C01

Testing Compliance to Design Principles

Zoran Horvat
PRINCIPAL CONSULTANT AT CODING HELMET

@zoranh75 csharpmentor.com

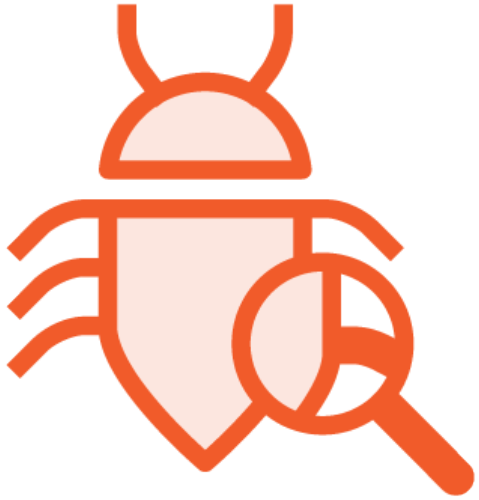
Testing Disposable Pattern Implementation

No assert on the SUT

Testing Disposable Pattern Implementation

No assert on the SUT

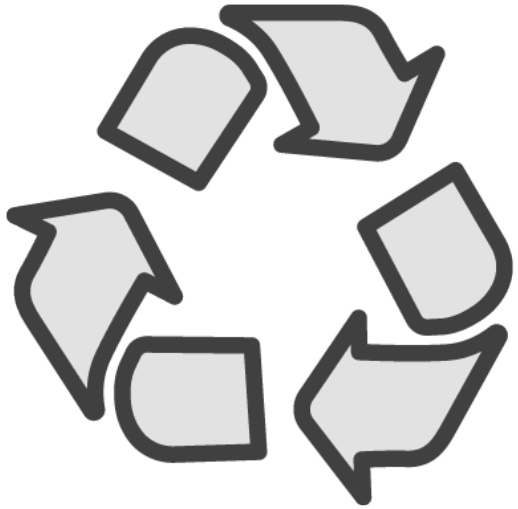
Assert interaction with dependency



Failing to call Dispose() is the bug

No more interaction with dependency beyond Dispose()

Disposable Pattern



Garbage collection

We never know ***when*** and ***if*** a resource will be released

Explicit release

Connections, handles, etc. disposed explicitly

Implementation

Implement IDisposable

Call Dispose()

Constraints

No calls on an object after Dispose()

Disposable Pattern

Throw ObjectDisposedException from method calls after Dispose() has been invoked

InvalidOperationException

↓

ObjectDisposedException

C04

```
DateTime date = DateTime.Today;

if (this.Target == someday.Date)
{
  ...
}

birthday.Date

dueDate.Date

thisDay.Date = thatDay.Date

...

class Date
{
  ...
}
```

- This all looks like we are lacking an abstraction
  - Reduce typing
  - Reduce errors from repeated code

```
public class Date
{
  public int Year => FullTime.Year;
  public int Month => FullTime.Month;
  public int Day => FullTime.Day;

  private DateTime FullTime { get; }

  public Date(int year, int month,
              int day)
  {
    FullTime =
      new DateTime(year, month, day);
  }

  public Date(DateTime d)
   : this(d.Year, d.Month, d.Day) { }

  public Date AddDays(int days) =>
      new Date(FullTime.AddDays(days));
}
```

- Sample date implementation

- Wraps around common DateTime

  This class will demonstrate how to use and test value types

```
int x;
int y = x + 5
```

- Use integer as a simple value

```
Date today;
Date nextWeek = today.AddDays(5);
```

- Use a date as a simple value
  Original value does not change
  Values are immutable

```
Car car = new Car();
```

- Entity differs from a value
  Long-lived entity has a persistent ID
- No identity in a value
  Entire content identifies a value

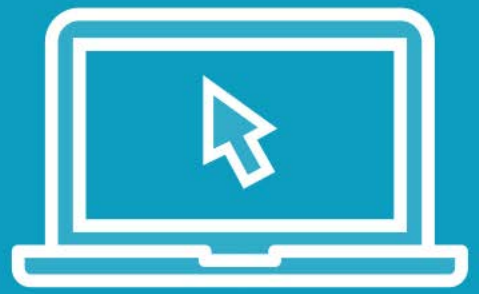
```
if (x == 5) ...

if (Date.Today == birthday) ...
```

- Values can be tested for equality
  This does not stand for entities
  year, month, day – all three must be equal in equal dates

C06

Value-Typed Equality



Override Equals() from System.Object

Override GetHashCode() from System.Object

- Equal objects to return same hash code

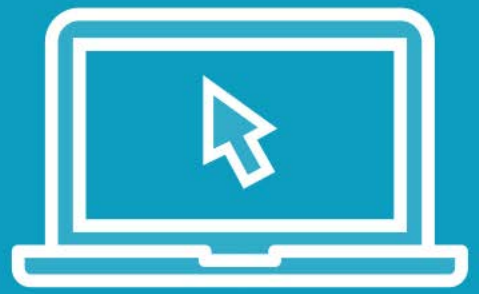
Override equality operator (==)

- Otherwise, typing a == b would be a bug

Override inequality operator (!=)

