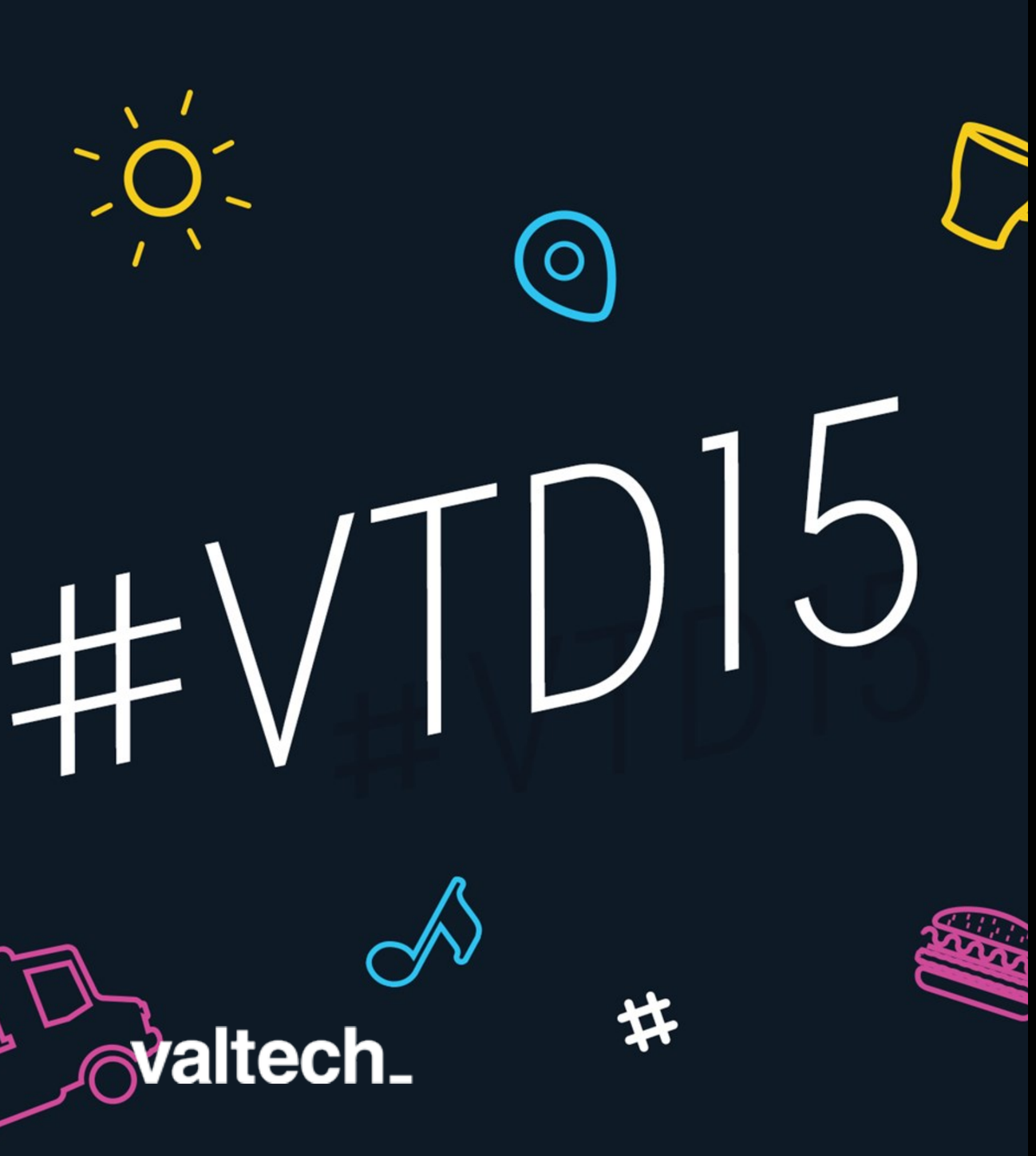


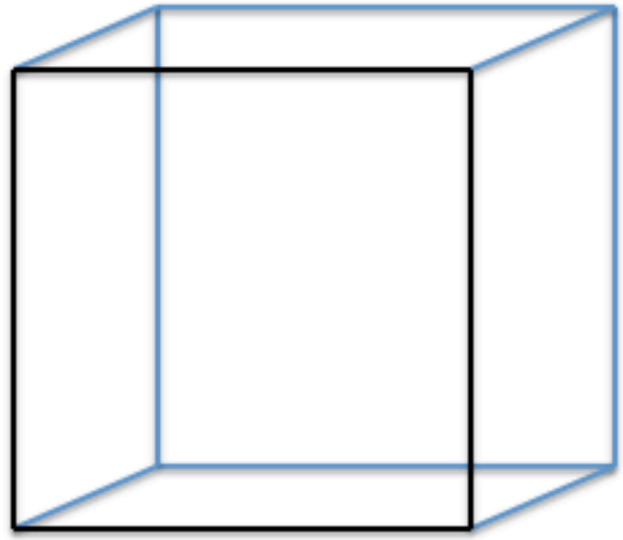
Property-Based Testing with F

Mikael Lundin

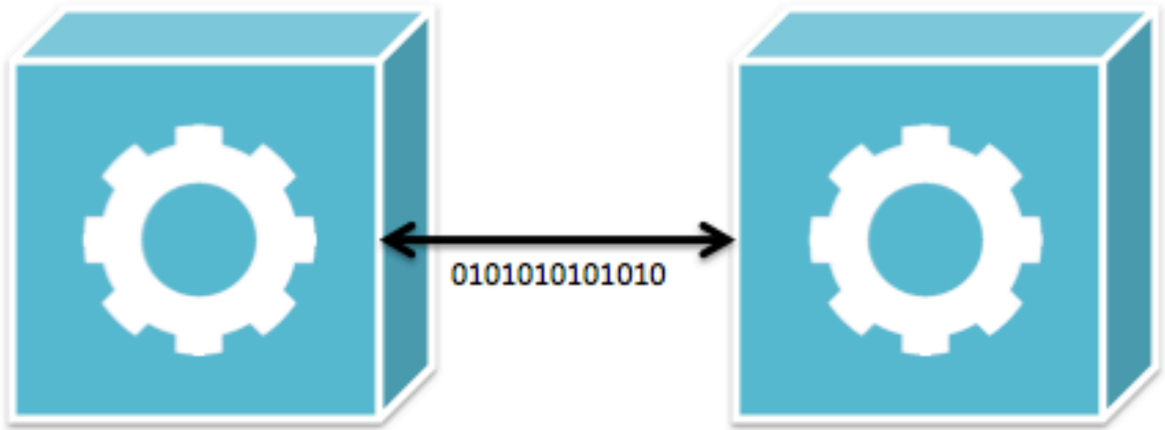
Unit Testing

Specifies what the code is supposed to do.

```
public class CommitTransactionShould
{
   [Test]
   public void SetStatusToCommitted()
   {
