The Importance of Knowing What to Test

Zoran Horvat

PRINCIPAL CONSULTANT AT CODING HELMET

@zoranh75 csharpmentor.com

“There’s a saying of Goethe’s which
Mr. Magee likes to quote.
Beware of what you wish for in youth
because you will get it in middle life.”

James Joyce, Ulysses

“Beware of what you add to the design today, because that is what you will maintain tomorrow.”

Anonymous

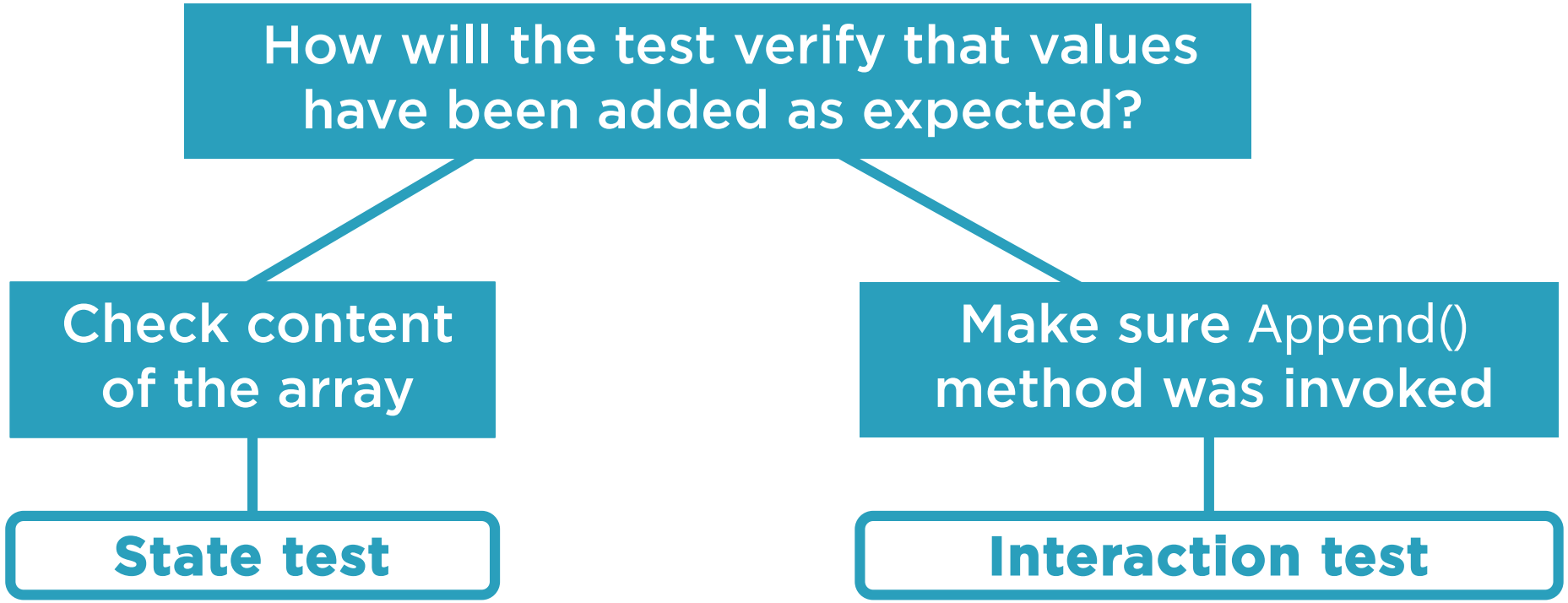
Tests are a liability

- They are limiting our freedom to change production code later

Make right decisions on what to test

- That will incur maintenance savings later

```
void AddTargetPoints(MyArray toArray, int count)
{
  for (int i = 0; i < count; i++)
  {
    toArray.Append(3 + 2*i);
  }
}
```



```
public class MyArray
{
  private int[] Data { get; set; }
  public int Length =>
    this.Data.Length;

  public MyArray()
  {
    Data = new int[0];
  }

  public void Append(int value)
  {
    int[] data = this.Data;
    Array.Resize(ref data,
                 data.Length + 1);
    data[data.Length - 1] = value;
    this.Data = data;
  }
}
```

- **How do we implement a state test for this class?**
  1. Perform the operation
  2. Read state after the fact
  3. Compare to expected state
- **The class must expose (part of) its state**

  How does this relate to the encapsulation principle?

