Introduction to Property Based Testing

- Sebastian Grail
- iOS Developer at Canva
- @sebastiangrail
- github: sebastiangrail

Testing a `max` function with conventional unit tests

To illustrate what propert based testing is, I'm going to run through an example of standard TDD workflow

Test Driven Development

- write a "single" unit test describing an aspect of the program
- run the test, which should fail because the program lacks that feature
- write "just enough" code, the simplest possible, to make the test pass
- "refactor" the code until it conforms to the simplicity criteria
- repeat, "accumulating" unit tests over time

https://www.agilealliance.org/glossary/tdd/

```
XCTAssertEqual(max(1, 2), 2)
```

🔴

Write a single failing test

```swift
XCTAssertEqual(max(1, 2), 2)

func max(_ x: Int, _ y: Int) -> Int {
  return 2
}
```

✅

Write "just enough" code to make the test pass. It's so simple that we don't refactor

```swift
XCTAssertEqual(max(1, 2), 2)
XCTAssertEqual(max(1, 3), 3)

func max(_ x: Int, _ y: Int) -> Int {
  return 2
}
```

🔴

Write another failing test

```swift
XCTAssertEqual(max(1, 2), 2)
XCTAssertEqual(max(1, 3), 3)

func max(_ x: Int, _ y: Int) -> Int {
  return y
}
```

✅

Write "just enough" code to make the test pass. It's so simple that we don't refactor

```swift
XCTAssertEqual(max(1, 2), 2)
XCTAssertEqual(max(1, 3), 3)
XCTAssertEqual(max(3, 1), 3)

func max(_ x: Int, _ y: Int) -> Int {
  return y
}
```

🔴

Another failing test

```swift
XCTAssertEqual(max(1, 2), 2)
XCTAssertEqual(max(1, 3), 3)
XCTAssertEqual(max(3, 1), 3)

func max(_ x: Int, _ y: Int) -> Int {
  if x == 3 {
    return x
  }
  return y
}
```

✅

And "just enough" code. Again, there isn't much to refactor

```swift
XCTAssertEqual(max(1, 2), 2)
XCTAssertEqual(max(1, 3), 3)
XCTAssertEqual(max(3, 1), 3)
XCTAssertEqual(max(4, 1), 4)

func max(_ x: Int, _ y: Int) -> Int {
  if x == 3 {
    return x
  }
  return y
}
```

🔴

And yet another failing test

```swift
XCTAssertEqual(max(1, 2), 2)
XCTAssertEqual(max(1, 3), 3)
XCTAssertEqual(max(3, 1), 3)
XCTAssertEqual(max(4, 1), 4)

func max(_ x: Int, _ y: Int) -> Int {
  if x == 3 || x == 4 {
    return x
  }
  return y
}
```

✅

Enough code to make the test pass. Maybe we can refactor this.

```swift
XCTAssertEqual(max(1, 2), 2)
XCTAssertEqual(max(1, 3), 3)
XCTAssertEqual(max(3, 1), 3)
XCTAssertEqual(max(4, 1), 4)

func max(_ x: Int, _ y: Int) -> Int {
  if x >= 3 {
    return x
  }
  return y
}
```

✅

One way to refactor, the tests still pass. So we should write even more tests

```swift
XCTAssertEqual(max(1, 2), 2)
XCTAssertEqual(max(1, 3), 3)
XCTAssertEqual(max(3, 1), 3)
XCTAssertEqual(max(4, 1), 4)
XCTAssertEqual(max(0, -3), 0)

func max(_ x: Int, _ y: Int) -> Int {
  if x >= 3 {
    return x
  }
  return y
}
```

🔴

```swift
XCTAssertEqual(max(1, 2), 2)
XCTAssertEqual(max(1, 3), 3)
XCTAssertEqual(max(3, 1), 3)
XCTAssertEqual(max(4, 1), 4)
XCTAssertEqual(max(0, -3), 0)

func max(_ x: Int, _ y: Int) -> Int {
  if x > y {
    return x
  }
  return y
}
```

✅

We finally found the correct implementation and can't write any more failing tests

```
XCTAssertEqual(max(1, 2), 2)
XCTAssertEqual(max(1, 3), 3)
XCTAssertEqual(max(3, 1), 3)
XCTAssertEqual(max(4, 1), 4)
XCTAssertEqual(max(0, -3), 0)
```