      // AAA
   }
}
```



Integration Testing



Specifies how the system fits together.

```
public class SaveToDatabseShould
{
  [Test]
  public void ReturnEntityWithAutoIncrementedID()
  {
    // AAA
  }
}
```

Functional Testing

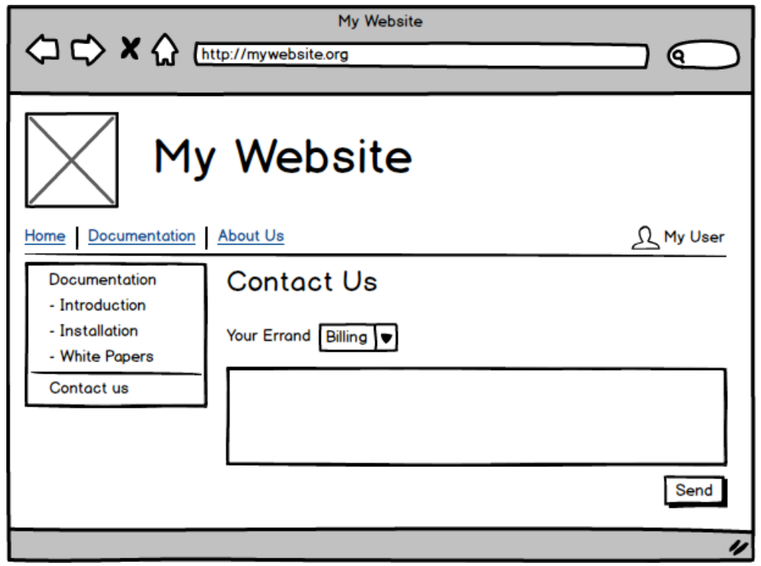
Specifies features the functionality will sustain.

