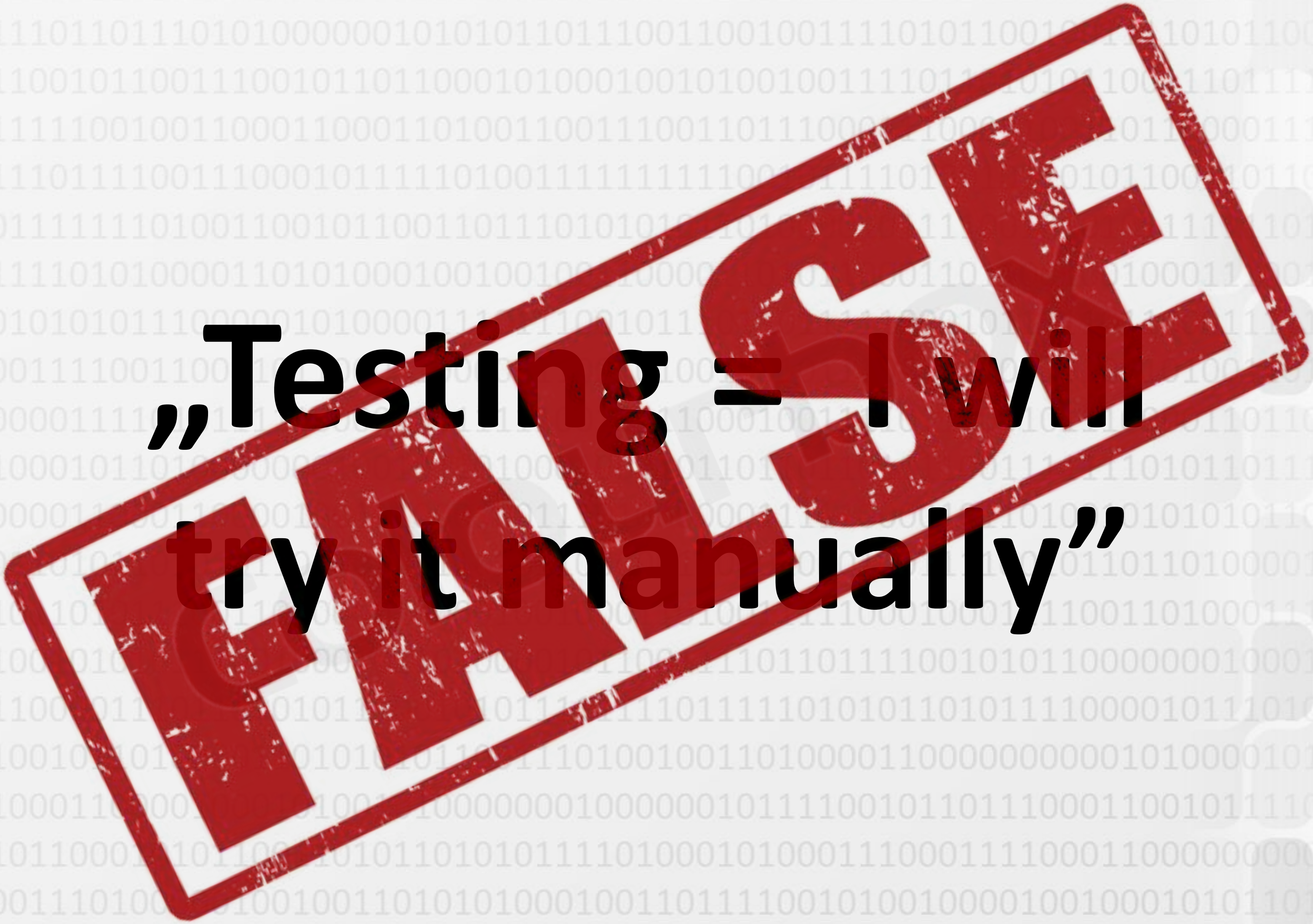
# Advanced development techniques

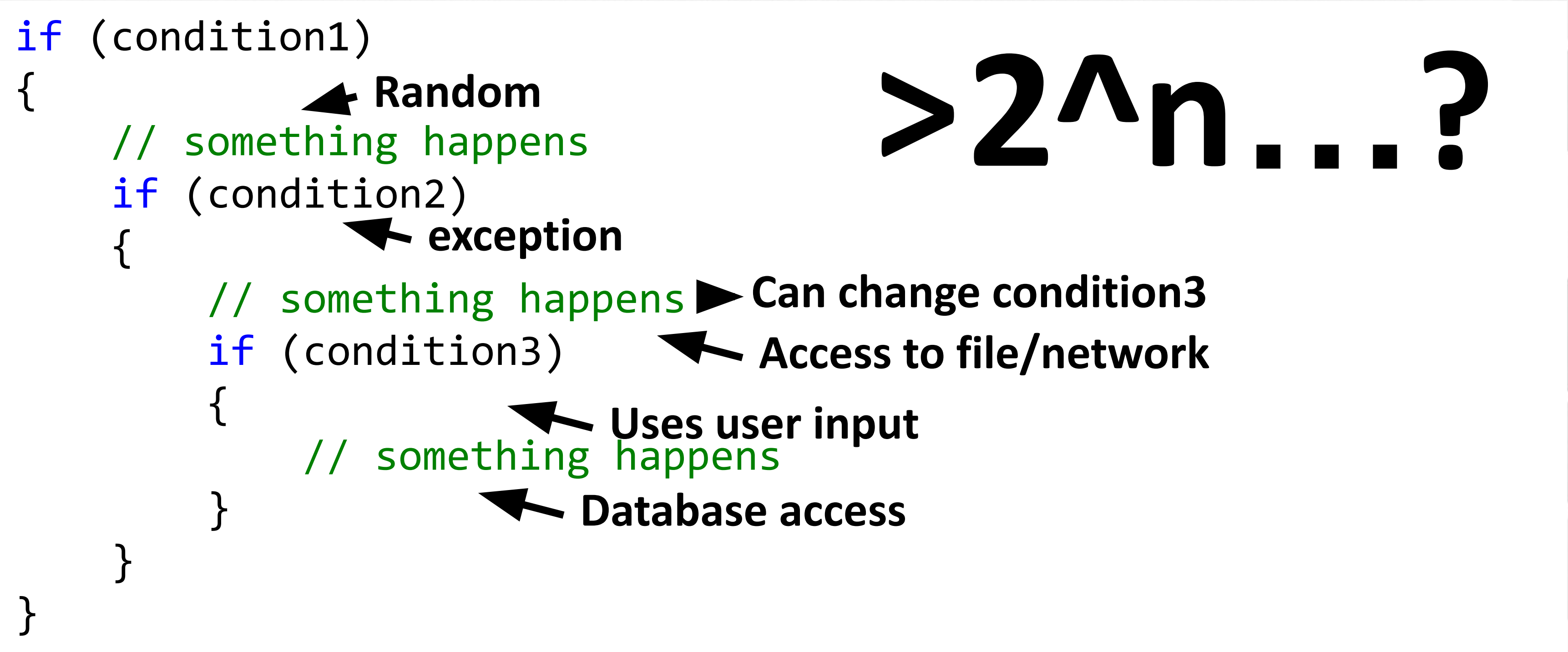
Simple Unit Tests
NUnit

# „Testing = I will try it manually”



# Automatic tests are required

- **Number of test cases?**



```
if (condition1)
{
    // something happens
    if (condition2)
    {
        // something happens
        if (condition3)
        {
            // something happens
        }
    }
}
```

**Random**

**exception**

**Can change condition3**

**Access to file/network**

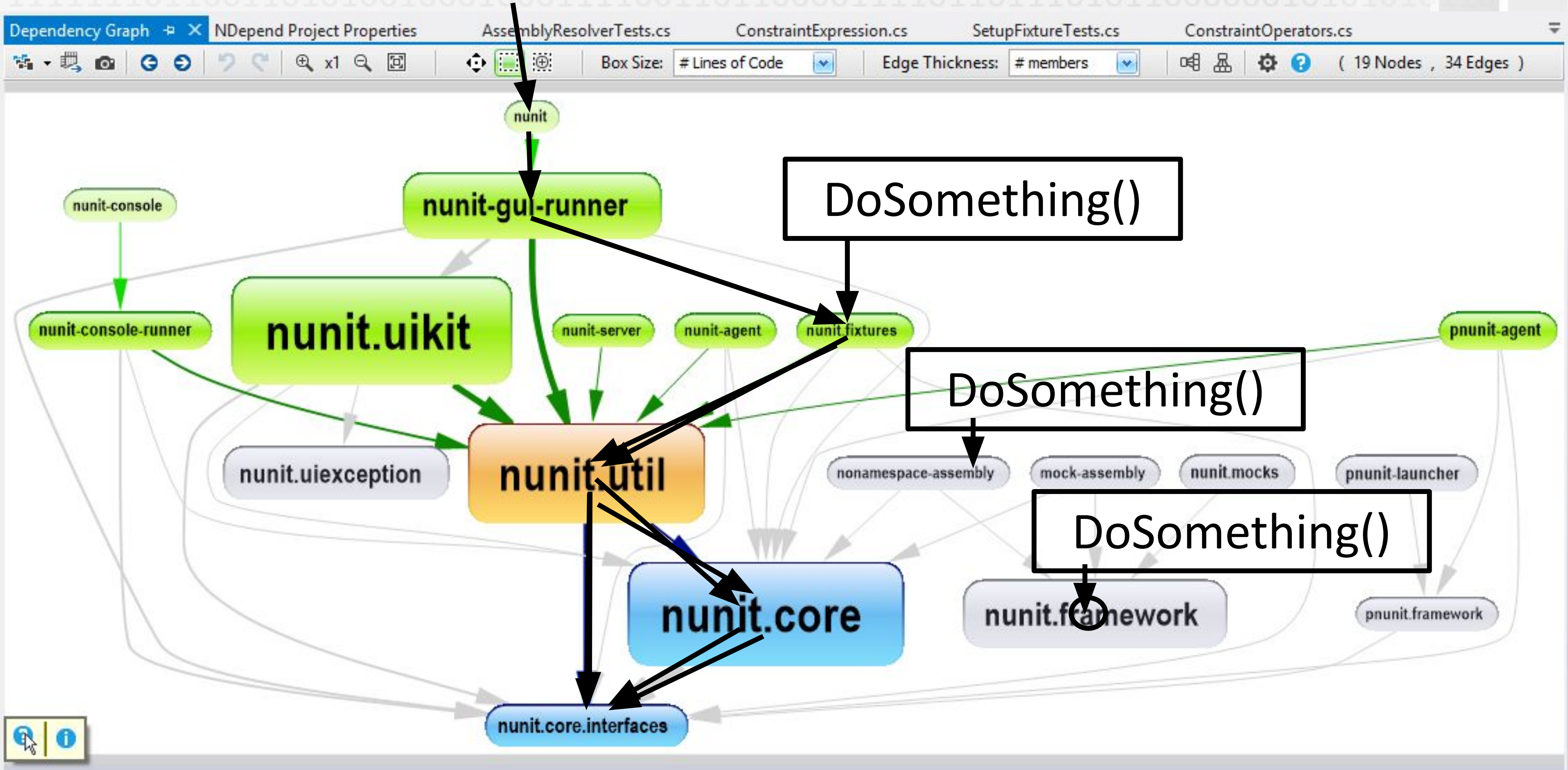
**Uses user input**

**Database access**

**>2^n...?**

- **Projects are usually A LOT BIGGER than 3 blocks!**
- **Automatic tests are developed during the development phase (“later” will mean “never”, so that is NOT an option!)**

