Intro to PBT with

ScalaCheck

https://github.com/ssanj/intro-to-property-based-testing

intro-to-property-based-testing-simple-msug

EBT

EBT
PBT

EBT
PBT
ScalaCheck

EBT
PBT
ScalaCheck

Choosing Properties

EBT
PBT
ScalaCheck
Choosing Properties
Examples

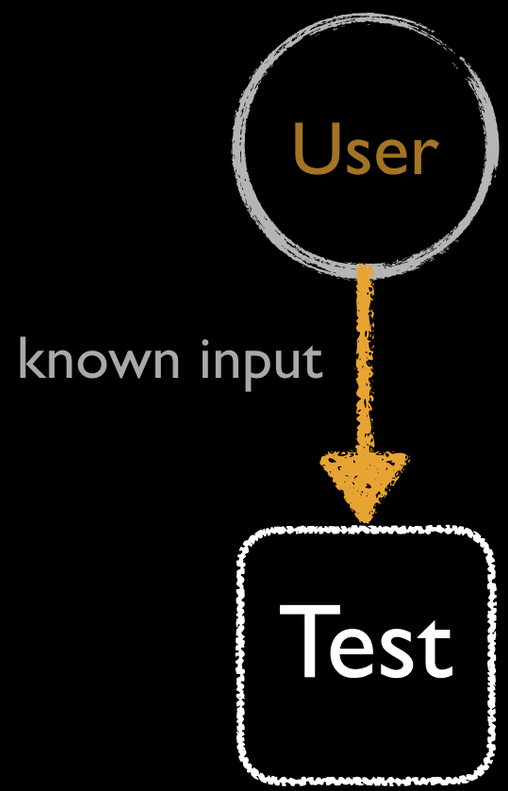
EBT
PBT
ScalaCheck
Choosing Properties
Examples
Summary

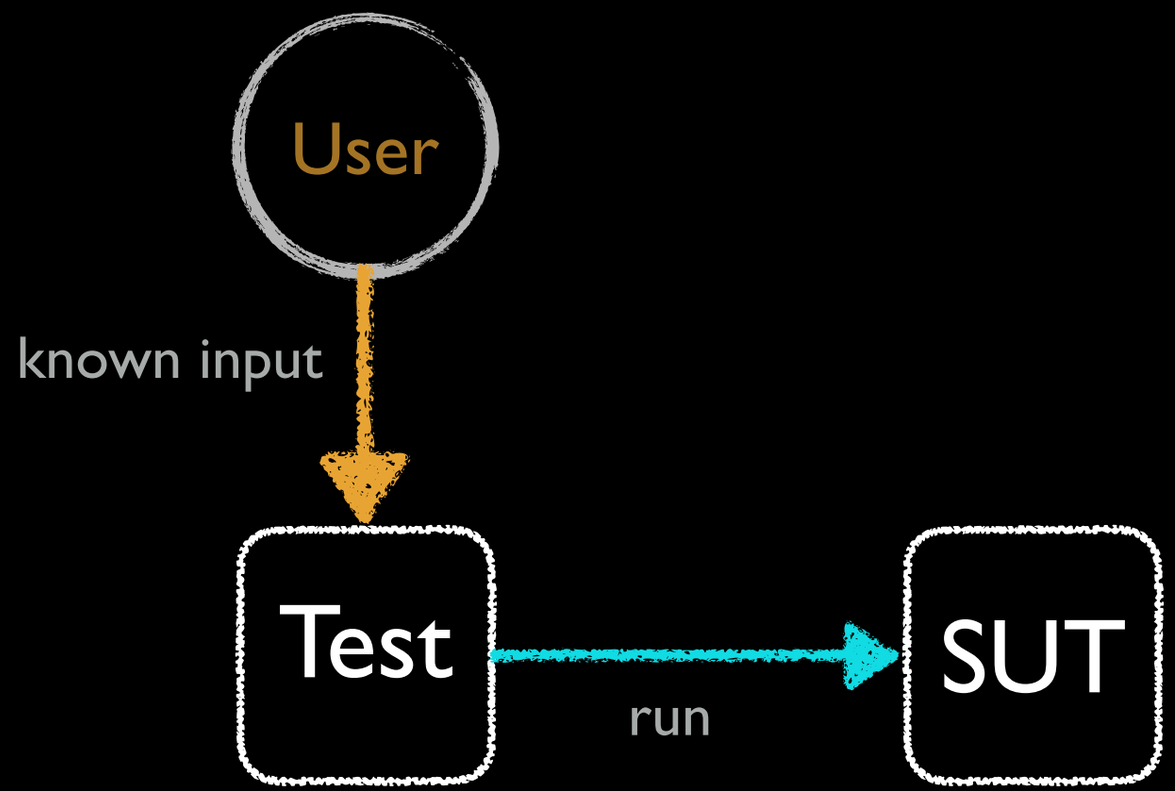
Testing shows the presence,
not the absence of bugs

