MATHEMA

Property-Based Testing

in .NET

Patrick Drechsler

MATHEMA
CAMPUS

Patrick Drechsler

- Software Developer
- Work: C#
- Interests:
  - Software Crafting
  - Test-Driven Development
  - Functional Programming
  - Domain-Driven Design
- Slides are online: See QR-Code

Roadmap

- Problem w/ Example-Based Testing
- Examples solving the problem manually
- Property-Based Testing Frameworks
  - Vocabulary w/ examples
  - introduction to Generators & Shrinkers
  - more examples w/ Generators
- PBT Strategies
- Summary

Example based tests

Example:

```C#
public int Add(int a, int b)
  => a + b;
```

Unit test:

```C#
[Fact]
public void Add_works()
  => Add(1, 2).Should().Be(3);
```

More examples with parameterized test:

```C#
[Theory]
[InlineData(1, 2, 3)]
[InlineData(0, 1, 1)]
public void Add_works(int a, int b, int expected)
  => Add(a, b).Should().Be(expected);
```

😈 Developer From Hell (DFH)

- Test-Driven Development (TDD)
- Only implement enough to make the test pass

😈 DFH 1/3

```C#
[Fact]
public void List_123_returns_321()
{
  List<int> input = [1, 2, 3];
  var actual = MyReverse(input);
  List<int> expected = [3, 2, 1];
  Assert.Equal(expected, actual);
}
```

```C#
// 😈
public List<int> MyReverse(List<int> input)
  => [3, 2, 1];
```

😈 DFH 2/3

```C#
[Fact]
public void Empty_list_returns_empty_list()
{
  List<int> input = [];
  var actual = MyReverse(input);
  Assert.Equal([], actual);
}
```

```C#
// 😈
public List<int> MyReverse(List<int> input)
{
  if (input.Count == 0)
    return [];

  return [3, 2, 1];
}
```

😈 DFH 3/3

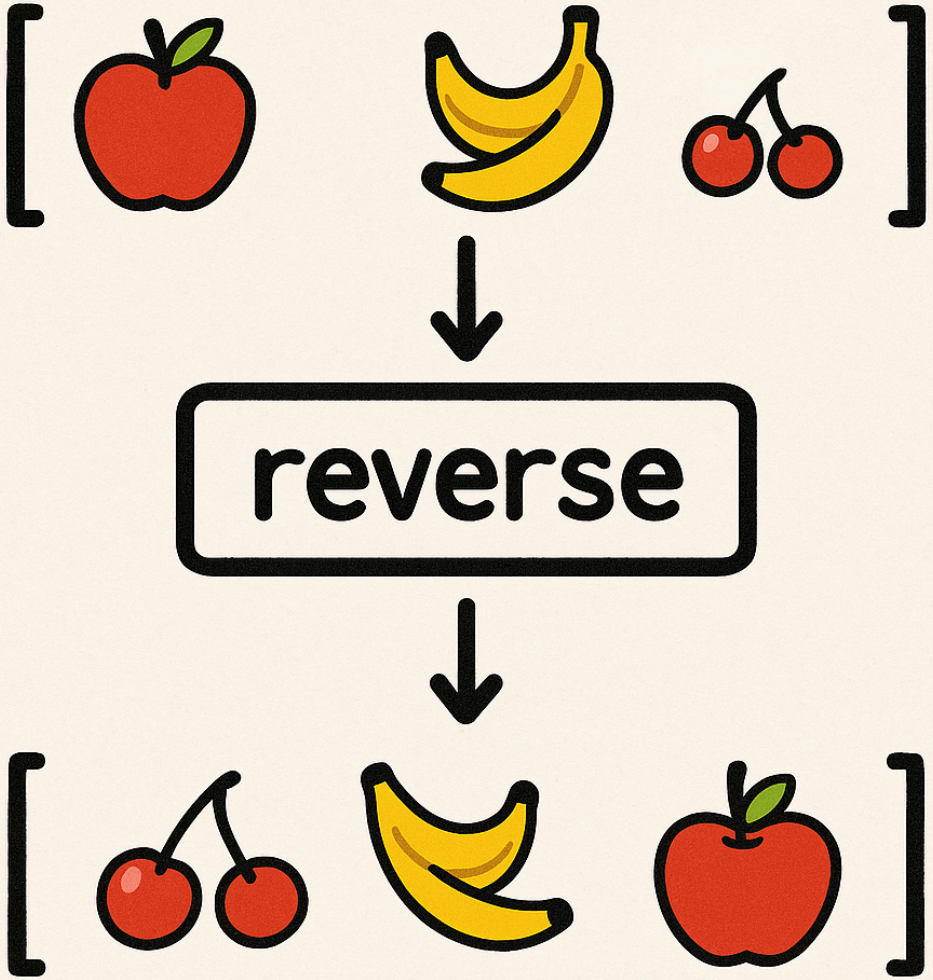
```C#
[Fact]
public void List_678_returns_876()
{
  List<int> input = [6, 7, 8];
  var actual = MyReverse(input);
  List<int> expected = [8, 7, 6];
  Assert.Equal(expected, actual);
}
```

```C#
// 😈
public List<int> MyReverse(List<int> input)
{
  if (input.Count == 0)
    return [];
  if (input.SequenceEqual((List<int>)[6, 7, 8]))
    return [8, 7, 6];

  return [3, 2, 1];
}
```