Implement IEquatable<T>

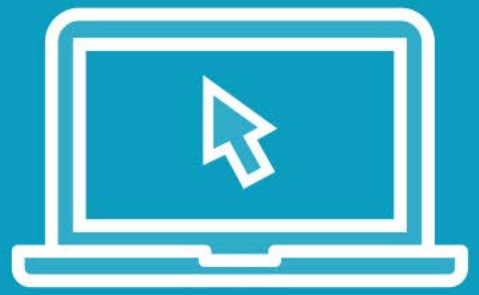
Value-Typed Equality



```
class A
{
  int field1;
}

class B : A
{
  // int field1; -- inherited
  int field2;
}
```

Value-Typed Equality



```
class A
{
  int field1;
}

class B : A
{
  // int field1; -- inherited
  int field2;
}
```

Comparing objects a and b

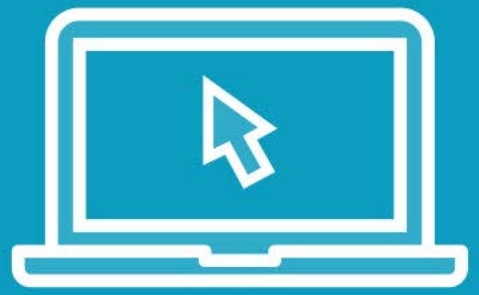
a.Equals(b) → True

b.Equals(a) → False

Algebraic equality definition

1. Reflexive: a = a
2. Symmetric: a = b if and only if b = a

Value-Typed Equality



```
class A
{
  int field1;
}

class B : A
{
  // int field1; -- inherited
  int field2;
}
```

Comparing objects a and b

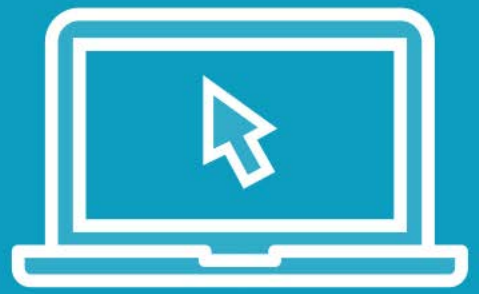
a.Equals(b) → True

b.Equals(a) → False

Algebraic equality definition

1. Reflexive: a = a
2. Symmetric: a = b if and only if b = a **(Broken!)**
3. Transitive: a = b and b = c implies a = c

Value-Typed Equality



Override Equals() from System.Object

Override GetHashCode() from System.Object

- Equal objects to return same hash code

Override equality operator (==)

- Otherwise, typing a == b would be a bug

Override inequality operator (!=)

Implement IEquatable<T>

Class must be sealed

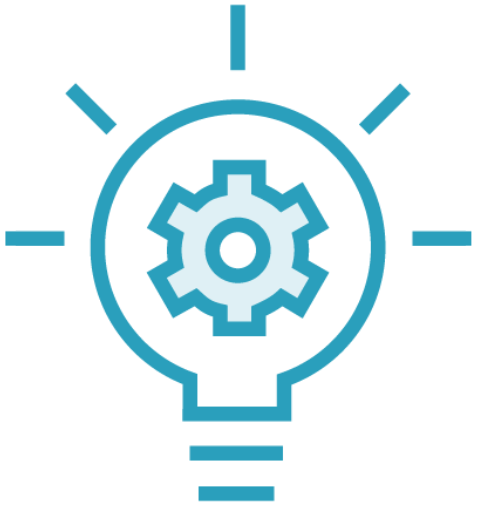
Object-level requirements

- Equals() – both variants, GetHashCode()
- a.Equals(null) = False, a != null
- null == null

C08

Writing Own Test Libraries

Isolate the concept you are testing



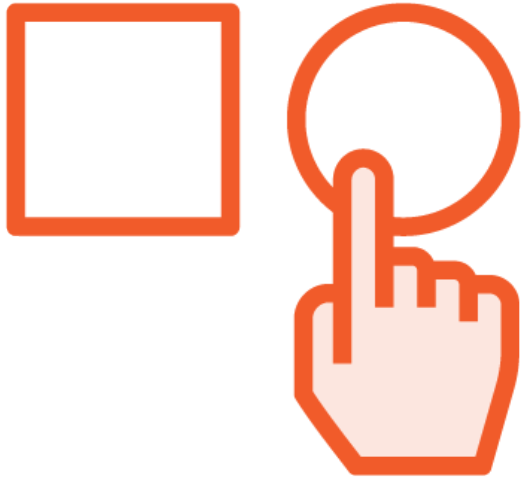
Encapsulate the rules into a reusable component

Clear the testing component of bugs

Clear it once

Keep the Unit Tests Straight-Forward

Wrap complexity in a reusable library



Let the library enforce straight-forward interface on unit tests

And then tests will be simple once again

C09

Summary

Testing design principles

- Design tests as for any other behavior

Example: Testing Disposable pattern

- Define rules of the concrete pattern implementation
- Embed rules in unit tests

Testing code reuse

- Keep testing rules in the base class, then derive from it to specialize SUT
- Keep testing rules in separate class, then use it as a component on a SUT

Summary

Example: Testing value-typed equality

- 29 rules were built dynamically
- The system of rules implemented as a reusable component
- General rules, not related to one SUT
- Encapsulate complexity in the testing component

GitHub repo with equality testing library

https://github.com/zoran-horvat/value-type-tests

Next module:
Covering negative scenarios