# Different test types



- **UI tests**
- **Integration tests**
- **Component tests**
- **Unit tests**

(Names might differ from team to team)

# „The tester will test the code”

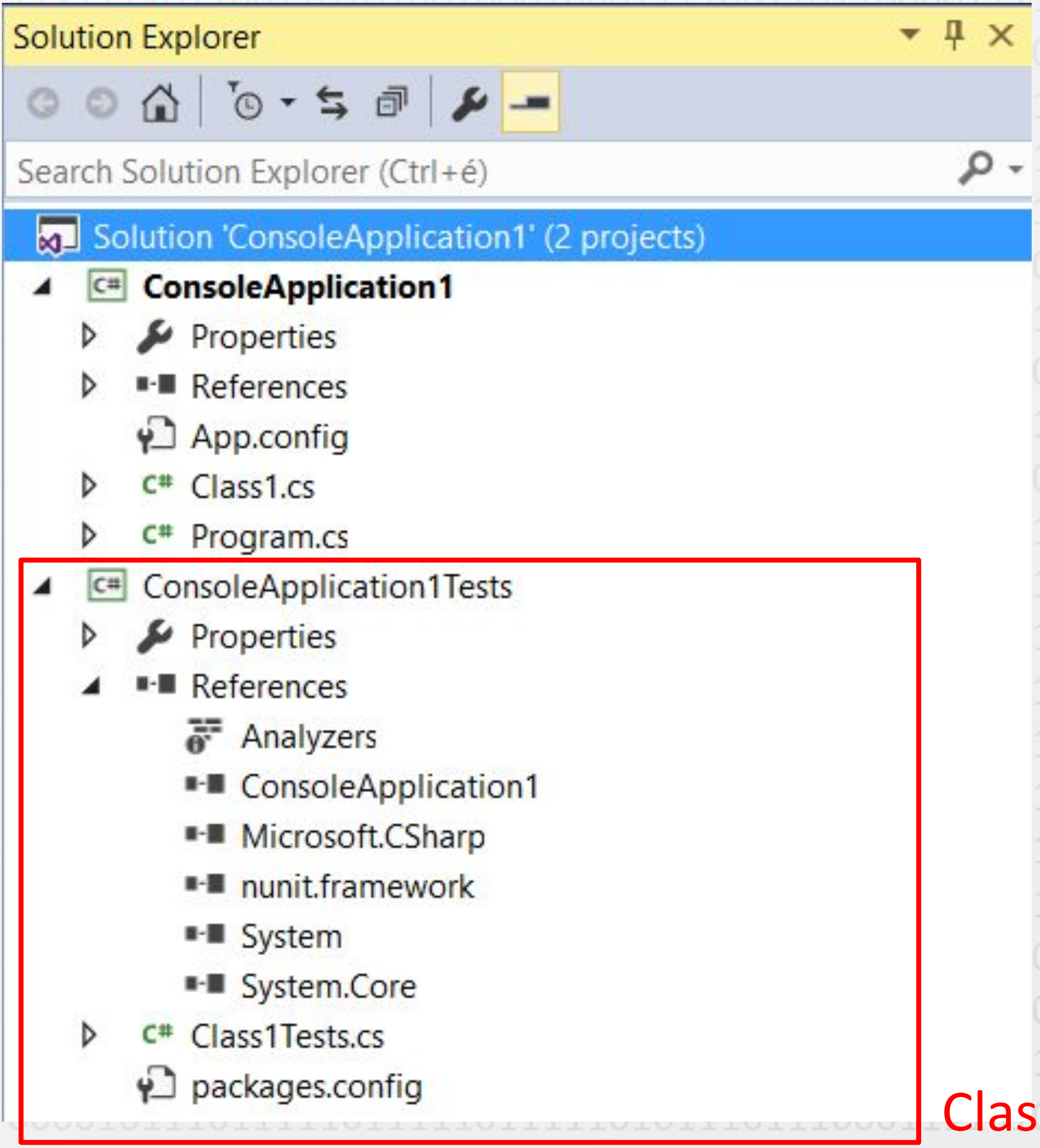
# Testing in the development work process

- **Testing moved closer to the implementation, because:**
  - Gives us an earlier feedback
  - Protects the covered code against bugs introduced later by accident
  - Helps us in clearing up requirements
  - Enforces us to write clear and well-structured code
- **(In some approaches, testing is even done BEFORE coding)**
- **„Agile crossfunctional team"**
  - The team is responsible for all aspects of the product/functionality: planning, implementation, quality check, testing…
  - More and more the maintenance and the support is also included
  - The boundaries between the old roles are grey or missing – **„crossfunctional team"**

# NUnit

- **Instead: unit test framework for .NET languages (Java: jUnit)**
- **Part of a bigger family of utilities**
  - Unit testing: **NUnit** (vs. Visual Studio Unit Testing Framework)
  - Mocking: NSubstitute (vs. **Moq**, Rhino Mocks)
  - IoC container: NInject (vs. Spring.Net, Castle Windsor, Unity)
  - Test coverage: NCover (vs. dotCover)
- **More widespread (…)**
  - Better/readable syntax
  - Good for almost all different types of tests (Unit, Integration, Performance…)
  - GUI support
  - Command line Test Runner (Dotnet Core: beta)
  - Test Explorer inside VisualStudio (with NUnit3TestAdapter)

# Simple unit test project

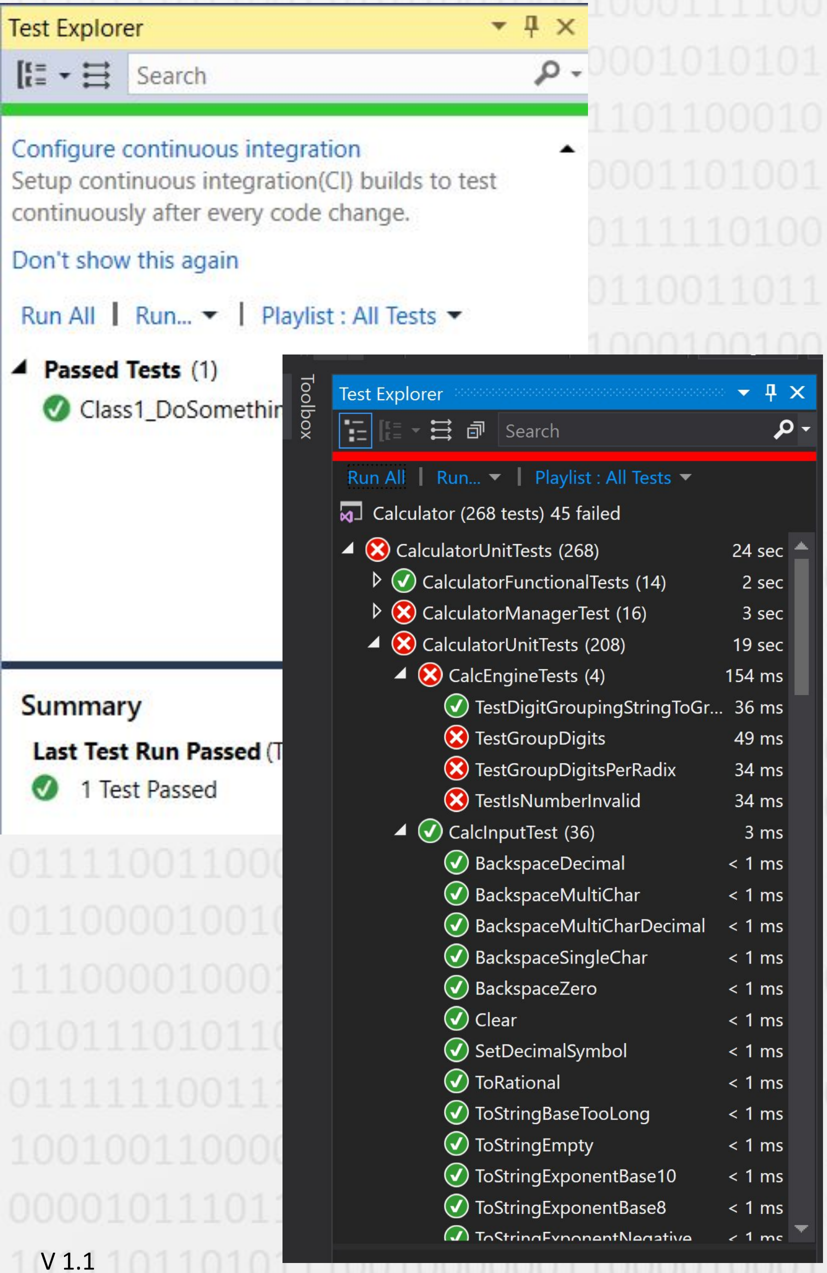


Solution 'ConsoleApplication1' (2 projects)
- **ConsoleApplication1**
  - Properties
  - References
  - App.config
  - Class1.cs
  - Program.cs
- ConsoleApplication1Tests
  - Properties
  - References
    - Analyzers
    - ConsoleApplication1
    - Microsoft.CSharp
    - nunit.framework
    - System
    - System.Core
  - Class1Tests.cs
  - packages.config

- **Live project(s) + test project(s) (Class Library)**
- **References of the test project:**
  - The project to be tested
  - Test framework
- **In the test project we want to test the public parts of the other project**
  - Internal parts: possible but rarely used [InternalsVisibleTo(„ConsoleApplication1Tests”)]
  - Private parts: forbidden
- **With unit tests we only test and check the external working of the class/module.**

Class Library

# Simple unit test project



Test Explorer

Search

Configure continuous integration

Setup continuous integration(CI) builds to test continuously after every code change.