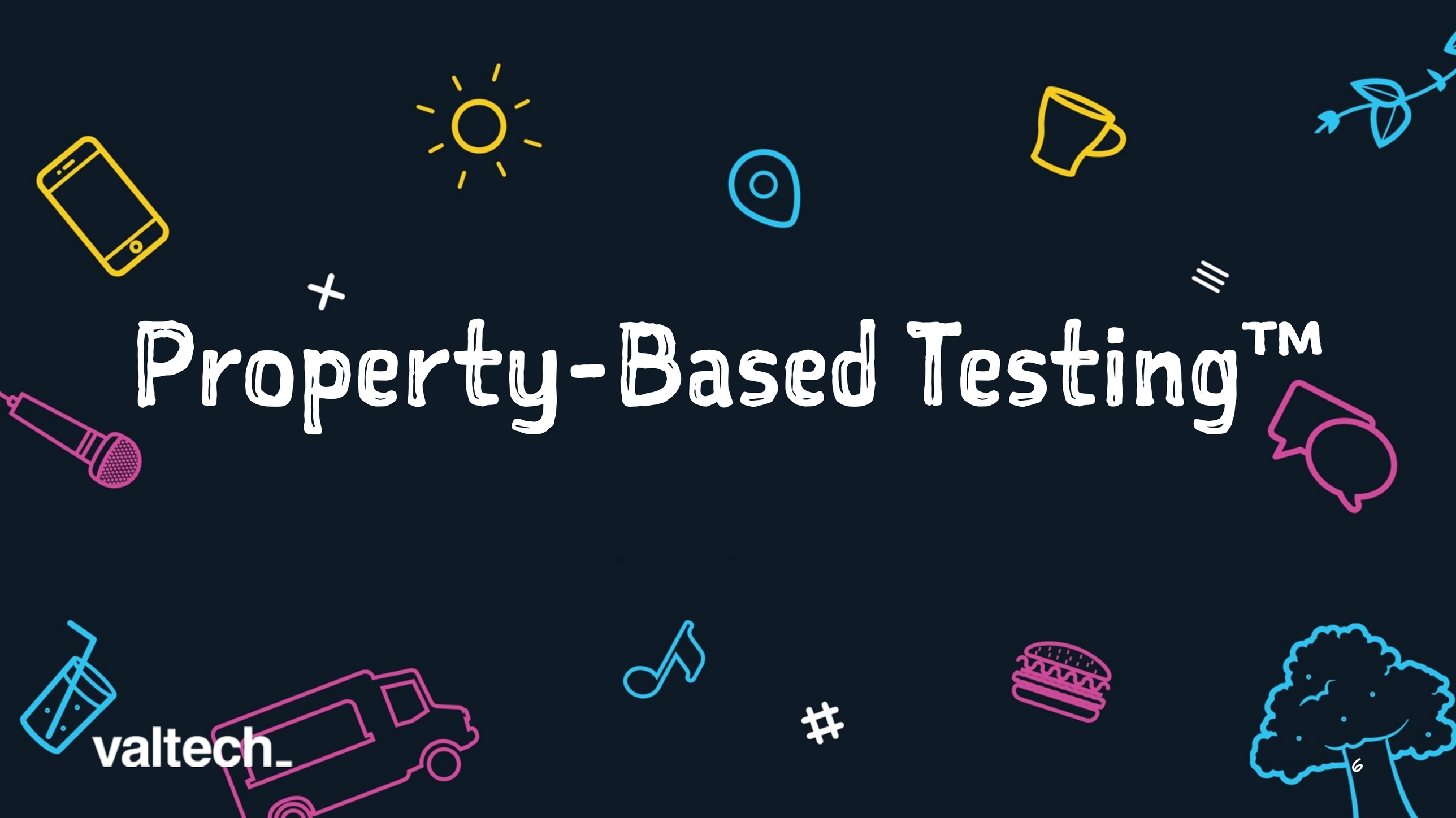
```
Feature: Order Checkout

Scenario: Pushing order button gives confirmation with order number
  Given an order at the order checkout page
  And invoice is selected as payment option
  When order button is clicked
  Then order confirmation should be displayed
  And order number should be visible
```

Manual Testing

How well does this feature solve the problem?

