Software Testing for Continuous Delivery

Seminar 10: Dependencies and Mocks

Dr. Byron J. Williams
September 30, 2019

Herbert Wertheim College of Engineering
Department of Computer & Information Science & Engineering
UNIVERSITY *of* FLORIDA

FICS Research

Legacy Code - Testing Challenges

- “Our system can’t be tested”
- “Unit testing/test-driven development only works in green field projects.”
- complex or botched architecture, inconsistent design, or simply code written with everything but testability in mind
- Problem: dependencies
  - Different parts of a system depend on each other in different ways, and the exact nature of these dependencies affects testability.

Relationship b/w Program Elements

- In object-oriented systems, the tested object will make use of other objects, from now on called ***collaborators***
  - Some are heavyweight and deeply entrenched in the system;
  - others are simple and provide very narrow functionality
- We use **test doubles** to deal with collaborators
- systems are usually composed of thousands of classes, and their instances form intricate webs of relations between collaborating objects

Collaborator Example

```
public class Raffle
{
    private ISet<int> tickets;

    public int TicketCount
    {
        get { return tickets.Count; }
    }

    public Raffle()
    {
        tickets = new HashSet<int> { 3, 10, 6 };
    }
}
```

- the constructor creates another object, and thus relies on indirect input, to produce a class that's small and yet can be difficult to test
- difficult to write a unit test to establish a relation between the object created in the constructor and the class's public interface—*TicketCount* property

An Alternative

```
public class Raffle
{
    private ISet<int> tickets;

    public int TicketCount
    {
        get { return tickets.Count; }
    }

    public Raffle(ISet<int> tickets)
    {
        this.tickets = new HashSet<int>(tickets);
    }
}
```

- making collaborators explicit by passing them around is the simplest and most obvious way to increase testability
- downside is the increase in complexity and sometimes decrease in intuitiveness, especially in trivial cases

To Test

```
[TestMethod]
public void RaffleHasFiveTickets()
{
    var testedRaffle = new Raffle
        (new HashSet<int> { 1, 2, 3, 4, 5 });
    Assert.AreEqual(5, testedRaffle.TicketCount);
}
```

- passing in collaborators using constructors or setters is usually appropriate when the dependent object isn’t short lived and is at the same level of abstraction as the object that uses it

Using a Factory Method

```
public class Raffle
{
    private ISet<int> tickets;

    public int TicketCount
    {
        get { return tickets.Count; }
    }

    public Raffle()
    {
        tickets = CreateTickets();
    }

    protected virtual ISet<int> CreateTickets()
    {
        return new HashSet<int> { 1, 2, 3 };
    }
}
```

- factory method to create tickets
- a reasonable trade-off between complexity and readability - approach often saves the day in legacy code
- however, calling overridable methods from a constructor may be considered bad practice because such methods can easily reference uninitialized member variables and crash the application by doing so

System Resource Dependencies

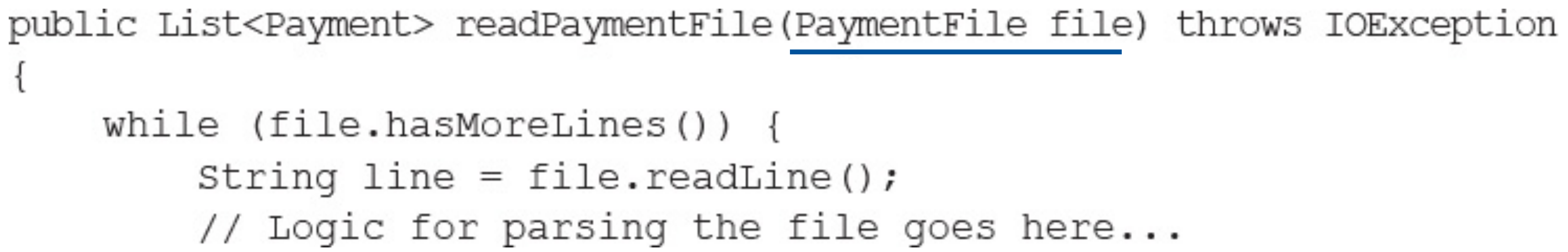
- **System resource** refers to an abstraction of an operating system artifact e.g., a file, the system clock, a network socket, etc.
- Although such resources are abstracted appropriate language constructs (e.g., classes), they still have an impact on unit tests
- Its use could trigger behavior and side effects way outside the *test harness*, like writes to disk or blocking reads