Don't show this again

Run All | Run... | Playlist : All Tests

Passed Tests (1)

✓ Class1_DoSomethin

Summary

Last Test Run Passed (T

✓ 1 Test Passed

Test Explorer

Search

Run All | Run... | Playlist : All Tests

```
Calculator (268 tests) 45 failed
◢ ✗ CalculatorUnitTests (268)                24 sec
  ▷ ✓ CalculatorFunctionalTests (14)          2 sec
  ▷ ✗ CalculatorManagerTest (16)              3 sec
  ◢ ✗ CalculatorUnitTests (208)              19 sec
    ◢ ✗ CalcEngineTests (4)                 154 ms
        ✓ TestDigitGroupingStringToGr...     36 ms
        ✗ TestGroupDigits                    49 ms
        ✗ TestGroupDigitsPerRadix            34 ms
        ✗ TestIsNumberInvalid                34 ms
    ◢ ✓ CalcInputTest (36)                    3 ms
        ✓ BackspaceDecimal                  < 1 ms
        ✓ BackspaceMultiChar                < 1 ms
        ✓ BackspaceMultiCharDecimal         < 1 ms
        ✓ BackspaceSingleChar               < 1 ms
        ✓ BackspaceZero                     < 1 ms
        ✓ Clear                             < 1 ms
        ✓ SetDecimalSymbol                  < 1 ms
        ✓ ToRational                        < 1 ms
        ✓ ToStringBaseTooLong               < 1 ms
        ✓ ToStringEmpty                     < 1 ms
        ✓ ToStringExponentBase10            < 1 ms
        ✓ ToStringExponentBase8             < 1 ms
        ✓ ToStringExponentNegative          < 1 ms
```

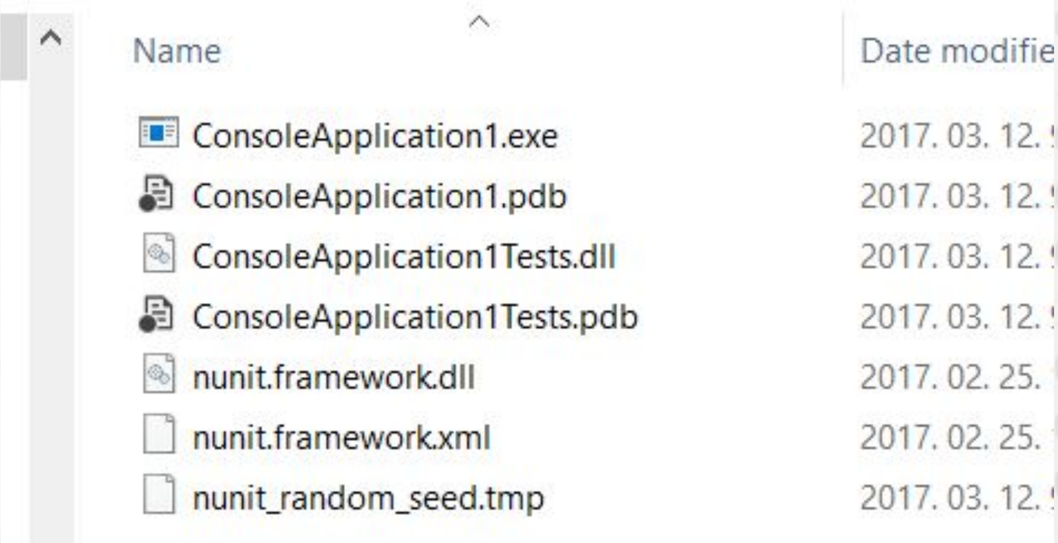
- **Test / Windows / Test Explorer**
- **NUnit3TestAdapter required! (Nuget)**

- **Depending on the development team and the tests, execute tests**
  - always,
  - or after an important change in the code,
  - before/after a merge,
  - Before push ALWAYS

- **NEVER PUSH NON-PASSING CODE!**

# How does it work?

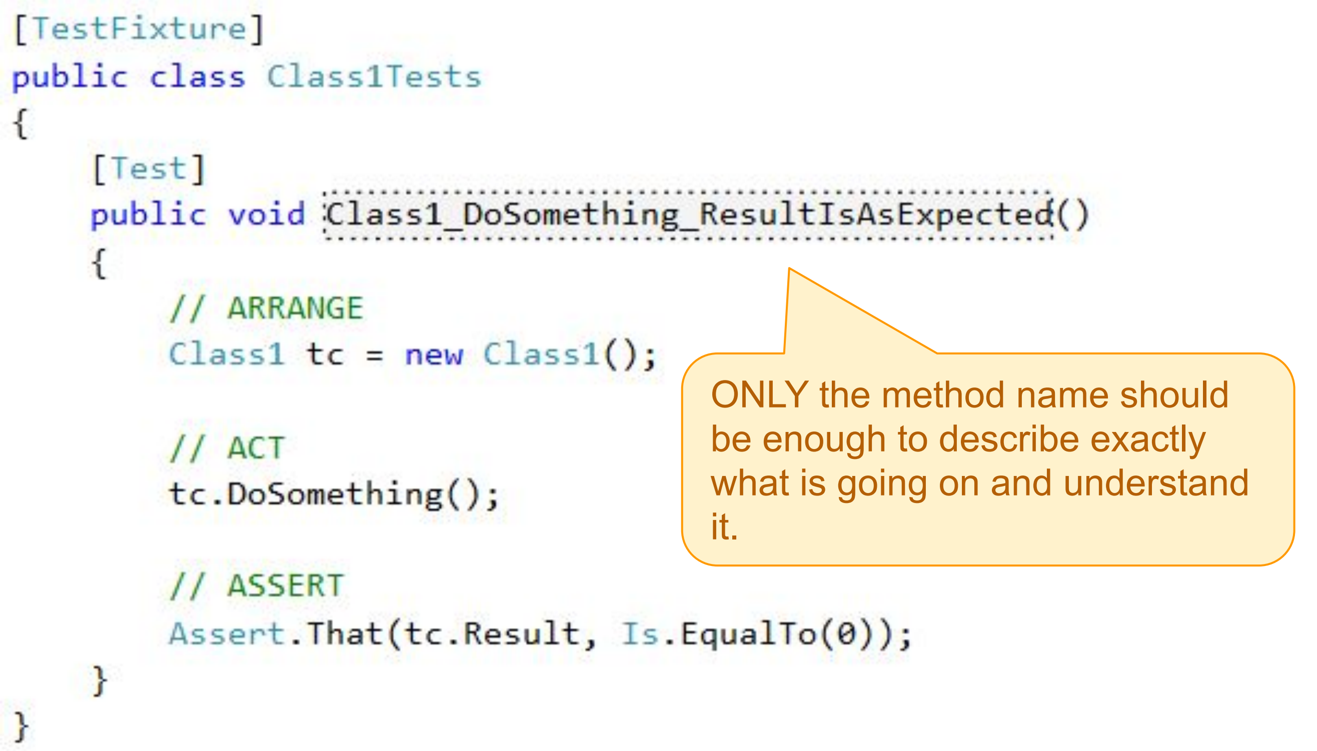
- **Reflection**



| Name | Date modifie |
|---|---|
| ConsoleApplication1.exe | 2017. 03. 12. |
| ConsoleApplication1.pdb | 2017. 03. 12. |
| ConsoleApplication1Tests.dll | 2017. 03. 12. |
| ConsoleApplication1Tests.pdb | 2017. 03. 12. |
| nunit.framework.dll | 2017. 02. 25. |
| nunit.framework.xml | 2017. 02. 25. |
| nunit_random_seed.tmp | 2017. 03. 12. |

- **Studio Test Explorer => NUnit**
  - NUnit uses reflection to look for the **[TestFixture]** attribute in all classes in the solution (exe/dll)
  - The [TestFixture] class has methods marked with [Test], those methods are located via reflection
  - It interprets the additional configuration attributes ([TestCaseSource], [Explicit], ... ... ... ) that change the input/operation/…
  - Then the test methods are executed using reflection

# NUnit

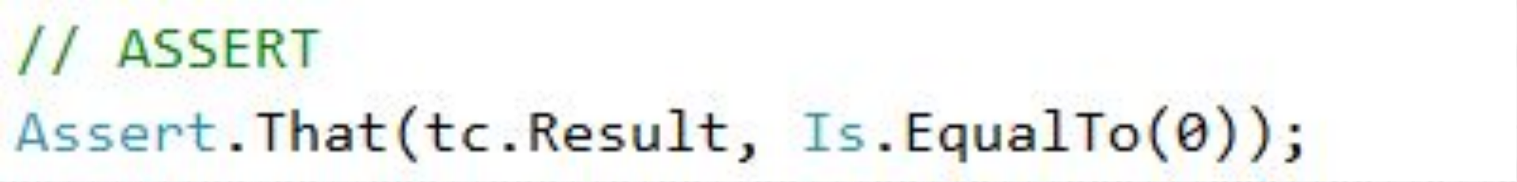


```
[TestFixture]
public class Class1Tests
{
    [Test]
    public void Class1_DoSomething_ResultIsAsExpected()
    {
        // ARRANGE
        Class1 tc = new Class1();

        // ACT
        tc.DoSomething();

        // ASSERT
        Assert.That(tc.Result, Is.EqualTo(0));
    }
}
```

ONLY the method name should be enough to describe exactly what is going on and understand it.

