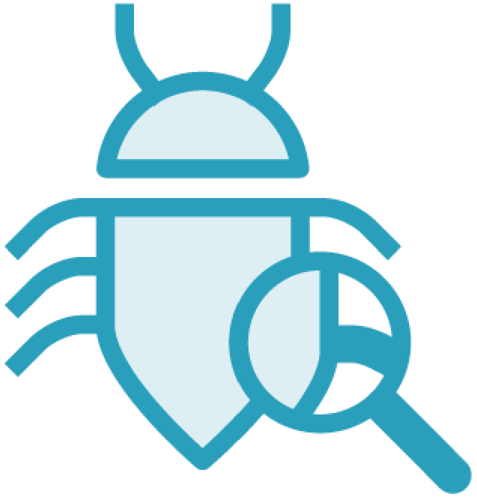
Writing Highly Maintainable Unit Tests

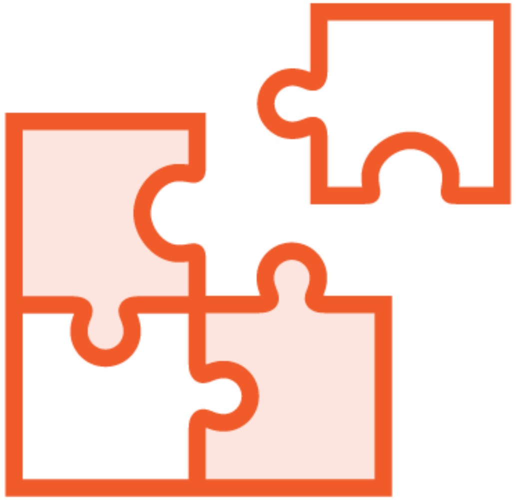
UNDERSTANDING PROVABLE CODE CORRECTNESS

Zoran Horvat
PRINCIPAL CONSULTANT AT CODING HELMET
@zoranh75 csharpmentor.com

Unit Test Defined



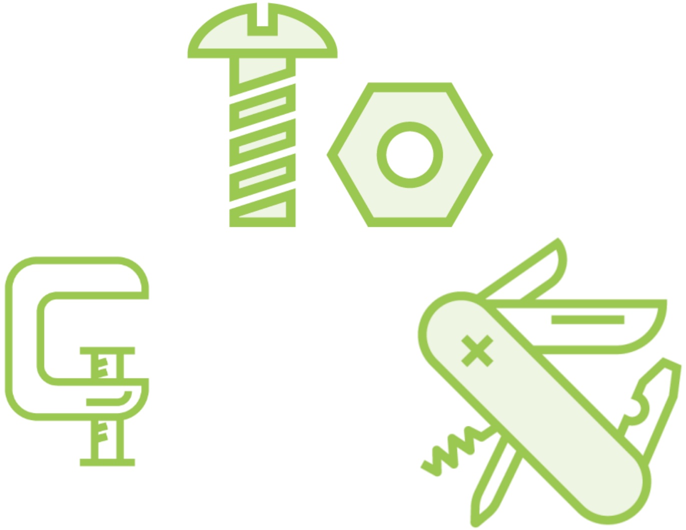
Is one claim about one piece of code true?



Piece of code = **Unit**

Checking the claim = **Test**

Maintainability Defined



Can we modify/extend code over time?

If not – adding a new feature becomes increasingly difficult

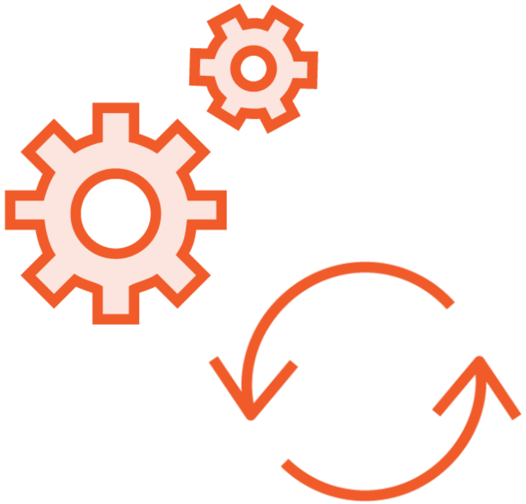
Fundamental Questions of Testing

What does it take to create a proper unit test?