File Dependency Example

```
public List<Payment> readPaymentFile(String filename) throws IOException {
    File paymentFile = new File(filename);
    BufferedReader reader
        = new BufferedReader(new FileReader(paymentFile));

    String line;
    while ((line = reader.readLine()) != null) {
        // Logic for parsing the file goes here...
```

- Passing in a filename to a method is providing input to indirect input (read this sentence again)
- The paymentFile variable is indirect input, whereas the filename parameter is its input

Provide Your Own Abstraction



```
public List<Payment> readPaymentFile(PaymentFile file) throws IOException
{
    while (file.hasMoreLines()) {
        String line = file.readLine();
        // Logic for parsing the file goes here...
```

- Imagine that we introduced a simple abstraction called PaymentFile that wrapped an instance of File to improve readability and testability of the readPaymentFile method

Provide Your Own Abstraction

```
public List<Payment> readPaymentFile(String filename) throws
IOException {
    return readFileContents(new FileInputStream(filename));
}

List<Payment> readFileContents(InputStream inputStream) throws IOException {
    List<Payment> parsedPayments = new ArrayList<>();
    BufferedReader reader = new BufferedReader(
            new InputStreamReader(inputStream));

    String line;
    while ((line = reader.readLine()) != null) {
        String[] values = line.split(";");
        parsedPayments.add(new Payment(parseReference(values[0]),
                parseAmount(values[1]),
                parseDate(values[2])));
    }
    return parsedPayments;
}
```

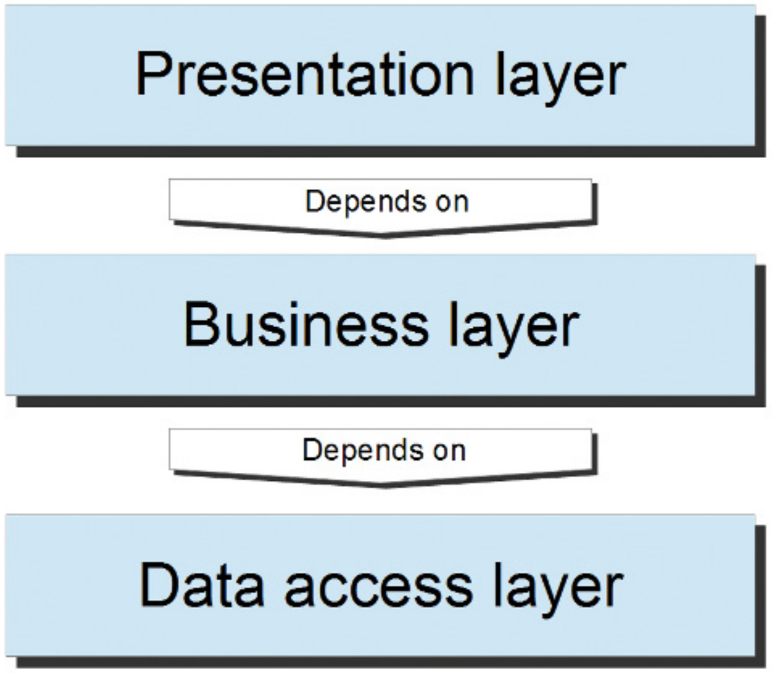
- file opened
- programming language provides us with a convenient abstraction of its contents (e.g., stream object, list of file contents).
- All these can easily be controlled by a unit test
- splitting the pure file I/O from whatever's done with the contents of the file is a good refactoring that not only benefits testability, but also promotes separation of concerns

