How to increase unit test quality by removing avoidable mocks?

Crelder
Brown Bag, March 15th 2022

Problem

Unit tests with many mocks:

- Bind test to **implementation details** of your production code
- Test the **HOW** and not the WHAT

```
public function test…()
{
   $rule = ['type' => 'REPLACE_TEXT',
            'params' => [
                  'string' => ['find' => ['real'], 'replace' => ['Real']],
                  'fields' => [31 => 'Title'],
   ];

   $attribute = m::mock(Attribute\Entity::class);
   $attribute->shouldReceive('getId')->andReturn('31');
   $attribute->shouldReceive('getName')->andReturn('Title');

   $atom = m::mock(CommandAtom\Entity::class);
   $atom->shouldReceive('getAttribute')->andReturn($attribute);
   $atom->shouldReceive(['getItem->getId' => 123]);
   $atom->shouldReceive('getModifiedValue')->andReturn('This is a test!');
   $atom->shouldReceive('modify');
   $atom->shouldReceive('setModifiedValue');

   $rule = m::mock(EnrichmentRuleModel\Entity::class);
   $rule->shouldReceive('getTitle')->andReturn('TestRuleTitle');

   $this->actionFactory
        ->returnAction(json_encode($object))->process($atom, $rule);

   $atom->shouldHaveReceived('getAttribute')->times(1);
   $atom->shouldHaveReceived('getModifiedValue')->times(4);
   $atom->shouldNotHaveReceived('modify');
   $atom->shouldNotHaveReceived('setModifiedValue');

   $attribute->shouldNotHaveReceived('getName');
}
```

Two schools of unit testing

Classical School

```
public class CustomerTests
  {
  public void Purchase_succeeds_when_enough_inventory()
     {
     // Arrange
     var store = new Store();
     store.AddInventory(Product.Shampoo, 10);
     var customer = new Customer();

     // Act
     bool success = customer
                    .Purchase(store, Product.Shampoo, 5);

     // Assert
     Assert.True(success);
     Assert.Equal(5, store.GetInventory(Product.Shampoo));
     }
  }
```

Isolation of	A unit is	Uses mocks for
Unit tests	A unit of behavior (one or more classes)	shared dependencies

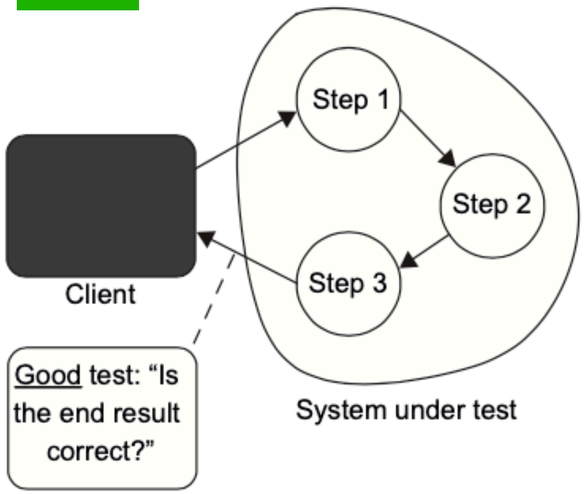
Mockist School

```
public class CustomerTests
  {
  public void Purchase_succeeds_when_enough_inventory()
     {
     // Arrange
     var storeMock = new Mock<IStore>();
     storeMock
        .Setup(x => x.HasEnoughInventory(Product.Shampoo, 5))
        .Returns(true);
     var customer = new Customer();

     // Act
     bool success = customer
                    .Purchase(storeMock.Object, Product.Shampoo, 5);

     // Assert
     Assert.True(success);
     storeMock.Verify(x => x.RemoveInventory(Product.Shampoo, 5),
              Times.Once);
     }
  }
```