- 👍 100% code coverage
- 👎 Not exhaustive
- 👎 Not robust against changes
- 👎 Not good documentation
- 👎 Confusing

Next: A different Approach, still TDD, but now we try to avoid the issues in this solution

```
let (a, b) = (Int.random, Int.random)
XCTAssert(max(a, b) == a || max(a, b) == b)
```


This time we start with a stronger assertion, given any two random values, max should return one of them.

```swift
func max(_ x: Int, _ y: Int) -> Int {
  return 0
}

let (a, b) = (Int.random, Int.random)
XCTAssert(max(a, b) == a || max(a, b) == b)
```

✅

(if one of the `Int.random` **returned** `0`**)**

The simplest implementation might succeed by chance

```swift
func max(_ x: Int, _ y: Int) -> Int {
  return 0
}

for _ in 0..<100 {
  let (a, b) = (Int.random, Int.random)
  XCTAssert(max(a, b) == a || max(a, b) == b)
}
```

🔴

If we run it 100 times, we can be more confident about our test

```swift
func max(_ x: Int, _ y: Int) -> Int {
  return x
}

for _ in 0..<100 {
  let (a, b) = (Int.random, Int.random)
  XCTAssert(max(a, b) == a || max(a, b) == b)
}
```

✅

Write the simplest code to make our test pass

```swift
func max(_ x: Int, _ y: Int) -> Int {
  return x
}

for _ in 0..<100 {
  let (a, b) = (Int.random, Int.random)
  XCTAssert(max(a, b) == a || max(a, b) == b)
}
for _ in 0..<100 {
  let (a, b) = (Int.random, Int.random)
  XCTAssert(max(a, b) >= a && max(a, b) >= b)
}
```

🔴

Our second assertion checks that the returned value is not smaller than any of the inputs, i.e. it is greater or equal to both

```swift
func max(_ x: Int, _ y: Int) -> Int {
  return x > y ? x : y
}

for _ in 0..<100 {
  let (a, b) = (Int.random, Int.random)
  XCTAssert(max(a, b) == a || max(a, b) == b)
}
for _ in 0..<100 {
  let (a, b) = (Int.random, Int.random)
  XCTAssert(max(a, b) >= a && max(a, b) >= b)
}
```

✅

We now can *only* write a corrcect implementation

```swift
XCTAssert(max(a, b) == a || max(a, b) == b)
XCTAssert(max(a, b) >= a && max(a, b) >= b)
```

- 👍 100% code coverage
- 👍 Exhaustive[1]
- 👍 Robust against changes
- 👍 Documents what the function does
- 👎 Harder to read and write at first
- 👍👎 The test data changes with every run

---

[1] Eventually we will have tested *every* input

Now we are left with 2 assertions that precisely describe our requirements

Properties

```
XCTAssert(max(a, b) == a || max(a, b) == b)
XCTAssert(max(a, b) >= a && max(a, b) >= b)
```

- $\forall a \in \mathbb{Z}, b \in \mathbb{Z} : max(a, b) = a \vee max(a, b) = b$
- $\forall a \in \mathbb{Z}, b \in \mathbb{Z} : max(a, b) \geq a \wedge max(a, b) \geq b$

We can describe these assertions in two mathematical properties

Properties

- $\forall a \in \mathbb{Z}, b \in \mathbb{Z} : max(a, b) = a \vee max(a, b) = b$

```
forAll { (a: Int, b: Int) in
  max(a,b) == a || max(a,b) == b
}
```

- $\forall a \in \mathbb{Z}, b \in \mathbb{Z} : max(a, b) \geq a \wedge max(a, b) \geq b$

```
forAll { (a: Int, b: Int) in
  max(a,b) >= a && max(a,b) >= b
}
```


And can then translate those properties back into pseudo Swift code

SwiftCheck

SwiftCheck is a testing library that automatically generates random data for testing of program properties.

https://github.com/typelift/SwiftCheck

SwiftCheck is a Swift implementation of the Haskell testing library QuickCheck

There are several QuickCheck implementation for other languages, including Java and Kotlin

SwiftCheck

```swift
func testMax() {
  property("max returns one of its inputs") <- forAll { (a: Int, b: Int) in
    max(a,b) == a || max(a,b) == b
  }

  property("the output is >= to both inputs") <- forAll { (a: Int, b: Int) in
    max(a,b) >= a && max(a,b) >= b
  }
}
```


The pseudo code before is actually valid code in SwiftCheck. Here it is inside an XCTestCase test* method

We give the property a name and assign or logical proposition to it. SwiftCheck then takes care of the details and runs the closure 100 times

Examples of Properties

Reflexivity

```
property("Integer equality is reflexive") <-
  forAll { (x: Int) in
    x == x
  }
```

- is subset of
- is divisible by

Other reflexive functions: is subset of, is divisble.