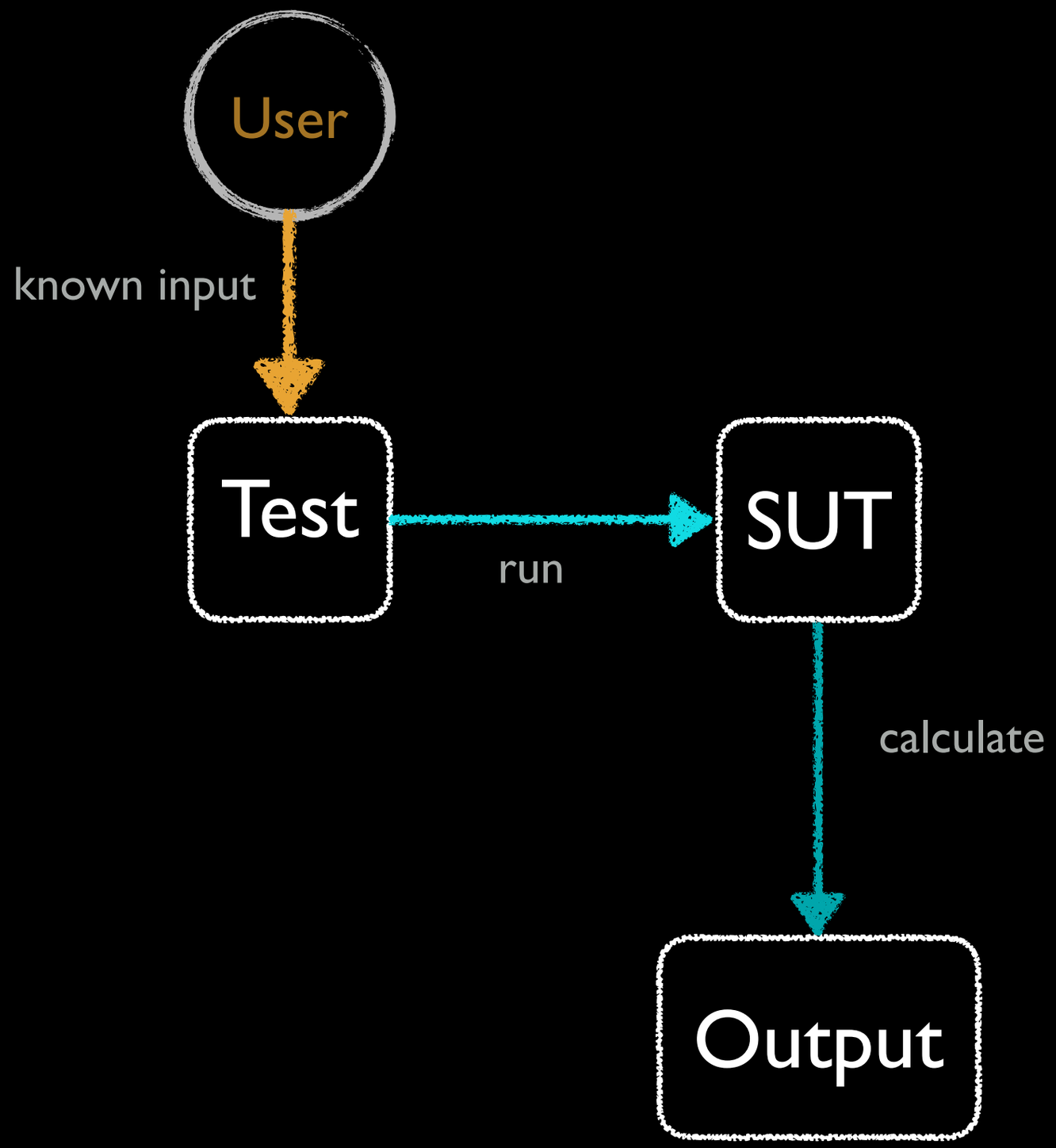
Dijkstra

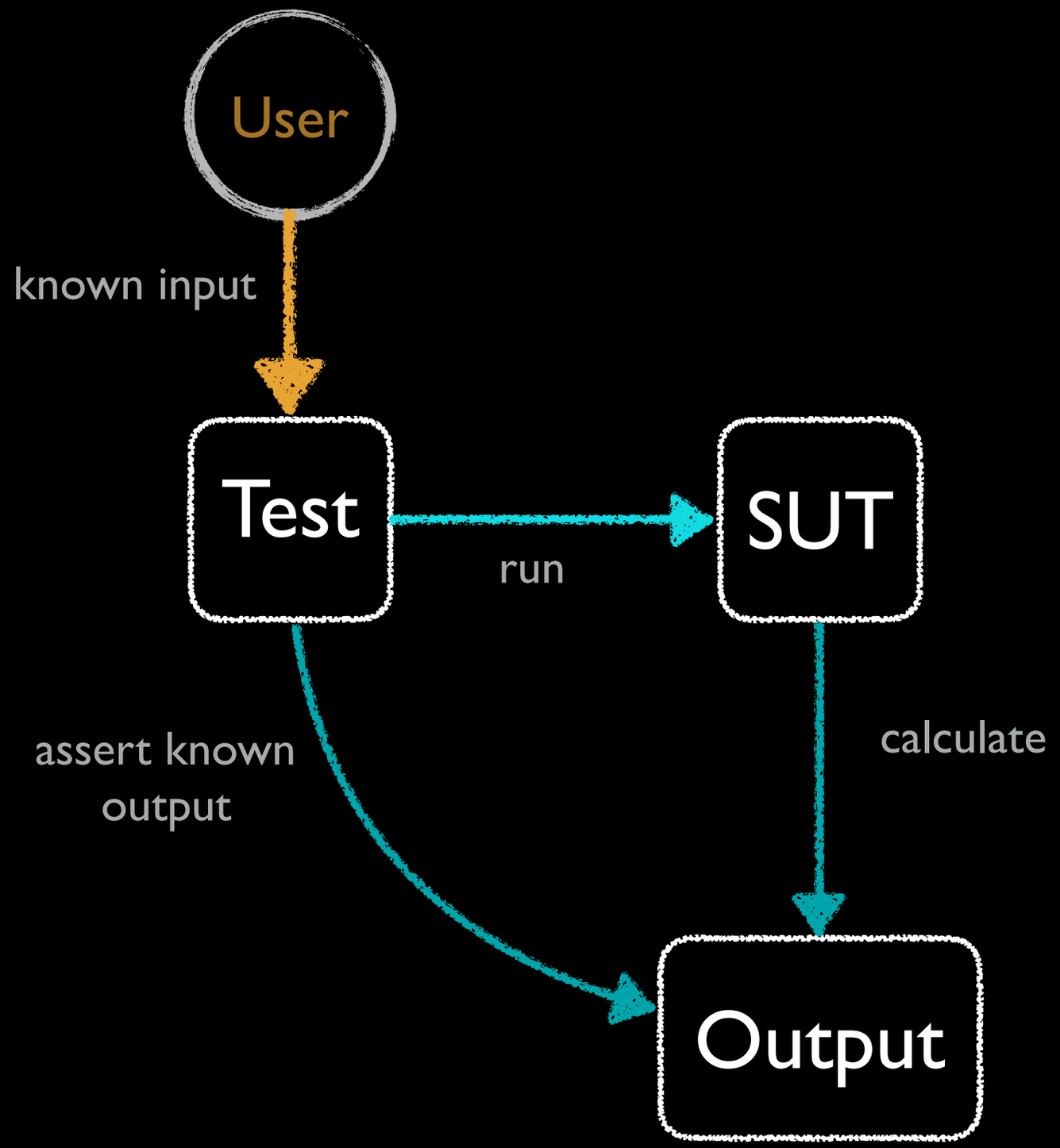
Example-Based Testing

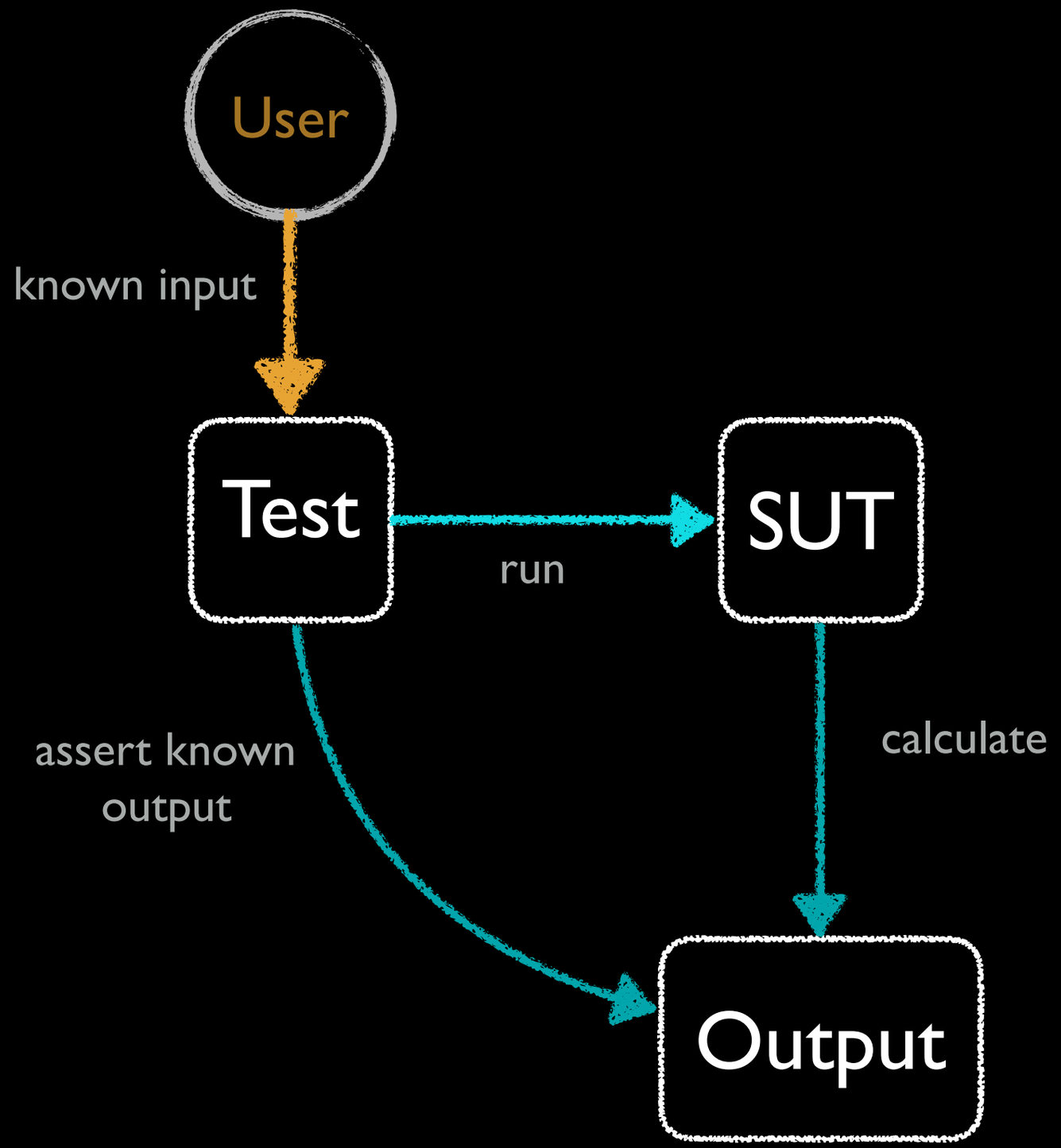
Test


User
known input
Test


User
known input
Test
run
SUT

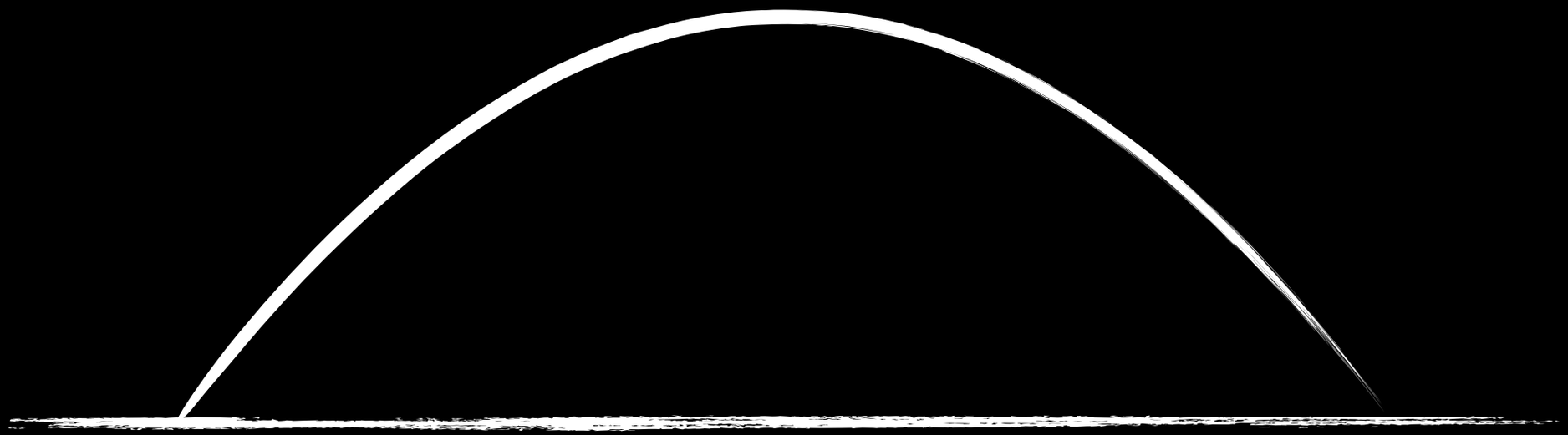

User
known input
Test
run
SUT
calculate
Output


User
known input
Test
run
SUT
assert known
output
calculate
Output

