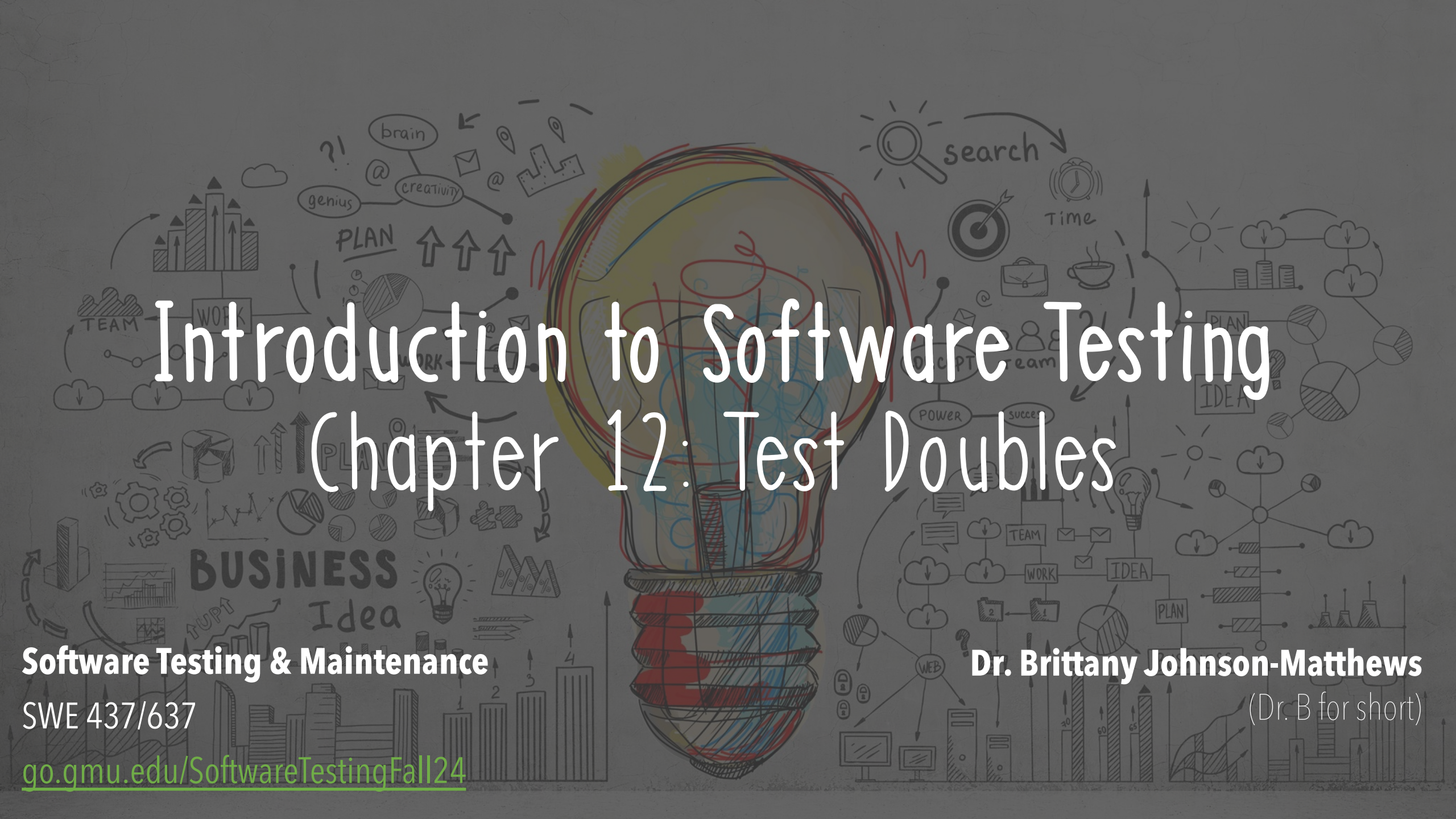


Introduction to Software Testing
Chapter 12: Test Doubles

Software Testing & Maintenance

SWE 437/637

go.gmu.edu/SoftwareTestingFall24

Dr. Brittany Johnson-Matthews

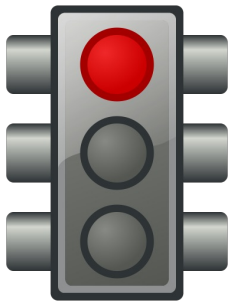
(Dr. B for short)

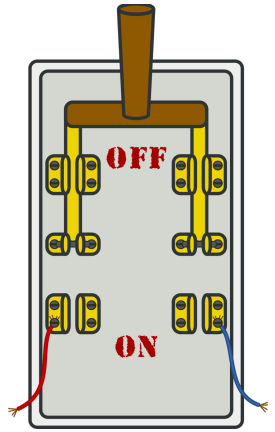
Why do we need test doubles?