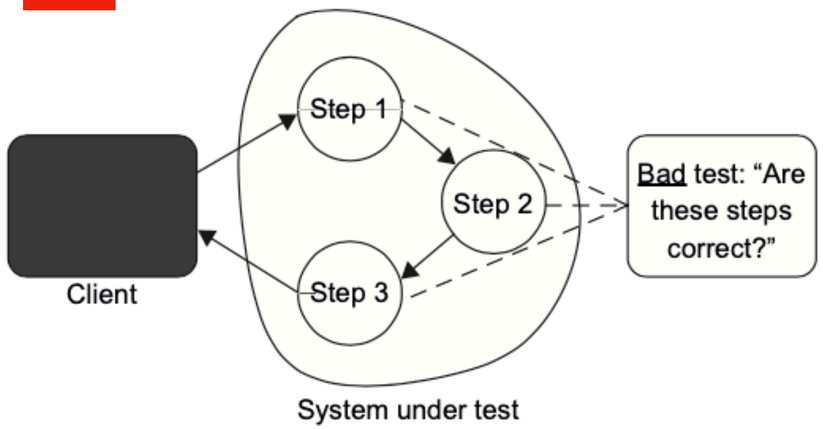
Isolation of	A unit is	Uses mocks for
Units	A class	all but immutable dependencies

Observable behavior vs. implementation details

Good = tests the WHAT



Bad = tests the HOW



- Tests will probably fail when you **refactor** this code because you bind your tests to implementation details.
- Therefore you will have more **false alarms**, hence decreasing the test accuracy!

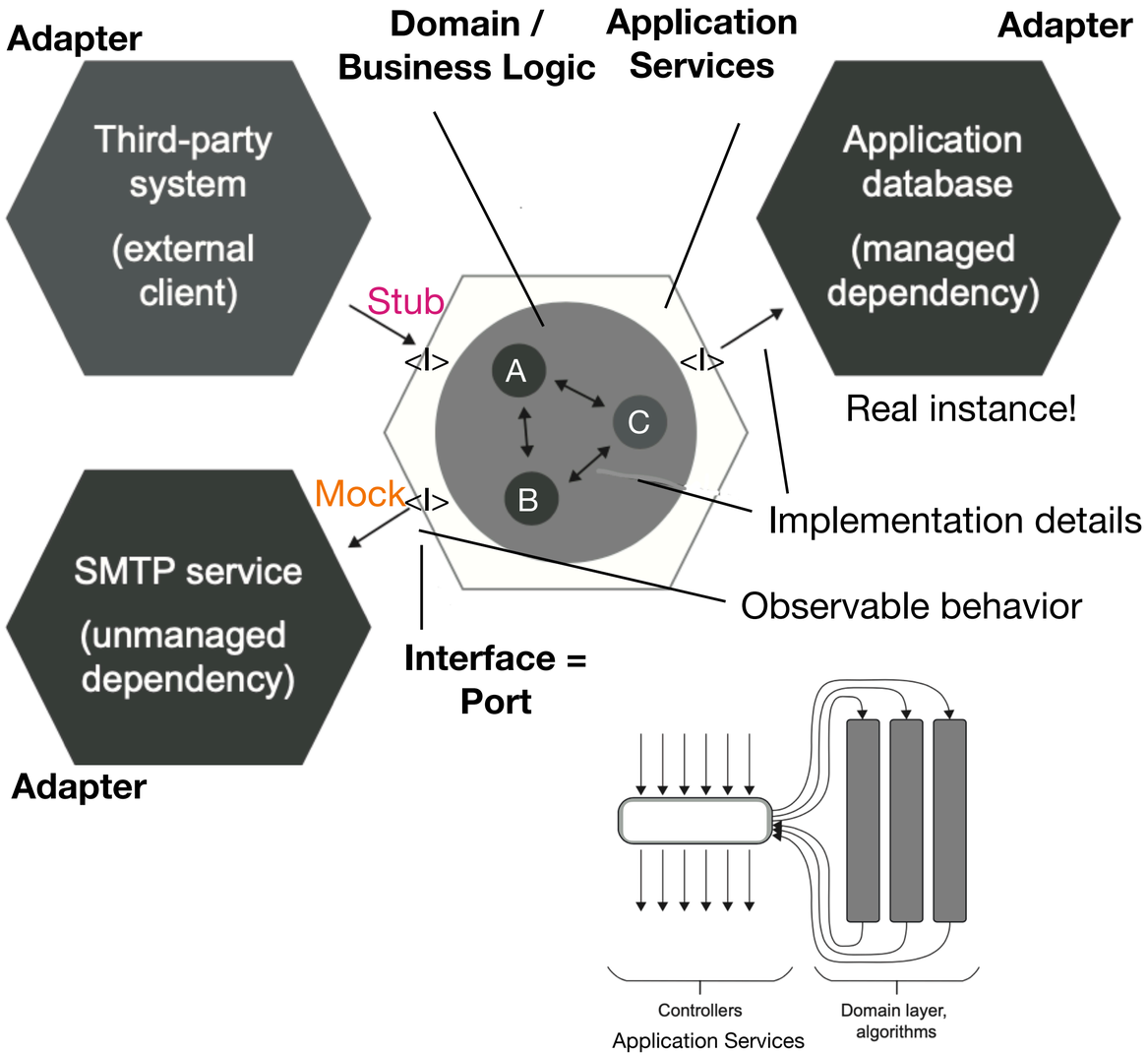
$$Test\ Accuracy = \frac{number\ of\ bugs\ found}{number\ of\ false\ alarms\ raised}$$