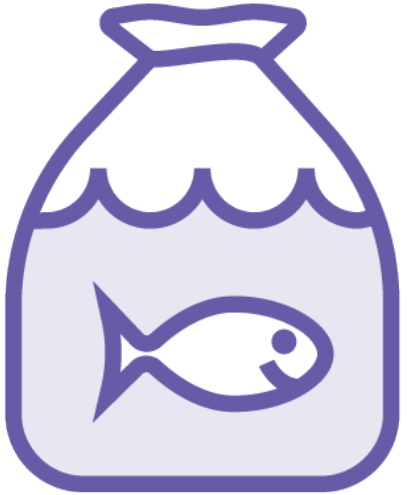
  Encapsulation doesn't mean the state is inaccessible

  It prevents indiscriminate exposure of state

Liberal View on Encapsulation

Allow both read and write on state

Throw exceptions from property setters



Keep the object consistent all the time

Rigid View on Encapsulation

Turn all fields and properties private

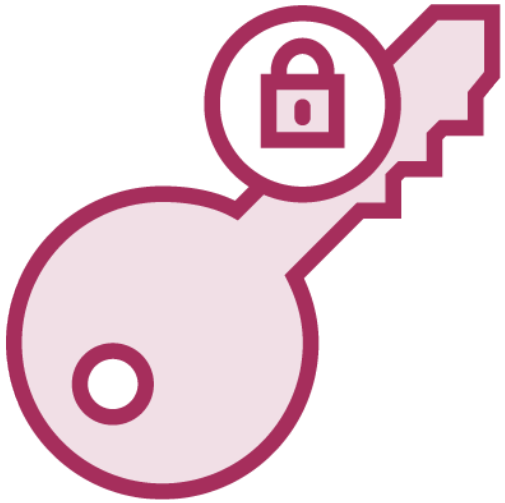
Only expose operations

Object decides how to perform an operation

Balanced View on Encapsulation

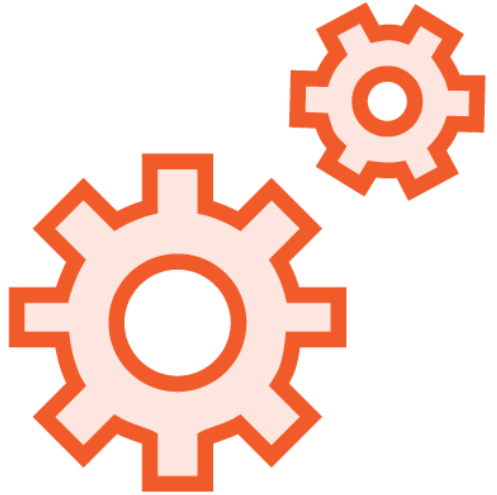
Public State

Others pick state from an object when they need an operation



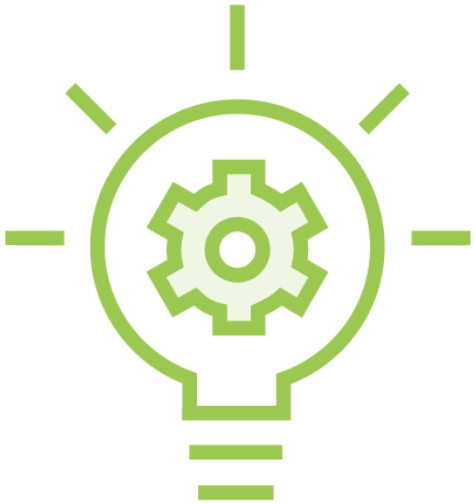
Private state

Others call the object when they need an operation

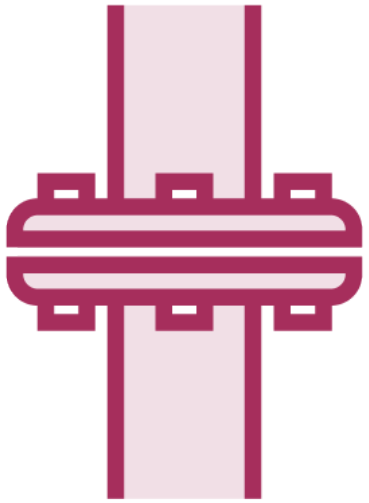


Both extremes cause other classes to tightly couple to the class containing state