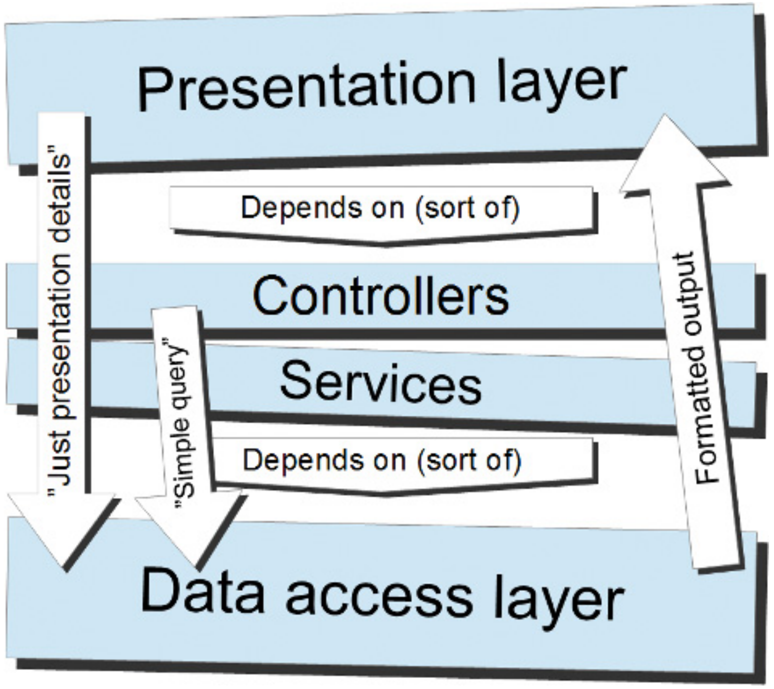
Test File Dependency

```
@Test
public void parseLineIntoPayment() throws Exception {
    String line = "912438784;1000.00;20151115\n";

    List<Payment> payments = new PaymentFileReader().readFileContents(
            new ByteArrayInputStream(line.getBytes()));

    Payment expectedPayment = new Payment("912438784",
            new BigDecimal(1000.00,
                    new MathContext(2, RoundingMode.CEILING)),
            LocalDate.of(2015, Month.NOVEMBER, 15));
    assertEquals(expectedPayment, payments.get(0));
}
```

- The corresponding test would set up the file contents as a string and create a stream from it

Dependencies b/w Layers





UF Herbert Wertheim College of Engineering
UNIVERSITY *of* FLORIDA

POWERING THE NEW ENGINEER TO TRANSFORM THE FUTURE

DEPARTMENT OF COMPUTER & INFORMATION SCIENCE & ENGINEERING
FLORIDA INSTITUTE FOR CYBERSECURITY RESEARCH (FICS)
Dr. Byron J. Williams

Unit Testing++

Medium Tests (Google), Integration Tests (JPM), Fast Medium Test (“Developer Testing”)

Unit Testing

- A test that invokes a **small**, **testable unit** of work in a software system and then **checks a single assumption** about the resulting output or behavior
- Key concept: **Isolation** from other software components or units of code
- Low-level and focused on a tiny part or “unit” of a software system
- Usually written by the **programmers themselves** using common tools
- Typically written to be **fast** and run along with other unit tests in **automation**

References: Some concepts from martinfowler.com/bliki/UnitTest.html and artofunittesting.com/definition-of-a-unit-test/

RECAP

Trouble with Defining Unit Tests

- **Tests are not classified**— not every developer cares about whether a test is a unit test, an integration test, or an end-to-end test (or simply doesn't know)
- **The test suite is small**—If the test suite consists of a 100 unit tests in total, does it matter if 30 of them take a few extra seconds to run?
- **Laziness**— running all tests as unit tests requires practically no effort
- **Hurry**—Sometimes you want to go really fast, especially at the beginning of a project. In such cases, growing into a more advanced build as you go along isn't a bad idea. In this stage, fast medium tests may live in the unit test suite.

See Almost Unit Test Notes on:

- Tests Using In-memory Databases
- Test-specific Mail Servers
- Tests Using Lightweight (web) Containers
- Tests of Web Services

Challenges with Execution Time / Speed

- **Slower developer feedback**— run fast, but slower than unit tests — developers, used to quick feedback, may stop running them while writing code — not a good thing, especially if they abandon the habit of using tests to get feedback about their code.
- **Slower in a continuous integration (CI) environment**— difference in performance will be apparent when running the unit tests, which will become not-so-fast tests on weaker machines
- **Shaky portability**— preceding examples could easily be affected by local permissions, disk space, firewall settings, or occupied sockets
- **Confusing**—fast medium tests tend to blur the line between different types of tests — developers who are new on the team will wonder what goes where
- **Sluggish unit test suite**—One in-memory database test takes one second. Ten such tests take two seconds. Combine that with some other almost unit tests, and the unit test suite will start getting sluggish. Not sluggish enough to trigger any rework, but slow enough to annoy somebody on a bad day. If the sluggishness passes a certain threshold, the tests will no longer be executed.

