COMS W4156 Advanced Software Engineering (ASE)

October 4, 2022

Agenda

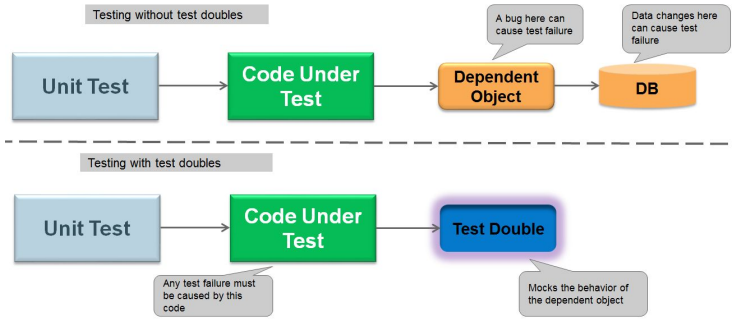
1. Test Doubles (Mocking)
2. API Testing and Mocking demo

Unit Testing in Isolation

Unit testing aims to test each method and class *in isolation from the rest of the program*

The method, or several methods in the same class, is called from a test runner (aka test driver), not from the other code in the rest of the program

When the methods under test would normally call other code in the rest of the program outside the class, or otherwise interact with the environment outside the class, that other code and/or environment is **faked**



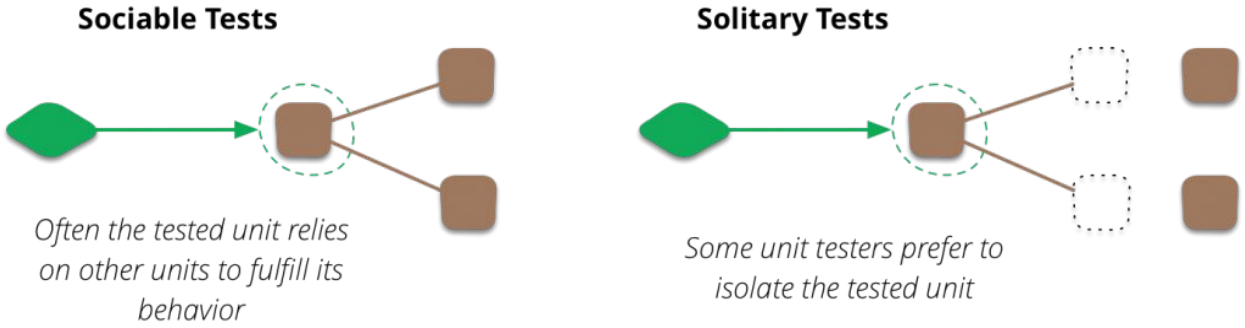
Exceptions: basic libraries like math and string processing

Sociable Tests vs. Solitary Tests

Solitary tests are more expensive than sociable tests

Both require a test driver, but in sociable tests the method under tests calls other code in the codebase and external resources more-or-less the same way it would during production (although usually in a separate testing environment)

Solitary tests instead call special other code (**fakes**) that stands in for that other code and external resources



Unit Tests with Side-Effects

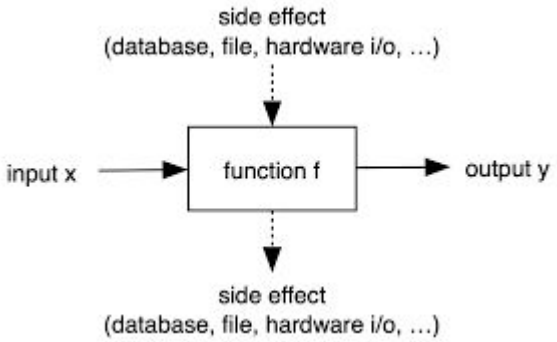
Avoid when possible!

But if your units have, or might have, side-effects outside the class (i.e., beyond the object's own fields), then unit testing needs to consider what to do with these outputs that change global shared state and/or produce externally visible output

Need to **fake** that global shared state and/or external visible output so it isn't really global and it isn't really externally visible

Unit tests need to be very fast, so they can run whenever the unit is changed.
Real resources often too slow

Bugs found during unit testing should be in the unit under test, not in some other code