Balanced View on Encapsulation



Expose limited amount of state



Let others implement operations with least amount of coupling

Try to maintain classes with that much state exposed

```
public class MyArray
{
  public int Length { get; }
  // Implementation based
  // on common array
}
```

Much later...

```
public class MyArray
{
  public int Length { get; }
  // Implementation based
  // on linked list
}
```

- **Imagine an array class exposing the** Length **property**

- **We have to maintain the** Length **property**

  New implementation may have to count elements

  Length does not exist as an explicit state anymore!

  Now we have to provide backward compatible Length

Public vs. Internal Test-related State

Public State

Forces you to make
a well designed class

Internal State

Looks like you can get
away with poor design

Better go with public state straight away and try to do that well

```
// Arrange
...
// Act
...
// Assert
Assert.AreEqual(3, array[0]);
Assert.AreEqual(5, array[1]);
```

- **Test method with two assertions**
  Common wisdom says that one unit test should have one assertion

```
Assert.NotNull(array);          ← fail!
Assert.AreEqual(3, array[0]);   ← inconclusive
```

- **A different claim to assert**
  Test variable against null
  Test variable's content
- **What if the first assertion fails?**
  The second assertion might pass
  Or it might fail as well
  The second test is inconclusive
- **Avoid multiple assertions if they are testing unrelated claims**

Test #1

```
// Arrange
...
// Act
...
// Assert
Assert.NotNull(array);
```

Test #2

```
// Arrange
...
// Act
...
// Assert
Assert.AreEqual(5, array[1]);
```

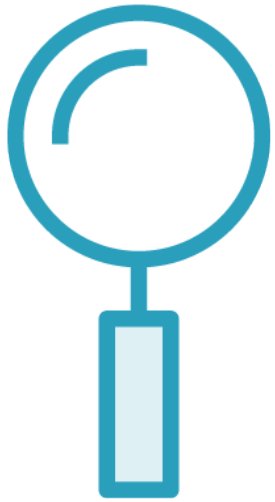
- **Separate test methods**

- **This test only asserts the value**

  But what if array is null?

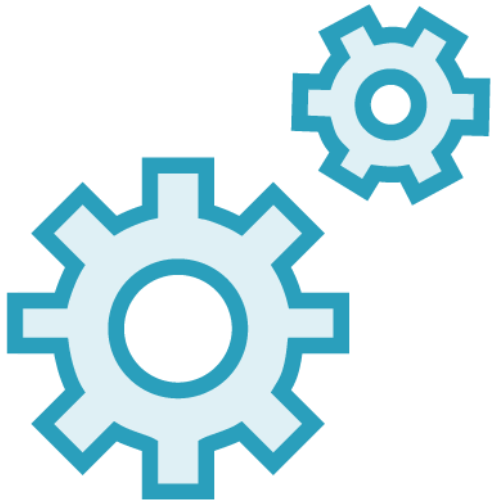
  It will fail with unhandled NullReferenceException

Evaluating Usefulness of Tests



State tests

Passing



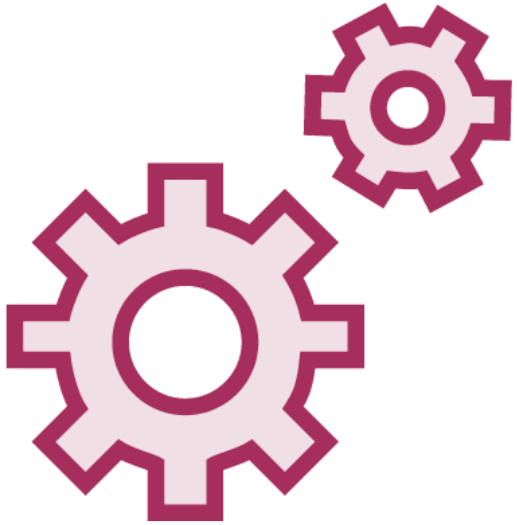
Interaction tests

Passing

Now which of these are better for us?