Herbert Wertheim College of Engineering
UNIVERSITY *of* FLORIDA

POWERING THE NEW ENGINEER TO TRANSFORM THE FUTURE

DEPARTMENT OF COMPUTER & INFORMATION SCIENCE & ENGINEERING
FLORIDA INSTITUTE FOR CYBERSECURITY RESEARCH (FICS)
Dr. Byron J. Williams

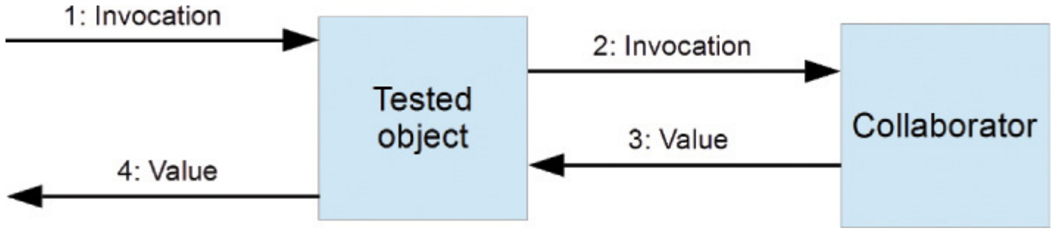
Test Doubles

Test Doubles

- **Test double** — a general term for an object that replaces a collaborator
- Different kinds of test doubles have different tasks, spanning from replacing the collaborator and making it return predefined values, to monitoring every single call to it
- Generally 5 kinds of test doubles: stubs, fakes, mock objects, spies, and dummies.

Testing Objects with Collaborators

- simplest and most generic test of an object that depends on another object



```
[TestMethod]
public void CanonicalTest()
{
    var tested = new TestedObject(new Collaborator());
    Assert.AreEqual(?, tested.ComputeSomething());
}
```

```
public int ComputeSomething()
{
    return 42 * collaborator.ComputeAndReturnValue();
}
```

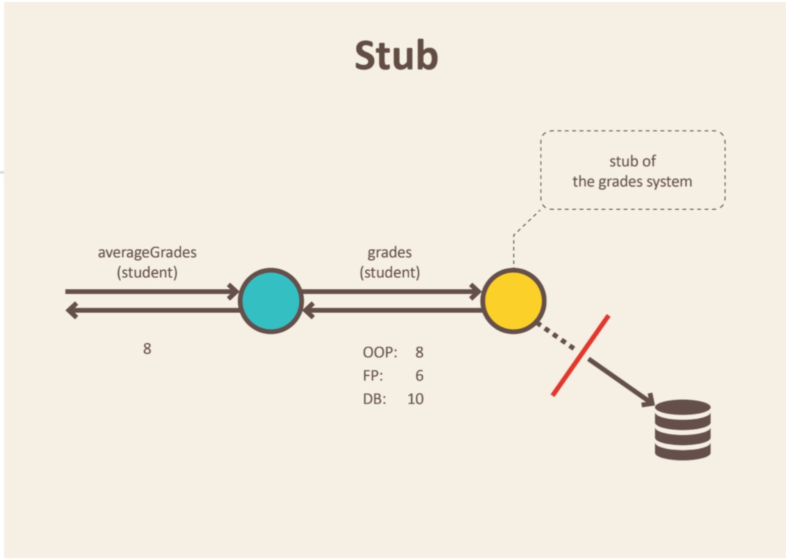
Stubs

```
class CollaboratorStub : ICollaborator
{
    public int ComputeAndReturnValue()
    {
        return 10;
    }
}


   [TestMethod]
   public void CanonicalTestWithStub()
   {
       var tested = new TestedObject(new CollaboratorStub());
       Assert.AreEqual(420, tested.ComputeSomething());
   }
```