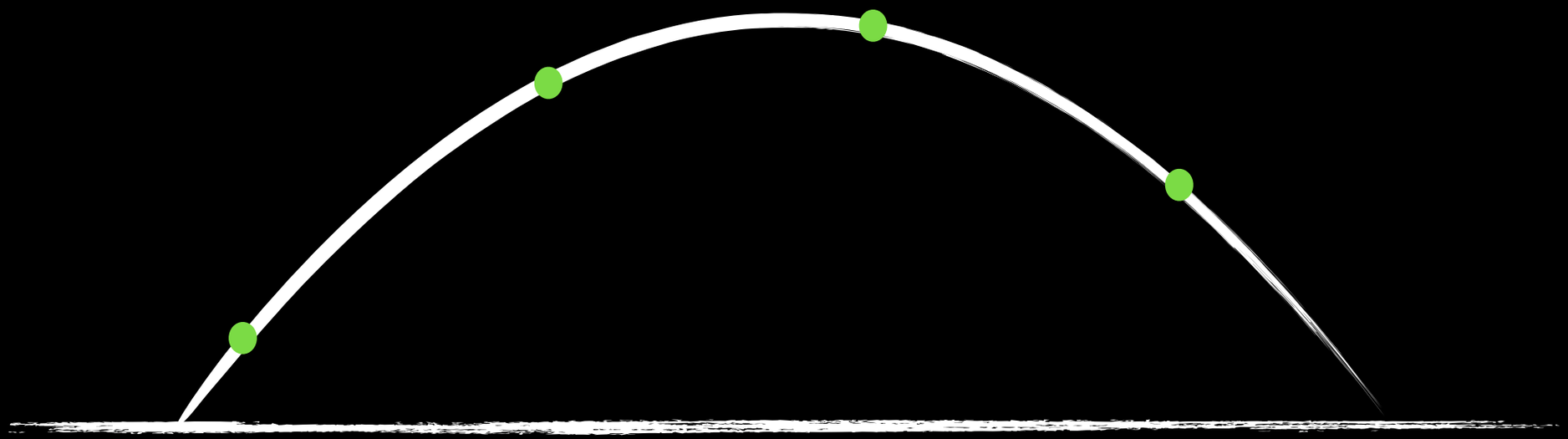

User
known input
Test
run
SUT
assert known output
calculate
Output


known inputs
known output
add(1, 2) should be 3


SENORGIF.COM



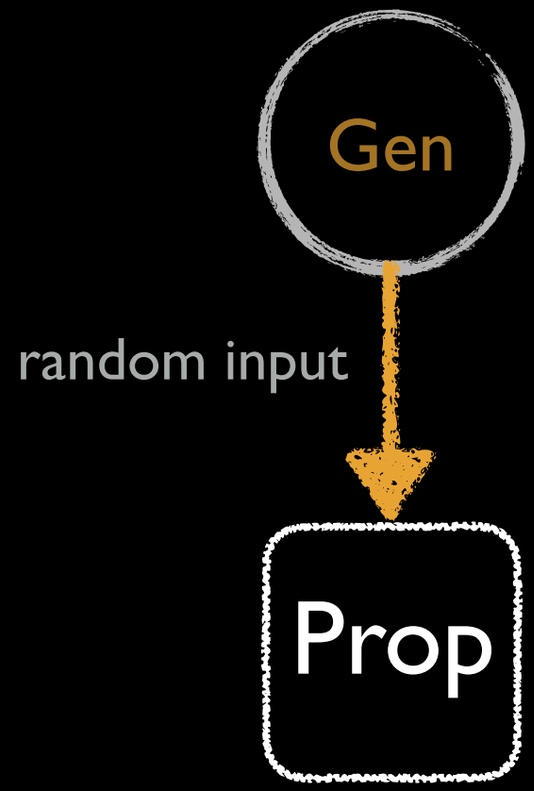
Input Sample

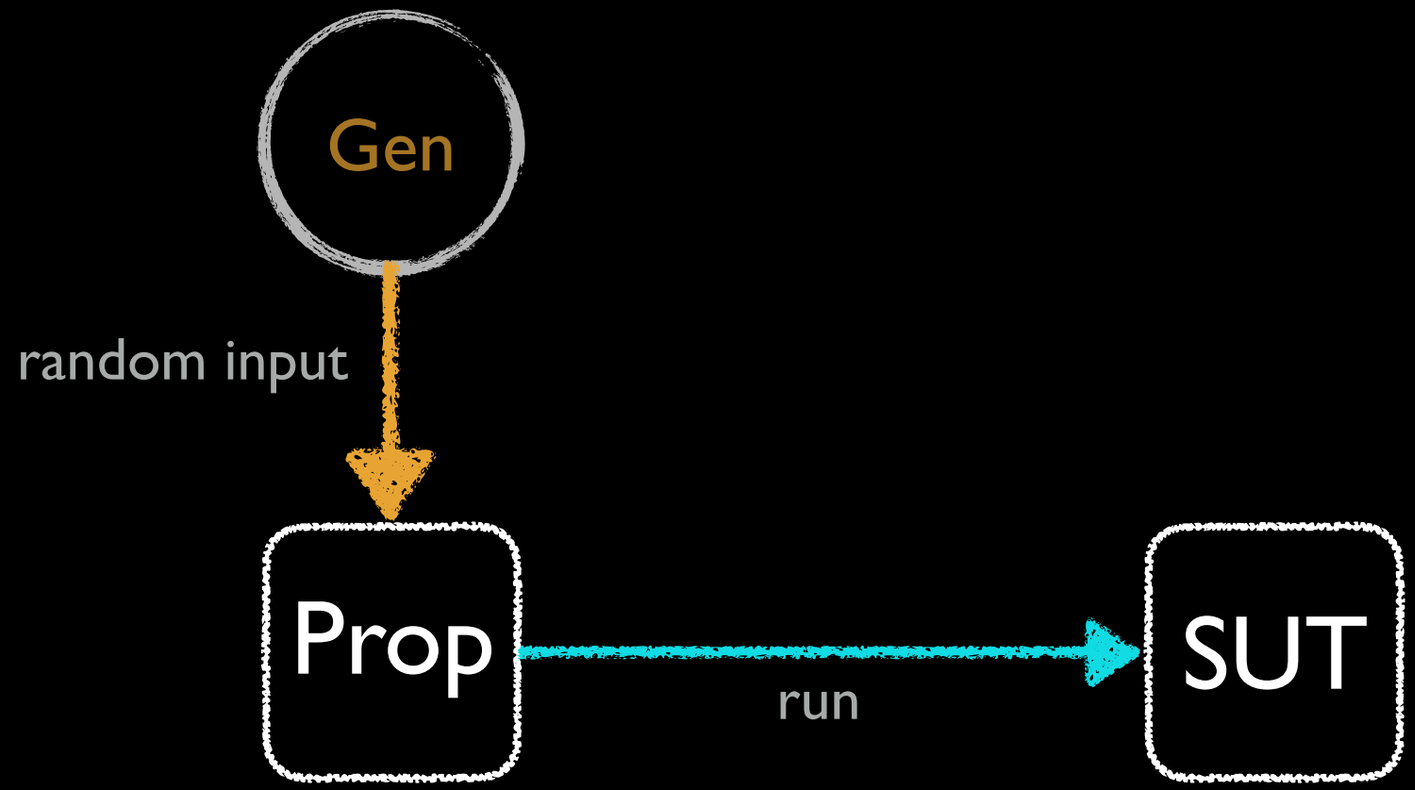


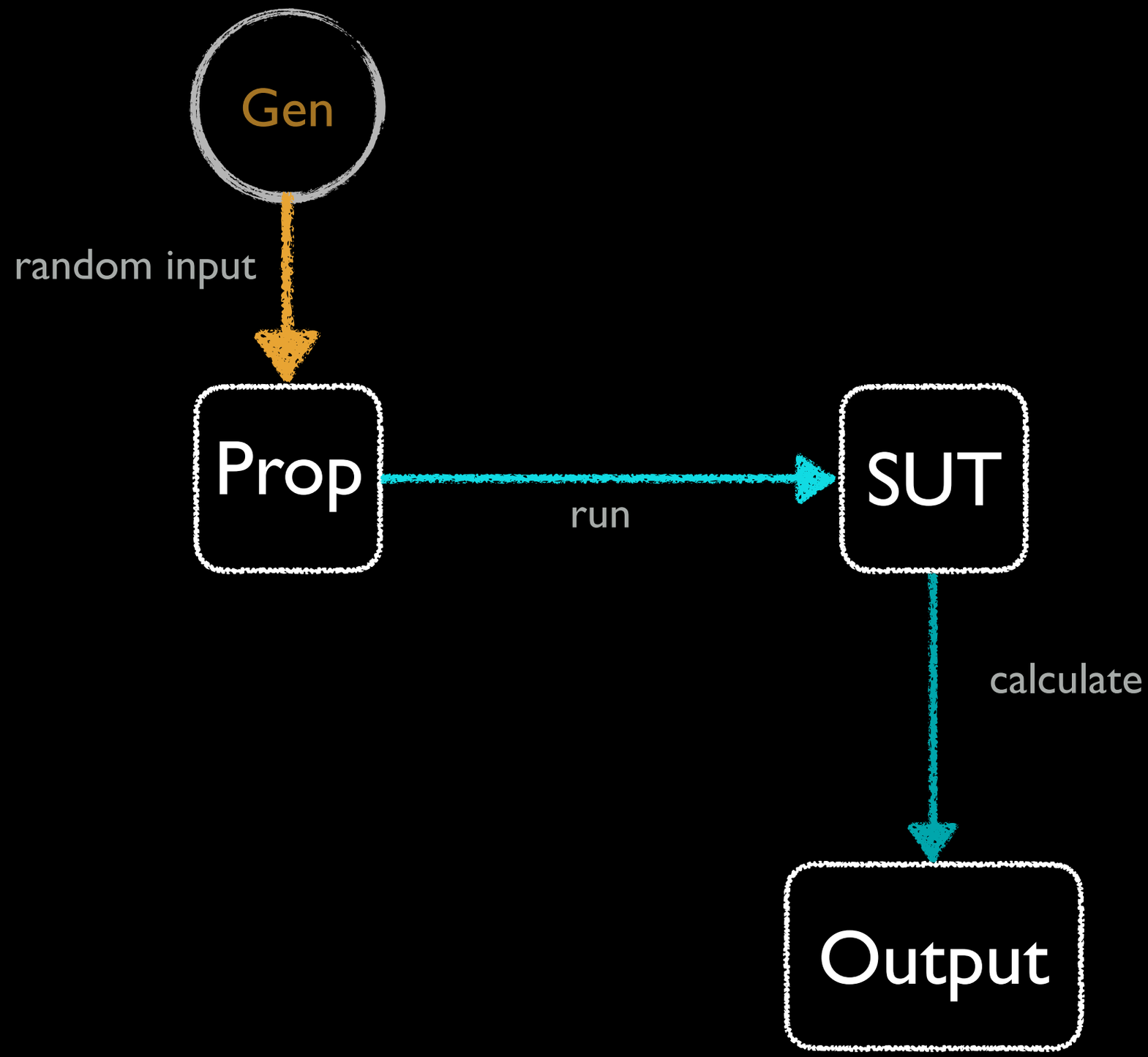
Input Sample

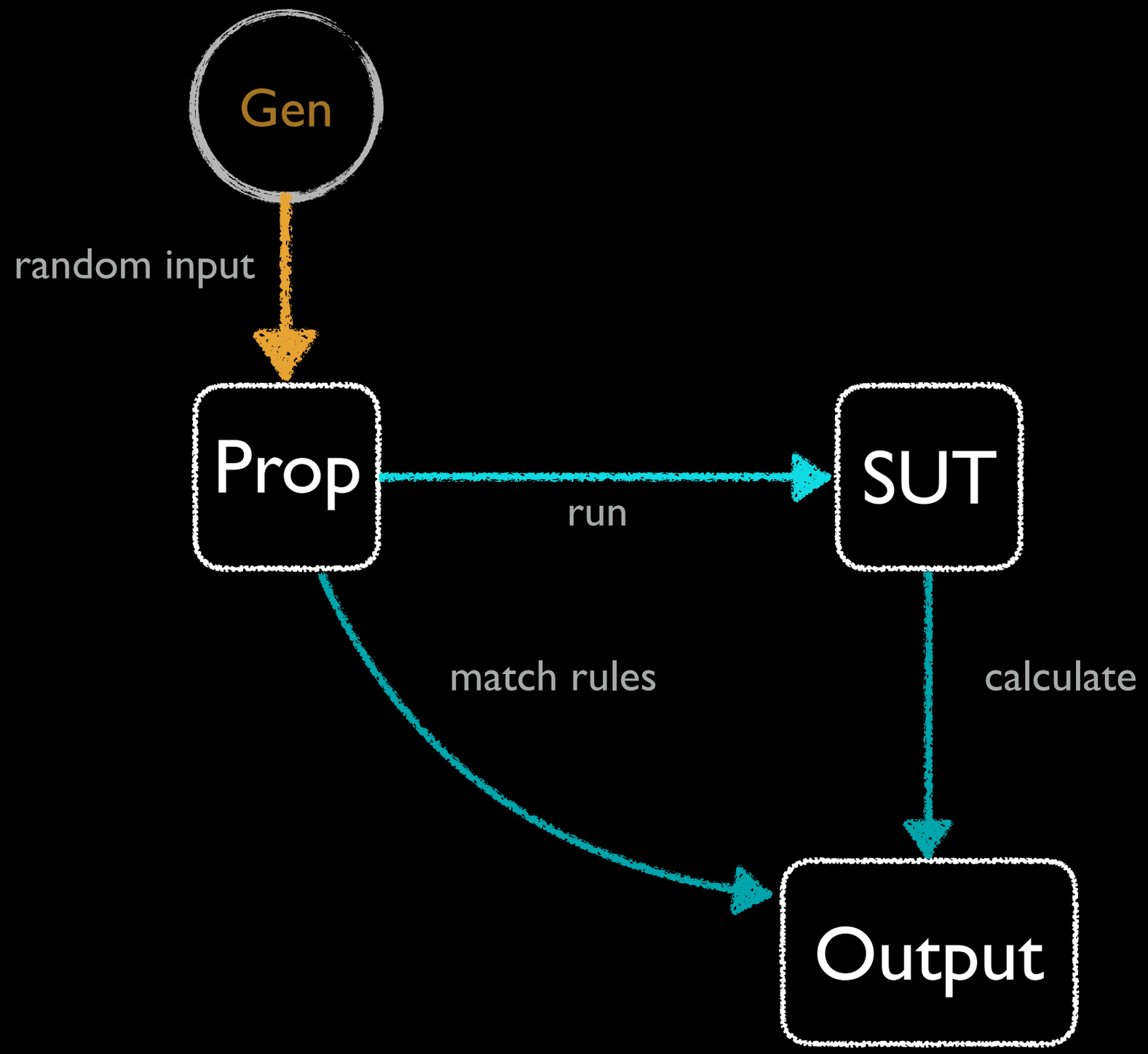
Property-Based Testing