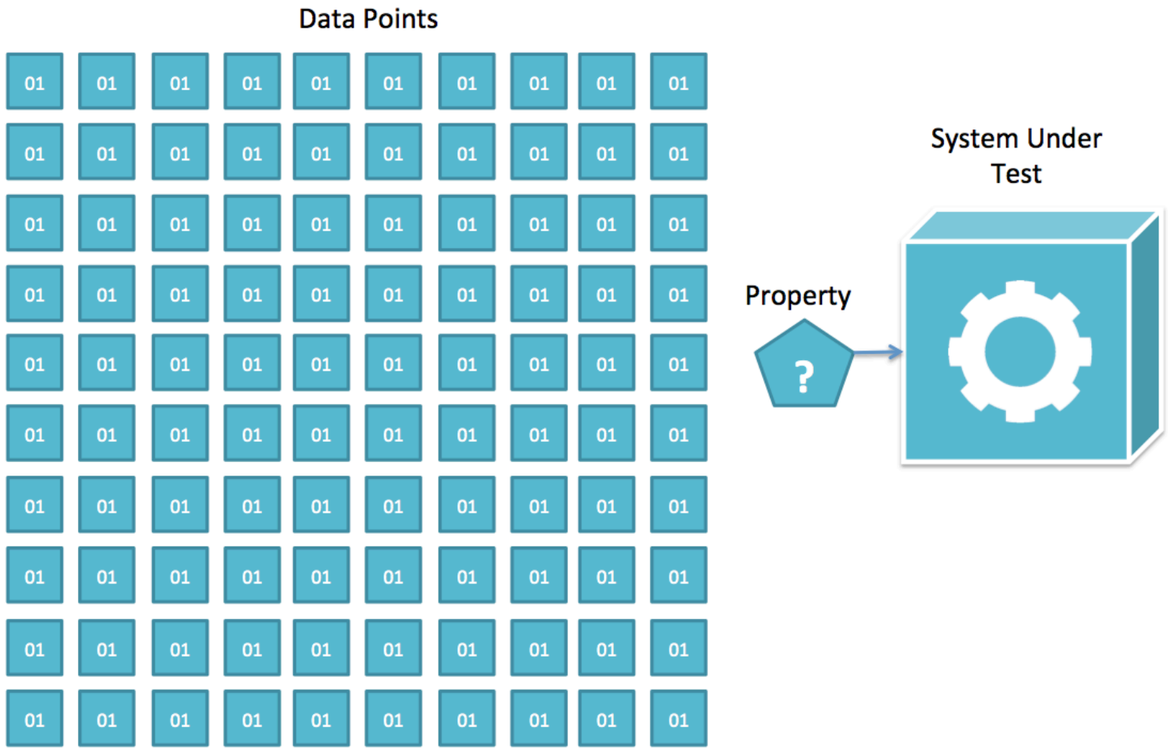



Property-Based Testing™
valtech_
6

Property

High-level specification of behavior that should hold for a range of data points.

Property



Example

Sorting

```
[Test]
public void ShouldReturnListOfNumberInOrder()
{
  // setup
  var data = new [] { 1, 13, 5, 8, 2, 3, 1 };

  // test
  var result = ArrayUtils.BubbleSort(data);

  // assert
  Assert.That(result, Is.EqualTo(new [] { 1, 1, 2, 3, 5, 8, 13}));
}
```

Example

Sort-properties

- Sorting the list once = sorting the list twice
- First item is smallest
- Last item is largest
- All items are present from original list

Bubblesort

```
public static T[] BubbleSort<T>(T[] input) where T : IComparable
{
    var result = new T[input.Length];
    input.CopyTo(result, 0);

    T temp = result[0];

    for (int i = 0; i < result.Length; i++)
    {
        for (int j = i + 1; j < result.Length; j++)
        {
            if (result[i].CompareTo(result[j]) > 0)
            {
                temp = result[i];
                result[i] = result[j];
                result[j] = temp;
            }
        }
    }

    return result;
}
```

Properties

- Sorting the list once = sorting the list twice

Install FsCheck

```
Install-Package FsCheck
```

Implement property

```
open PropertyBasedTesting.CSharp

let ``Sorting the list once = sorting the list twice`` (input : int array) =
    ArrayUtils.BubbleSort input = ArrayUtils.BubbleSort(ArrayUtils.BubbleSort input)
```