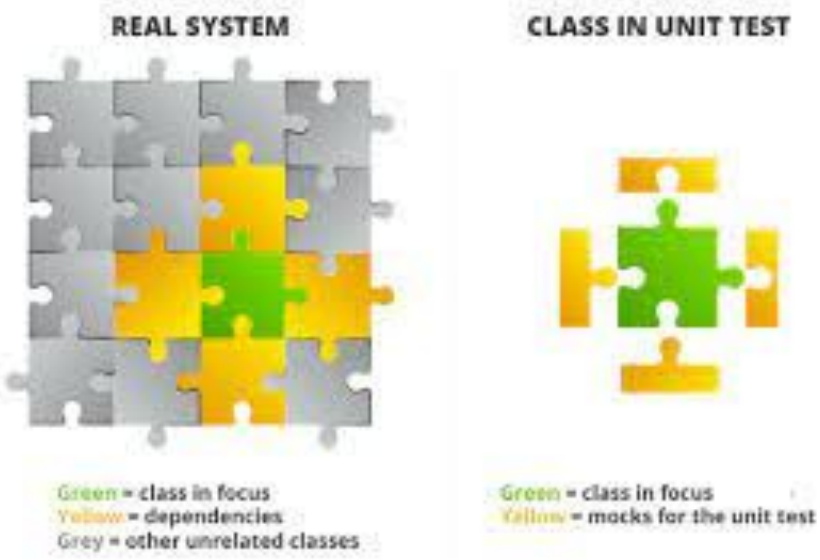


Class Testing

Class (module, component) 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

Complicated when paths and dependencies between units in the same class *go through other code not in the class*, then need specialized **fakes**



Integration Testing in Isolation

Integration testing should also **fake** dependencies outside the code under test

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

However, some integration testing specifically tests interfaces with external resources - in which case real resources are used after initial testing with fakes

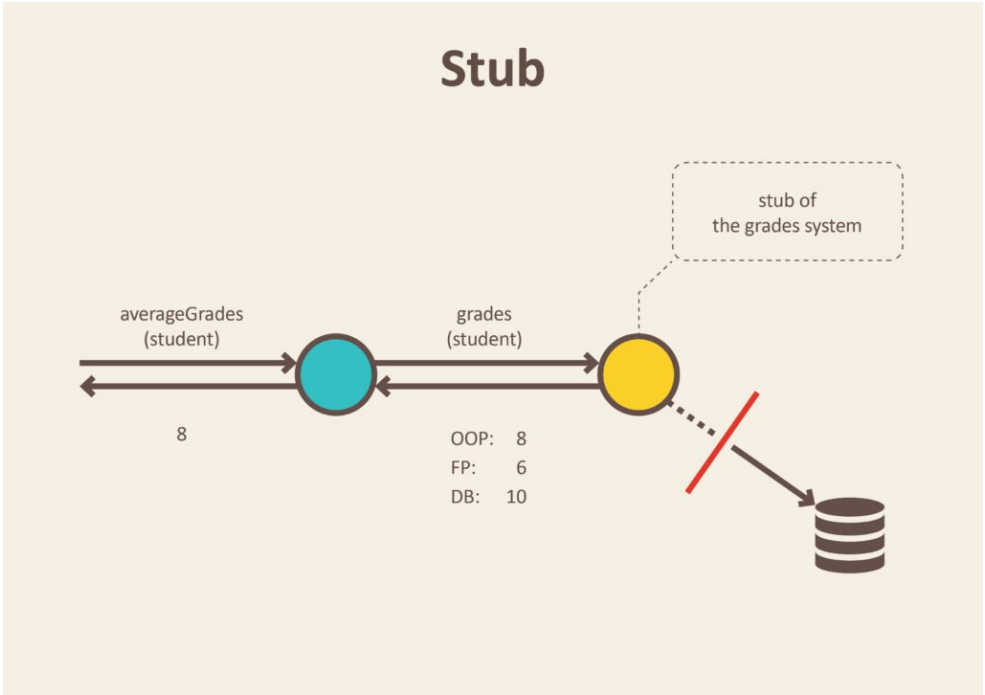
What Are Test Doubles?

The fakes used in testing are called “test doubles”

A test double is anything that replaces a production object or service for testing purposes

Analogous to stunt doubles who perform dangerous action sequences in movies

Several Kinds of Test Doubles



Stubs support *state verification*, meaning they provide canned (hardcoded) data

When called during a test, they always return the same value or always do the same thing

Stubs

```
private class FooStub implements Foo
{
    public String bar()
    {
        return "bar";
    }
}


public class FooCollectionTest
{
    @Test
    public void it_should_return_joined_bars()
    {
        FooCollection sut = new
FooCollection();
        sut.add(new FooStub);
        sut.add(new FooStub);
        assertEquals("barbar", sut.joined());
    }
}
```