Prop

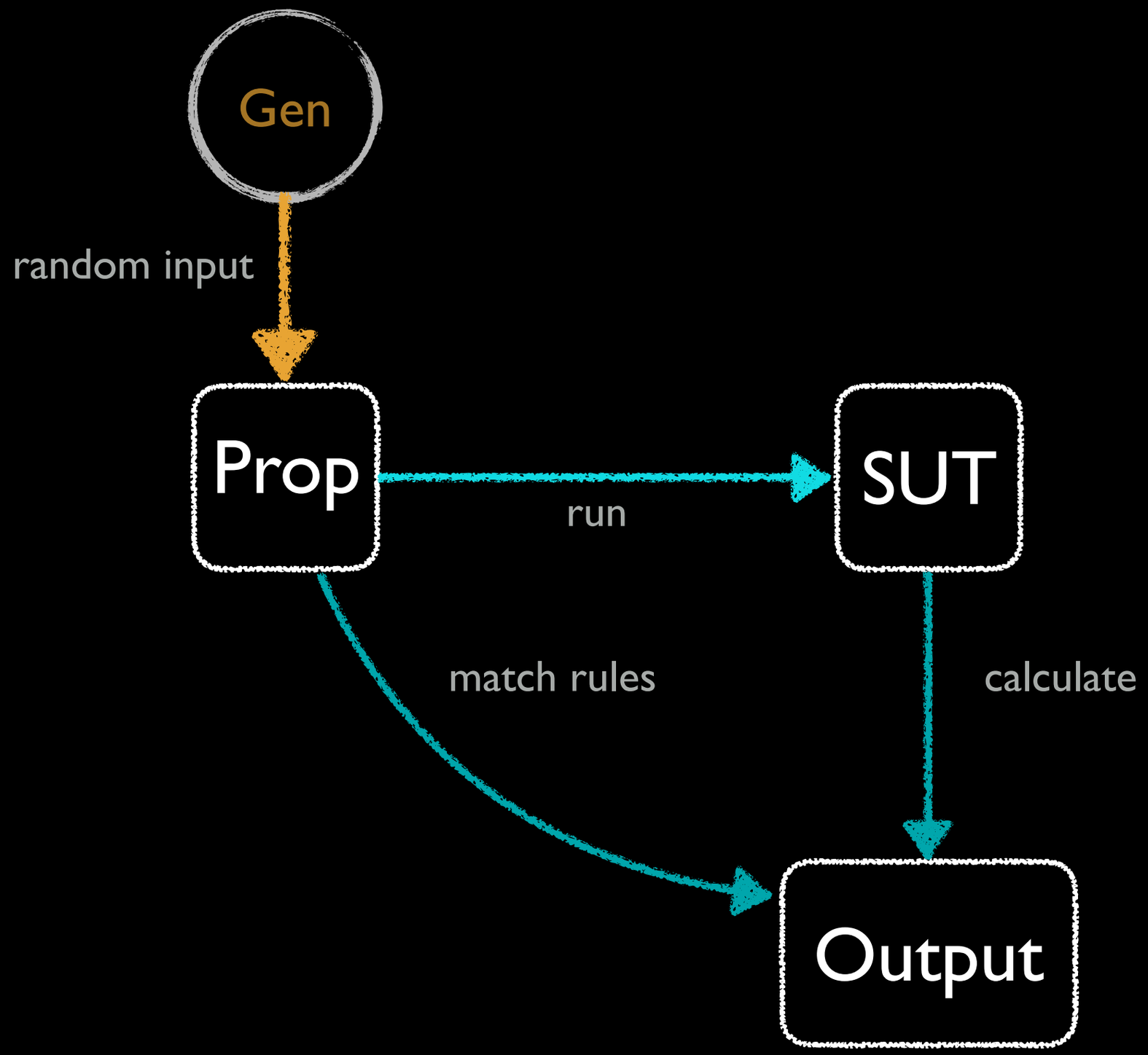

Gen
random input
Prop

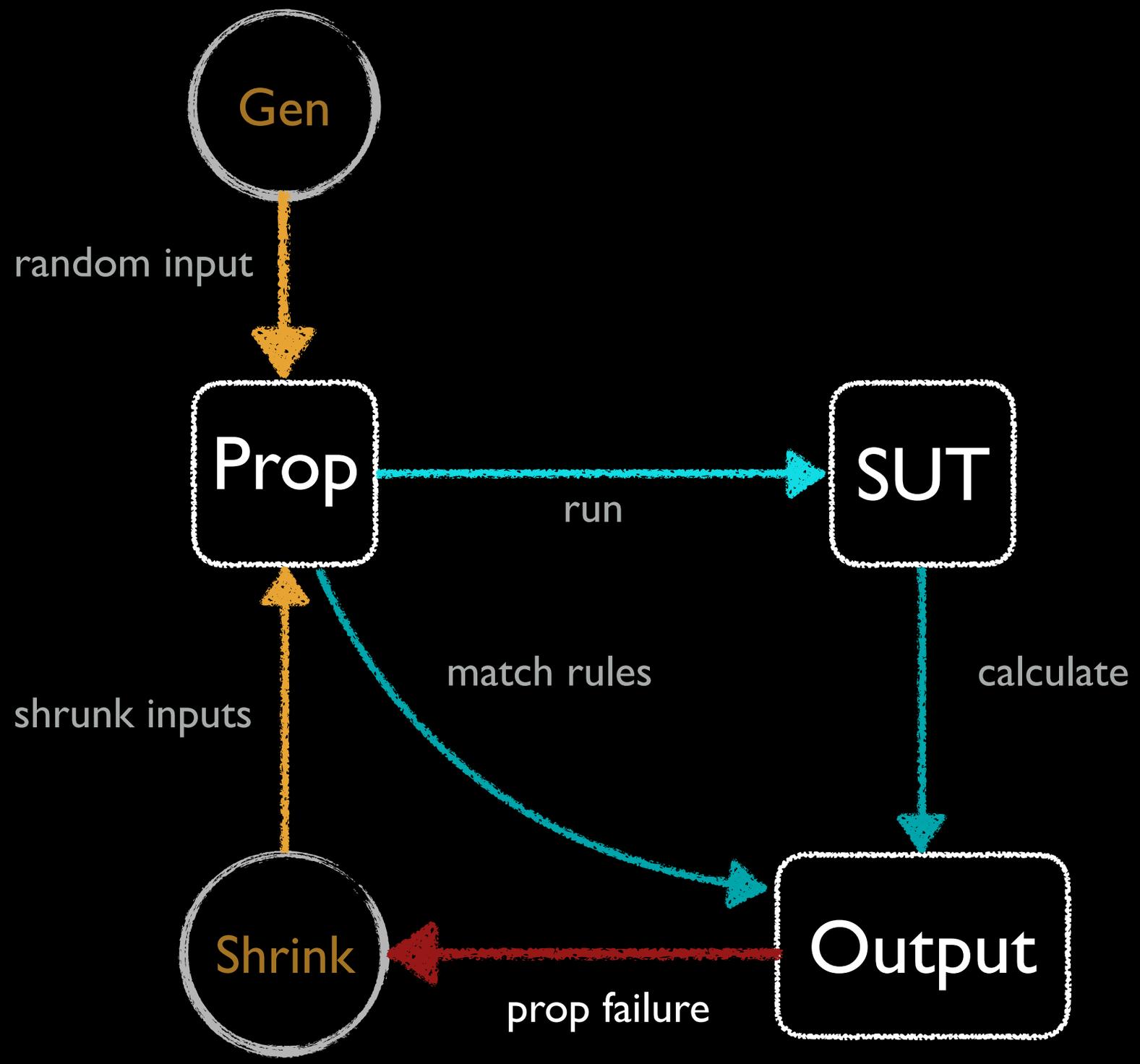

Gen
random input
Prop
run
SUT

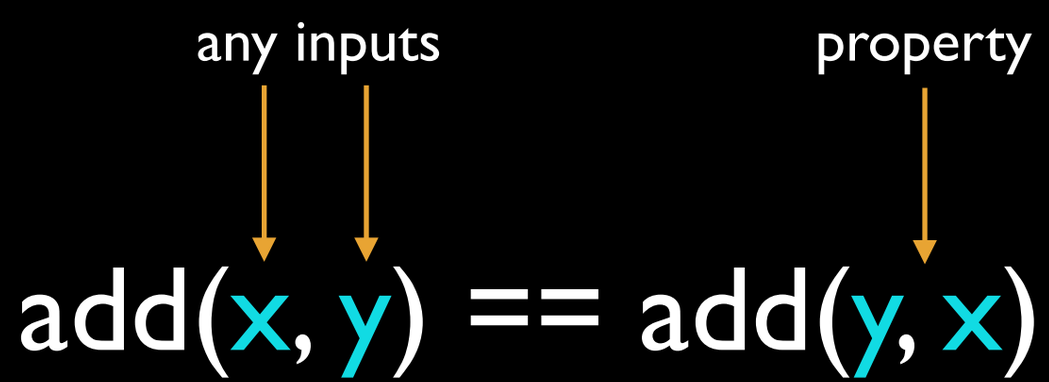

Gen
random input
Prop
run
SUT
calculate
Output

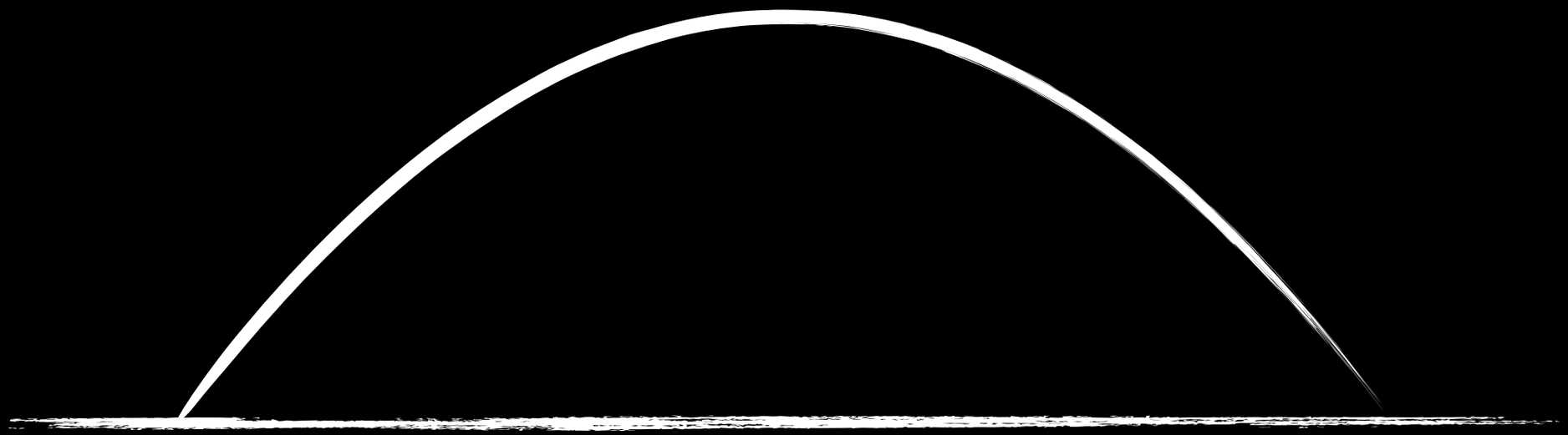

Gen
random input
Prop
run
SUT
match rules
calculate
Output

x100

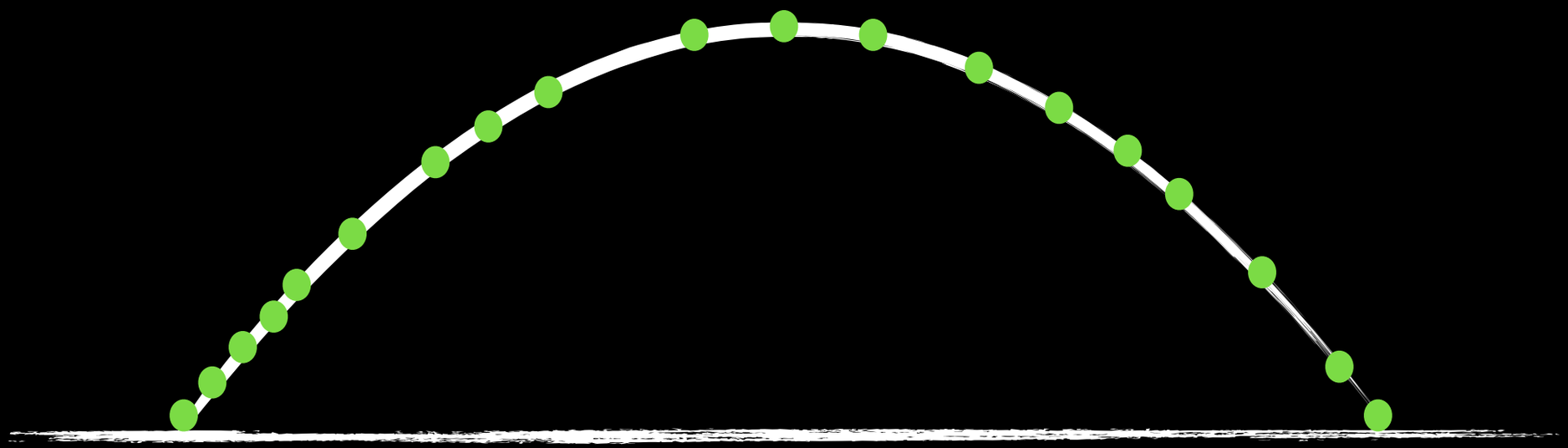