- **Common test parts („AAA”)**
  - **Arrange**: preparation, instance creation, setting values, etc.
  - **Act**: executing the **single** step that we want to test
  - **Assert**: comparing the actual result with the expected outcome of the call:
    - result → eg. 2 + 2 equals 4?!
    - state change → eg. after a method call some property’s value should be X, is that true or not?
    - behaviour (exceptions, events) → eg. I can check that the expected exception happened or not?
    - circumstance (backend calls) → eg. during the registration does the user received email or not? (was the sendEmail method called or not, from an other method)
- **Other naming schemes:**
  - Given…When…Then()
  - Spec / SpecFlow (BDD - Behavior Driven Development)

# NUnit



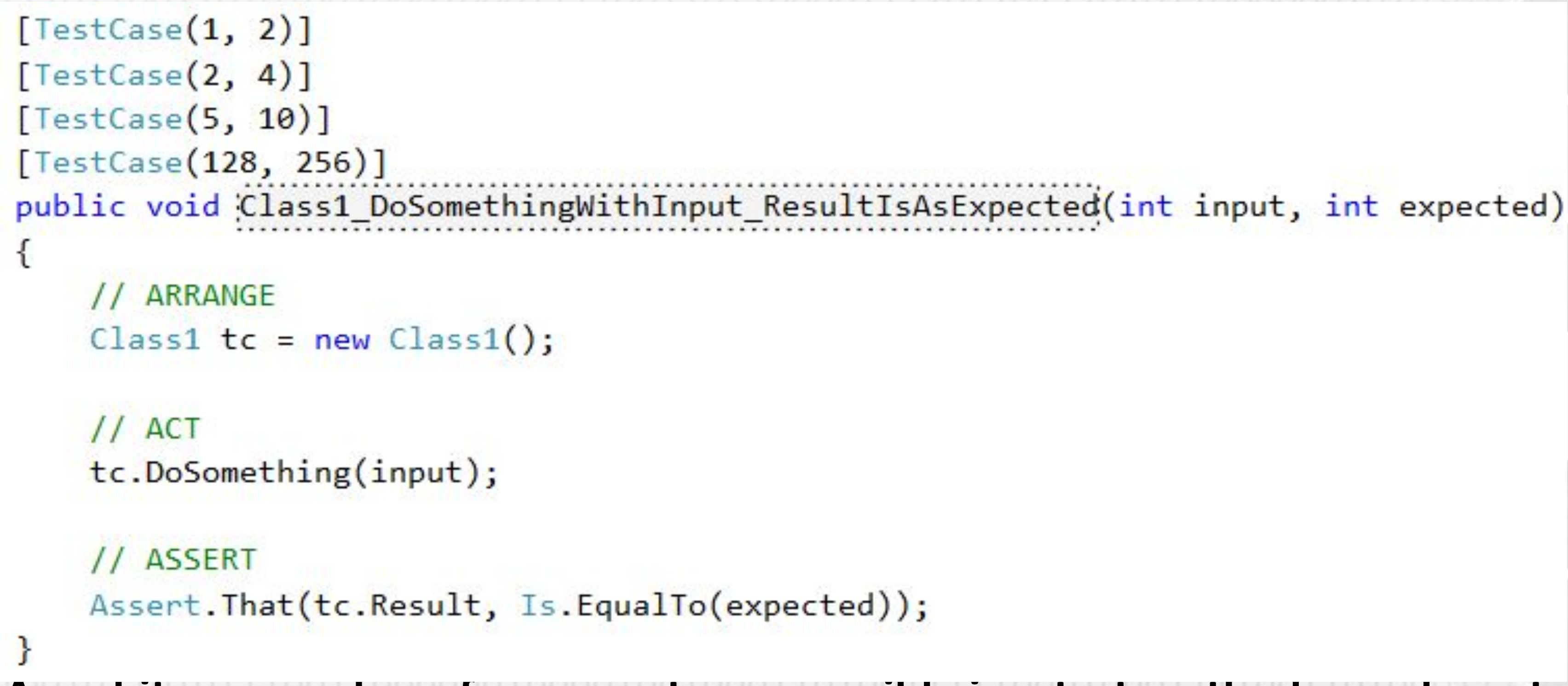
```
// ASSERT
Assert.That(tc.Result, Is.EqualTo(0));
```

- **Lot of possibilities**

```
// Values:
Assert.That(result, Is.EqualTo("MyResult"));
Assert.That(result, Is.Null);
Assert.That(result, Is.LessThan(2));
Assert.That(result, Is.SameAs(otherReferenceToTheSameObject));
Assert.That(result, Is.Not.Null); // Negate

// Exceptions:
Assert.That(() => t.MyTestedMethod(),
    Throws.TypeOf<NullReferenceException>());
Assert.That(() => t.MyTestedMethod(), Throws.Nothing);

// Old syntax: (same, as Is.EqualTo)
Assert.AreEqual(result, 42);
```

You can read out like english sentences (fluent syntax). Better, should be used.

# NUnit

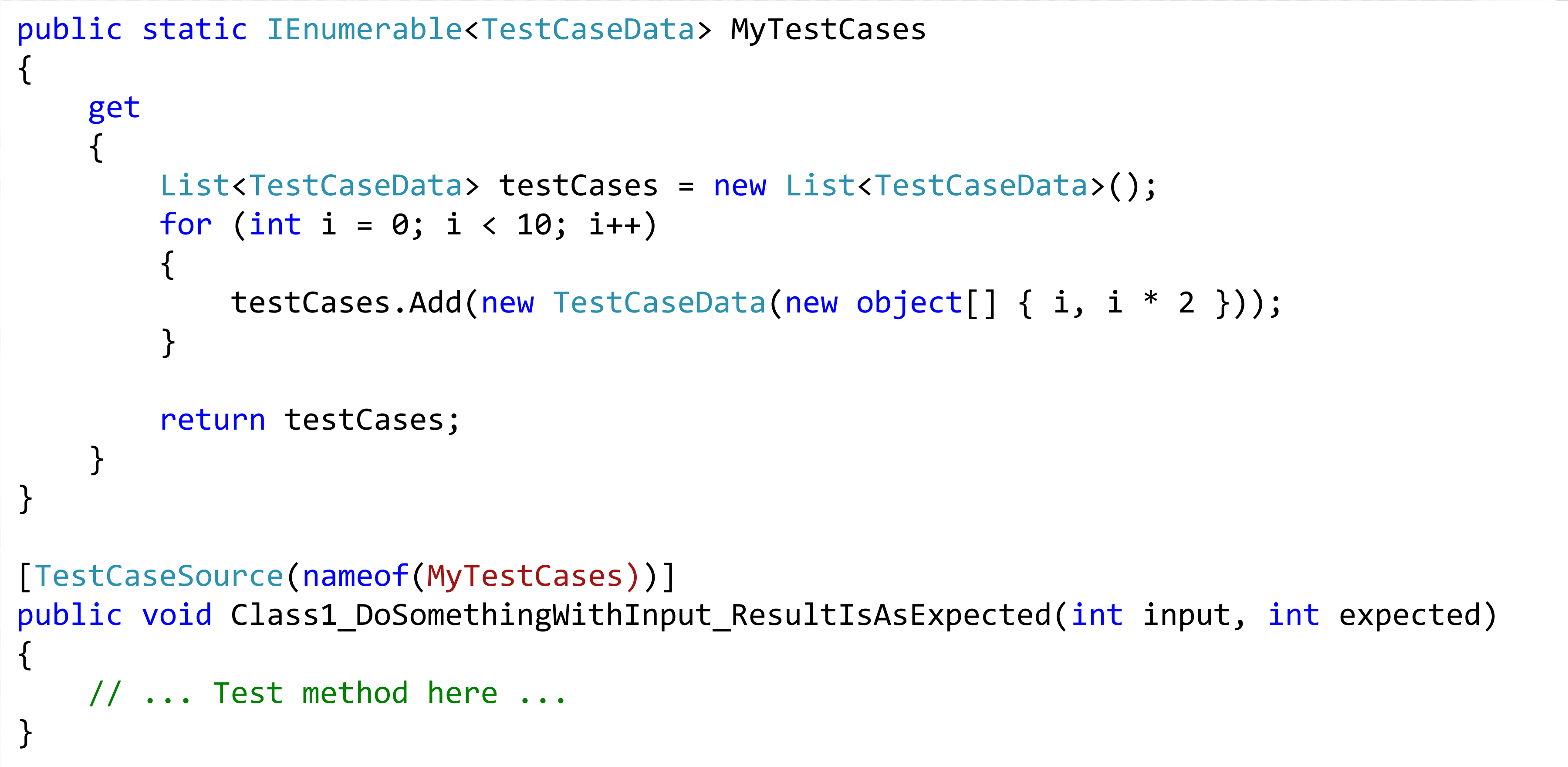


```
[TestCase(1, 2)]
[TestCase(2, 4)]
[TestCase(5, 10)]
[TestCase(128, 256)]
public void Class1_DoSomethingWithInput_ResultIsAsExpected(int input, int expected)
{
    // ARRANGE
    Class1 tc = new Class1();

    // ACT
    tc.DoSomething(input);

    // ASSERT
    Assert.That(tc.Result, Is.EqualTo(expected));
}
```

- **An arbitrary number of arguments are possible in a test method – we have to use the same number of arguments in the TestCase attribute**
  - Here we can use only constants -> TestCaseSource!
- **Similar:**
  - [Sequential] – the input parameters are used sequentially
  - [Combinatorial] – The input parameters are taken with all possible combinations
  - [Pairwise] – optimized (results in smaller number of test cases)

# NUnit

- **TestCaseSource**
  - To use dynamically generated parameters for test cases
  - If wanted, can be used in alternative ways: instead of object[], we can use a TestCaseData descendant; instead of a property we can use a method/class…



```
public static IEnumerable<TestCaseData> MyTestCases
{
    get
    {
        List<TestCaseData> testCases = new List<TestCaseData>();
        for (int i = 0; i < 10; i++)
        {
            testCases.Add(new TestCaseData(new object[] { i, i * 2 }));
        }

        return testCases;
    }
}

[TestCaseSource(nameof(MyTestCases))]
public void Class1_DoSomethingWithInput_ResultIsAsExpected(int input, int expected)
{
    // ... Test method here ...
}
```

# NUnit

If I run 1000 tests, **Setup/TearDown** will run 1000 times (before/after every test) which can mean a lot of extra work for the CPU → **tests will be slow(**er)!

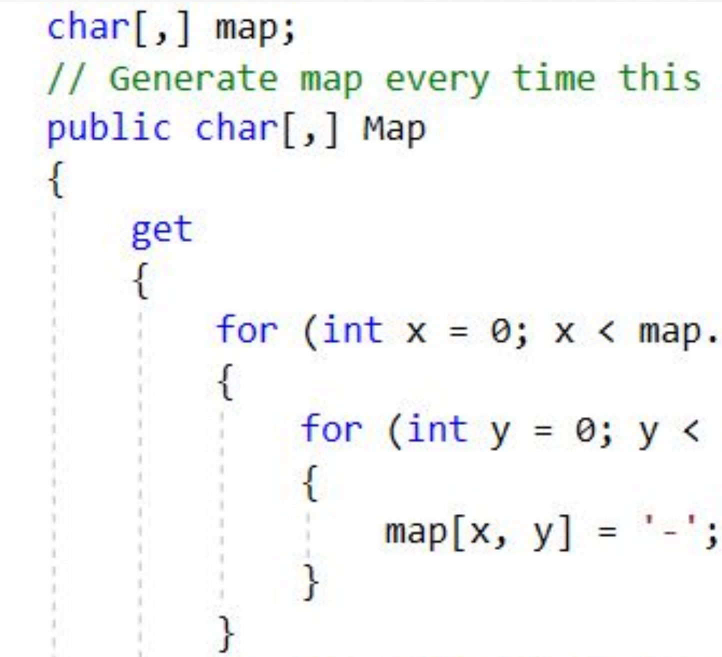
- **[Setup]**
  - The marked method will be executed **before every single** test or test case
- **[TearDown]**
  - The marked method will be executed **after every single** test or test case
- **[TestFixtureSetUp] / [OneTimeSetUp]**
  - The marked method will be executed **once before the tests** of a [TestFixture]
  - We can create common resources – look out, the tests must be independent!
- **[TestFixtureTearDown] / [OneTimeTearDown]**
  - The marked method will be executed **once after the tests** of a [TestFixture]
- **[SetUpFixture]**
  - Can be applied to a class, namespace-level setup/teardown

It can be solved using **TestFixtureSetup** where it will run only once, but in that case if 1000 tests use the same object/entity, it can be **problematic**!