Verify the property

```
open FsCheck

Check.Quick ``Sorting the list once = sorting the list twice``
```

Result

```
Falsifiable, after 3 tests (0 shrinks) (StdGen (1204045486,296013961)):
[||]
with exception:
System.IndexOutOfRangeException: Index was outside the bounds of the array.
```

Alas! A bug.

```
public static T[] BubbleSort<T>(T[] input) where T : IComparable
{
    if (input.Length == 0)
    {
        return input;
    }

    var result = new T[input.Length];
    input.CopyTo(result, 0);

    T temp = result[0];

    for (int i = 0; i < result.Length; i++)
    {
        for (int j = i + 1; j < result.Length; j++)
        {
            if (result[i].CompareTo(result[j]) > 0)
            {
                temp = result[i];
                result[i] = result[j];
                result[j] = temp;
            }
        }
    }

    return result;
}
```

Verify the property

```
open FsCheck

Check.Quick ``Sorting the list once = sorting the list twice``
```

Result

```
Ok, passed 100 tests.
```

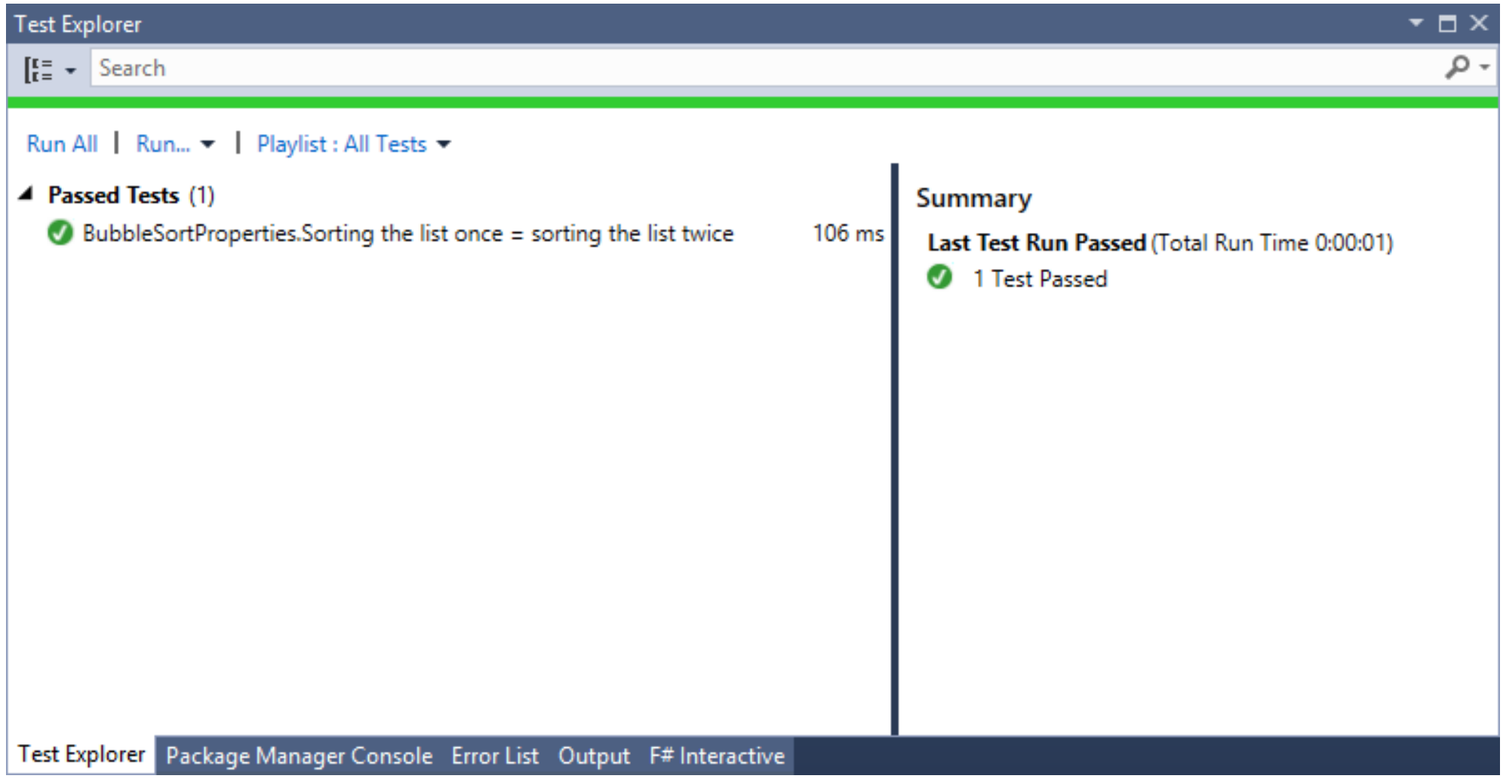
Make it reproducible

```
Install-Package FsCheck.xUnit
Install-Package Xunit.runner.visualstudio
```

```
open FsCheck.Xunit
open PropertyBasedTesting.CSharp

[<Property>]
let ``Sorting the list once = sorting the list twice`` (input : int array) =
    ArrayUtils.BubbleSort input = ArrayUtils.BubbleSort(ArrayUtils.BubbleSort input)
```