All production code can be categorised into:

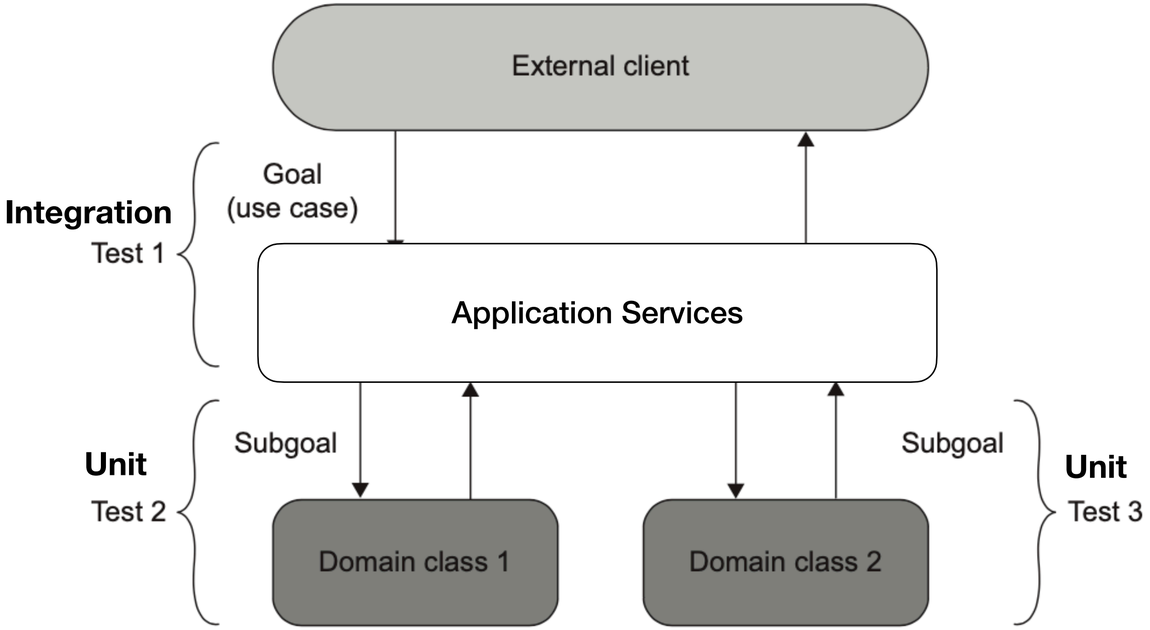
	Observable behaviour	Implementation detail
Public	Good API	Bad API
Private	-	Good API

Hexagonal Architectur, Clean Architecture, Ports and Adapters

Production Code



Testing the Production Code



Conclusion and Summary

Why I favour the classical over the mockist school of unit testing

	Isolation of	A unit is	Uses test doubles for
Mockist school	Units	A class	All but immutable dependencies
Classical school	Unit tests	A unit of behaviour (a class or a set of classes)	Shared dependencies

Mockist School “Advantages”:

* Better granularity with a clear rule: one class, one test
* If a test fails, you know for sure which functionality has failed

* Easier unit testing a larger graph of interconnected classes

Classical School “Counter-Arguments”:

=> Test shouldn't test units of code, but rather **units of behavior**

=> Running your tests regularly, you know what caused the bug (look at the **last change**)

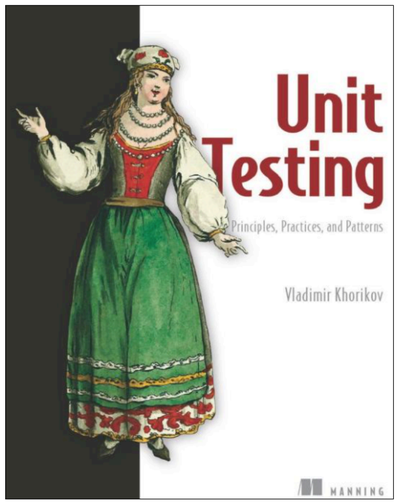
=> Focus on **not having such a graph of classes** in the first place! Good thing that tests point out this problem!