# It is hard to write a good test!

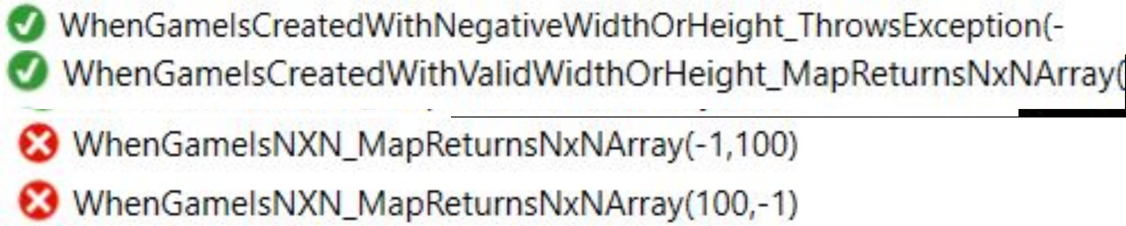
- **The tests must be fast**
  - Slow tests are impossible to run over and over again  test will not be executed, bugs will be found later
- **The tests must be independent**
  - Order, timing, etc. must not affect the results
- **Naming convention must be easy-to-read**
  - A good test list is basically a ***requirement-list*** that documents the capabilities of the program
- **We must not cover every possible inputs**
  - Examples are good
  - ***Finding the corner cases are important!***
- **Only test a single feature of a single class**
  - Always independent from the live data (database/settings)
  - We can substitute the dependencies too: Dependency Injection + fake dependencies, mocking... (=> Moq)

# Test Cases – simple code?



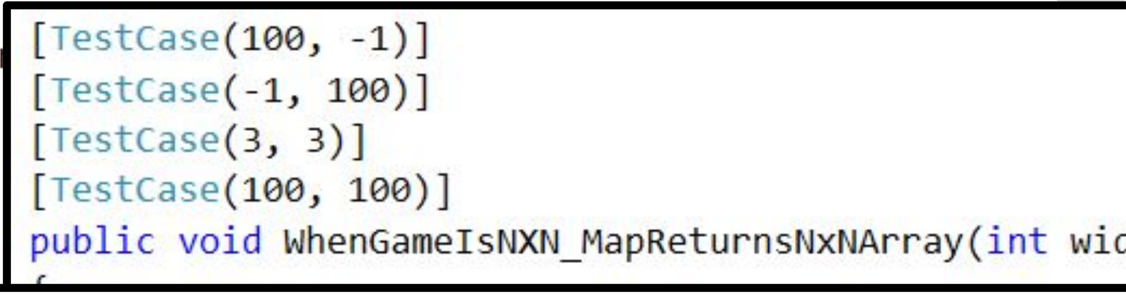
```
char[,] map;
// Generate map every time this
public char[,] Map
{
    get
    {
        for (int x = 0; x < map.GetLength(0); x++)
        {
            for (int y = 0; y <
            {
                map[x, y] = '-';
            }
        }
```

```
- - - - - - - - - -
- - - - - - - - - -
- - - - x - - - - -
- - - - - - - - - -
```

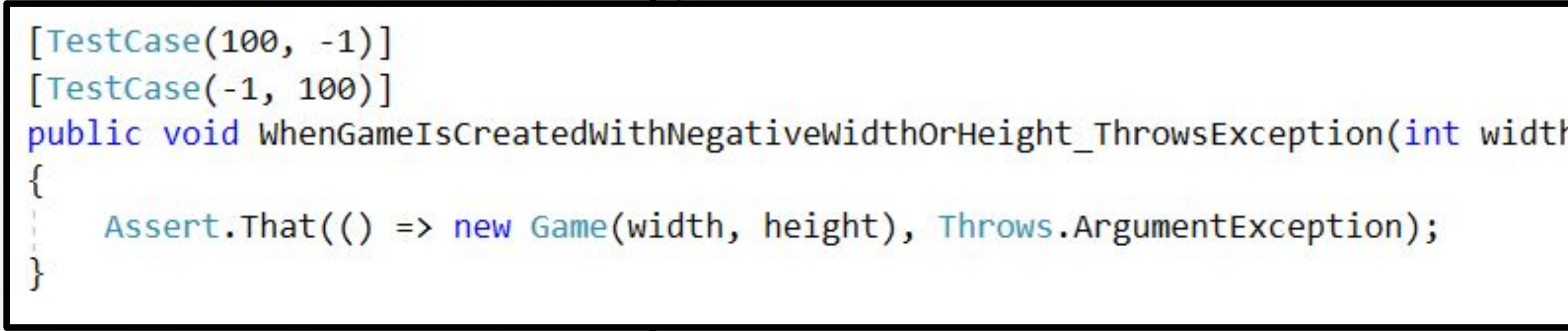


✓ WhenGameIsCreatedWithNegativeWidthOrHeight_ThrowsException(-
✓ WhenGameIsCreatedWithValidWidthOrHeight_MapReturnsNxNArray(
✗ WhenGameIsNXN_MapReturnsNxNArray(-1,100)
✗ WhenGameIsNXN_MapReturnsNxNArray(100,-1)

```
- - - - - - - - - -
- - - - - - - - - *
```



```
[TestCase(100, -1)]
[TestCase(-1, 100)]
[TestCase(3, 3)]
[TestCase(100, 100)]
public void WhenGameIsNXN_MapReturnsNxNArray(int wid
```

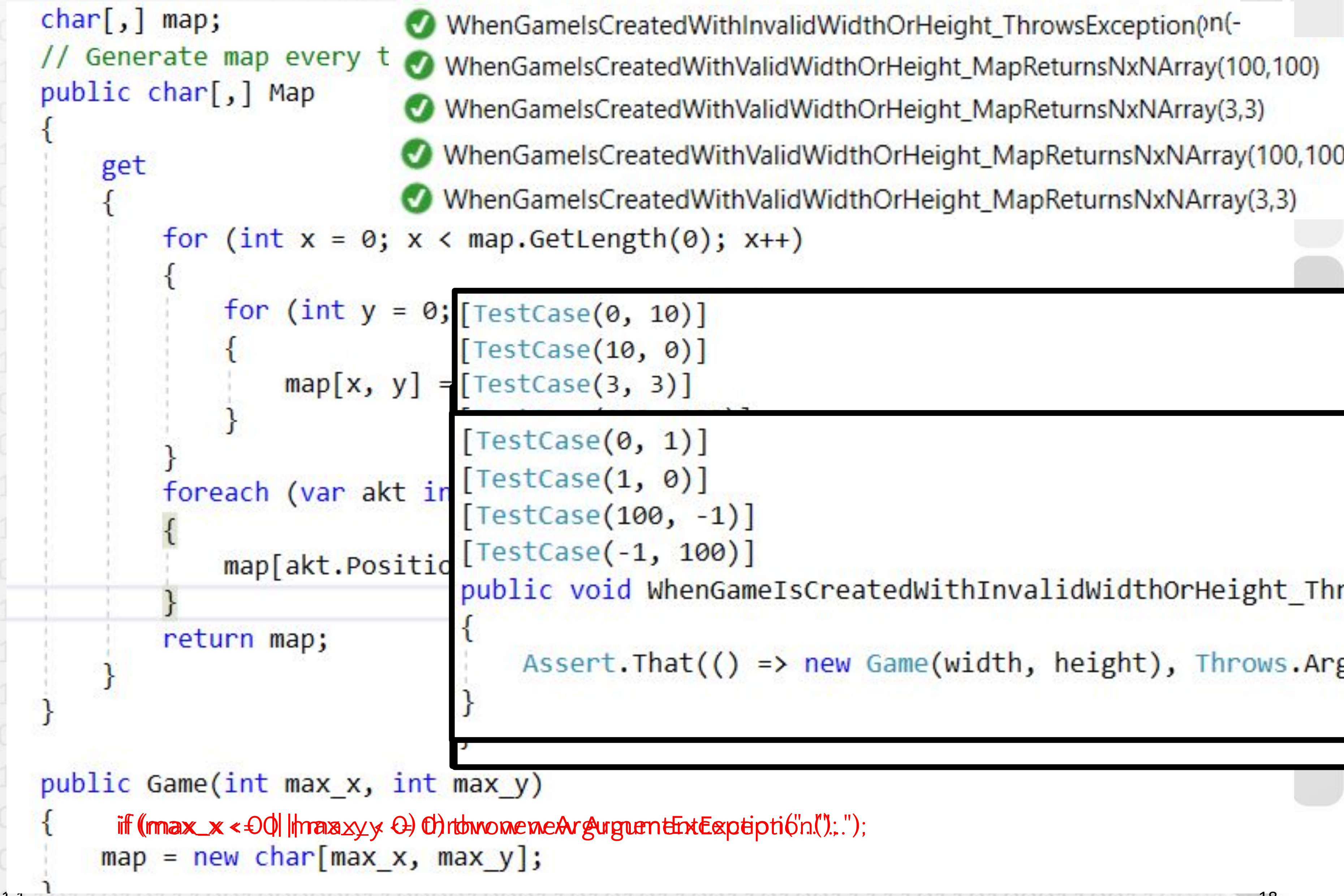


```
[TestCase(100, -1)]
[TestCase(-1, 100)]
public void WhenGameIsCreatedWithNegativeWidthOrHeight_ThrowsException(int widt
{
    Assert.That(() => new Game(width, height), Throws.ArgumentException);
}
```

```
public Game(int max_x, int max_y)
{
    map = new char[max_x, max_y];
```

if (max_x < 0 || max_y < 0) throw new ArgumentException("...");