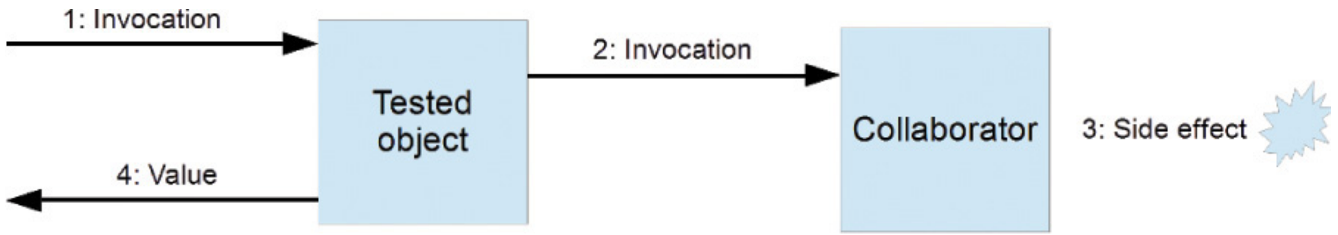
- To take control of a dependency like this, a stub is needed. The primary motivation behind **stubbing is to control the tested object's indirect input**. Because the collaborator is injected in the constructor of the tested object, creating a stub is very straightforward. All that's needed is an implementation that returns a hard-coded value.

Stubs

- collaborator doesn't return anything
- Assuming that side effects aren't the focus of the test, we just need a way to get rid of them. To do that, all we need to do is to implement **an "empty" stub** that replaces the side effect–ridden code

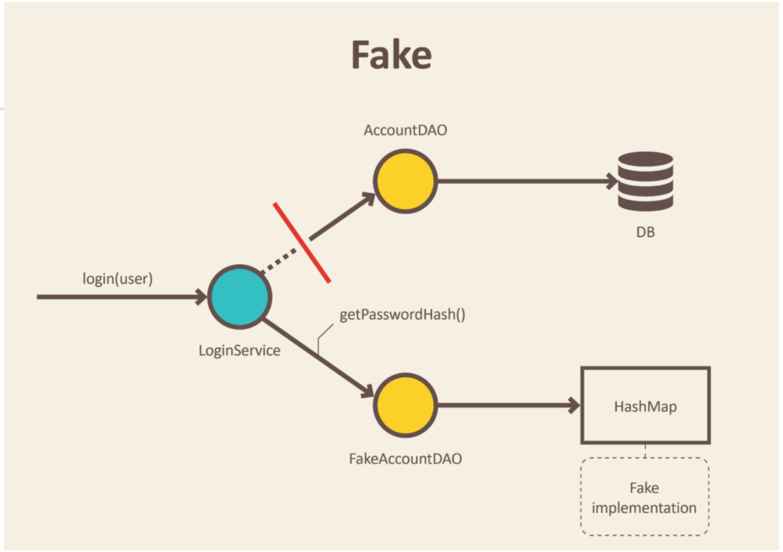


https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da



Fakes

- When stubbing is not enough:
  - The behavior that would be stubbed away is required by the tested object. On the other hand, it comes with side effects and shenanigans that would break a unit test
    - a fake object may be a reasonable trade-off
  - Fake objects are lightweight implementations of collaborators, and their primary purpose is to provide something that's self-consistent from the perspective of the caller