Let’s find some properties for reversing a list...

- result list has the same size
- result first element is last element of input list
- an empty list returns an empty list
- reversing a list twice returns the original list
- etc...



Live Coding

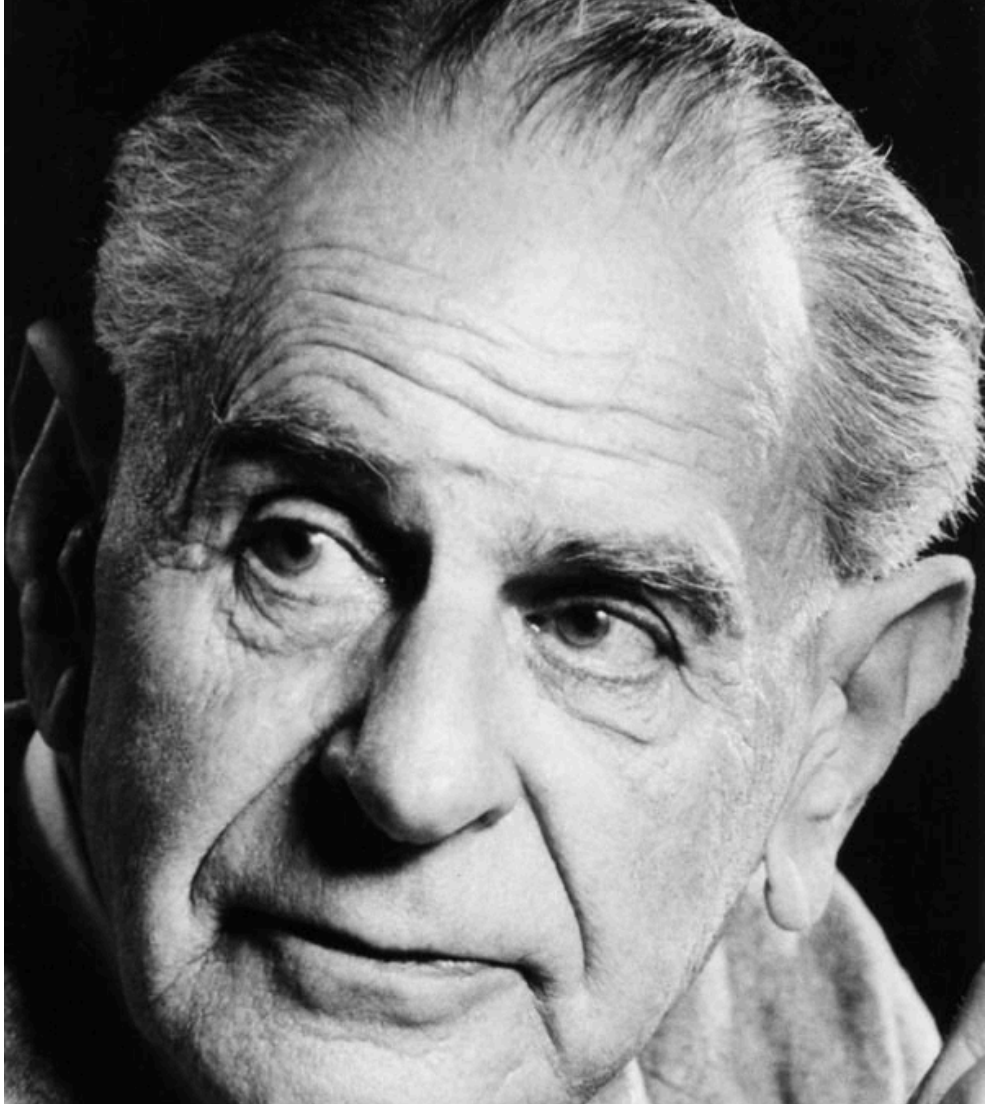
- Setup **without** a framework:
  - Adding 2 numbers
  - Reversing a list

PBT Framework Basics

- **Generator**
  - describe the input data!
- **Shrinker**
  - framework gives us minimal examples
- **Generator ➕ Shrinker 👉 Arbitrary**
- If something fails we not only get a falsifiable result: We get the closest result that does not fail.
- Some PBT Frameworks are called "Hypothesis" (i. e. in Python). Why?