Gen
random input
Prop
run
SUT
match rules
calculate
shrunk inputs
Shrink
prop failure
Output


any inputs
property
add(x, y) == add(y, x)



Input Sample



Input Sample

ScalaCheck

ScalaCheck

Rickard Nilsson

QuickCheck

John Hughes & Koen Claessen

Gen[A]

Arbitrary[A]

Gen[A]

Arbitrary[A]

PersonGenerator

```
Arbitrary[String]
```

Arbitrary[String]

ﾐﾘ嚽巹⍰蟩●銀霘煲⍰釷哰⍰隔沯授禮¦ﾊﾞｰﾚﾙ♌璦濊析ώ䁢豪箇Lبش舩「栴⍰⍰فخ⏬鑣걝尯◌ৃ令♈듷郷霏ㄷ釗瞛흰

```
  oneOf[T](
g0: Gen[T],
g1: Gen[T],
gn: Gen[T]*):
Gen[T]
```

```
  oneOf[T](
g0: Gen[T],
g1: Gen[T],
gn: Gen[T]*): Gen[T]
```

```
oneOf(alphaLowerChar, alphaUpperChar, numChar)
```

```
  oneOf[T](
g0: Gen[T],
g1: Gen[T],
gn: Gen[T]*):
```

Gen[T]

```
oneOf(alphaLowerChar, alphaUpperChar, numChar)
```

3,m,m,3,7,G,5,X,0,i

