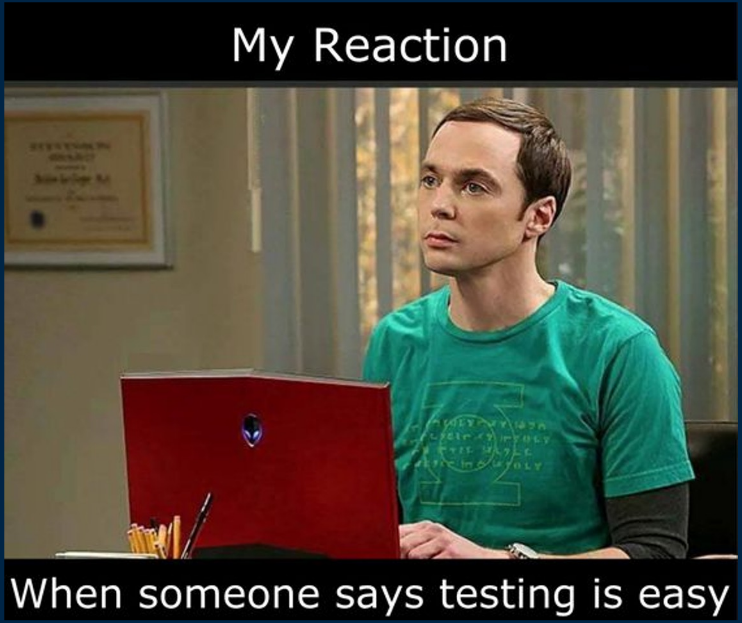
Introduction to Unit Tests



Agenda

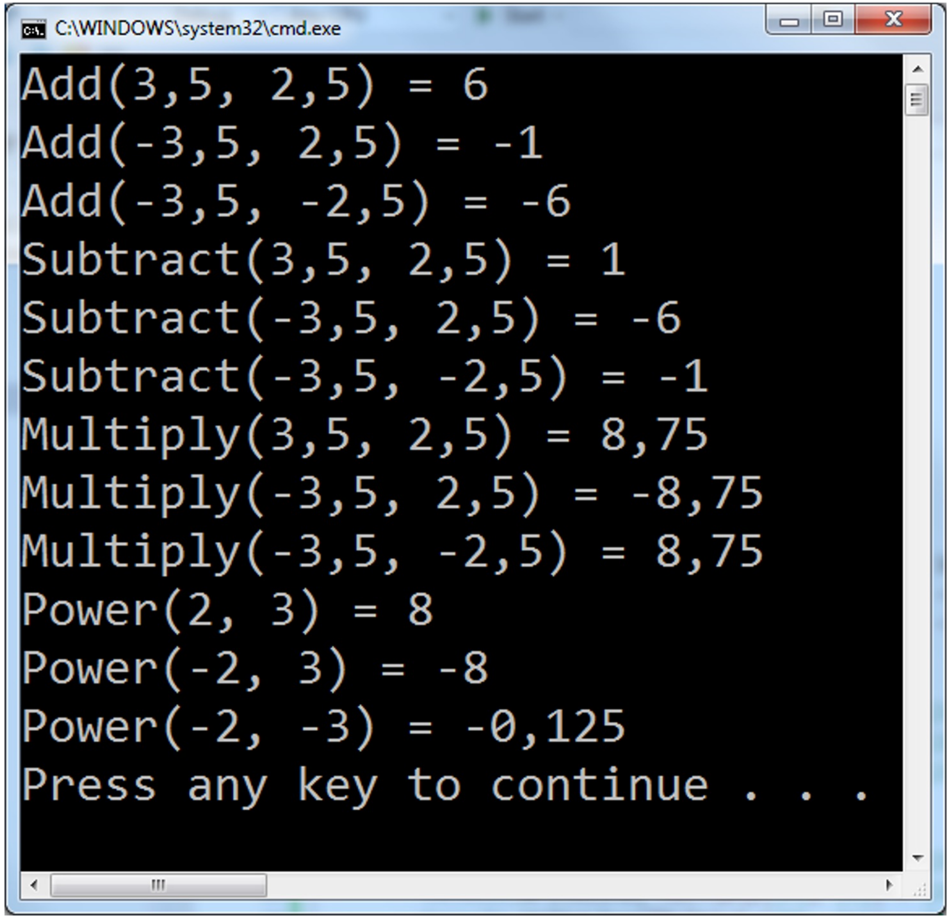
- Motivation
- Case study: NUnit
- Testing
    - Basic parts of a unit tests
    - What does a unit test look like?

