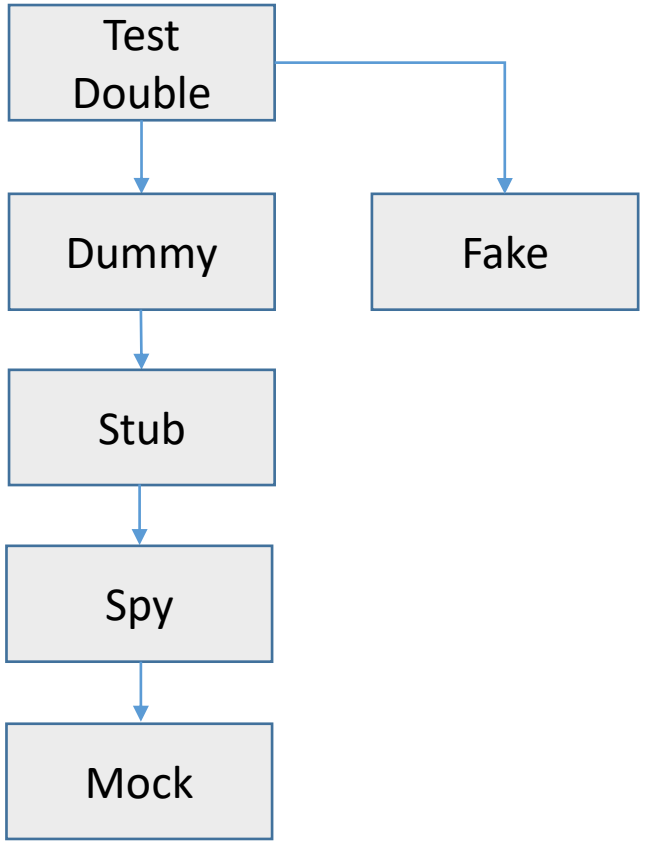
Unit Testing in C#

Schrijven van Test Doubles

Inhoud

1. Wat zijn test doubles en waarvoor dienen ze
2. Soorten Test doubles: **fakes, dummies, stubs, spies en mocks**
3. Manueel schrijven van test doubles versus mocking frameworks gebruiken
4. Mocking frameworks: beperkte en onbeperkte – voor- en nadelen
5. Welke manier van aanpak/benadering bij het schrijven van tests

Test Doubles

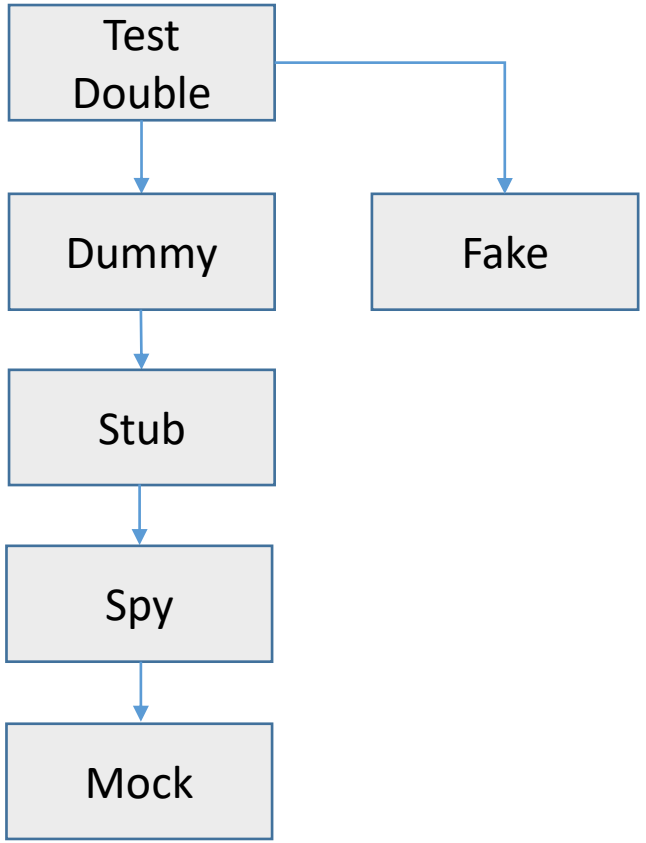


Wat is een Dummy?