Commutativity

```
property("Integer addition is commutative") <-
  forAll { (x: Int, y: Int) in
    x + y == y + x
  }
```

- Many maths operations are commutative (e.g. `max`)
- Set insertion

This is useful for all operations where the order of arguments doesn't matter.

For some operations, commutativity holds for part of the output, e.g. writing to a database might be independent of the order when discarding IDs. Writing to a cache might be commutative if it is big enough for both values

Associativity

```
property("appending strings is associative") <-
  forAll { (x: String, y: String, z: String) in
    (x + y) + z == x + (y + z)
  }
```

- Addition and multiplication of numbers, vectors
- Matrix multiplication
- Union and intersection of sets
- Can be performed in parallel on large data sets

Inverses

```
property("reversing an array twice is identity") <-
 forAll { (xs: [Int]) in
    xs.reversed().reversed() == xs
  }
```

- `reversed` is the inverse of itself
- Works similar for other inverses
- E.g. serialising/deserialising

While this check is usually insufficient by itself, it makes sense in a suite of property tests

Distributivity

```
property("dot product distributes over vector addition")
<- forAll { (a: Vector, b: Vector, c: Vector) in
  let left = a.dot(b + c)
  let right = a.dot(b) + a.dot(c)
  return left.isCloseTo(right)
}
```

- Many maths operations are distributive
- map distributes over function composition

Often two functions need to work together in a specific way

Invariants

```
property("zip returns a sequence the length of its shortest argument")
<- forAll { (xs: [Bool], ys: [Bool], zs: [Bool]) in
  Array(zip(xs, ys, zs)).count == min(xs.count, min(ys.count, zs.count))
}
```

- `zip` returns prefixes of all its arguments
- `map` doesn't change the structure of a type
- Sorting doesn't add or remove elements

Many functions have invariants, i.e. part of the input is unchanged in the output

Other examples include `map` doesn't change the structure of a type, sorting doesn't add or remove elements

Replicating Test Failures

```swift
func max(_ x: Int, _ y: Int) -> Int {
  return x
}
```

```
*** Failed! Proposition: the output of max is greater or equal to both inputs
...
failed - Falsifiable; Replay with 1731542611 351985798 and size 1
```


The output includes the seeds for the random number generator and the size for the generator

```swift
func max(_ x: Int, _ y: Int) -> Int {
  return x
}
```

```
*** Failed! Proposition: the output of max is greater or equal to both inputs
...
failed - Falsifiable; Replay with 1731542611 351985798 and size 1
```

```swift
let arguments = CheckerArguments(replay: (StdGen(1731542611, 351985798), 1))
```


We can use this information to create custom CheckerArguments

```swift
func max(_ x: Int, _ y: Int) -> Int {
  return x
}
```

```
*** Failed! Proposition: the output of max is greater or equal to both inputs
...
failed - Falsifiable; Replay with 1731542611 351985798 and size 1
```

```swift
let arguments = CheckerArguments(replay: (StdGen(1731542611, 351985798), 1))

property("the output of max is greater or equal to both inputs",
         arguments: arguments)
   <- forAll { (x: Int, y: Int) in
    let result = max(x, y)
    return result >= x && result >= y
  }
```


We pass the arguments into the `property` call and can now set a breakpoint and debug

Generators

Property tests are only as good as the generators they use. Let's take a closer look where those generators come from.

Generators

```
func forAll <A> (pf: @escaping (A) -> Testable)
    -> Property where A : Arbitrary

func forAll <A, B> (pf: @escaping (A, B) -> Testable)
    -> Property where A : Arbitrary, B : Arbitrary

...
```


This is the (simplified) signature of the `forAll` function we used before.