A stub implements a required interface and delivers pre-defined inputs to the caller when the stub's methods are called. Stubs are programmed only for the particular test’s scope and might provide inputs intended to force a specific execution path in the caller

Stubs can replace standard or third-party libraries as well as code from your codebase

Stubs can raise exceptions as well as return values. Suppose we need to simulate a hardware failure or transaction timeout, then the stub would always raise that failure or timeout exception

Dummy Objects

A *dummy object* is a special case of a stub that does not even return a value

It’s like a stunt double that does no stunts

In movies, sometimes a double doesn't perform anything; they just appear on the screen. For example, the actor’s character needs to stand in a crowded place where it would be risky for the real actor to go

Dummy Objects

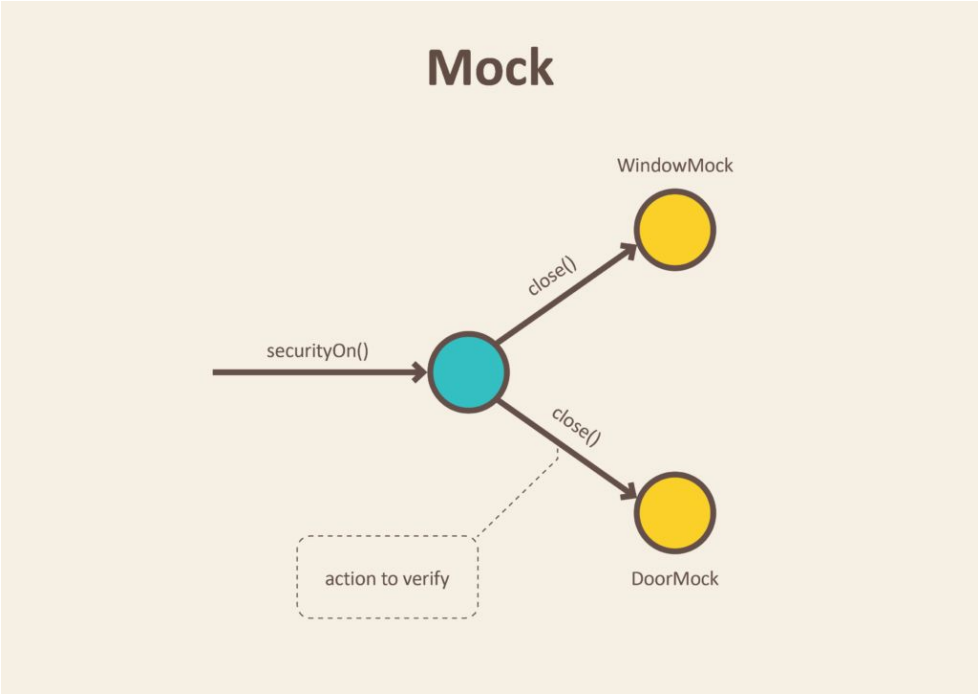
A dummy object might be passed as a mandatory parameter object

The dummy object is not directly used in the test or in the code under test, but it is required to call the code under test

Not the same as a null object - which can indeed be used, e.g., in a conditional - if this parameter is null then do x else do y

```
private class FooDummy implements Foo
{
    public String bar() {
        return null;
    }
}

public class FooCollectionTest
{
    @Test
    public void it_should_maintain_a_count()
    {
        FooCollection sut = new FooCollection();
        sut.add(new FooDummy);
        sut.add(new FooDummy);
        assertEquals(2, sut.count());
    }
}
```