Dummy is een test double (methode) die **null, niets of 0 teruggeeft**.

Dummies **hebben eigenlijk geen code en doen niets en dienen enkel ter vervanging van methoden waarvan de te testen code afhankelijk is**.

Test Doubles



Wat is een Stub?

Een **stub** is een **vervanging van een methode**.

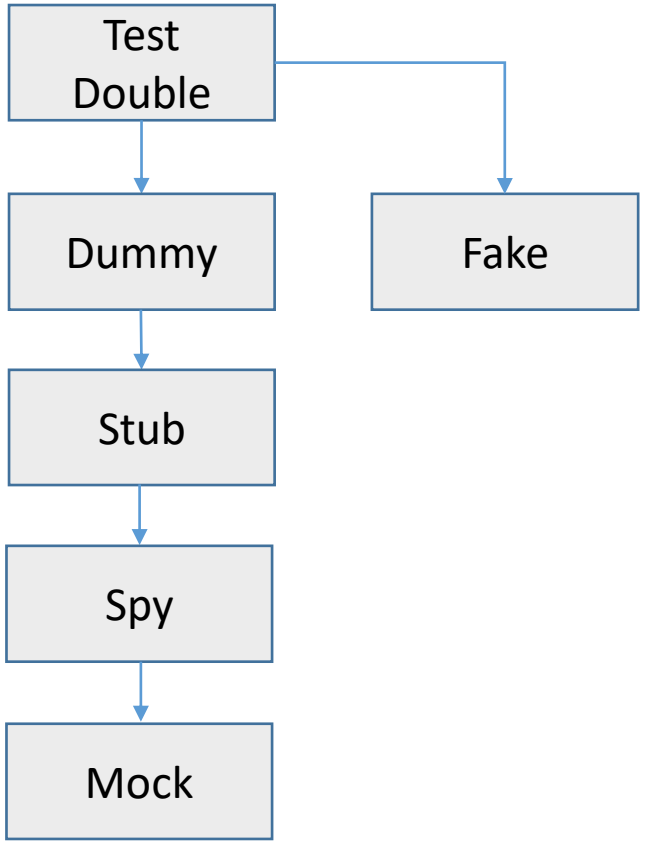
De stub **heeft dan een implementatie, maar die sterk afwijkt van de (echte) implementatie.**

Hierdoor kan een **module(-onder-test) geïsoleerd worden (getest) van andere modules.**

Stub is een dummy die wordt ingesteld **om (een) voorgedefinieerde constante waarde(n) terug te geven** vanuit de methoden van het object die wordt vervangen door de stub.

Gebruik stubs om **de uitvoering volgens bepaalde paden te laten verlopen**.

Test Doubles



Wat is een Spy?

Spy is een stub die **bepaalde informatie bijhoudt** over de aanroepen **die op de spy gebeuren**.

Het verifieert **welke argument-waarden** er worden doorgegeven aan de methoden, **hoeveel keer** methoden worden aanroepen en **in welke volgorde** de methoden worden aangeroepen.

Wat is een Mock?

Een **mockobject** is een (software)object speciaal gemaakt om **de eigenschappen en gedragingen te simuleren van een of meerdere objecten**

Mock is dus een ‘spy’, maar een mock bevat zelf assertions, dus een **mock kan een test doen falen.**

Interactie Testen (Interaction Testing)

‘Interaction testing’ is testen **hoe** een object boodschappen stuurt **(methoden aanroept)** naar andere objecten.

Gebruik interactie testen wanneer het aanroepen van een ander object het resultaat is van een specifieke werkeenheid (unit of work).