Consider testing a system which implements the following constraints:

If the light is red and the valve is open, then release the monster.

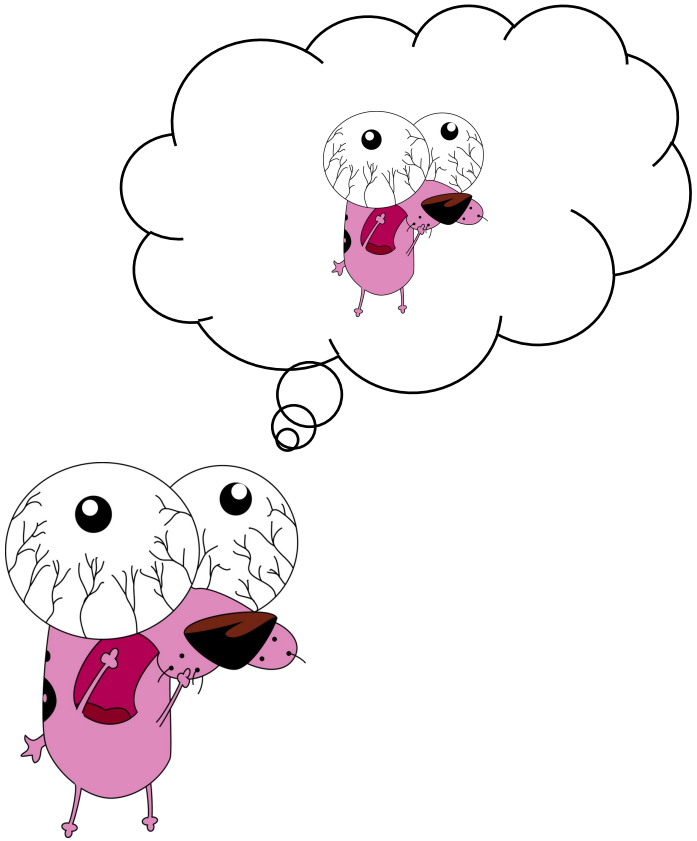
If the valve is open and the switch is on, release the monster.





Why do we need test doubles?

1. If the light is red and the valve is open then release the monster.
2. If the valve is open and the switch is on then release the monster.



Why do we need test doubles?

1. If the light is red and the valve is open then release the ~~monster.~~ puppy
2. If the valve is open and the switch is on then release the ~~monster.~~ puppy

All the (necessary) functionality of the monster, without the mess!

Why do we need test doubles?

We need some dependency-like functionality without the mess

The necessary dependency is **not implemented yet** – but I can't afford to wait

There may be **unrecoverable actions** that result from using the dependency – releasing the monster, sending customer emails, executing a security lockdown, launching a missile

The dependency may have **nondeterministic properties** – intentional randomness, timing, intermittent failures, etc.

Irrelevant changes to the dependency later on may **break our tests** – this is a *major source* of test maintenance effort

Types of test doubles

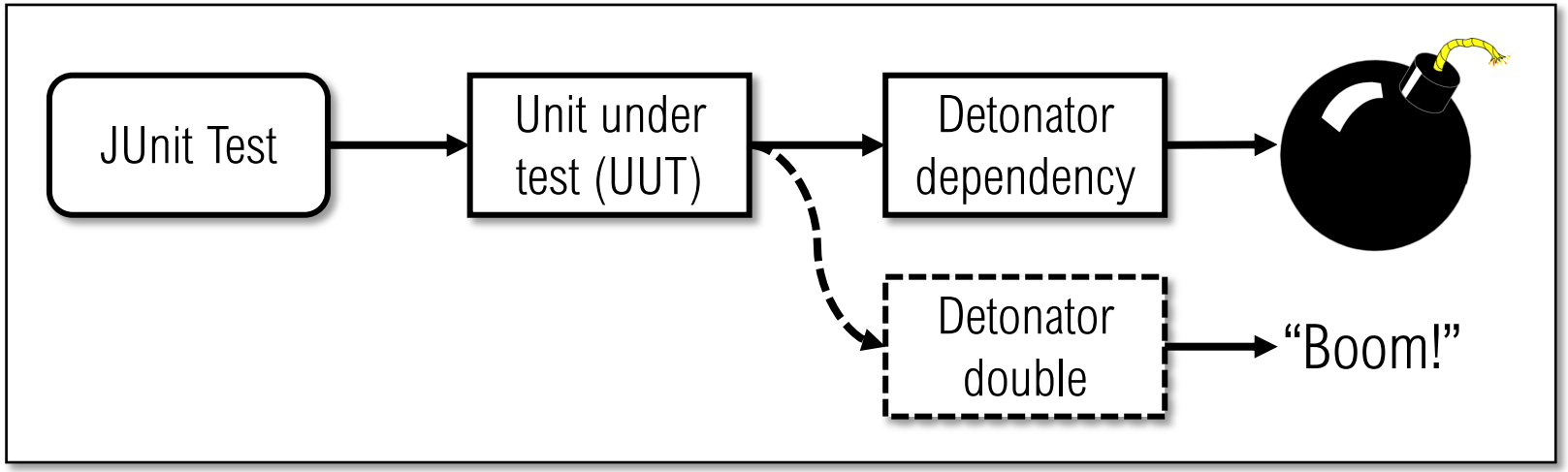
Stubs are custom-developed bits of functionality that simulate the dependency through custom code.

