create and configure mock
    MyClass test = Mockito.mock(MyClass.class);
    when(test.getUniqueId()).thenReturn(43);


    // call method testing on the mock with
parameter 12
    test.testing(12);
    test.getUniqueId();
    test.getUniqueId();


    // now check if method testing was called with
the parameter 12
    verify(test).testing(ArgumentMatchers.eq(12));

    // was the method called twice?
    verify(test, times(2)).getUniqueId();

    // other alternatives for verifiying the number
of method calls for a method
    verify(test, never()).someMethod("never
called");
    verify(test, atLeastOnce()).someMethod("called
at least once");
    verify(test, atLeast(2)).someMethod("called at
least twice");
    verify(test, times(5)).someMethod("called
five times");
    verify(test, atMost(3)).someMethod("called at
most 3 times");
    // This let's you check that no other methods
where called on this object.
    // You call it after you have verified the
expected method calls.
    verifyNoMoreInteractions(test);
}
```

Mockito Example (Spies)

```
List list = new LinkedList();
List spy = spy(list);

//optionally, you can stub out some methods:
when(spy.size()).thenReturn(100);

//using the spy calls *real* methods
spy.add("one");
spy.add("two");

//prints "one" - the first element of a list
System.out.println(spy.get(0));

//size() method was stubbed - 100 is printed
System.out.println(spy.size());

//optionally, you can verify
verify(spy).add("one");
verify(spy).add("two");
```

https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html

Documentation for Mockito:

https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html

Mocking with Mockito Exercise

Testing with Mockito

The objective of this exercise is to illustrate the concept of Behavioral Driven Development (BDD) and to introduce Mockito, a mocking framework. The exercise involves a small example project, PayRoll.

The primary class of this project is EmployeeManager which is invoked to pay salaries to all the employees of a company. This class has two dependencies — Bank and Company. Both of them are interface, thus have no implemented methods. Bank class is used by EmployeeManager to pay salary to an individual. Company class is used to retrieve the list of all employees of the class.

The goal of the exercise is to test the EmployeeManager class' functionality with mock Bank and Company class. Clone the project from GitHub (link below) and import it on your IDE. Using Mockito complete the empty test methods inside EmployeManagerTest.

https://github.com/marufzaber/Payroll