The function takes any type that conforms to arbitrary and returns any type that is `Testable`.

Bool conforms to `Testable`

Generators

```
func forAll <A> (pf: (A) -> Testable) -> Property where A : Arbitrary

public protocol Arbitrary {
  public static var arbitrary: Gen<Self> { get }
  public static func shrink(_: Self) -> [Self]
}
```


Any type conforming to `Arbitrary` can produce a generator and shrink itself. We'll come back to shrinking later.

Generators

```
func forAll <A> (pf: (A) -> Testable) -> Property where A : Arbitrary

public protocol Arbitrary {
  public static var arbitrary: Gen<Self> { get }
  public static func shrink(_: Self) -> [Self]
}

public struct Gen<A> {
  ...
}
```


`Gen` is a generic struct that provides various ways of constructing generators

Generators

- `Gen` represents a generator for random arbitrary values of type `A`.
- `Gen` wraps a function that, when given a random number generator and a size, can be used to control the distribution of resultant values.
- Create with single value, range or collection of values
- Create a new generator by modifying an existing generator
- Compose multiple generators into a new generator
- SwiftCheck comes with `Arbitrary` implementations for many Swift types

Value types that uses a random number generator as a dependency. This makes testing deterministic

```swift
struct Vector {
  let dx: Double
  let dy: Double
}

extension Vector: Arbitrary {
  public static var arbitrary: Gen<Vector> {
    return Gen<(Double, Double)>
      .zip(Double.arbitrary, Double.arbitrary)
      .map { Vector(dx: $0, dy: $1) }
  }
}
```


Often we can compose existing generators into generators for our custom types.

zip turns a tuple of generators into a generator of tuples.

map modifies generated values

Custom Generators

Custom Generators

```
func forAll <A> (
  gen: Gen<A>,
  pf: @escaping (A) -> Testable)
  -> SwiftCheck.Property where A : Arbitrary
```


One overload of the `forAll` function takes a generator as an argument

Custom Generators

```
forAll { (x: Double) in
  ...
}
```

```
let gen = Double.arbitrary
forAll(gen) { x in
  ...
}
```


These two snippets are equivalent. Note that in the second we can omit the type for the closure argument

suchThat

```
let gen = Double.arbitrary.suchThat { $0 >= 0 }
forAll(gen) { x in
   ...
}
```

- Generates only values that satisfy the predicate
- Discards values that don't satsify the predicate
- Can be slow if a lot of values fail

There's usually a better alternative

map

```
let gen = Double.arbitrary.map { abs($0) }
forAll(gen) { x in
  ...
}
```

- Modifies values
- Fast, because no values need to be discarded

There are many more operators, we'll see some of them later in the talk

Testing Custom Types

Let's look at some more ways of composing generators

```swift
struct User {
    let name: String
    let verified: Bool
    let age: Int
}

extension User: Arbitrary {
    static var arbitrary: Gen<User> {
        return Gen<User>.compose { composer in
            return User(
                name: composer.generate(),
                verified: composer.generate(),
                age: composer.generate())
        }
    }
}
```


Compose provides a very readable (and imperative) way of combining generators

`generate` returns a generic, the type is inferred from the context

```
User(name: "", verified: true, age: 0)
User(name: "", verified: true, age: 1)
User(name: "×$", verified: false, age: 2)
User(name: "úÏö", verified: false, age: -1)
User(name: "
ëd", verified: true, age: 1)
User(name: "½în", verified: true, age: -4)
User(name: "\\tyïÏ", verified: true, age: -1)
User(name: "kþóß:Õ", verified: false, age: -3)
```


This is great if we want to test edge cases in names and age, but what if `User` has a failable initialiser?

```swift
let nonNegativeNumbers = Int.arbitrary.map { abs($0) % 200 }

let validAges = Gen<Int>.fromElements(in: 0...200)
```


Multiples ways of creating a more realistic age generator

`fromElements` chooses any value from a range with equal probability

```swift
struct User {
    let name: String
    let verified: Bool
    let age: Int
}

extension User: Arbitrary {
    static var arbitrary: Gen<User> {
        return Gen<User>.compose { composer in
            return User(
                name: composer.generate(),
                verified: composer.generate(),
                age: composer.generate(using: validAges))
        }
    }
}
```