How to make a large test suite maintainable in the long run?

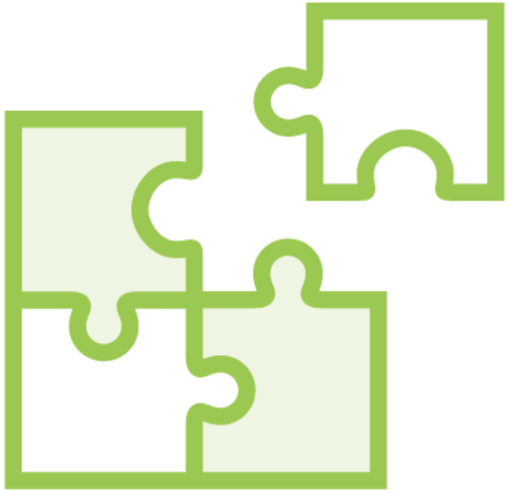
Unit Test Defined

Performed on
a method, a class,
a single instruction...



Must be automated

Has to run repeatedly



Unit + automated test

It proves a single claim
on a single piece of code

Unit Test Defined

Remove everything else to get hold of a unit

Remove all other claims about that unit

Now repeat this endlessly

(And keep it maintainable while you're there)

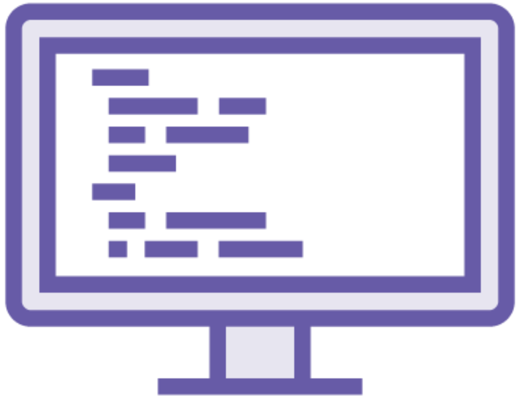
Prerequisites

Been writing unit tests for a while

Been using a testing framework

Speak OO language like a pro

Been maintaining a large project with tests



Examples will be in C#

Testing techniques should be universal

Importance of Maintainability

What if the application is hard to maintain?

What if it remains non-maintainable in the long run?

It will have to be replaced in the end

Customers will eventually abandon it

What Follows in This Course

Introducing a context within which classes are tested

One way of instantiating a class in tests

Another way of instantiating the same class in production

What Follows in This Course

Writing tests against implementation (a.k.a. white box testing)

Proving that concrete method implementation is right

This practice may lead to non-maintainable tests

Anyway, white box testing has its application

What Follows in This Course

Managing class dependencies in unit tests

Dependency injection (DI) is a commonplace today

Still, dependencies have a profound effect on tests

Poorly designed dependencies lead to poor tests

What Follows in This Course

Introducing Abstract Data Types (ADTs)

ADT defines abstract behavior

Class defines concrete implementation

Start writing tests against abstract behavior

What Follows in This Course

Putting generic classes under test

Generic derivation is similar to common derivation

Yet it has some qualities of its own

Testing generic classes may get complicated

What Follows in This Course

Putting entire concepts under test

Class under test, method under test, etc.

Why not design pattern under test?

Why not value-typed semantic under test?