Motivation: Manually test

```
public class Calculator {
    public double Add(double a, double b) {
        return a + b;
    }

    public double Subtract(double a, double b) {
        return a - b;
    }

    public double Multiply(double a, double b) {
        return a * b;
    }

    public double Power(double a, double b) {
        return Math.Pow(a,b);
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        // Declare the unit-under-test
        var uut = new Calculator();

        // Test Add()
        Console.WriteLine("Add({0}, {1}) = {2}", 3.5, 2.5, uut.Add(3.5, 2.5));
        Console.WriteLine("Add({0}, {1}) = {2}", -3.5, 2.5, uut.Add(-3.5, 2.5));
        Console.WriteLine("Add({0}, {1}) = {2}", -3.5, -2.5, uut.Add(-3.5, -2.5));

        // Test Subtract()
        Console.WriteLine("Subtract({0}, {1}) = {2}", 3.5, 2.5, uut.Subtract(3.5, 2.5));
        Console.WriteLine("Subtract({0}, {1}) = {2}", -3.5, 2.5, uut.Subtract(-3.5, 2.5));
        Console.WriteLine("Subtract({0}, {1}) = {2}", -3.5, -2.5, uut.Subtract(-3.5, -2.5));

        // Test Multiply()
        Console.WriteLine("Multiply({0}, {1}) = {2}", 3.5, 2.5, uut.Multiply(3.5, 2.5));
        Console.WriteLine("Multiply({0}, {1}) = {2}", -3.5, 2.5, uut.Multiply(-3.5, 2.5));
        Console.WriteLine("Multiply({0}, {1}) = {2}", -3.5, -2.5, uut.Multiply(-3.5, -2.5));

        // Test Power()
        Console.WriteLine("Power({0}, {1}) = {2}", 2.0, 3.0, uut.Power(2.0, 3.0));
        Console.WriteLine("Power({0}, {1}) = {2}", -2.0, 3.0, uut.Power(-2.0, 3.0));
        Console.WriteLine("Power({0}, {1}) = {2}", -2.0, -3.0, uut.Power(-2.0, -3.0));
    }
}
```

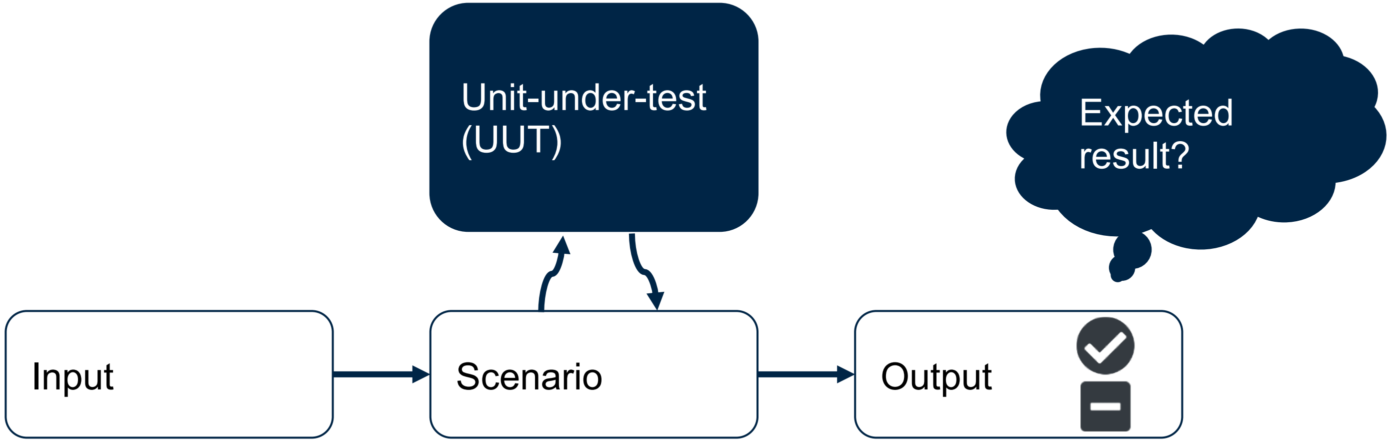
Validate result?



```
Add(3,5, 2,5) = 6
Add(-3,5, 2,5) = -1
Add(-3,5, -2,5) = -6
Subtract(3,5, 2,5) = 1
Subtract(-3,5, 2,5) = -6
Subtract(-3,5, -2,5) = -1
Multiply(3,5, 2,5) = 8,75
Multiply(-3,5, 2,5) = -8,75
Multiply(-3,5, -2,5) = 8,75
Power(2, 3) = 8
Power(-2, 3) = -8
Power(-2, -3) = -0,125
Press any key to continue . . .
```