# Test Cases – simple code?



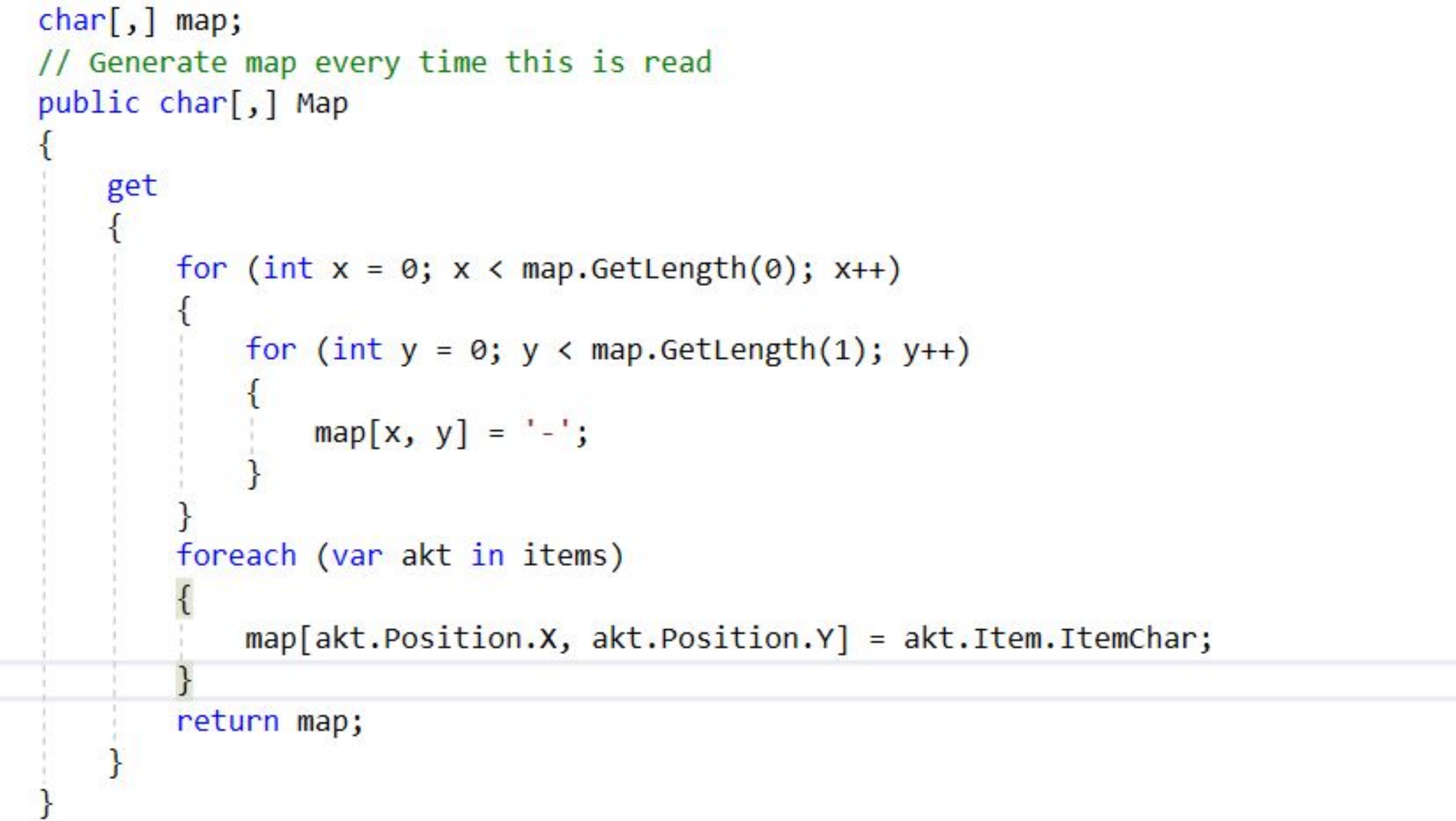
```
char[,] map;
// Generate map every t
public char[,] Map
{
    get
    {
        for (int x = 0; x < map.GetLength(0); x++)
        {
            for (int y = 0;
            {
                map[x, y] =
            }
        }
        foreach (var akt in
        {
            map[akt.Positio
        }
        return map;
    }
}

public Game(int max_x, int max_y)
{
    if (max_x < 0 || max_y < 0) throw new ArgumentException("..");
    if (max_x <= 0 || max_y <= 0) throw new ArgumentException("..");
    map = new char[max_x, max_y];
```

```
✓ WhenGameIsCreatedWithInvalidWidthOrHeight_ThrowsException(n(-
✓ WhenGameIsCreatedWithValidWidthOrHeight_MapReturnsNxNArray(100,100)
✓ WhenGameIsCreatedWithValidWidthOrHeight_MapReturnsNxNArray(3,3)
✓ WhenGameIsCreatedWithValidWidthOrHeight_MapReturnsNxNArray(100,100
✓ WhenGameIsCreatedWithValidWidthOrHeight_MapReturnsNxNArray(3,3)
```

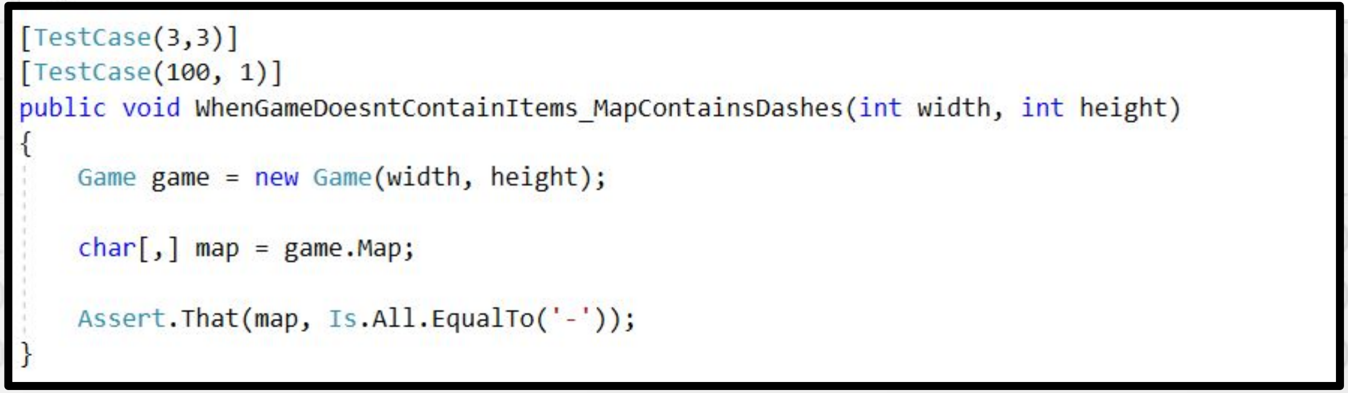
```
[TestCase(0, 10)]
[TestCase(10, 0)]
[TestCase(3, 3)]
```

```
[TestCase(0, 1)]
[TestCase(1, 0)]
[TestCase(100, -1)]
[TestCase(-1, 100)]
public void WhenGameIsCreatedWithInvalidWidthOrHeight_Thr
{
    Assert.That(() => new Game(width, height), Throws.Arg
}
```

# Test Cases – more complex code

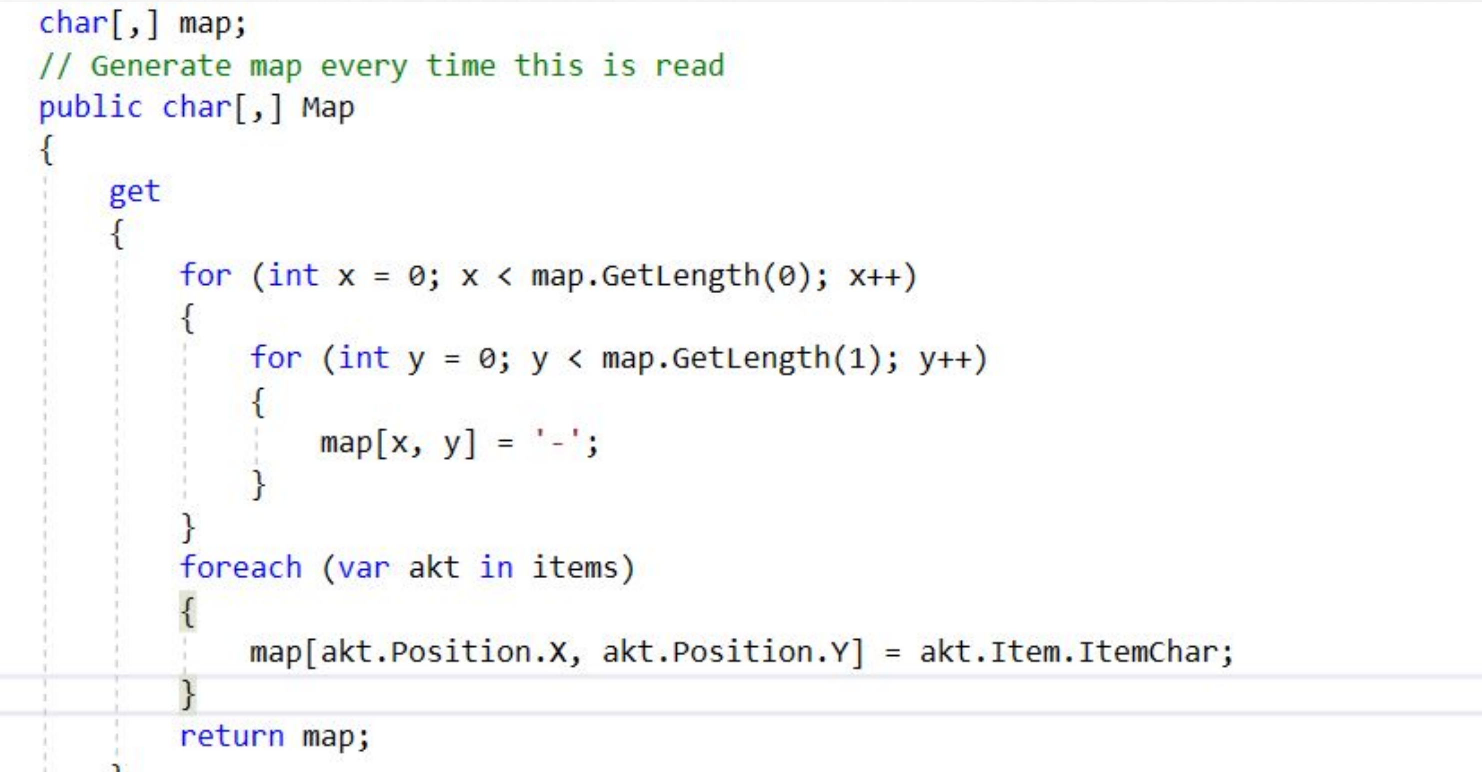


```
char[,] map;
// Generate map every time this is read
public char[,] Map
{
    get
    {
        for (int x = 0; x < map.GetLength(0); x++)
        {
            for (int y = 0; y < map.GetLength(1); y++)
            {
                map[x, y] = '-';
            }
        }
        foreach (var akt in items)
        {
            map[akt.Position.X, akt.Position.Y] = akt.Item.ItemChar;
        }
        return map;
    }
}
```

