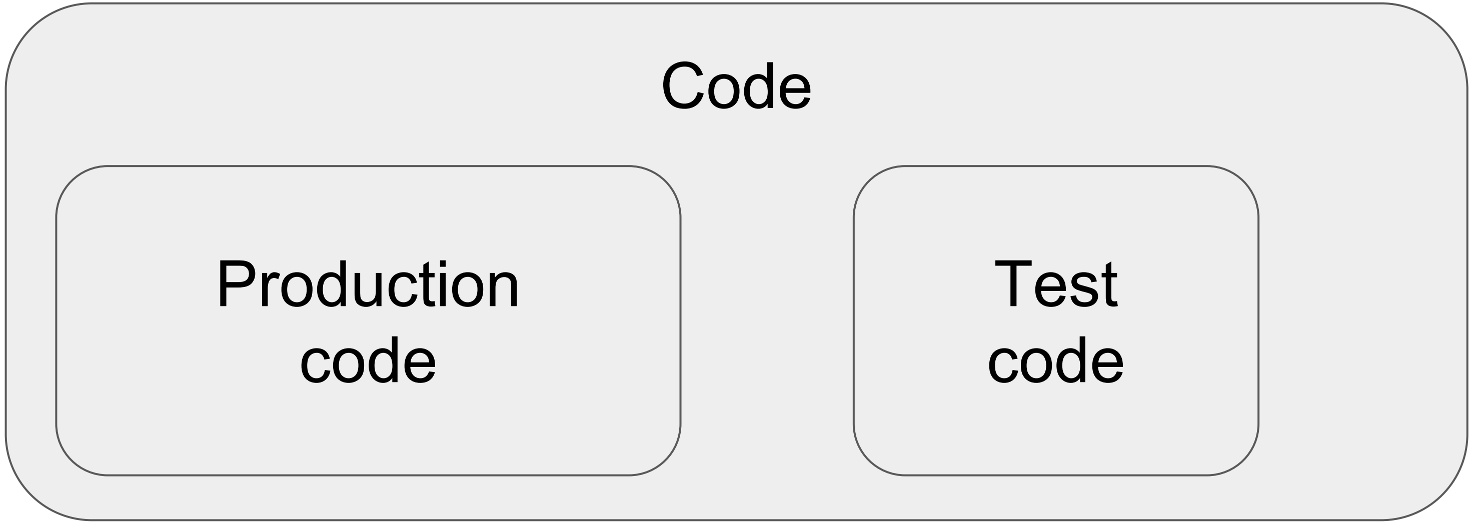
Unit testing

Unit testing

The practice of **writing code** to **test the code**, and then run those tests automatically.



Manual vs. automated

```
public double CalculateTax(double price){
    if (x) return ...;
    if (y) return ...;

    return ...;
}
```

Manual:

1. Launch the app
2. Login
3. Go to tax calculation page
4. Fill out a form
5. Verify result on the screen

Automated:

```
var result = CalculateTax(2.22);

Verify(result == 0.22);
```

Repeatability

Unit-tests are repeatable. We can repeat tests

- before we deploy the code (make software available for users)
- when we commit the code to repository
- while we make a change in the code

Instead of spending hours to test all the code manually we can run **thousands** of unit tests in seconds.

Confidence

When there are lots of unit-tests developers can be confident that there are *no bugs.* Any obvious bugs will be immediately caught with unit tests.

Developer can be confident that his change did not break anything.

Developer can be confident to refactor or edit the code.