```
oneOf[T](
   t0:T,
   t1:T,
   tn:T*):
```

Gen[T]

```
oneOf("Red", "Green", "Blue")
```

"Red","Blue","Blue","Blue","Green"

```
listOf[T]
(g: => Gen[T]):
Gen[List[T]]
```

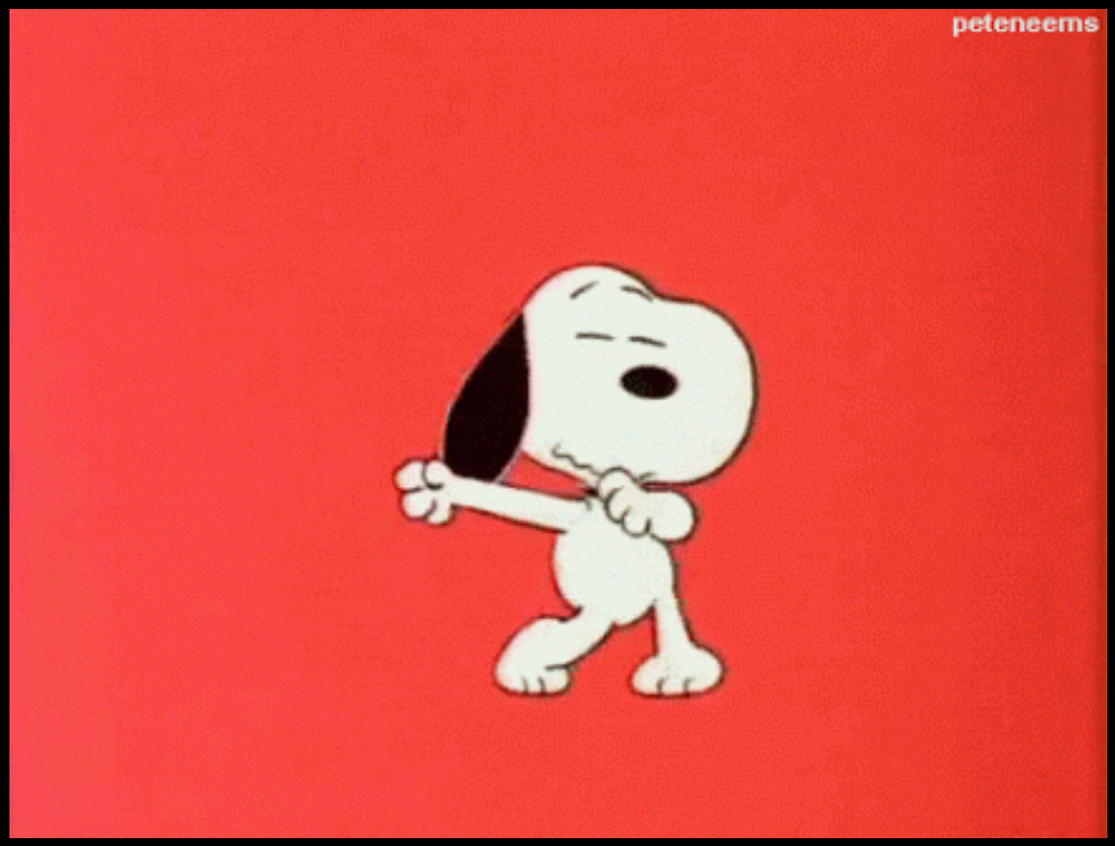
```
def listOf[T]
(g: => Gen[T]):
Gen[List[T]]
```

```
listOf(posNum[Int])
```

```
listOf[T]
(g: => Gen[T]):
Gen[List[T]]
```

```
listOf(posNum[Int])
```