Aktuel status
Diagnoser
Planlægning
Resultater
Medicin
Rekvisition / Svar
Korrespondance
Personoplysninger
Patientadministration
Resultater
Målinger
Læs journal:
Kontaktårsag
Tidligere
Aktuelt
Øvrige organsystemer
Objektivt
Start
lenovo

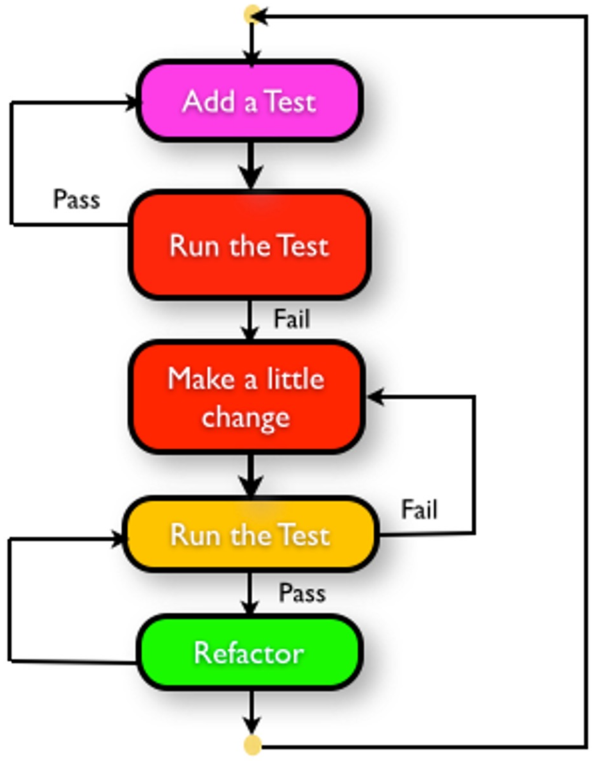
A unit tests



Icons: FrICONiX.com

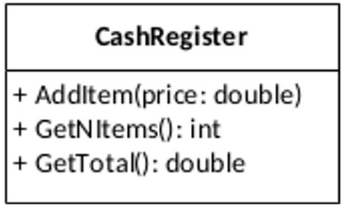
How to plan and execute test

1. Define a scenario
2. Write the `[TestCase]`
3. Run the test
4. Implement the code
5. Repeat



Demo: A cash register

- We will implement and test a class CashRegister



- Your turn: What test cases do we need for each of the class' methods?
  - What is the scenario?
  - What is the test input?
  - What is the expected result?

Test also gives...