We can now pass our custom generator into the `generate` function inside `compose`.

```
User(name: "", verified: false, age: 72)
User(name: "ô", verified: false, age: 33)
User(name: "#º×g", verified: false, age: 131)
User(name: "°|¿z", verified: false, age: 110)
User(name: "H¡ó§Á", verified: true, age: 67)
User(name: "z¹ûx9¡", verified: true, age: 200)
User(name: "*ÏÓ P½Áº", verified: true, age: 3)
```

```swift
let charGenerator: Gen<Character> = Gen<Character>.fromElements(in: "a"..."z")
```


This generator produces a `Character` from 'a' to 'z' with equal probability.
We can use it as a building block for the name

```
let charGenerator: Gen<Character> = Gen<Character>.fromElements(in: "a"..."z")

// Gen<Int>
let lowersGenerator = Gen<Int>.choose((3, 19))
```


We now build a generator for the lowercase suffix

First we create a generator that will give us the length

```swift
let charGenerator: Gen<Character> = Gen<Character>.fromElements(in: "a"..."z")

// Gen<[Character]>
let lowersGenerator = Gen<Int>.choose((3, 19))
    .flatMap { n -> Gen<[Character]> in
        let generators = Array(repeating: charGenerator, count: n)
        return sequence(generators) // [Gen<T>] -> Gen<[T]>
    }
```


We can use `flatMap` to create generators that depend on other generators

For every value generated, we return a new generator that depends on that value

In this case, given a generated length, we create a generator of arrays of that length

```swift
let charGenerator: Gen<Character> = Gen<Character>.fromElements(in: "a"..."z")

// Gen<String>
let lowersGenerator = Gen<Int>.choose((3, 19))
    .flatMap { n -> Gen<[Character]> in
        let generators = Array(repeating: charGenerator, count: n)
        return sequence(generators) // [Gen<T>] -> Gen<[T]>
    }
    .map { String($0) }
```


In the end we can use `map` to create a string from the `Character` array

```swift
let charGenerator: Gen<Character> = Gen<Character>.fromElements(in: "a"..."z")

// Gen<String>
let lowersGenerator = Gen<Int>.choose((3, 19))
    .flatMap { n -> Gen<[Character]> in
        let generators = Array(repeating: charGenerator, count: n)
        return sequence(generators) // [Gen<T>] -> Gen<[T]>
    }
    .map { String($0) }

let nameGenerator = Gen<(String, String)>
    .zip(charGenerator.map { String($0).uppercased() }, lowersGenerator)
    .map { $0.appending($1) }
```


Here I've used `zip` and `map` to create a generator for the whole name

```
User(name: "Qfcgfofolfpnps", verified: false, age: 27)
User(name: "Zusquveglknr", verified: true, age: 43)
User(name: "Djsatcdioiefqqasctcw", verified: true, age: 67)
User(name: "Utnpnohyjbxopk", verified: false, age: 123)
User(name: "Umkkqgruxdpgnnzwsnbut", verified: false, age: 117)
User(name: "Covfefe", verified: false, age: 161)
User(name: "Tuoslidvouzmj", verified: true, age: 120)
User(name: "Sfafwbojao", verified: false, age: 10)
User(name: "Pgwlrlqxzitwzvncv", verified: true, age: 110)
```


If we use this generator for the `name` property we get something like this

Shrinking

A very powerful feature of SwiftCheck is its ability to find simple test data for failure cases.

Shrinking

```swift
func reverse <T> (_ xs: [T]) -> [T] {
  guard let first = xs.first else { return [] }
  return reverse(Array(xs.dropLast())) + [first]
  //                        ^
}
```

Succeeds for

- arrays with less than 2 values
- arrays where all values are equal

This function uses `dropLast` instead of `dropFirst`, but it still works for some inputs.

If the test fails with a huge array, it might be hard to debug.

Shrinking

^ `Arbitrary` **has a default implementation for** `shrink` **which returns the empty array**

Shrinking Examples

When a test fails, SwiftCheck uses `shrink` to find smaller test data and will report the smallest input that fails the test.

QuickCheck finds the smallest test case that doesn't satisfy the property

```
Falsifiable (after 6 tests and 8 shrinks):
[1, 0]
...
Replay with 1911878021 8651 and size 5
```

- `[1, 0]` is the smallest failing input that SwiftCheck found
- We can use `CheckerArguments` like before to replicate the failure.