More Kinds of Test Doubles



Mocks support *behavior verification*, meaning they actually do something

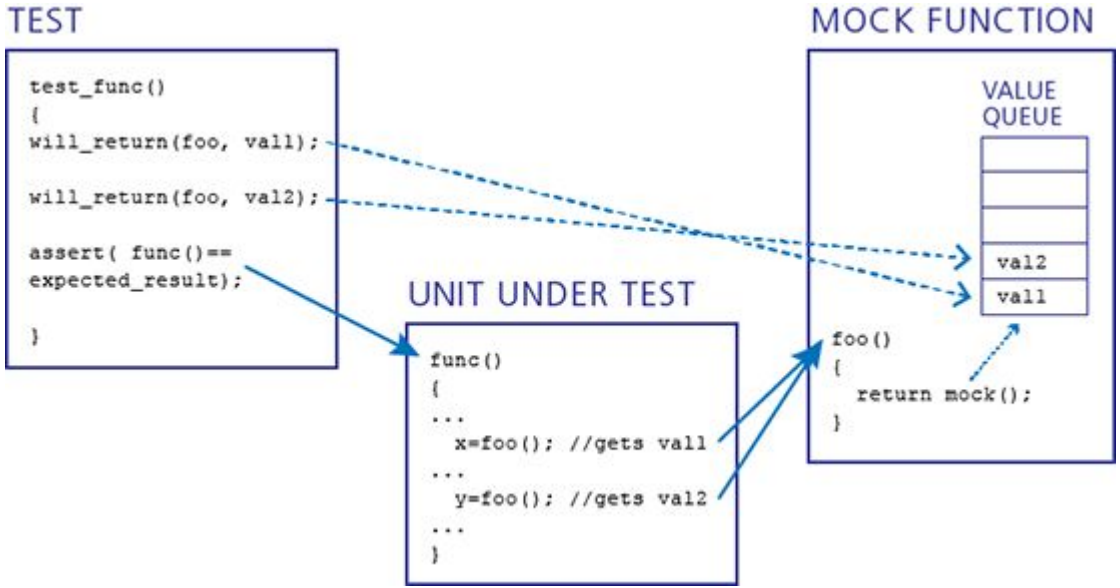
A mock generally does more than a stub

Mocks

A mock might return hardcoded values for certain calls, but it *remembers* those calls and values

We can use mocks to verify that all expected actions are performed in the expected order with no duplicates - mocks might throw an exception when they receive an unexpected call or don't receive an expected call

Spy objects are a special case of mocks added to stubs just to log calls to the stub (the remembering part of mocks)



Example

```
public class SecurityCentral {
    private final Window window;
    private final Door door;

    public SecurityCentral(Window
window, Door door) {
        this.window = window;
        this.door = door;
    }

    void securityOn() {
        window.close();
        door.close();
    }
}
```

```
public class SecurityCentralTest {
    Window windowMock =
mock(Window.class);
    Door doorMock = mock(Door.class);

    @Test
    public void
enabling_security_locks_windows_and_doors(
) {
        SecurityCentral securityCentral =
new SecurityCentral(windowMock, doorMock);
        securityCentral.securityOn();
        verify(doorMock).close();
        verify(windowMock).close();
    }
}
```

How do we know that SecurityCentral will close the real door and the real window when it needs to? We don’t - until integration testing with real resources

Fake Objects

During integration or system testing, we need to test that security central closes real doors and windows - without disturbing production

One way to do this is with a *fake object*

A fake object is a special case of a mock that provides a simplified and/or light-weight working implementation but is not suitable for production use, e.g., an in-memory database, a payments service without real money, a scaled-down network inaccessible by real users

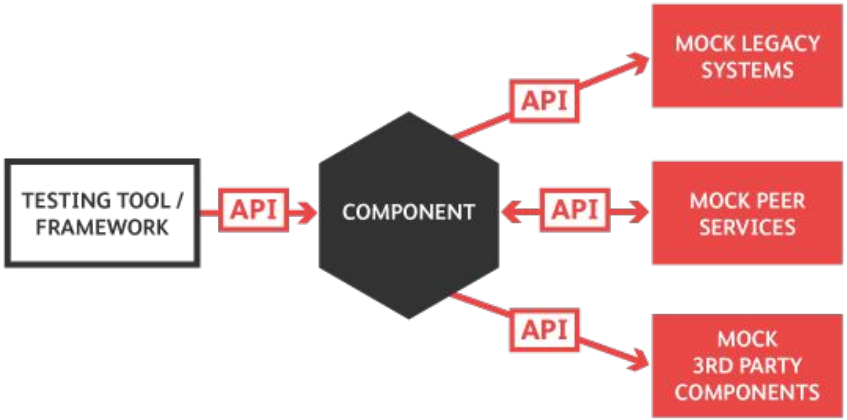
Mocks Combine Fakes with Spies on Fake Objects