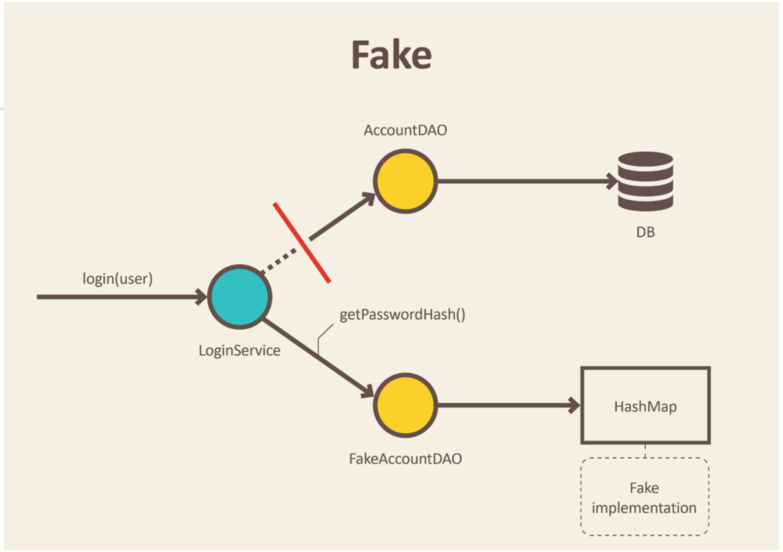


https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da

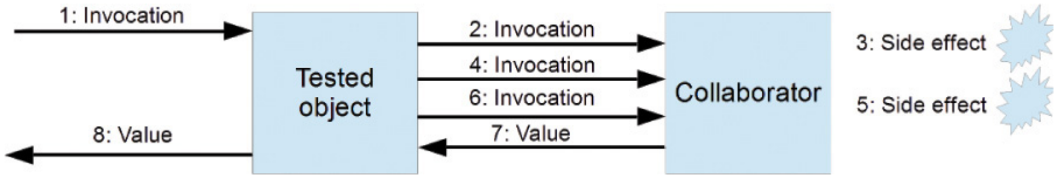
Fakes

- These calls not only affect its state, but also result in side effects. Afterward, the object expects a nontrivial result that is somehow based on those calls.

```
public Invoice MakePurchase(Customer customer,
        Product product, Discount discount)
{
    var purchase = purchaseFacade.CreatePurchase(customer);
    purchaseFacade.AddProduct(purchase, product);
    var invoice = purchaseFacade.CreateInvoice(purchase);

    if (discount != null)
    {
        invoice.ApplyDiscount(discount);
    }
    return invoice;
}
```

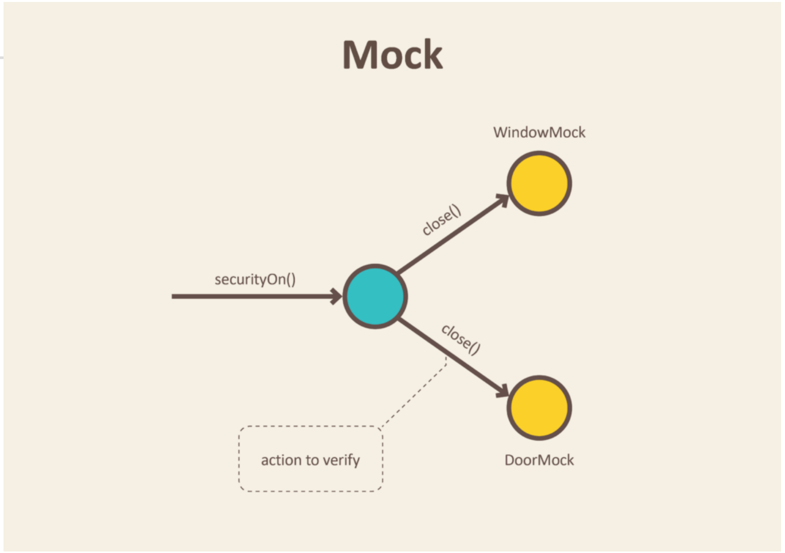


https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da



Mocks

- Stubs provide indirect input in a controlled manner
- Fakes replace collaborators with simpler self-consistent implementations
- The missing piece of the puzzle is the ability to **verify indirect output**. This is the purpose of a mock object, or more commonly just "mock."
  - they shift a test's focus from state to behavior



https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da

State vs. Behavior

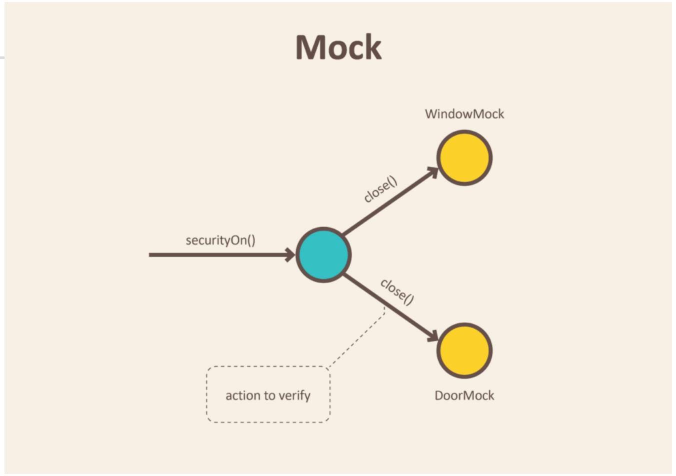
- Tests that focus on state end with assertions that check return values or somehow query the tested object's state

```
assertEquals(expectedValue, tested.computeSomething())
Assert.AreEqual(expectedValue, tested.Value).
```