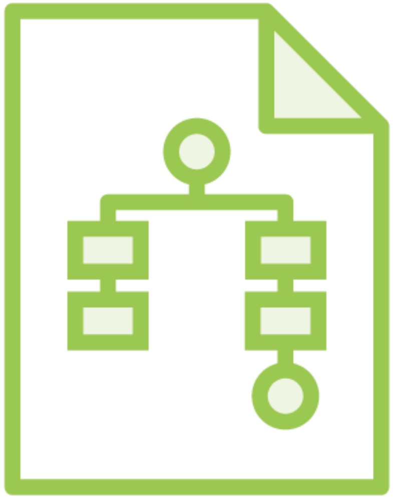
What Follows in This Course

Introducing Design by Contract (DbC)
and letting it affect unit tests

Tests can force the code to
follow principles of DbC

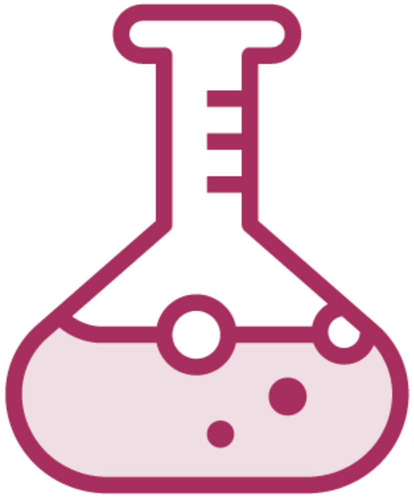
We will request the code
to be provably correct

Why Know so Much About Unit Testing?



Testing frameworks,
isolation frameworks,
test runners...

They make little difference



Design of tests,
design of the application...

Well, that can make a difference

```
int Maximum(int[] values)
{
  int max = values[0];

  for (int i = 1;
       i < values.Length;
       i++)
  {
    if (values[i] > max)
    {
      max = values[i];
    }
  }

  return max;
}
```

- **Finds greatest value in the array**
  Assume values array is non-null
  Assume values array contains at least one element
- **Can we offer a formal proof that this function works correctly?**
  Yes, we can offer a proof based on induction

```
int Maximum(int[] values)
{
  int max = values[0];

  for (int i = 1;
       i < values.Length;
       i++)
  {
    if (values[i] > max)
    {
      max = values[i];
    }
  }

  return max;
}
```

$max \leftarrow Max\{k = 0 | values[k]\}$

Entire loop skipped if
`values.Length = 1`

$N = values.Length$

$max \leftarrow Max\{0 \leq k < N | values[k]\}$

$Q.E.D.$

```
int Maximum(int[] values)
{
  int max = values[0];

  for (int i = 1;
       i < values.Length;
       i++)
  {
    if (values[i] > max)
    {
      max = values[i];
    }
  }

  return max;
}
```

$max \leftarrow Max\{k = 0|values[k]\}$

$N = values.Length > 1$

Loop invariant (which must hold true):

$max \leftarrow Max\{0 \leq k < i|values[k]\}$

True when we enter the loop for $i = 1$

$max \leftarrow Max\{0 \leq k < i + 1|values[k]\}$

$N = values.Length$

$max \leftarrow Max\{0 \leq k < N|values[k]\}$

$Q.E.D.$

```
int Maximum(int[] values)
{
  int max = values[0];

  for (int i = 1;
       i < values.Length;
       i++)
  {
    if (values[i] > max)
    {
      max = values[i];
    }
  }

  return max;
}
```

- **What if the input array could be empty as well?**
  We must prove that we are only accessing the array within its bounds

```
int Maximum(int[] values)
{
  int max = values[0];

  for (int i = 1;
       i < values.Length;
       i++)
  {
    if (values[i] > max)
    {
      max = values[i];
    }
  }

  return max;
}
```

$$withinBounds = \text{true}$$

$$N = values.Length$$

$$withinBounds = \cancel{(oldWithinBounds = \text{true})}\ \textbf{AND}\ (\cancel{0 \leq}\ 0 < N)$$

```
int Maximum(int[] values)
{
  int max = values[0];

  for (int i = 1;
       i < values.Length;
       i++)
  {
    if (values[i] > max)
    {
      max = values[i];
    }
  }

  return max;
}
```