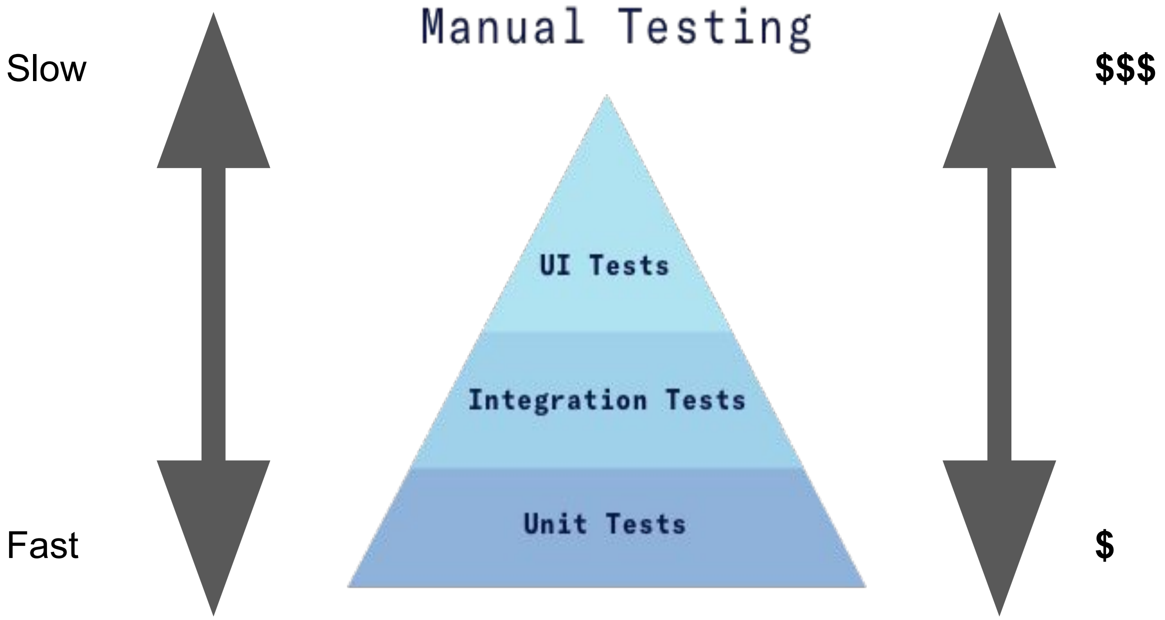
Types of tests

End-to-end (e2e) - tests the whole application from start to end

Integration - tests which combine multiple parts of the program and test them together

Unit - tests which test unit (a part) of the program separately

Test pyramid



Unit tests

Unit tests are ideal for complex algorithms, functions with a lot if logic inside.

```
public double CalculateTax(double price){
    if (x) return ...;
    if (y) return ...;

    return ...;
}
```

Tools to unit test in C#

In C# the most popular tools to unit test are:

- MSTest - made by Microsoft
- NUnit - old and trusty
- xUnit - new and cool

First tests

Good Unit Tests

1) As important as the production code
2) Clean, readable and maintainable
3) No logic (no if, else, foreach etc...)
4) Isolated (no calling other tests from tests)
5) Not too specific / not too general

What to test?

Test **outcome** of the function! Outcome is also called **behavior.**

Clean functions are easier to test. The less lines and less logic you have in a function - the easier it is to test it.

Outcome might be a return value of a function, or state it modifies.

What NOT to test?

- Language features
- 3rd-party code
- Implementation

How many tests?

In the best case: *number of tests >= execution paths*

How to name test?

Name of the test should clearly specify what behaviour or rule it tests.

Name of the tests could be used as documentation for the system, to show what code is doing.

{MethodName}_{Scenario}_{ExecutionTest}

Fundamentals of Unit-tests

Trustworthy tests

Trustworthy tests are tests which test the right thing and will definitely catch a bug.

How to write trustworthy tests?

1) Write tests BEFORE the code. This is called TDD.
2) Intentionally introduce a bug and see if test fails.

Writing tests

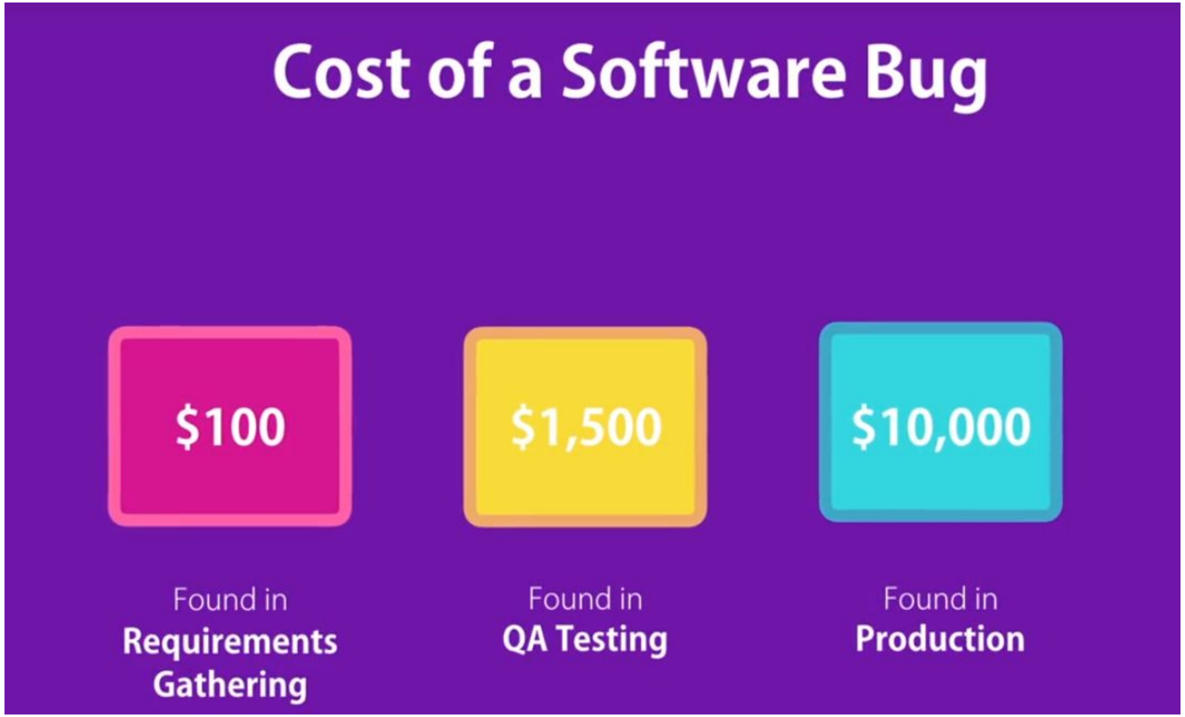
Never write a lot of tests because it is not worth to

vs.