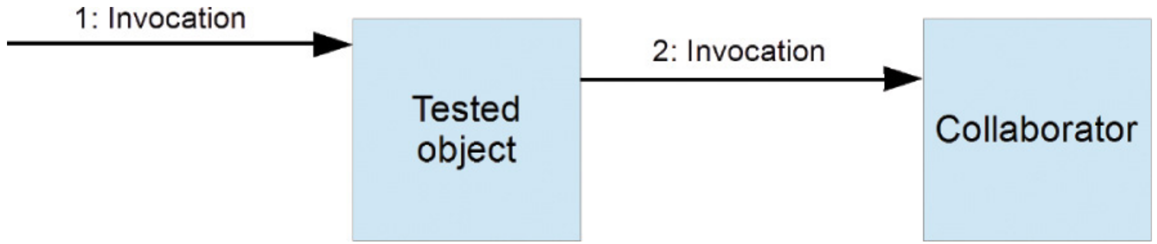
- Behavior-based tests are fundamentally different. Their goal is to v**erify that certain interactions have occurred** between the mock object and the tested code (or collaborator)
- The two preceding assertions care about what a method returns and what value a property has, a behavior-based test making use of mock objects would care about whether tested.computeSomething **has been called**, and possibly how many times, and whether the Value property **has been queried**

Verifying Indirect Output

- Suppose that the tested object invokes a method on another object and gets nothing back; that is, it calls a void method. Furthermore, that method may produce one or more side effects that simply won't work with unit tests. Such a dependency could just be stubbed away, but in this case the goal of the test is to make sure that the collaborating object is actually called properly.



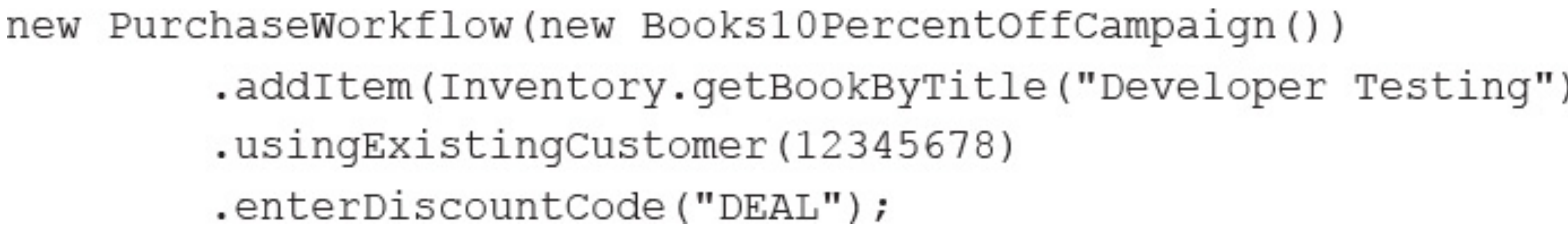
https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da



mock objects are preprogrammed with expectations on the interactions to come
— the primary use of mock objects: **verification of interactions**