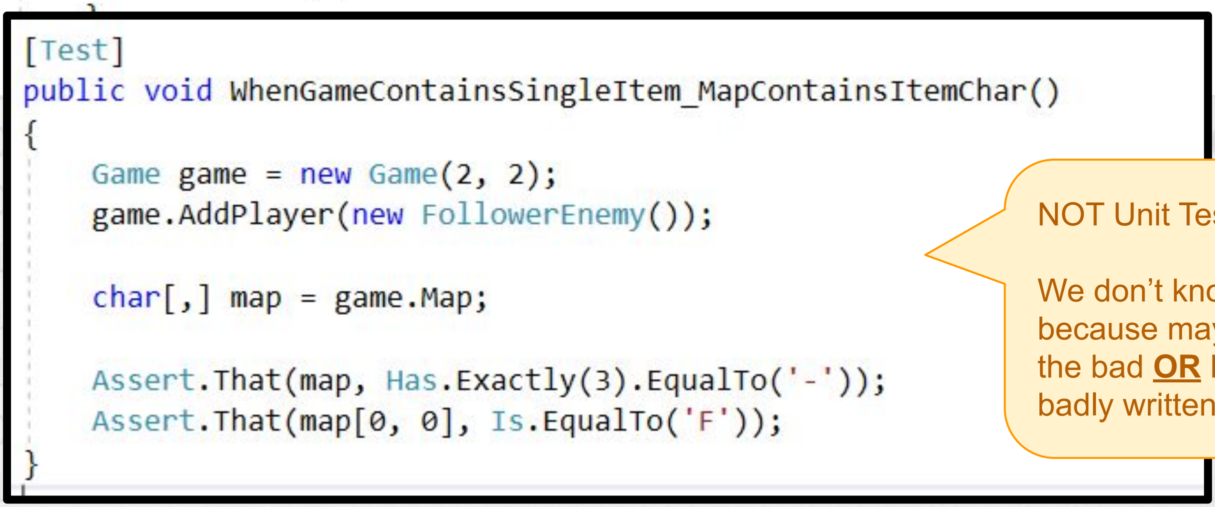


```
[TestCase(3,3)]
[TestCase(100, 1)]
public void WhenGameDoesntContainItems_MapContainsDashes(int width, int height)
{
    Game game = new Game(width, height);

    char[,] map = game.Map;

    Assert.That(map, Is.All.EqualTo('-'));
}
```

# Test Cases – more complex code



```
char[,] map;
// Generate map every time this is read
public char[,] Map
{
    get
    {
        for (int x = 0; x < map.GetLength(0); x++)
        {
            for (int y = 0; y < map.GetLength(1); y++)
            {
                map[x, y] = '-';
            }
        }
        foreach (var akt in items)
        {
            map[akt.Position.X, akt.Position.Y] = akt.Item.ItemChar;
        }
        return map;
```

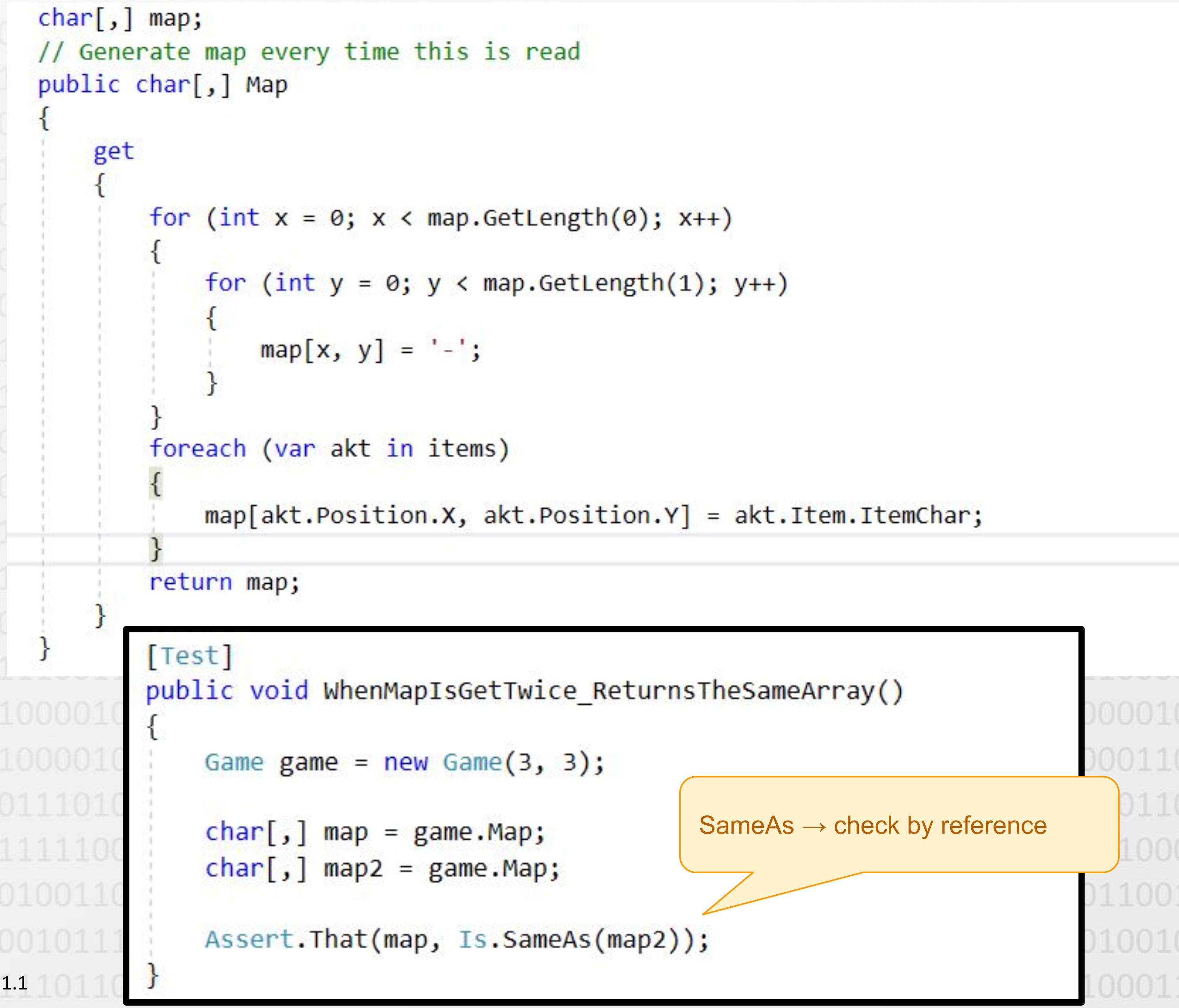


```
[Test]
public void WhenGameContainsSingleItem_MapContainsItemChar()
{
    Game game = new Game(2, 2);
    game.AddPlayer(new FollowerEnemy());

    char[,] map = game.Map;

    Assert.That(map, Has.Exactly(3).EqualTo('-'));
    Assert.That(map[0, 0], Is.EqualTo('F'));
}
```

NOT Unit Test!

We don’t know what is the problem, because maybe the FollowerEnemy is the bad **OR** Map is bad **OR** the test is badly written.

# Test Cases – more complex code



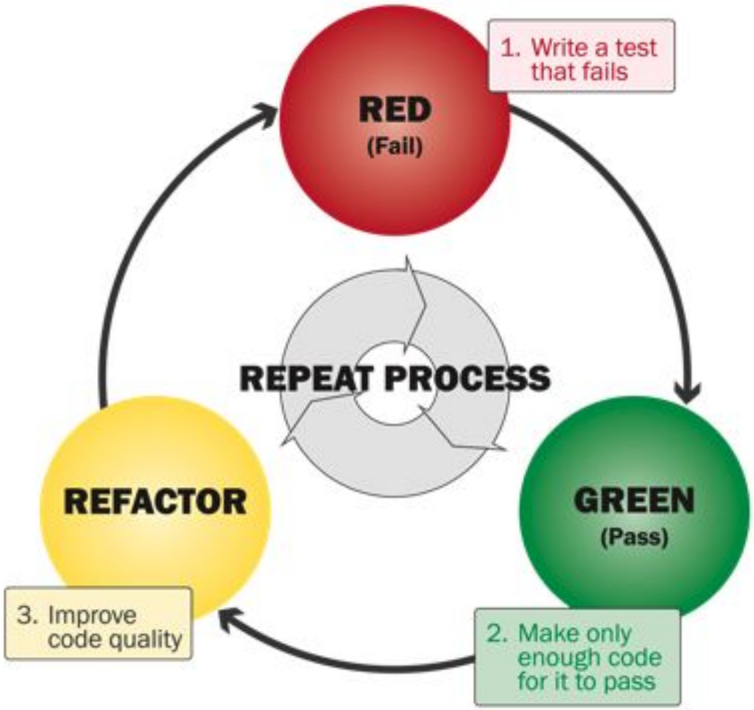
```
char[,] map;
// Generate map every time this is read
public char[,] Map
{
    get
    {
        for (int x = 0; x < map.GetLength(0); x++)
        {
            for (int y = 0; y < map.GetLength(1); y++)
            {
                map[x, y] = '-';
            }
        }
        foreach (var akt in items)
        {
            map[akt.Position.X, akt.Position.Y] = akt.Item.ItemChar;
        }
        return map;
    }
}
```

```
[Test]
public void WhenMapIsGetTwice_ReturnsTheSameArray()
{
    Game game = new Game(3, 3);

    char[,] map = game.Map;
    char[,] map2 = game.Map;

    Assert.That(map, Is.SameAs(map2));
}
```