Philosophy and natural sciences teach us...

- we can’t prove anything
- but we can **falsify** a **Hypothesis** ("Kritischer Rationalismus")
- finding a good hypothesis is the difficult part



Hello World FsCheck (1/2)

```C#
using FsCheck;        // ⚠️
using FsCheck.Fluent; // ⚠️

[Fact]
public void Reversing_a_list_twice_gives_the_original_list_v1()
{
  // This "Property" must return bool
  static bool CheckFn(List<int> list)
  {
    return MyReverse(MyReverse(list)).SequenceEqual(list);
  }

  // The lambda creates the test data input `list`
  Prop.ForAll((List<int> list) ⇒ CheckFn(list))  // ⚠️
    .QuickCheckThrowOnFailure();                 // ⚠️
}
```

- `Prop.ForAll`: creates sample data
- `QuickCheckThrowOnFailure`: throws on first failure

Hello World FsCheck (2/2)

```C#
using FsCheck;        // ⚠️
using FsCheck.Fluent; // ⚠️
using FsCheck.Xunit;  // ⚠️

[Property] // ⚠️
public bool Reversing_a_list_twice_gives_the_original_list_v2(List<int> list) // ⚠️
{
  var actual = MyReverse(MyReverse(list));
  var expected = list;
  return actual.SequenceEqual(expected);
}
```

- use `Property` attribute
- test must return `bool`
- test must have input parameter(s)

Generators

- data is generated via reflection by default
- generators can be fine-tuned: https://fscheck.github.io/FsCheck//TestData.html
- Example: `Gen.Choose`

```C#
// Generate random numbers between 1 and 100
// which are divisible be 3
var arb = Arb.From(
  Gen.Choose(1, 100)
    .Select(x => x * 3));
```

- don’t forget to wrap the Generator in an Arbitrary…

Shrinking

```C#
private static bool IsValid(string s)
{
  return s.StartsWith("a");
}

[Property]
public void Validation_works(string input)
{
  Assert.True(IsValid(input));
}
```

- returns a **minimal** falsifiable example:

```
Falsifiable, after 1 test (2 shrinks) (1523449830301185687,2773856807235509173)
Last step was invoked with size of 2 and seed of (16350581963149029034,15962072172286382299):
Original:
"\012"" (At least one control character has been escaped as a char code, e.g. \023)
Shrunk:
""
```

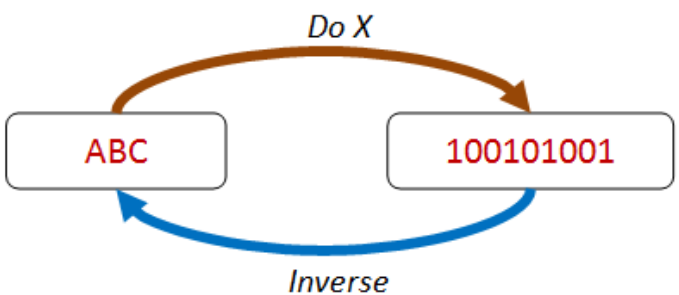
Live Coding

- Shrinker Demos
- FizzBuzz (C#, F#)

Strategies: "There and Back Again"

- When you have a **forward and backward** function, i. e. Serialize/Deserialize
- Example: Reversing a list twice returns the original list

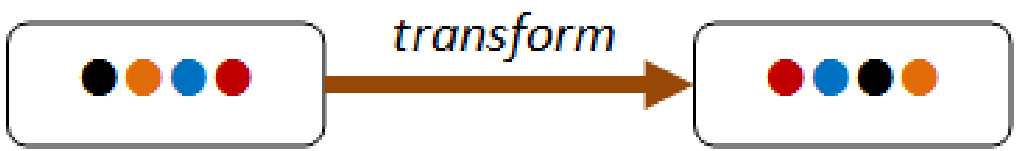
```fsharp
[<Property>]
let ``reversing a list twice returns the original list`` (aList: int list) =
  let actual = aList ▷ List.rev ▷ List.rev
  let expected = aList
  actual = expected
```



Strategies: "Some Things Never Change"

- When there is an **invariant** available
- Example: Sorting or Mapping a list never changes the length of the list