Mocks

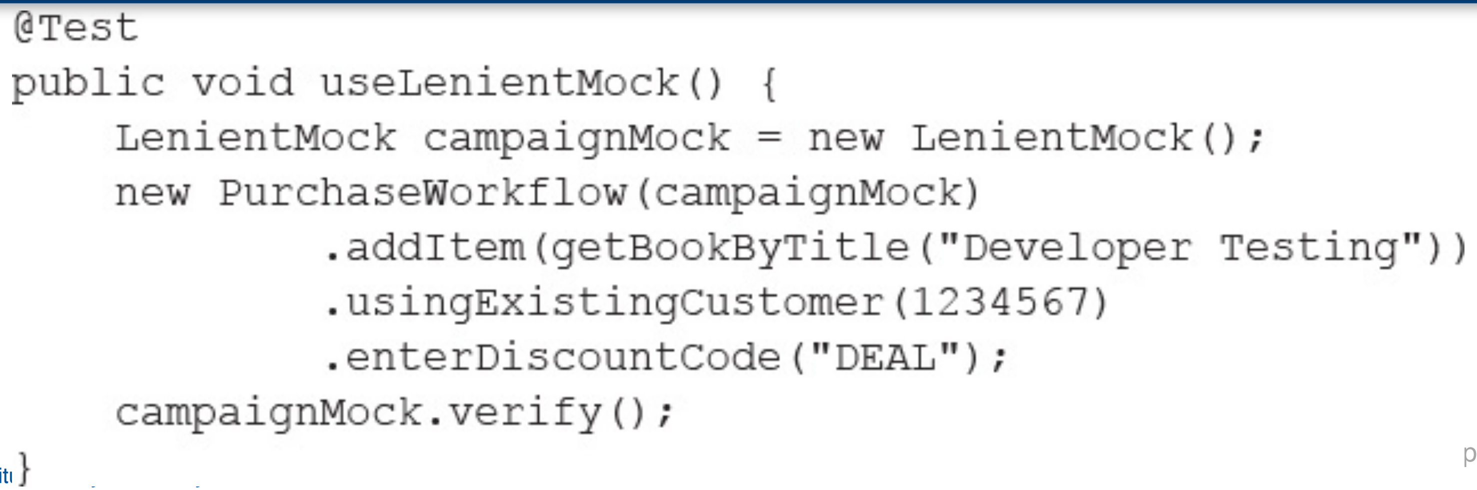
- Let’s assume that we’re modeling a shopping workflow, the kind that you go through when buying things online: You pick the items you want to buy, identify yourself, and finally you apply a discount code (if you have one) before checking out



```
new PurchaseWorkflow(new Books10PercentOffCampaign())
        .addItem(Inventory.getBookByTitle("Developer Testing")
        .usingExistingCustomer(12345678)
        .enterDiscountCode("DEAL");
```

Mocks

- suppose that we want to test how this purchase flow interacts with the object that represents a campaign. We want to make sure that the campaign’s applyDiscount method is indeed invoked and that its arguments are correct. Thus, a test using a mock object instead of a real campaign object verifies the indirect output of the PurchaseWorkflow class when applying a campaign discount.



```
@Test
public void useLenientMock() {
    LenientMock campaignMock = new LenientMock();
    new PurchaseWorkflow(campaignMock)
            .addItem(getBookByTitle("Developer Testing"))
            .usingExistingCustomer(1234567)
            .enterDiscountCode("DEAL");
    campaignMock.verify();
}
```

Mocks

```
private class LenientMock implements Campaign {

    private boolean wasInvoked = false;

    @Override
    public void applyDiscount(Long customerNumber,
                              String discountCode,
                              Purchase purchase) {
        wasInvoked = true;
    }

    public void verify() {
        assertTrue(wasInvoked);
    }
}
```

- Here we just want to verify that the PurchaseWorkflow class indeed calls a campaign's applyDiscount method
- The corresponding mock object confirms the interaction without caring about the parameters passed to applyDiscount. Note that the verify method contains an assertion! This is the mock object's way of telling that it knows what to verify.

More Examples in Notes

Spies

- The distinction between spies and mocks is academic
  - mocks are implemented so that they fail a test if their expectations aren't met
  - spies capture their interactions and the associated parameters for later use.
  - the difference is that the mock itself uses the captured values to determine whether the interaction happened correctly, whereas the spy leaves this decision to the test.

Dummies

- Dummies are values you don’t care about from the perspective of the test. They’re typically passed as arguments, although they can be injected or referenced statically at times.

```
[TestMethod, ExpectedException(typeof(ArgumentOutOfRangeException))]
public void ShouldFailForTooYoungCustomers()
{
    int age = 10;
    string ignoredFirstName = "";
    string ignoredLastName = "";
    CustomerVerifier.Verify(age, ignoredFirstName,
    ignoredLastName);
}
```