Note that the arguments replicate the *first* failure, SwiftCheck can't find the seeds to recreate the shrunk values.

Subsequent calls however will use the values produced by `shrink`

Shrinking a Custom Type

```
public static func shrink(_ vector: Vector) -> [Vector] {
  let dxs = Double.shrink(vector.dx)
  let dys = Double.shrink(vector.dy)
  var result: [Vector] = []
  for dx in dxs {
    for dy in dys {
      result.append(Vector(dx: dx, dy: dy))
    }
  }
  return result
}
```


We can create a customer shrinker for `Vectors` by shrinking each component and combining the results

Shrinking with Custom Generators

```
forAll(Int.arbitrary.map { -abs($0) }) { n in
  ...
}
```

- Uses the standard shrinker for `Ints`
- Runs the tests with positive numbers

When using a modified generator, SwiftCheck will still use the standard shrinker for that type

Shrinking with Custom Generators

```
forAll(Int.arbitrary.map { -abs($0) }) { n in
  return (x <= 0) ==> {
    ...
  }
}
```


The ==> operator, takes a boolean on the left and discards the test run if it is `false`

Careful: This can lead to tests where all runs are discarded

Shrinking with Custom Generators

```
forAllNoShrink(Int.arbitrary.map { -abs($0) }) { n in
  return ...
}
```


with `froAllNoShrink`, SwiftCheck doesn't shrink the input

Providing Custom Shrinkers

```
func forAllShrink<A>(
    _ gen: SwiftCheck.Gen<A>,
    shrinker: @escaping (A) -> [A],
    f: @escaping (A) throws -> Testable)
    -> SwiftCheck.Property
```

- No overloads for multiple arguments

Markov Chains using SwiftCheck Generators

Markov Chains using SwiftCheck Generators

```
let letterFrequency: [String: [(Double, String?)]] = [
  "_": [(11.291625661, "a"),
        (4.682943604, "b"),
  ...
  "a": [(7.837954860, nil),
        (0.021157184, "a"),
        (2.048922926, "b"),
  ...
]
```


Has an array of (probability in percent, next letter) for every letter.

Underscore represents start of word, nil represents end of word

Markov Chains using SwiftCheck

```
func string(following s: String) -> Gen<String?> {
    guard let successorGen = letterFrequency[s] else {
        return Gen.pure(nil)
    }
    return Gen<String?>.weighted(
        successorGen.map { (Int($0*100), $1) }
    )
}
```


Given a string, create a generator for the successor of that string with the same frequency as in English

Immediately terminate if the letter isn't in the dictionary

Use `weighted` to create a generator from a sequence of (relative distribution, value)

Markov Chains using SwiftCheck

```swift
func unfold <Value> (
  f: @escaping (Value) -> Gen<Value?>,
  initial: Value)
  -> Gen<[Value]>
{
  return f(initial).flatMap { value -> Gen<[Value]> in
    guard let value = value else {
      return Gen<[Value]>.pure([])
    }
    return unfold(f: f, initial: value).map { [value] + $0 }
  }
}
```


Unfold is the opposite of a fold, i.e. given a starting value and a function, it creates a sequence of values.

Note how the signature of the function argument matches the function from the previous slide

Call the function with the initial value.
 Use `flatMap` to create a generator that depends on a value
 If the value is nil, the recursion terminates
 Otherwise we recurse, using the value as the new initial, and concatenating the results

```swift
let generator = unfold(f: string(following:), initial: "_")
                .map { $0.joined() }
                .suchThat { $0.characters.count >= 3 }
```

There's a ~ 1 in 2^{27} chance that it will generate "swift"

The two functions together will produce a generator of "English-like" words

Use *map* to join the strings, suchThat to get more interesting words

Problems I Encountered

- Finding properties is **hard**
- Swift's type system: Missing conditional conformance
- Sometimes the type system gives up
- Danger of repeating implementation

Thank You

- Slides with notes and sample code available at github.com/sebastiangrail/property-based-testing-talk
- SwiftCheck is open source at github.com/typelift/SwiftCheck
- Haskell Programming from first principles at haskellbook.com is one of the best books on functional programming