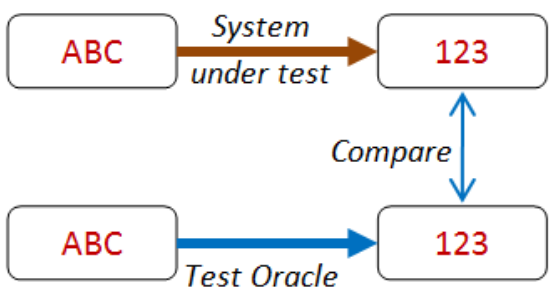
```fsharp
[<Property>]
let ``sorting a list does not change its size`` (aList: int list) =
  let actual = aList ▷ List.sort ▷ List.length
  let expected = aList.Length
  actual = expected
```



Strategies: "Test Oracle"

- When 2 functions should return the same result
- Example: Compare output of `LegacyFn` with `RefactoredFn`

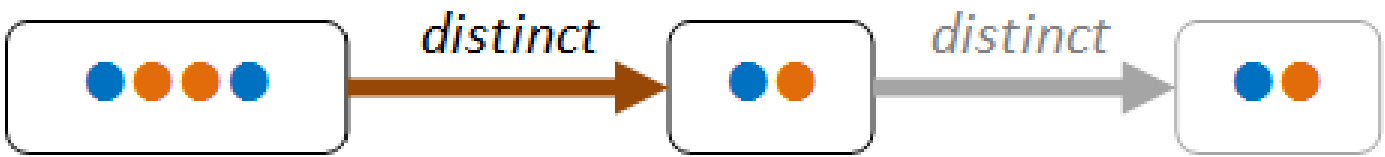
```F#
[<Property>]
let ``test oracle example`` (c: int, d: int) =
  let add1 a b = a + b
  let add2 a b = a * b
  let actual1 = add1 c d
  let actual2 = add2 c d
  actual1 = actual2
```



Strategies: "The More Things Change, The More They Stay The Same"

- When there are **idempotent** properties
- Example: Adding `` `0` ``, multiplying by `` `1` ``

```F#
[<Property>]
let ``adding zero is does not change the input`` (number: int) =
  let add a b = a + b
  let actual = add number 0
  let expected = number
  actual = expected
```



Strategies: "Nuclear Exception"

- Just verify that the function **does not throw an exception**
- Example: Generate valid inputs, and assert that no exception is thrown

```F#
[<Property>]
// Replace `str: NonEmptyString` with assumed valid input
let ``myFunction does not throw`` (str: NonEmptyString) =
  let myFunction s = if String.IsNullOrEmpty(s) then failwith "ups" else s
  myFunction str.Get = str.Get
```

- Great for API validations

Generators: Also for custom types

- Since most PBT frameworks use reflection under the hood, any type can be generated.
- Example: Gilded Rose

```F#
type ItemArb =
  static member Generate() =
    gen {
      let! name =
        Gen.elements [
          "Aged Brie"
          "Sulfuras, Hand of Ragnaros"
          "Backstage passes to a TAFKAL80ETC concert"
        ]
      let! sellIn = Gen.choose (1, 100)
      let! quality =
        Gen.choose (1, System.Int32.MaxValue)
        |> Gen.map PositiveInt
      return generateItem name sellIn quality
    }
    |> Arb.fromGen
```

PBT Frameworks for .NET

- FsCheck: The most popular
- Hedgehog: also worth a look
- CsCheck: the new kid on the block

All can be used from C# and F#.

The frameworks mainly differ in syntax.

Available for most languages

Just search for "Property Based Testing" and your language!

- Haskell: QuickCheck
- Python: Hypothesis
- JavaScript: fast-check
- Java: jqwik
- Scala: ScalaCheck
- Clojure: Test.check
- etc

When shouldn’t you use PBT?

- When you don’t have standard unit tests
- When standard unit tests are sufficient
- When you want to test a specific case
- When you can’t identify suitable properties
- When your models are not suitable
- When your SUT is slow

Summary

- Property-based testing is a technique for testing statements of the type:
  - **For all x that satisfy some precondition then some predicates holds**
- It can give you confidence that your code behaves correctly across a wider range of inputs.
- It can help you find bugs in your code resulting from inputs you never would have thought to test.
- This technique can be used alongside existing unit tests.

Thank You!

- patrick.drechsler@mathema.de
- https://github.com/draptik
- https://draptik.github.io/talks/
- https://floss.social/@drechsler
- https://bsky.app/profile/drechsler.bsky.social
- https://www.linkedin.com/in/patrick-drechsler-draptik/

Slides 👇

- QR Code or
- https://draptik.github.io/2025-05-mathema-campus-property-based-testing/
- sample code: https://github.com/draptik/2025-property-based-testing

Image sources: FSharpForFunAndProfit, pixabay.com, chatgpt.com, and perchance.org/ai-photo-generator