

Introduction to Property Based Testing



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

Testing a `max` function with conventional unit tests

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)
```



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

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



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

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

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



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

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

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




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

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

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

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



```
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
}
```



```
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  
}
```



```
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  
}
```



```
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  
}
```



```
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
}
```



```
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
}
```



```
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


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

```
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)

```
func max(_ x: Int, _ y: Int) -> Int {  
    return 0  
}  
  
for _ in 0..  
    let (a, b) = (Int.random, Int.random)  
    XCTAssert(max(a, b) == a || max(a, b) == b)  
}
```



```
func max(_ x: Int, _ y: Int) -> Int {  
    return x  
}  
  
for _ in 0..  
    let (a, b) = (Int.random, Int.random)  
    XCTAssert(max(a, b) == a || max(a, b) == b)  
}
```



```
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)  
}
```



```
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)  
}
```



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

- 👍 100% code coverage
- 👍 Exhaustive¹
- 👍 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

Properties

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

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

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

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  
}
```

SwiftCheck

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

<https://github.com/typelift/SwiftCheck>

SwiftCheck

```
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  
    }  
}
```

Examples of Properties

Reflexivity

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

- is subset of
- is divisible by

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

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

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

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

Replicating Test Failures

```
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

```
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

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

```
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

```
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  
  }
```

Generators

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
```

...

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]  
}
```

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> {  
    ...  
}
```

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

```
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) }  
    }  
}
```

Custom Generators

Custom Generators

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

Custom Generators

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

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

suchThat

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

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

map

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

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

Testing Custom Types

```
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())  
            }  
        }  
    }  
}
```

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

```
let nonNegativeNumbers = Int.arbitrary.map { abs($0) % 200 }  
  
let validAges = Gen<Int>.fromElements(in: 0...200)
```

```
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))  
            }  
        }  
    }  
}
```

```
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: "Hió§Á", verified: true, age: 67)
User(name: "z¹ ûx9 i", verified: true, age: 200)
User(name: "*ÏÓ P½Áº", verified: true, age: 3)
```

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



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

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

```
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]>
    }
```

```
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 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) }
```

```
User(name: "Qfcgfofolfpnps", verified: false, age: 27)
User(name: "Zusquveglknr", verified: true, age: 43)
User(name: "Djsatcdioiefqqasctcw", verified: true, age: 67)
User(name: "Utnpnohyjbxbpk", 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: "Pgwlrlqxzitzwzvncv", verified: true, age: 110)
```

Shrinking

Shrinking

```
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

Shrinking

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

Shrinking Examples

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.

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  
}
```

Shrinking with Custom Generators

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

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

Shrinking with Custom Generators

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

Shrinking with Custom Generators

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

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"),  
  
    ...  
]
```


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) }  
    )  
}
```

Markov Chains using SwiftCheck

```
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 }
  }
}
```

```
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"

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