Most important points to take away

- $$Test\ Accuracy = \frac{number\ of\ bugs\ found}{number\ of\ false\ alarms\ raised}$$
- Unit tests of **classical school** are less likely to **break when refactoring…**
- **…**which decreases the denominator in the formula above, therefore **increasing test accuracy!**
- **Avoid mocks** as much as possible!
- Avoid testing implementation details; **test observable behavior!**

- Not being able to easily unit test your code **predicts poor code design** with a high precision
- If you need **more than one operation** to invoke observable behavior, you likely have leaking implementation details

- Further reading: **Unit testing Khorikov (2020)**



I took several concepts, code examples, and figures from this book for this presentation.

Thank you for listening!

Test doubles

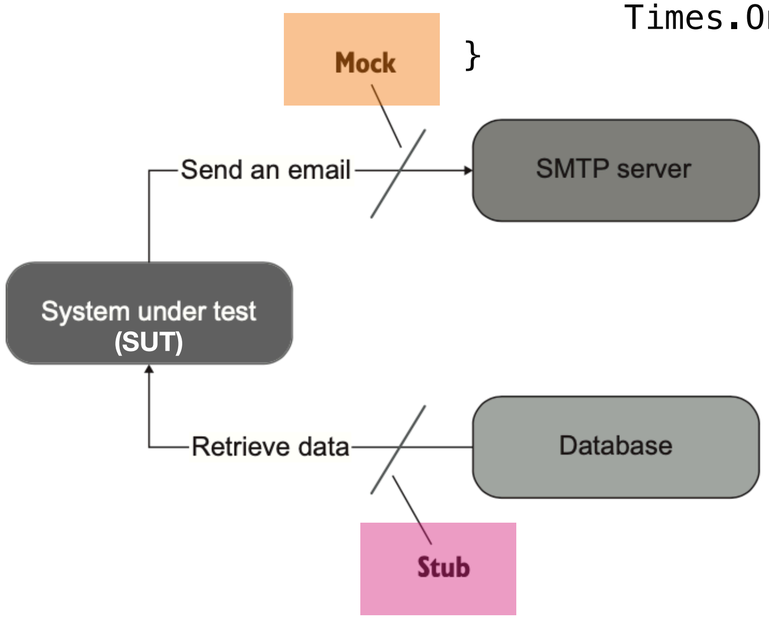
- “Test double”[1] is the overarching term for:
  1. Mock (mock, spy)
  2. Stub (stub, dummy, fake)

- Mock (the tool) vs. mock (the test double). Therefore a Mock tool can create a stub.

- Command query segregation principle:
  Mocks ≈ Commands
  Stubs ≈ Queries

[1]Compare our Mock tool in the monolith. Cited from their docs: “Mockery’s main goal is to help us create **test doubles**. It can create **stubs, mocks**, and spies.”

```
public void Sending_a_greetings_email()
{
    var emailGatewayMock = new Mock<IEmailGateway>();
    var sut = new Controller(emailGatewayMock.Object);

    sut.GreetUser("user@email.com");

    emailGatewayMock.Verify(
        x => x.SendGreetingsEmail("user@email.com"),
        Times.Once);
}
```



A mock helps to *emulate* and *examine* **outcoming** interactions.
These interactions are calls the SUT makes to its dependencies to change their state.

A stub helps to *emulate* **incoming** interactions.
These interactions are calls the SUT makes to its dependencies to get input data.

```
public void Creating_a_report()
{
    var stub = new Mock<IDatabase>();
    stub.Setup(x => x.GetNumberOfUsers()).Returns(10);
    var sut = new Controller(stub.Object);

    Report report = sut.CreateReport();

    Assert.Equal(10, report.NumberOfUsers);
}
```