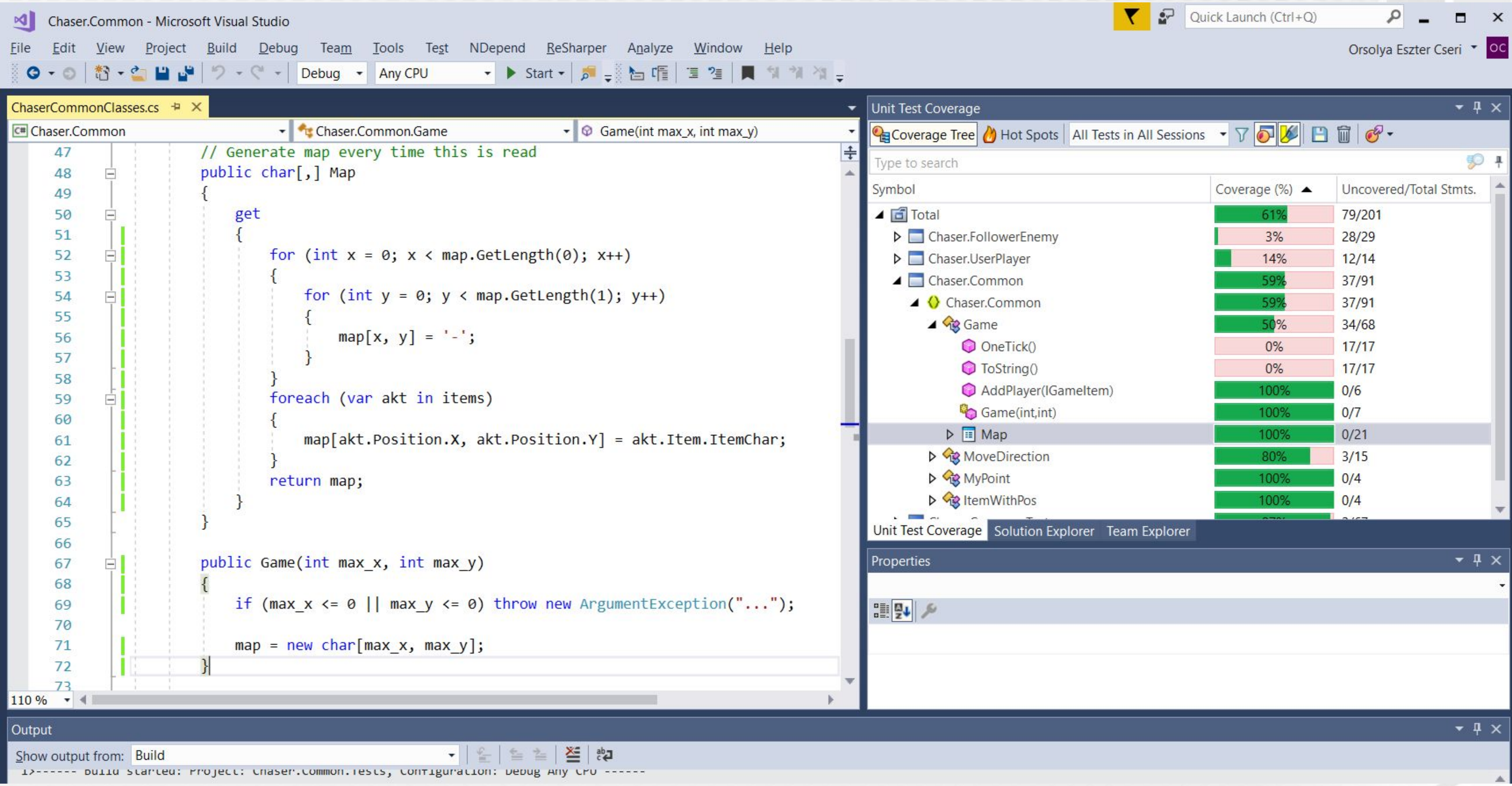
SameAs → check by reference

# Testing while developing

- **First the code, then the test**
  - Hard and not effective – also, “later” is many times “never”
- **Test First: create the tests first, then the code**
  - The requirements must be fixed beforehand – they have to be cleared up first!
  - This also ensures keeping the operational requirements when re-writing
- **TDD (Test-Driven Development):**
  - Done in pair programming
  - The code is written to be 100% “testable”
  - Development time: twice, but better quality
  - Not a generic all-around solution (typically good for the „algorithmic” tasks)
- **BDD, ATDD...**


1. Write a test that fails
RED
(Fail)
REPEAT PROCESS
REFACTOR
GREEN
(Pass)
3. Improve code quality
2. Make only enough code for it to pass

# Coverage



```
            // Generate map every time this is read
            public char[,] Map
            {
                get
                {
                    for (int x = 0; x < map.GetLength(0); x++)
                    {
                        for (int y = 0; y < map.GetLength(1); y++)
                        {
                            map[x, y] = '-';
                        }
                    }
                    foreach (var akt in items)
                    {
                        map[akt.Position.X, akt.Position.Y] = akt.Item.ItemChar;
                    }
                    return map;
                }
            }

            public Game(int max_x, int max_y)
            {
                if (max_x <= 0 || max_y <= 0) throw new ArgumentException("...");

                map = new char[max_x, max_y];
            }
```

| Symbol | Coverage (%) | Uncovered/Total Stmts. |
|---|---|---|
| Total | 61% | 79/201 |
| Chaser.FollowerEnemy | 3% | 28/29 |
| Chaser.UserPlayer | 14% | 12/14 |
| Chaser.Common | 59% | 37/91 |
| Chaser.Common | 59% | 37/91 |
| Game | 50% | 34/68 |
| OneTick() | 0% | 17/17 |
| ToString() | 0% | 17/17 |
| AddPlayer(IGameItem) | 100% | 0/6 |
| Game(int,int) | 100% | 0/7 |
| Map | 100% | 0/21 |
| MoveDirection | 80% | 3/15 |
| MyPoint | 100% | 0/4 |
| ItemWithPos | 100% | 0/4 |

```
1>------ Build started: Project: Chaser.Common.Tests, Configuration: Debug Any CPU ------
```

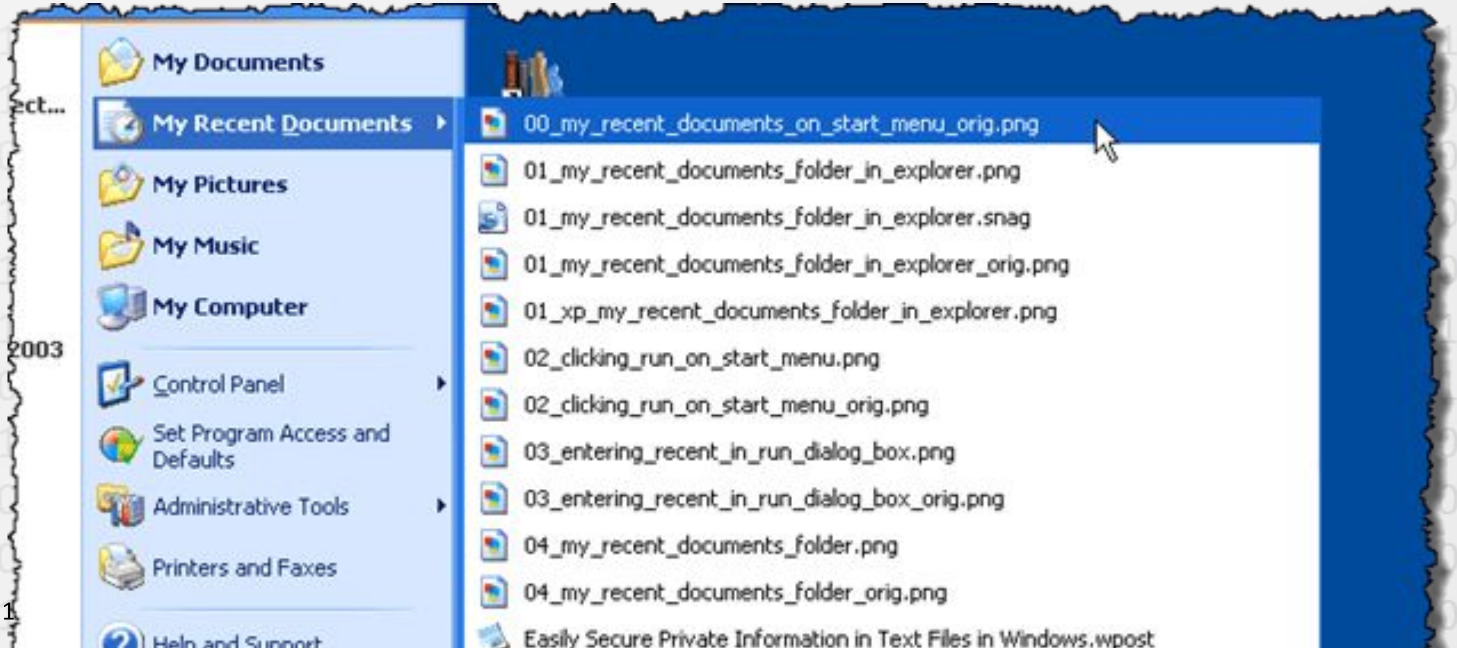
- How many percentage of my important codes are tested?!
- dotCover
- NCover
- OpenCover

# Coverage

- **Usually there is a minimum, but also a maximum (don’t aim the 100%, and also don’t be content if the coverage is 100%)**
- **More important measurements**
  - Statement Coverage: statements (if, while, for – excluded)
  - Branch Coverage: Conditions (if, else)
  - Condition Coverage: Every bool expression had true and false as well
  - Loop Coverage: Every loop was executed 0x, 1x and >1x, and if possible, then with the maximum limit and max+1 as well
  - Parameter Value Coverage: All significant values (e.g. string null, empty, whitespace…)
  - Inheritance Coverage: test for all possible return types
  - Use case coverage

# Exercise

- **Create a class that implements the Last Recently Used (LRU) functionality**
- **The class contains a list with a maximum capacity, and must have a *public void Add(object instance)* method**
- **During the development, follow the TDD approach**
  - Create an „it barely works” class
  - Write tests until we run into a red test
  - Fix the class to make the test green, then write tests…


ect...
2003
My Documents
My Recent Documents
My Pictures
My Music
My Computer
Control Panel
Set Program Access and Defaults
Administrative Tools
Printers and Faxes
Help and Support
00_my_recent_documents_on_start_menu_orig.png
01_my_recent_documents_folder_in_explorer.png
01_my_recent_documents_folder_in_explorer.snag
01_my_recent_documents_folder_in_explorer_orig.png
01_xp_my_recent_documents_folder_in_explorer.png
02_clicking_run_on_start_menu.png
02_clicking_run_on_start_menu_orig.png
03_entering_recent_in_run_dialog_box.png
03_entering_recent_in_run_dialog_box_orig.png
04_my_recent_documents_folder.png
04_my_recent_documents_folder_orig.png
Easily Secure Private Information in Text Files in Windows.wpost

# Practice Exercise

- **A bookshop wants to sell a 5-part book series. One copy of any of the five books costs 8 EUR. Discounts:**
  - 2 different books □ 5% discount on those two books.
  - 3 different books □ 10% discount on all three books.
  - 4 different books □ 20% discount on all four books.
  - 5 different books □ 25% discount on all five books
- **If you buy, four books, of which 3 are different titles, you get a 10% discount on the 3 that form part of a set, but the fourth book still costs 8 EUR:**
  - 1, 2, 3, 1 => (1, 2, 3) 10% discount, (1) 0% discount => 3x7.2 + 8 = 29.6
- **Some „baskets” can be grouped multiple ways!**
  - 1, 1, 2, 2, 3, 3, 4, 5 => (1, 2, 3, 4, 5) (1, 2, 3) or (1, 2, 3, 4) (1, 2, 3, 5) ?
- **Write a class (and tests) to calculate the price of a “basket”**
  - Grouping using the most simple way (always create the biggest group, even if this might not be the cheapest price)