- specifications built into the program
- confidence in code
- early error finding
- decoupled system
- better design

Your turn

For around 30 minutes do

1. Do '**Exercise 1: Plan your tests**'
2. Continue to **'Exercise 2: Prepare workspace**' when you are done with **1.**

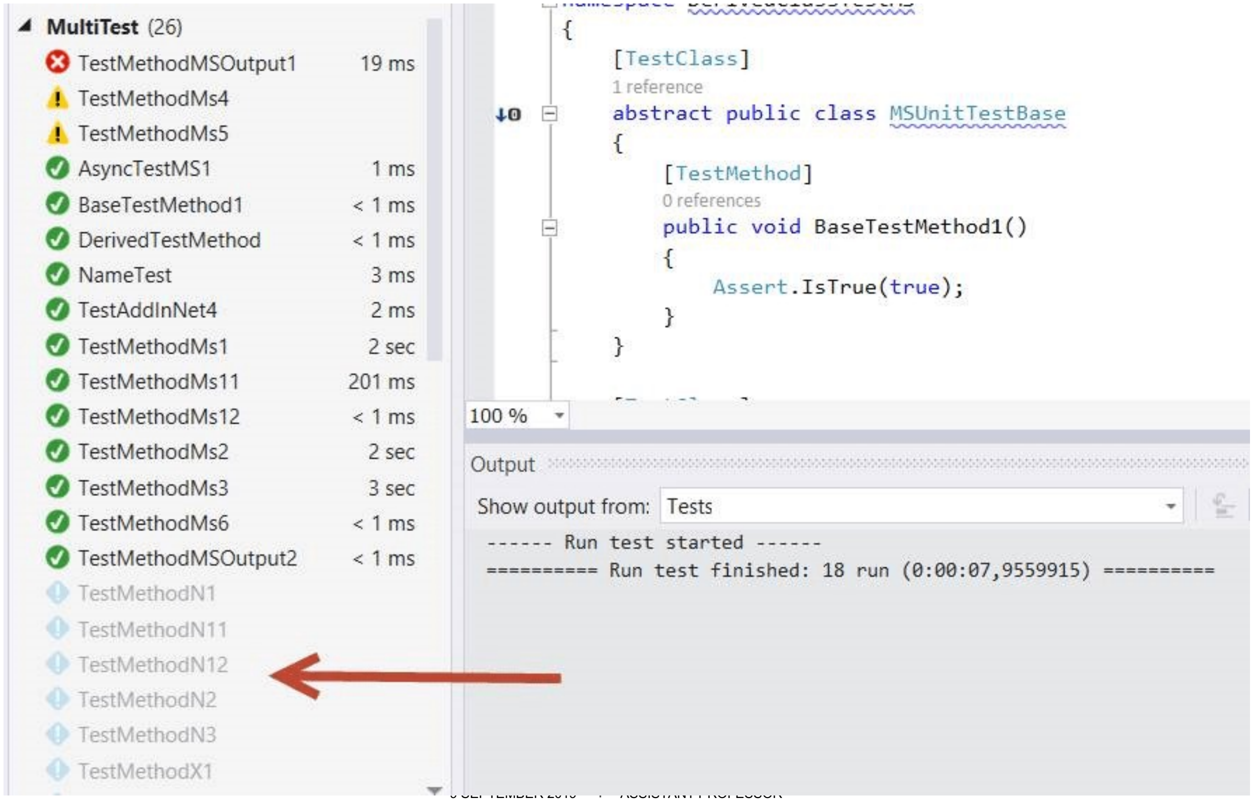
NUnit

Test framework

Gives:

- Support for automation
- Easy setup and removal
- Good assertion constructs
- Detailed test reports
- Nice IDE integration
- Testing styles

Automation / test report



```
◢ MultiTest (26)
  TestMethodMSOutput1    19 ms
  TestMethodMs4
  TestMethodMs5
  AsyncTestMS1            1 ms
  BaseTestMethod1       < 1 ms
  DerivedTestMethod     < 1 ms
  NameTest                3 ms
  TestAddInNet4           2 ms
  TestMethodMs1           2 sec
  TestMethodMs11        201 ms
  TestMethodMs12        < 1 ms
  TestMethodMs2           2 sec
  TestMethodMs3           3 sec
  TestMethodMs6         < 1 ms
  TestMethodMSOutput2   < 1 ms
  TestMethodN1
  TestMethodN11
  TestMethodN12
  TestMethodN2
  TestMethodN3
  TestMethodX1
```

```
{
    [TestClass]
    1 reference
    abstract public class MSUnitTestBase
    {
        [TestMethod]
        0 references
        public void BaseTestMethod1()
        {
            Assert.IsTrue(true);
        }
    }
```