$withinBounds = \text{true}$

$withinBounds = 0 < values.Length$

$1 \le i$

$$withinBounds = (oldWithinBounds = \text{true})\ \textbf{AND}\ (0 \le i < N)$$

```
int Maximum(int[] values)
{
  int max = values[0];

  for (int i = 1;
       i < values.Length;
       i++)
  {
    if (values[i] > max)
    {
      max = values[i];
    }
  }

  return max;
}
```

$$withinBounds = \text{true}$$

$$withinBounds = 0 < values.Length$$

$$1 \leq i < values.Length$$

$$withinBounds = (oldWithinBounds = \text{true})$$

$$withinBounds \Leftrightarrow values.Length > 0$$

```
int Maximum(int[] values)
{
  Debug.Assert(values.Length > 0);
  int max = values[0];
  for (int i = 1;
       i < values.Length;
       i++)
  {
    if (values[i] > max)
    {
      max = values[i];
    }
  }

  return max;
}
```

`Debug` **class from** `System.Diagnostics`

```
int Maximum(int[] values)
{
  Debug.Assert(values.Length > 0);
  int max = values[0];
  for (int i = 1;
       i < values.Length;
       i++)
  {
    if (values[i] > max)
    {
      max = values[i];
    }
  }

  return max;
}
```

$$
\begin{aligned}
withinBounds &= 0 < values.Length \\
&= \text{true}
\end{aligned}
$$

$$
\begin{aligned}
withinBounds &\Leftrightarrow values.Length > 0 \\
&= \text{true}
\end{aligned}
$$

```csharp
int Maximum(int[] values)
{
  Debug.Assert(values != null);
  Debug.Assert(values.Length > 0);
  int max = values[0];
  for (int i = 1;
       i < values.Length;
       i++)
  {
    if (values[i] > max)
    {
      max = values[i];
    }
  }
  return max;
}
```

- **More on assertions in the last module of this course**

  Assert will kill the process if its condition is false

  And our goal will be to never have a false condition to assert

- **This function will never fail**
- **Return value will always be correct**

```
int Maximum(int[] values)
{
  Debug.Assert(values != null);
  Debug.Assert(values.Length > 0);
  int max = values[0];
  for (int i = 1;
       i < values.Length;
       i++)
  {
    if (values[i] > max)
    {
      max = values[i];
    }
  }
  return max;
}
```