Mocks are tool-provided bits of functionality that simulate the dependency through specification.

There are additional terms in use

- “Dummy”, “spy”, “fake”, etc.
- Some sources differentiate between them, and others don’t – we’ll use *stubs* and *mocks*
- *Simulators* are more complex substitutes that have deep functional capabilities

How to insert test doubles



We could get our double into the test by:

- Having a different version of the UUT that doesn't use the real detonator

  But then we're not testing the deliverable software, and multiple versions are hard to maintain and will tend to get out of sync with each other

What is a seam?

A *seam* is a control point where we can change the behavior of the software

- With the compiler
- With the classpath or linker
- With inheritance
- With the JVM (or more broadly, any interpreter)

Compiler seams

Add code that enables a “test mode”

- `if (TEST_MODE) { // do something special }`
- Use a “test mode” constructor

Advantages

- Easy to understand and fast to implement

Disadvantages

- We’re not testing the same code that will be used in the delivered system

Classpath/linker seams

Replace dependency classes with test double classes by redirecting the classpath to a test directory (in Java) or by linking in alternate test double object files (in C++)

Advantages

- No change to the UUT, we test exactly what we'll deliver
- Fairly easy to implement

Disadvantages

- Maintenance of test double classes can be time-consuming with a risk of using the wrong one

Inheritance seams

Derive a new child class of the dependency and override the functionality with test functionality

- Alternate approach – define an interface and have real and test double classes implement it

```
public class BombDetonatorTestDouble implements BombDetonatorInterface {
   @Override
   public void Detonate () {
      System.out.println("Boom!");
   }
}
```

Advantages

- No change to the UUT, we test exactly what we'll deliver

Disadvantages

- Can be difficult to insert alternate classes, may require code changes to use static factories or *dependency injection*

Dependency injection

Always pass in the dependency via the constructor or setter method, probably as a base class or an interface

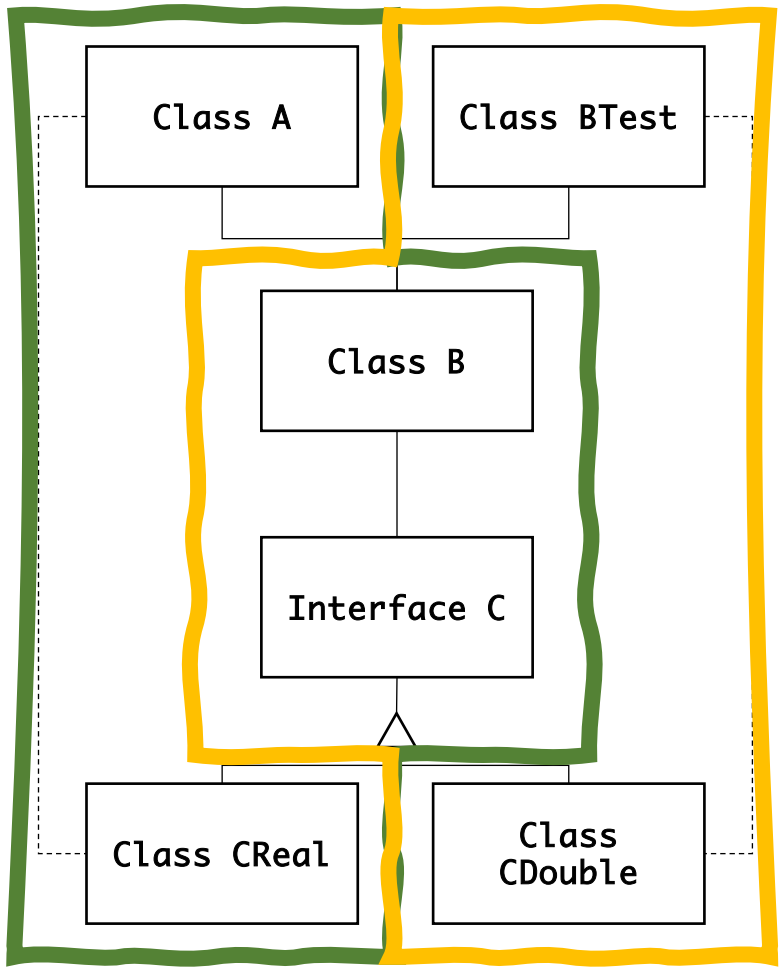
Advantages

- Very flexible, allows lots of types of stubs

Disadvantages

- Breaks encapsulation, as the calling class now needs to know about (and create) the dependency to pass it in – classes need to know about their "grandchild" objects