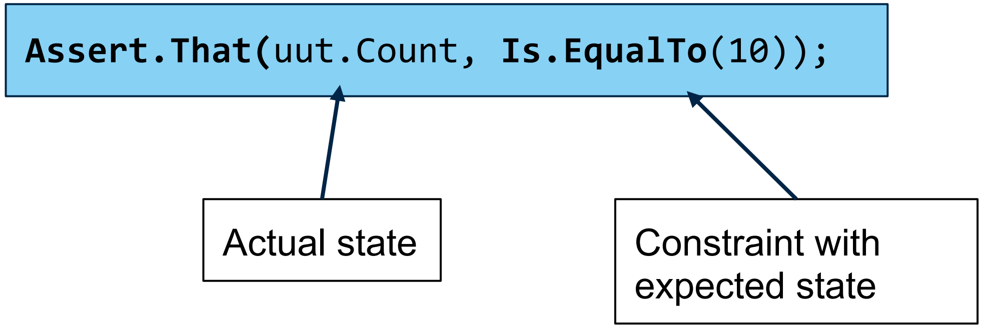
```
------ Run test started ------
========== Run test finished: 18 run (0:00:07,9559915) ==========
```

Nunit example

```
[TestFixture]
public class UnitTest1 {
    Calculator uut;
    [Setup]
    public void Setup() {
            uut = new Calculator();
    }

    [Test]
    public void Test_AddMethod() {
            double res = uut.Add(3.5, 2.5);
            Assert.AreEqual(res, 6);
    }
    …
}
```

NUnit assertion - constraints

- Constraint based assert model



```
Assert.That(uut.Count, Is.EqualTo(10));
```

- Other constraints examples - many more exists

```
Assert.That(uut.Count, Is.GreaterThan(3));
Assert.That(myString, Is.EqualTo("Hello"));
Assert.That(array, Has.Exactly(3).LessThan(100));
```

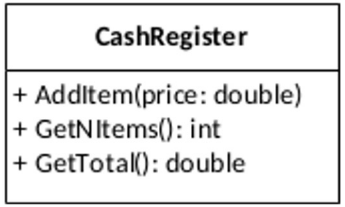
More constraints https://nunit.org/docs/2.5/constraintModel.html

Test framework

- **Assertions** Different ways to compare expected and actual test results in a readable way
- **Test case** Each test case - tests one specific, isolated aspect of the unit-under-test
- **Test fixture** Collects test cases, helps with setup, teardown, etc.
- **Test runner** Runs the tests and reports the result
- **Test reports** Result of the tests run