- **Why not formally prove that the entire code base is correct?**

  Because it gets complicated with growing code complexity

- **This function is short and simple**

  Still, adding another loop would complicate the proof

  Call to this function combines with other code

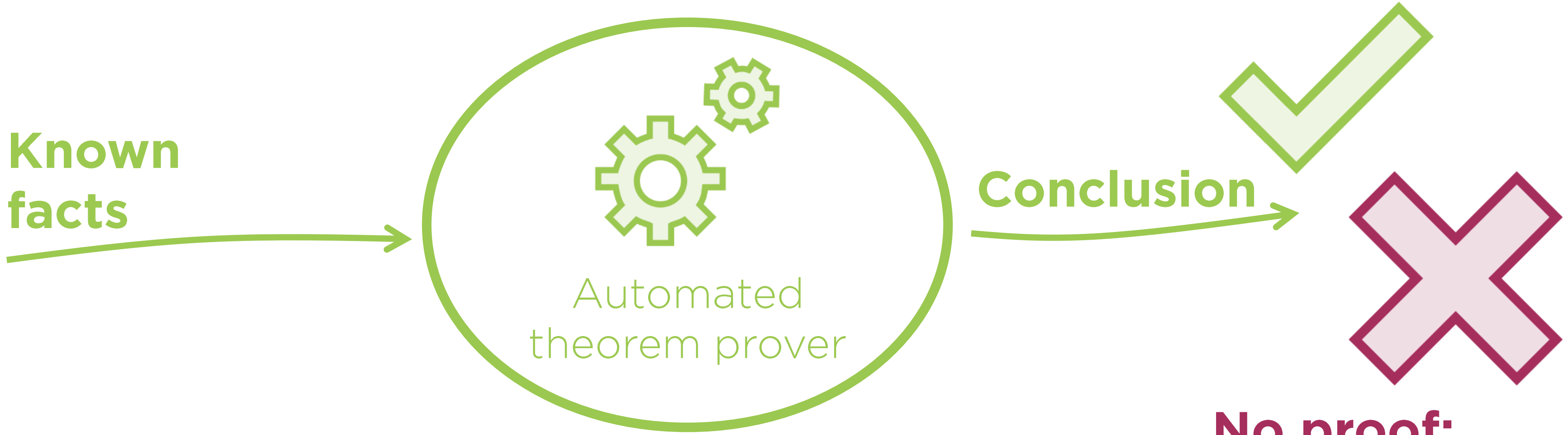
  Number of Boolean conditions to track grows exponentially

Tool Support Today

`obj.Method()`

Theorem to prove:
`obj` is never `null`

Proof found:
We can safely access the reference



No proof:
Reference *might* be null sometimes

Tools can detect accessing null
Testing access within bounds is harder
Other test get computationally very hard

```
int Maximum(int[] values)
{
  int max = values[0];

  for (int i = 1;
       i < values.Length;
       i++)
  {
    if (values[i] > max)
    {
      max = values[i];
    }
  }

  return max;
}
```

- **Test case #1**

  `Maximum([5]) = 5`

- **Test case #2**

  `Maximum([1, 2, 3, 5, 4]) = 5`

- **Boundary tests**

  `Maximum([5, 1, 2, 3, 4]) = 5`

  `Maximum([1, 2, 3, 4, 5]) = 5`

Proving vs. Testing Correctness

Are green tests proof the code is correct?

No!

Tests are not proving correctness of code

Tests are demonstrating that code runs as expected for certain input

```
int Maximum(int[] values)
{
  int max = values[0];

  for (int i = 1;
       i < values.Length;
       i++)
  {
    if (values[i] > max)
    {
      max = values[i];
    }
  }

  return max;
}
```

- **Test cases**

```
Maximum([5]) = 5
Maximum([1, 2, 3, 5, 4]) = 5
Maximum([5, 1, 2, 3, 4]) = 5
Maximum([1, 2, 3, 4, 5]) = 5
```

- **Four cases is small compared to number of all possible arrays**

  Real number of cases is virtually infinite

```
int Maximum(int[] values)
{
    return 5;
}
```

- **Test cases**

```
Maximum([5]) = 5
Maximum([1, 2, 3, 5, 4]) = 5
Maximum([5, 1, 2, 3, 4]) = 5
Maximum([1, 2, 3, 4, 5]) = 5
```

- **Four cases is small compared to number of all possible arrays**

  Real number of cases is virtually infinite

- **Tests may pass for no reason**

  Function with a defect can still satisfy tests by pure chance

  It takes a lot of debugging to find such bug

Testing vs. Proving Code Correctness

Tests do not prove code correctness

Logical inference is a formal proof of code correctness

But we can choose testing points wisely

Well-selected tests can uncover bugs

Following Examples

We will devise tests, rather than proofs

Formal logic helps select complete set of tests

E.g. boundary tests

Summary

Approach #1: Prove code correctness

- Run in production without fear
- We know the code will never fail
- Not realistic for large code

Approach #2: Write tests

- Tests do not *prove* correctness
- Tests run the code and try it

Summary

Important aspects of testing

- How to select test cases
- How to automate tests

What do we want?

- Well defined tests
- Automated tests

Next module:
Deciding what to test