Passing Test Equals... What?

Test is coding an expectation



Code works in respect to expectation

The code may still be correct or incorrect

Tests are an executable documentation

The Role of Tests During Maintenance

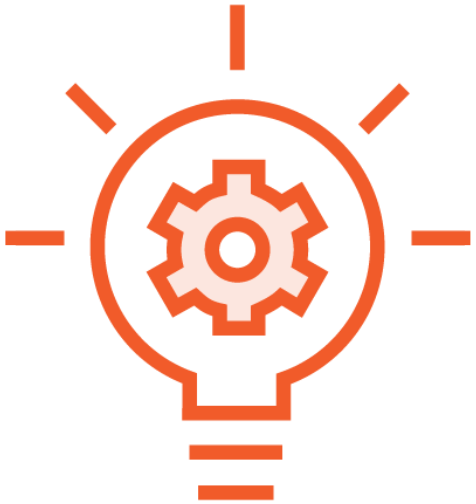
Will the test pass after code modification?

Failing test means regression

Failing test still doesn't mean we have a bug

Now this is weird!

The Purpose of Interaction Tests



Wrong

Test assumes certain implementation

Right

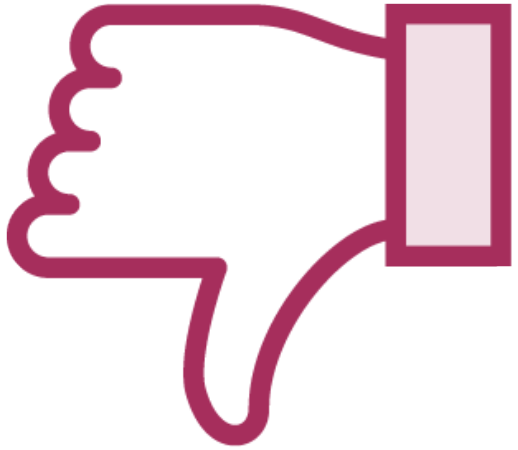
Test proves that certain mandatory interaction occurs

Useful when interaction itself is a requirement

The Purpose of Interaction Tests

E.g. a test which proves that a dependency was used

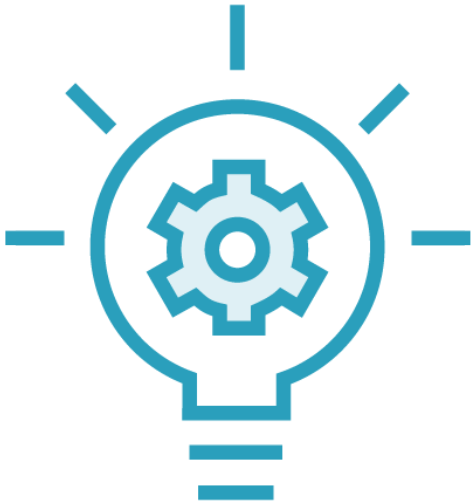
Implementation may change and stop using that dependency



The test will fail

But implementation is correct!

The Role of Tests



Not assume any particular implementation

Verify effects of an operation

Not stand in the way of a changing implementation

Summary

How to choose what to test?

First turn the class testable

- Expose (part of the) state
- Make calls interceptable
  - Make certain methods overridable
  - Implement an interface

Different test implementations

- State test
- Interaction test through class derivation
- Interaction test through interface

Summary

Resilience to change

- Class under test will change later

Interaction tests are fragile when production code is changing

- Changes to code may affect interactions
- Interaction test which assumes certain interaction is an over-specified test

General rules

- Use interaction tests when concrete interaction is the requirement
- Use state tests to test effects

Next module:
White-box testing