All the code should have tests to it

Answer: **Middleground**

“We don’t have time for unit tests”



Software is **expensive**!

Client either pays more for quality software

or

pays less for bad quality software and then pays again for fixing bugs

Testing strings

Testing void methods

*Test **outcome** of the function!*

How to test “void” function? (function which does not return anything)

You should test “**side-effects**” of such functions.

Testing void methods

Test Coverage

When there are enough tests?

To answer this question developers usually set some kind of target how much of the code should be tested, e.g. 90%.

Code coverage shows how many execution paths are covered by unit tests.

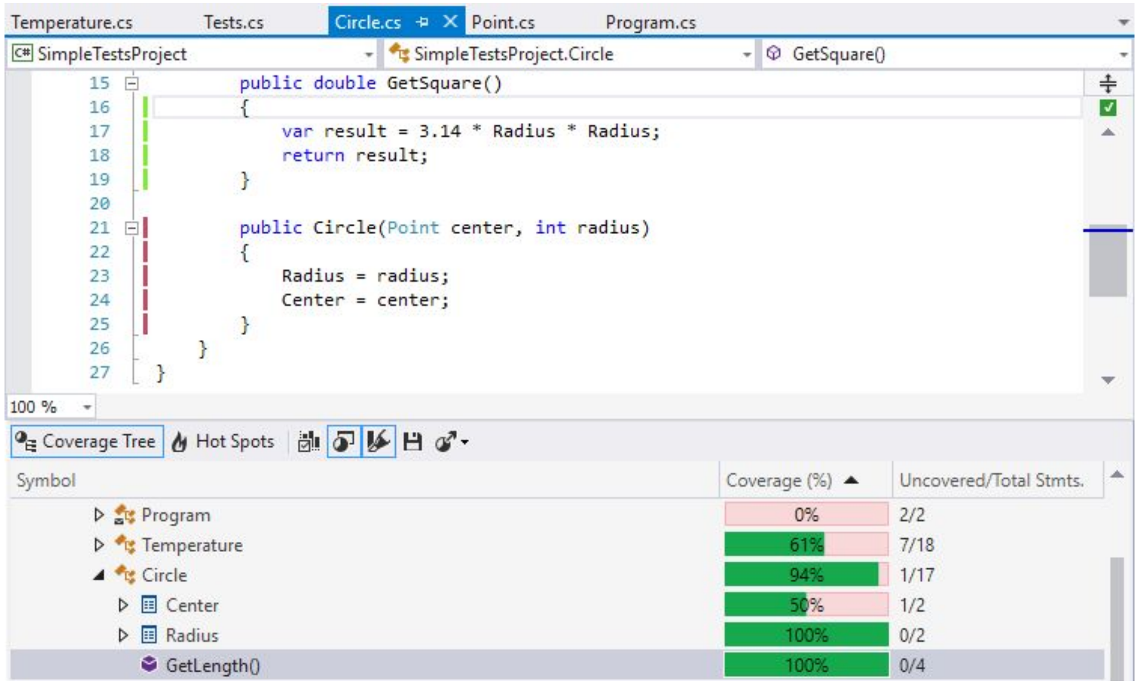
Using code coverage you can see what parts are not covered

Code coverage tools

…not free

- Visual Studio Enterprise (250$ / month)
- ReSharper dotCover (50$ / month)
- NCover (658$ / per user)

Code coverage demo



Temperature.cs | Tests.cs | Circle.cs | Point.cs | Program.cs

SimpleTestsProject | SimpleTestsProject.Circle | GetSquare()

```
        public double GetSquare()
        {
            var result = 3.14 * Radius * Radius;
            return result;
        }

        public Circle(Point center, int radius)
        {
            Radius = radius;
            Center = center;
        }
    }
}
```

100 %

Coverage Tree | Hot Spots

Symbol	Coverage (%)	Uncovered/Total Stmts.
Program	0%	2/2
Temperature	61%	7/18
Circle	94%	1/17
Center	50%	1/2
Radius	100%	0/2
GetLength()	100%	0/4

FringeBenefits

NOW: Your turn!

Write unit tests for: **FizzBuzz**!

- Think about this function as “Black Box”

- How many execution paths are there?
- How should you name them?

Write unit tests for: **DemeritPointsCalculator**!

- How many execution paths are there?
- How should you name them?

Write unit tests for:

- Stack
- PhoneNumber
- DateHelper