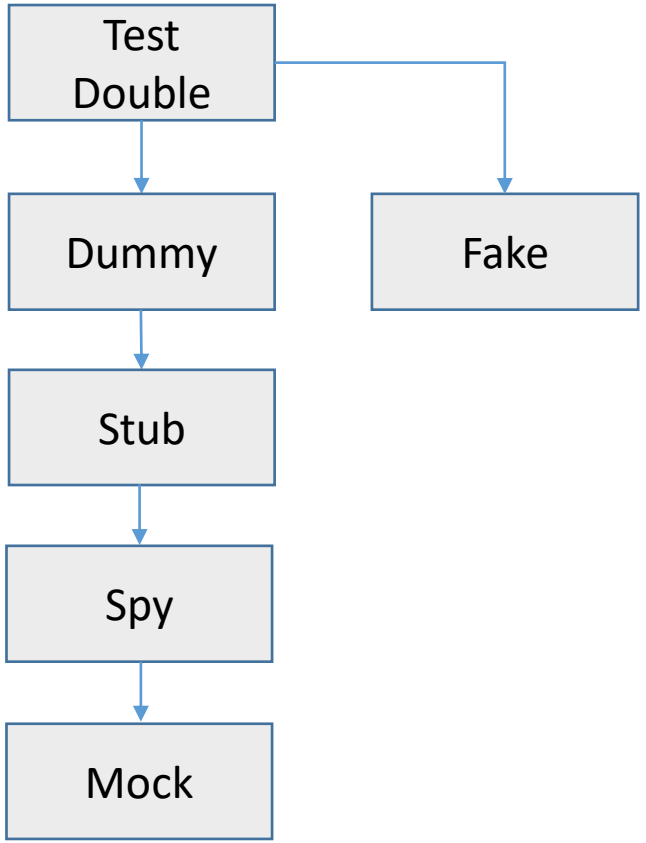
Voorbeeld van een Spy en mock

```
 public interface ILoggingService{
    void LogInformation(string message, params object[] parameters);
}
[TestFixture]
public class CustomerServiceTests
{
    private readonly CustomerService _sut;
    private readonly Mock<ICustomerRepository> _customerRepoMock = new Mock<ICustomerRepository>();
    private readonly Mock<ILoggingService> _loggerSpy = new Mock<ILoggingService>();
    public CustomerServiceTests()   {
        _sut = new CustomerService(_customerRepoMock.Object,_loggerMock.Object);
    }

    [Test]
    public async Task GetByIdAsync_ShouldLogAppropriateMessage_WhenCustomerExists()
    {
        // Arrange
        var customerId = Guid.NewGuid();
        var customerName = "Jos De Klos";
        var customerDto = new CustomerDto { Id = customerId.ToString(), FullName = customerName };
        // _customerRepository.GetByIdAsync(customerId).Returns(customerDto);
        _customerRepoMock.Setup(x => x.GetByIdAsync(customerId)).ReturnsAsync(customerDto);
        // Act
        await _sut.GetByIdAsync(customerId);

        // Assert
        _loggerSpy.Verify(x => x.LogInformation("Retrieved a customer with Id: {Id}", customerId),
               Times.Once);
        _loggerSpy.Verify(x => x.LogInformation("Unable to find a customer with Id: {Id}",
              customerId),Times.Never);
    }
```

Test Doubles



Wat is een ‘Fake’?

Fake is een ‘test double’ die het gedrag zo realistisch mogelijk simuleert van de echte dependency van de code die moet worden getest.

Voorbeeld van een ‘Fake’?

```
namespace Business.Tests.TestDoubles
{
    public class DbGatewayFake:IDbGateway
    {
        private Dictionary<int, WorkingStatistics> _storage =
                new Dictionary<int, WorkingStatistics>()
        {
            {1, new WorkingStatistics() {PayHourly = true, WorkingHours = 5, HourSalary = 10}},
            {2, new WorkingStatistics() {PayHourly = false, MonthSalary = 500}},
            {3, new WorkingStatistics() {PayHourly = true, WorkingHours = 8, HourSalary = 100}},
        };
        public WorkingStatistics GetWorkingStatistics(int id)
        {
            return _storage[id];
        }

        public bool Connected { get; }
    }
}
```

Nadelen van manuele Test Doubles

1. Het **manueel** schrijven van Test Doubles **vergt veel tijd, vooral voor interfaces met veel members**
2. **Veel dezelfde terugkerende code moet worden geschreven**
3. Het is **moeilijk om manueel geschreven code voor test doubles te hergebruiken voor andere tests**