Dependency injection



`Class B` uses `Interface C`

- Since B can either use the real C or a double for C, B only knows about a generic interface for C
- The real dependency `CReal` and the test double `CDouble` both implement `Interface C`

The class using B provides the appropriate implementation of `Interface C`

- Calling `Class A` creates class `CReal` and passes it to `Class B`
- Calling class `BTest` creates class `CDouble` and passes it to `Class B`

JVM seams

Define a mock object and the mocking framework will cause the JVM to use the mock instead of the real dependency

Advantages

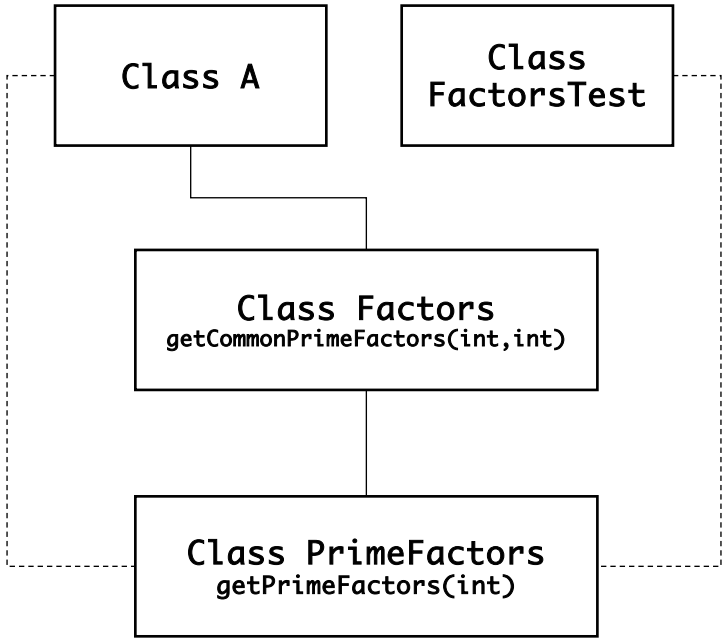
- No change to the UUT or to any other part of the system, *it's completely transparent*
- Easy with the right mocking framework

Disadvantages

Only works with Java

- Typically substitutes a single static mock in place of any/all instances of the real dependency

Mocking getPrimeFactors()



Class Factors

- Has method `getCommonPrimeFactors()` that returns the common prime factors of two integers
- Calls method `getPrimeFactors()` of class `PrimeFactors` to get the prime factors for each integer
- Is called by `Class A`
- Is tested by class `FactorsTest`

Mocking getPrimeFactors()

```
@RunWith(JMockit.class)
public class FactorsTest
{
   @Mocked
   PrimeFactors primeFactorsMock;

   @Test
   public void testCommonPrimeFactors() {
      // Define output data needed by the prime fact    mock
      List<Integer> factorsOf4 = Arrays.asList(2);
      List<Integer> factorsOf6 = Arrays.asList(2, 3);

      // Specify the mock behavior
      new Expectations() {{
         PrimeFactors.getPrimeFactors(4);
         returns(factorsOf4);

         PrimeFactors.getPrimeFactors(6);
         returns(factorsOf6);
      }};

      // Execute the test
      List<Integer> commonPrimeFactors = Factors.getCommonPrimeFactors(4, 6);
      assertEquals(1, commonPrimeFactors.size());
      assertEquals(2, commonPrimeFactors.get(0).intValue());
   }
}
```

Declare the mock – the JVM will automatically use it instead of the real PrimeFactors class

Declare data for the mock to return

Tell the mock what it should expect and what it should do

Execute the test and verify the expected results (as usual)

Stub, Mock, or Simulator?

For extremely simple dependency substitutions, like when return true is all that's needed, a stub may be sufficient

- But since it typically results in yet another test class, using a mock might still be easier

For more complex scenarios when you want to verify that calling expectations were met and want to return more complex sequences of values, a mock is preferable

For highly complex scenarios like generating real-time data streams based on physical systems, a simulator is appropriate

More about seams

For a details on how to exploit seams in legacy software for testing, see Michael Feathers' book *Working Effectively with Legacy Code*