Run it as a standard unit test



Conditions

```
// Check.Quick ``First item is smallest``
[<Property>]
let ``First item is smallest`` (input : int array) =
    (input.Length) > 0 ==>
        lazy((ArrayUtils.BubbleSort input).[0] = Seq.min input)

// Check.Quick ``Last item is largest``
[<Property>]
let ``Last item is largest`` (input : int array) =
    (input.Length) > 0 ==>
        lazy((ArrayUtils.BubbleSort input).[input.Length - 1] = Seq.max input)
```

Named properties

```
// Check.Quick ``Has same numbers as input list``
[<Property>]
let ``Has same numbers as input list`` (input : int array) =
    let sorted = ArrayUtils.BubbleSort input

    (sorted.Length = input.Length)
        |@ "same length" .&.
    (sorted |> Seq.forall (fun x -> Seq.exists ((=) x) input))
        |@ "all elements exists"
```

Another example

```
// Check.Quick ``Is ordered``
[<Property>]
let ``Is ordered`` (input : int array) =
    ArrayUtils.BubbleSort input
        |> Seq.pairwise |> Seq.forall (fun (a, b) -> a <= b)
```

Concrete Example: Rating

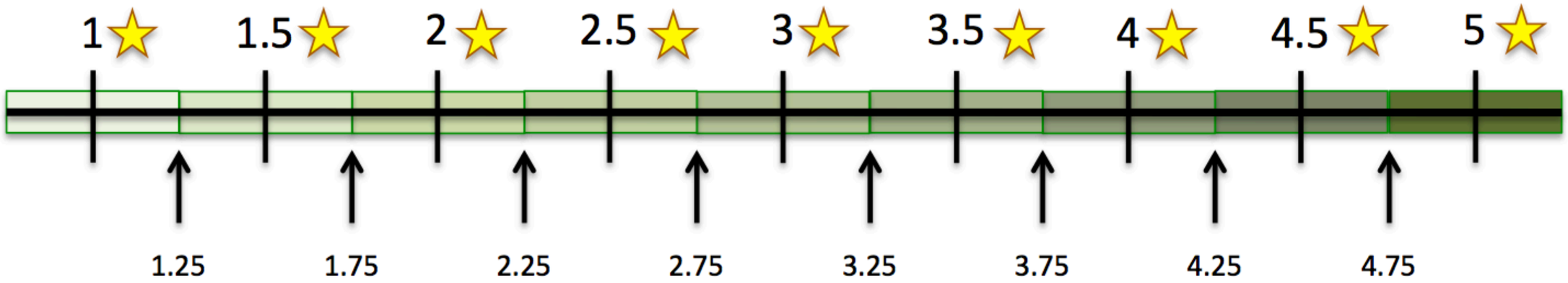
```
public class Rating
{
    /// <summary>
    /// Send in the user rating
    /// </summary>
    /// <param name="value">A number 1 - 5</param>
    public void Rate(int value)
    {
    }

    /// <summary>
    /// Returns the average rating value
    /// </summary>
    public double Average
    {
    }

    /// <summary>
    /// The number of stars to display, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5
    /// </summary>
    public double Stars
    {
    }
}
```