Demo: A cash register

- Let us implement the CashRegister

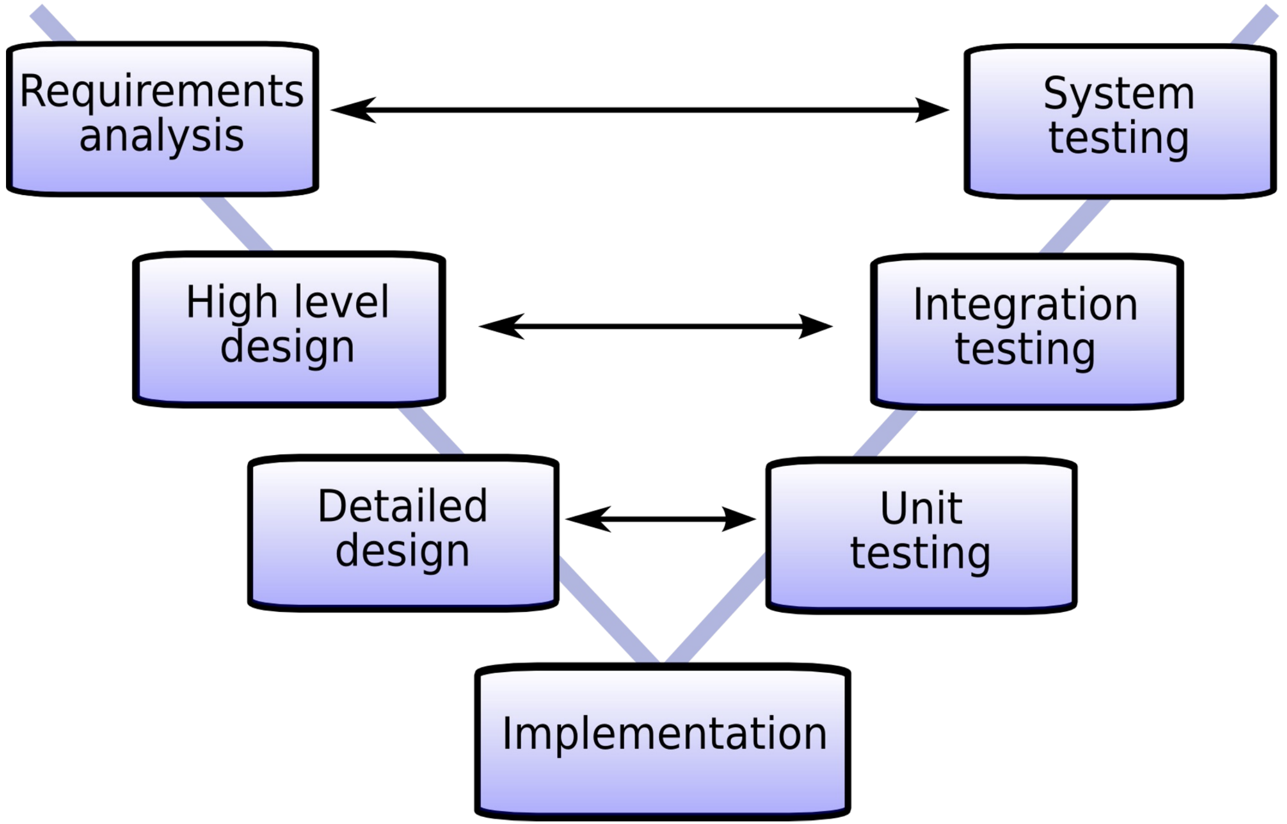


Your turn

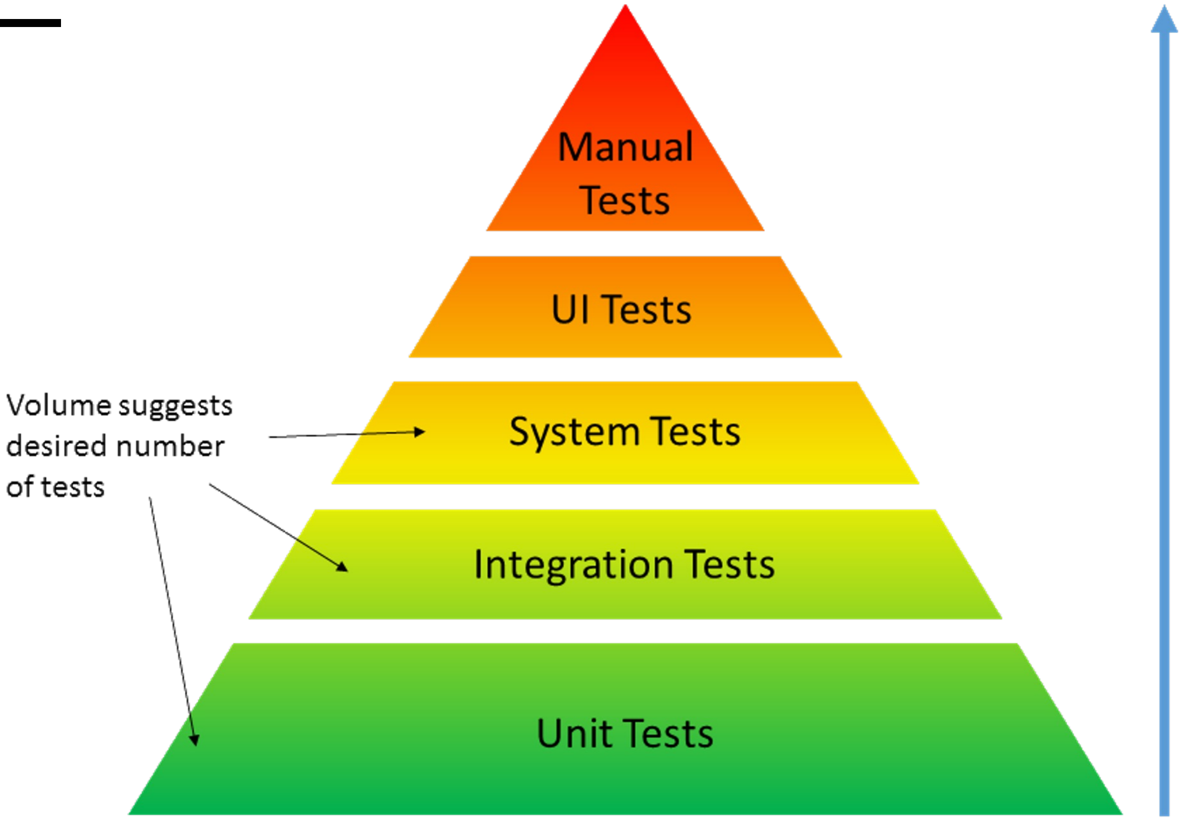
Continue with exercises

Testing

and Pitfalls


Requirements analysis
System testing
High level design
Integration testing
Detailed design
Unit testing
Implementation

Why test? Why unit test?

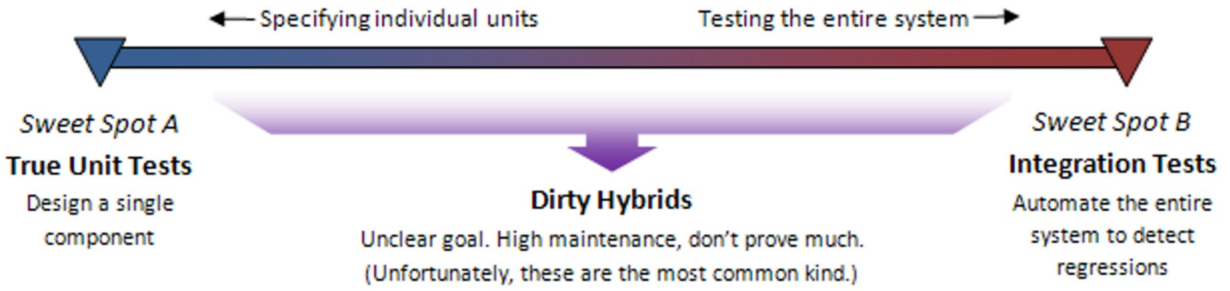


Rising with the pyramid:

- Complexity
- Fragility
- Cost of maintenance
- Execution time
- Time to locate bug on test failure

Pitfalls

- True unit test contains information about design and behaviour of UUT (Unit-under-test)
  - **Do not** make any assumption about other parts
- Integration test **do not** tell anything about how code base is broken down into units
  - Make assumptions about the whole system behaviour
- In between
  - Small changes breaks unrelated test
  - Tests breaks - but system works as “expected”


Specifying individual units
Testing the entire system
Sweet Spot A
True Unit Tests
Design a single component
Dirty Hybrids
Unclear goal. High maintenance, don't prove much.
(Unfortunately, these are the most common kind.)
Sweet Spot B
Integration Tests
Automate the entire system to detect regressions

SOLIDUM PETIT IN PROFUNDIS · UNIVERSITAS ARHUSIENSIS ·

Reference

TDD: agilefaqs.com/services/training/test-driven-development

Dancing man: https://giphy.com