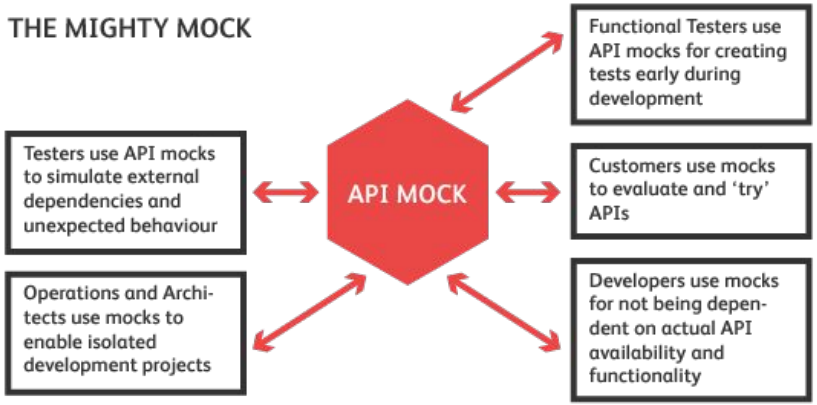
```
public class FooCollectionTest
{
  @Test
  public void it_should_return_joined_bars()
  {
      // instance
      Foo fooMock = mock(Foo.class);
      // behavior
      when(fooMock.bar()).thenReturn("baz", "qux");

      FooCollection sut = new FooCollection();
      sut.add(fooMock);
      sut.add(fooMock);

      assertEquals("bazqux", sut.joined());
      // verify
      verify(fooMock, times(2)).bar();
  }
}
```

Mocking can prevent some tests from being “flaky”, i.e., passing sometimes and failing other times when nothing has changed - covered later in semester

For example, using an in-memory database avoids timeouts and other issues that can arise with a disk-based (or over-the-network) database

Mocks can Serve as Fake APIs





Mocking Frameworks

There are mocking frameworks that work together with unit testing frameworks for most major languages

Java: Mockito, PowerMock, EasyMock, JMockIt

C++: gMock comes with googletest, FakeIt, trompeloeil, mockcpp

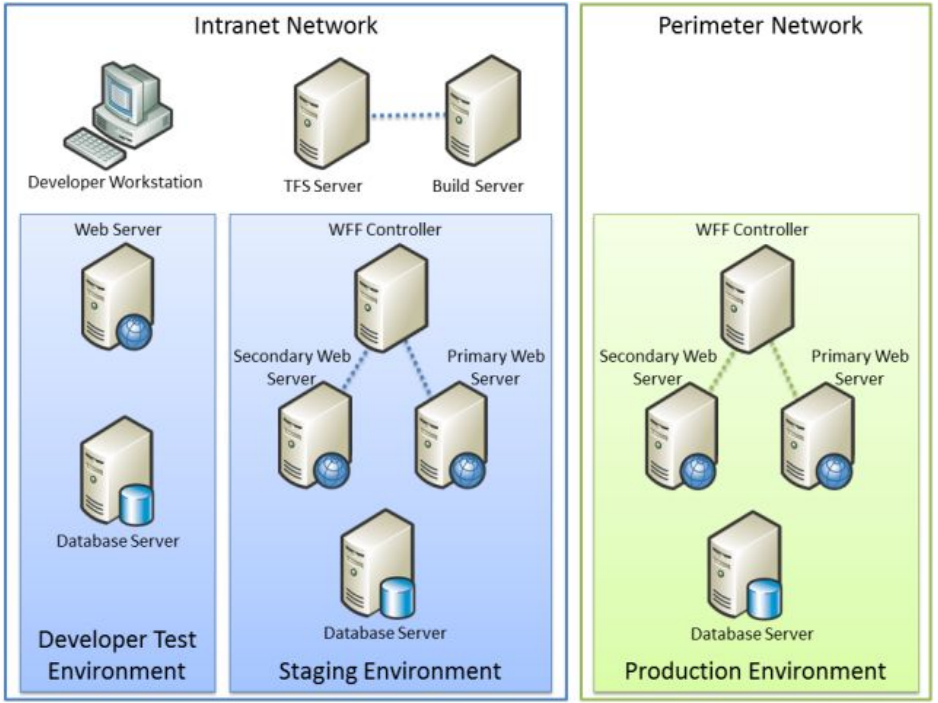
You should use mocks for testing your team project

When NOT To Use Test Doubles

End-to-end system testing should use real resources, but not usually the *production* instances of those resources

End-to-end system testing for financial services does not move real money, it does not send “final notices” to real customers, etc.

Imagine a newly hired developer using their laptop to test their modified code on the production data for amazon, netflix, facebook, … your account on your favorite internet service



Agenda

1. Test Doubles (Mocking)
2. API Testing and Mocking demo

API Testing and Mocking demo: Yunhao Wang

demo presentation

Read The Docs

readthedocs.org Documentation Hosting

GitHub Pages

Upcoming Assignments

Revised Project Proposal: due yesterday but will accept until October 10 - meet with your Mentor to discuss your preliminary proposal *before* submitting revision

First Iteration due October 24

First Iteration Demo due October 31

Next Class

finish Mocking demo

API Testing

if time permits: Equivalence partitions and boundary analysis

Ask Me Anything