Mocking Frameworks

- **Mocking** frameworks zijn onafhankelijk van unit testing frameworks
- Er bestaan 2 typen van mocking frameworks: **Beperkte (Constrained) en onbeperkte (Unconstrained)**
- **Voorbeeld Moq, Mockito en NSubstitute zijn mocking frameworks en zij kunnen beide worden gebruikt met unit testing frameworks zoals Nunit, xUnit,…**

Constrained Frameworks

- Hebben **beperkte mogelijkheden**: kunnen o.a**. geen mocking geven voor private en static methoden.**
- **Genereert code ‘at runtime’ die afgeleid zijn** van basisclassen of interfaces implementeren
- **Je kan ‘test doubles’ enkel creëren voor de code waarvoor je manueel test doubles kan schrijven**

Constrained Frameworks

1. NSubstitute
2. Moq
3. RhinoMock
4. Mockito

Onbeperkte (Unconstrained) Frameworks

- Geven **bijna alle mogelijkheden. Je kan een test double maken voor zowat alles: ook private en static methoden**,…
- Voorbeelden van unconstrained testing **frameworks**: **Typemock Isolator, JustMock, and Moles (a.k.a. MS Fakes)**

Nadelen van onbeperkte (Unconstrained) Frameworks

- **Goede unit tests kunnen enkel public members** van een object aanroepen.
  Tests waarvoor je **test doubles voor private methoden zijn zeer bedenkelijk.**
- Meestal is er **een probleem met het ontwerp van de software indien je niet via de publieke members kan testen van een object**
- **Refactor** eerder de productie code om unit testing mogelijk te maken.
- unconstrained frameworks kunnen (op lange termijn) **leiden tot moeilijk verstaanbare en onderhoudbare tests.**

Gebruik geen onbeperkte (unconstrained) mocking frameworks, tenzij je geen andere keuze hebt.

De verschillende mocking frameworks zijn vrij gelijkaardig.

Eens je ervaring hebt met een framework, is het gemakkelijk om testen in een ander framework te schrijven.

Detroit (Klassieke aanpak)

De status (waarden) worden voornamelijk geverifieerd na de act-fase.

Dit gebeurt door de **terugkeerwaarden** (**return values**) of bv Property-waarden te **verifiëren op juistheid** na het aanroepen van het te testen object.

Voorbeeld:

```
//act

var result = testee.DoStuff("param");

//assert

Assert.That(result, Is.EqualTo("expected"));
```

London (mockisten)

De **focus ligt in het testen van interacties tussen objecten en verifieert** of:

- De **interactie** van het test-object (testtee) **met zijn afhankelijke objecten**(mock-collaboratoren) op de juiste manier gebeurt:

    - Aanroepen van de **juiste methoden,**
    - In de **juiste volgorde**,
    - Met de **juiste parameters**,
    - En het **juist aantal keren**

Voorbeeld :

```
testee.DoStuff();
test:
_mockCollaborator.Verify(x =>
                    GetDataForStuff("param"),
                        Times.Once);
```

Pragmatische aanpak: probeer het midden te houden tussen klassieke en mockist – aanpak om zoveel mogelijk voordelen uit testen te halen.

Samenvatting

1. Maak gebruik van **test doubles om afhankelijkheden te verwijderen van het object dat je wil testen.**
2. Er bestaan **verschillende soorten test doubles**, zoals fakes, dummies, stubs, spies en mocks
3. Er zijn verschillende **mocking frameworks** (bv Mock, Mockito, Nsubstitute,…)
4. Er is een verschil in aanpak: ‘klassiek’ en ‘mockist’.

   Klassieke aanpak: Zo mogelijk vermijden van het gebruik van mocking

   Mockist aanpak: de focus ligt op het testen van interacties tussen het test-object en zijn afhankelijke objecten

   Mocks en interfaces vormen de basis voor het vinden van een system.