Rating Property

- Number of stars should be closest proximity discrete star to the average value



Generator

```
// generate list with values 1-5
static member ratingValues =
    let range = Arb.generate<int> |> Gen.suchThat ((>) 6) |> Gen.suchThat ((<) 0)
    Gen.listOf<int> range |> Arb.fromGen
```

```fsharp
open FsCheck
open PropertyBasedTesting.CSharp

// Arb.registerByType (typeof<RatingProperties>)
// Check.QuickAll (typeof<RatingProperties>)
type  RatingProperties =
    // generator
    static member ratingValues =
        let range = Arb.generate<int> |> Gen.suchThat ((>) 6) |> Gen.suchThat ((<) 0)
        Gen.listOf<int> range |> Arb.fromGen

    // property
    static member ``Number of stars should be closest proximity discrete star to the average value`` (input : int list) =
        // create sut
        let rating = Rating()
        // send in ratings
        List.iter (fun v -> rating.Rate(v)) input
        // trace
        printf "Rating: %A\n" rating
        // assert
        match rating.Stars with
        | 0.0 -> rating.Average = 0.0
        | 1.0 -> rating.Average < 1.25
        | 1.5 -> 1.25 <= rating.Average && rating.Average < 1.75
        | 2.0 -> 1.75 <= rating.Average && rating.Average < 2.25
        | 2.5 -> 2.25 <= rating.Average && rating.Average < 2.75
        | 3.0 -> 2.75 <= rating.Average && rating.Average < 3.25
        | 3.5 -> 3.25 <= rating.Average && rating.Average < 3.75
        | 4.0 -> 3.75 <= rating.Average && rating.Average < 4.25
        | 4.5 -> 4.25 <= rating.Average && rating.Average < 4.75
        | 5.0 -> 4.75 <= rating.Average
        | x -> false
```

Implementation

```
public double Stars
{
    get
    {
        return Math.Round(2 * Average, MidpointRounding.AwayFromZero) / 2;
    }
}
```

```
--- Checking RatingProperties ---
Rating: {Quantity: 1; Average: 1.00; Stars: 1.0}
Rating: {Quantity: 2; Average: 2.50; Stars: 2.5}
Rating: {Quantity: 5; Average: 2.40; Stars: 2.5}
Rating: {Quantity: 3; Average: 3.00; Stars: 3.0}
Rating: {Quantity: 0; Average: 0.00; Stars: 0.0}
Rating: {Quantity: 1; Average: 1.00; Stars: 1.0}
Rating: {Quantity: 2; Average: 1.00; Stars: 1.0}
Rating: {Quantity: 3; Average: 3.33; Stars: 3.5}
Rating: {Quantity: 3; Average: 2.67; Stars: 2.5}
Rating: {Quantity: 8; Average: 2.88; Stars: 3.0}
Rating: {Quantity: 2; Average: 3.00; Stars: 3.0}
Rating: {Quantity: 0; Average: 0.00; Stars: 0.0}
Rating: {Quantity: 5; Average: 2.40; Stars: 2.5}
Rating: {Quantity: 12; Average: 3.00; Stars: 3.0}
...
Rating: {Quantity: 4; Average: 2.25; Stars: 2.5}
Rating: {Quantity: 40; Average: 2.48; Stars: 2.5}
Rating: {Quantity: 13; Average: 2.85; Stars: 3.0}
Rating: {Quantity: 27; Average: 2.59; Stars: 2.5}
Rating: {Quantity: 50; Average: 3.22; Stars: 3.0}
Rating: {Quantity: 9; Average: 2.44; Stars: 2.5}
Rating: {Quantity: 7; Average: 2.71; Stars: 2.5}
Rating: {Quantity: 6; Average: 3.33; Stars: 3.5}
Rating: {Quantity: 3; Average: 3.67; Stars: 3.5}
Rating: {Quantity: 7; Average: 3.57; Stars: 3.5}
Rating: {Quantity: 78; Average: 2.97; Stars: 3.0}
Rating: {Quantity: 14; Average: 2.79; Stars: 3.0}
RatingProperties.Number of stars should be closest proximity discrete star to the average value-Ok, passed 100 tests.
```

Summary

- High-level explorative properties of your system under test
- Challenge your system with loads of data
- Easily find edge cases with your code
- Will make you think of your program outside-in

Thank you!

mikael.lundin@valtech.se

@mikaellundin

→ http://valte.ch/PropertyBasedTestingSlides

→ http://valte.ch/PropertyBasedTestingCode