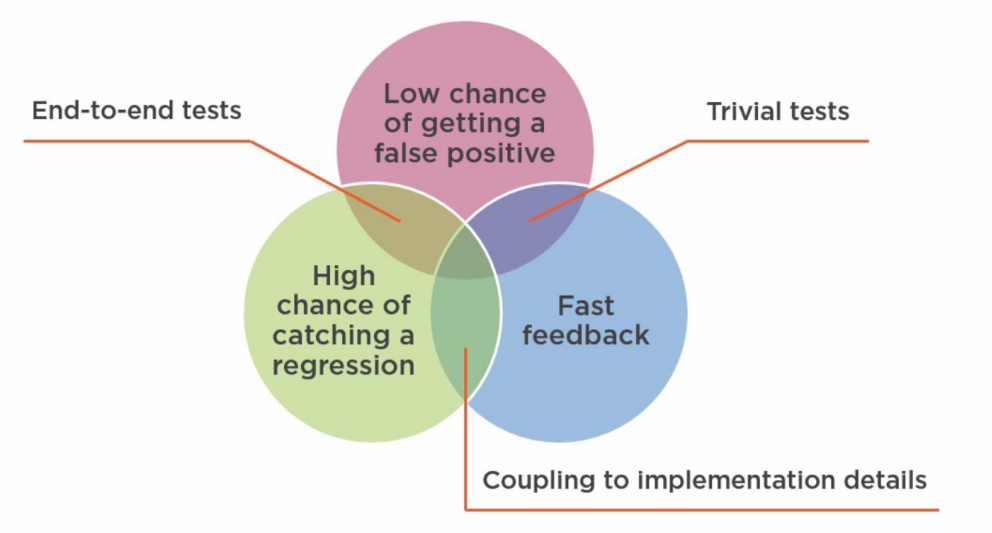
Test-Driven Development

Beste praktijken bij Unit Testing

Garanties van Unit Tests

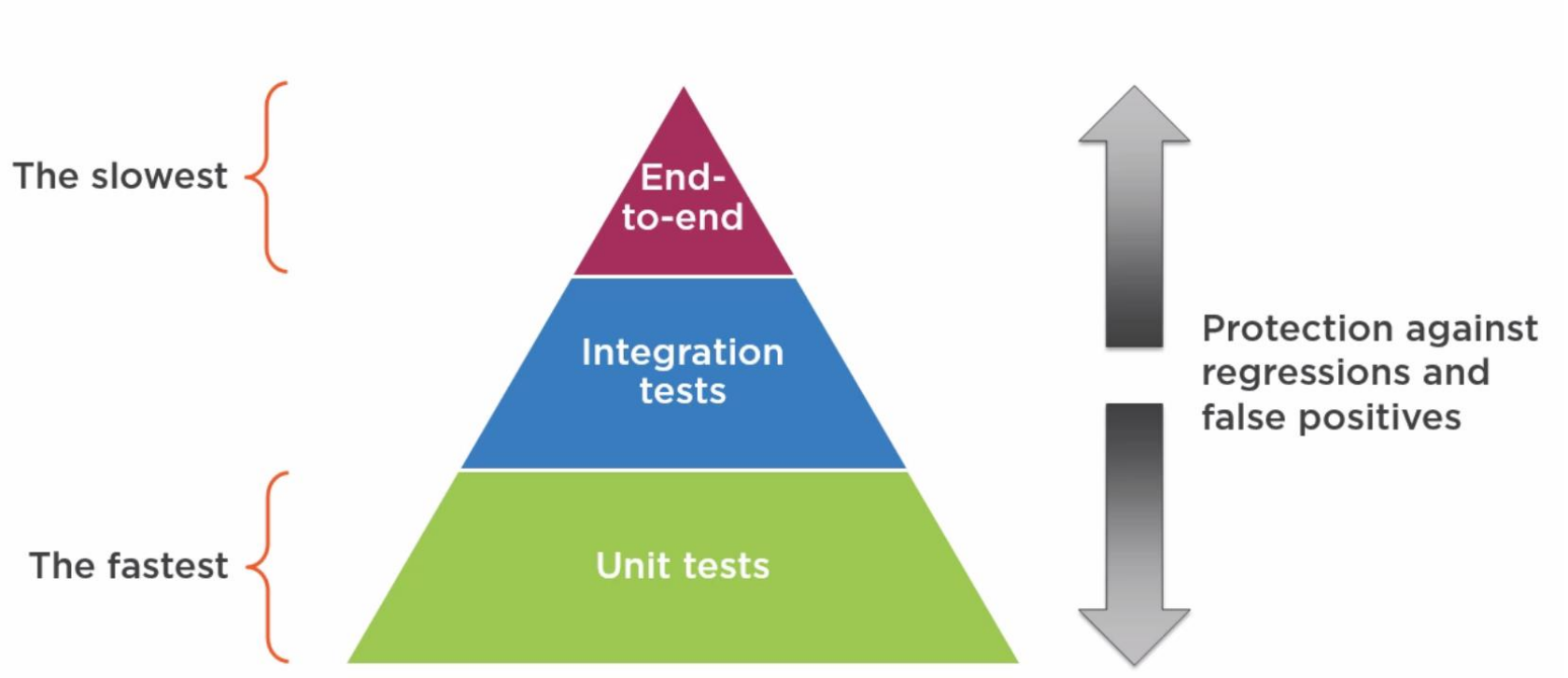
- Unit tests op zichzelf garanderen helemaal niets
- Unit tests vereisen evenveel aandacht (en dus werk) als de productiecode zelf.
- Schrijven van unit tests is werk dat studie en aandacht vergt.

Wat is een geldige en waardevolle Test ?



1. **Een test met hoge kans om een fout te vinden**
2. **Een test met Lage kans op een ‘vals positief’ resultaat**
3. **Een test die snelle feedback geeft**
4. **Een test met lage onderhoudskost**

Wat maakt een Test waardevol?



Wanneer Mocks (niet) schrijven?

- **Interacties testen is veel moeilijker**
- Maak **geen Unit tests om interacties tussen domein-objecten te testen**. Deze testen zouden **veel 'vals positieve' resultaten** geven en deze testen zijn **moeilijk te onderhouden, aangezien hier veel mocks voor moeten worden geschreven**.
- **Kies steeds interactie testen als laatste optie.**
  Probeer zoveel mogelijk **tests te schrijven die return waarden van methoden of de waarden van properties van objecten te verifiëren.**

Triviale Code: niet testen

- **Schrijf geen unit tests voor getters en setters**. Deze worden indirect getest d.m.v. de unit tests van andere methoden.
- **Schrijf geen unit tests voor triviale code of one-line methoden/functies**. Deze worden indirect getest d.m.v. de unit tests van andere methoden.

Test ‘Single Concern’

‘One-Assert-Per-Test’ principe:

Test enkel één aspect of gedrag per Unit test

Kenmerken van een goede Unit Test

- Betrouwbaarheid
- Onderhoudbaarheid
- Leesbaarheid

Wat te vermijden

- Vermijd control flow operatoren (controlestructuren) in unit testen
- Vermijd Duplicaties
- Zet geen Test Doubles op in de [SetUp] methode
- Vermijd tests die in bepaalde volgorde moeten worden uitgevoerd
  - creëert tijdelijke koppeling tussen unit tests
  - Je moet bij elke toevoeging van een unit test nadenken in welke volgorde welke test zou moeten worden uitgevoerd.
  - Testen die in een bepaalde volgorde moeten worden uitgevoerd, runnen trager
  - Testen die in een bepaalde volgorde moeten worden uitgevoerd zijn moeilijk te onderhouden

Streefdoel bij unit test

Onafhankelijkheid en isolatie

Referenties

https://docs.microsoft.com/en-us/visualstudio/test/unit-test-basics?view=vs-2019

https://www.youtube.com/watch?v=hsfVPPYoc9o

https://jakeydocs.readthedocs.io/en/latest/mvc/controllers/testing.html

https://raaaimund.github.io/tech/2019/05/07/aspnet-core-unit-testing-moq/

https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-nunit

https://wakeupandcode.com/unit-testing-in-asp-net-core/

https://www.starringjane.com/blog/tdd-test-driven-development/

https://nl.wikipedia.org/wiki/Testtool

https://nl.wikipedia.org/wiki/Stub

https://nl.wikipedia.org/wiki/Mockobject

https://nl.wikipedia.org/wiki/Testen_(software)

https://en.wikipedia.org/wiki/Test_double

https://piazza.com/class_profile/get_resource/j11t8bsxngk3r3/j2lw6zcyt5t6lu