List(72, 5, 86, 34, 76, 68, 39, 26, 6, 36, 83, 40),
List(58, 93, 34, 29, 4, 10, 12, 47, 98, 45, 38, 43, 20, 47, 18, 63, 54, 71),
List(84, 57, 94, 14, 85, 37, 25, 67, 41, 91, 46, 69, 88, 83, 63, 68, 97, 68, 8, 8, 57, 66, 76, 45, 99, 35, 21, 54, 32, 51, 89, 59,
75, 26, 64, 49, 21, 57, 45, 65, 38, 64, 83, 4, 58, 32, 13, 13, 100, 49, 45, 30, 15, 94, 90, 19, 77, 46, 31, 16, 52, 43)



There are many more

http://bit.ly/2oxLFLV

Can be used with EBT

Can be used with EBT

gen.sample

[U]niversally Quantified Properties

[U]niversally Quantified Properties

=> Prop

[U]niversally Quantified Properties

=> Prop
=> Boolean

[U]niversally Quantified Properties

=> Prop
=> Boolean (implicit => Prop)

forAll with Gen

```
forAll(genPerson) { (px: Person) =>
  px.age >= 1 && px.age <= 120
}
```

```
forAll[T1](Gen[T1])
  (T1 => Boolean):
Prop
```

simplified function definition

x8

```
forAll[T1](Gen[T1])
  (T1 => Boolean):
Prop
```

simplified function definition

forAll with Arbitrary

```
forAll { (n1: Int, n2: Int) =>
  add(n1, n2) == add(n2, n1)
}
```

```
forAll[T1]
(T1 ⇒ Boolean):
Prop
```

simplified function definition

x8

```
forAll[T1]
(T1 ⇒ Boolean):
Prop
```

simplified function definition

Shrinking

```
forAll { n: Int =>
  n / 100 == 0
}
```

```
T => Stream[T]
```

Choosing Properties

Mλthematical Laws

Laws of Int Addition

Associativity	(a+b)+c	== a+(b+c)
Commutativity	(a+b)	== (b+a)
Identity	(a+∅)	== a
	(∅+a)	== a
Distribution	x(a+b)	== xa+xb

If your problem domain has laws, used them!

Questions

How can I explain this to a deceptive computer?

How can I explain this to a deceptive computer?
How is this similar to …?

How can I explain this to a deceptive computer?
How is this similar to …?
How is this different from …?

How can I explain this to a deceptive computer?
How is this similar to …?
How is this different from …?
Can I verify this without duplicating the CUT?

How can I explain this to a deceptive computer?
How is this similar to …?
How is this different from …?
Can I verify this without duplicating the CUT?
How can I make it fail?

Patterns

Scott Wlasin

fsharpforfunandprofit

Charles O'Farrell

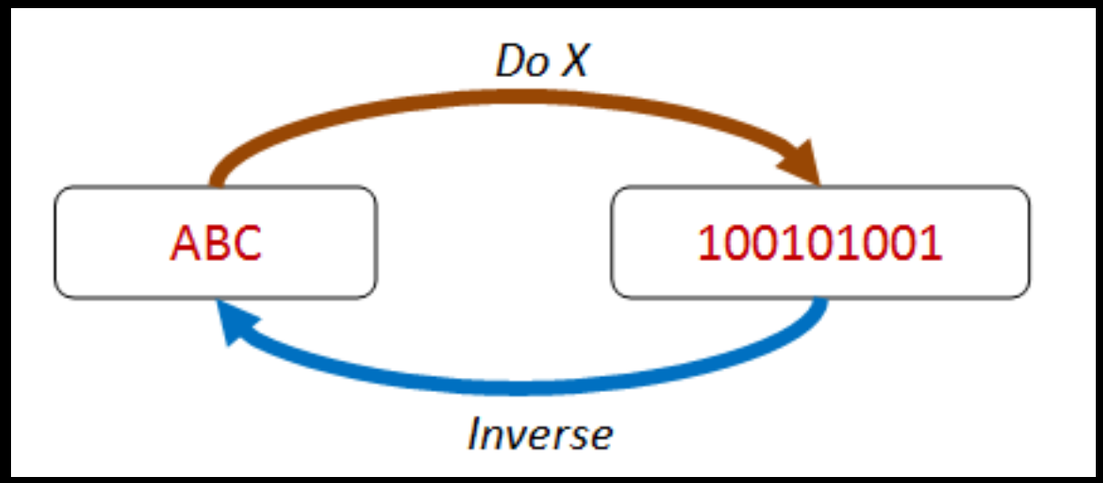
Yow! Lamda Jam

Invariants

Length
Contents

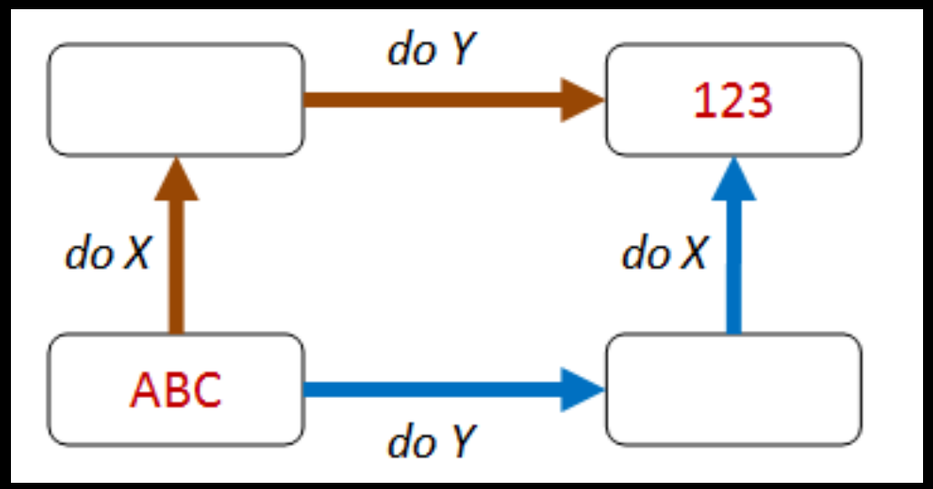
list.sorted.length == list.length

Round-tripping



json.parse.toJson == json

Different Order Same Result



list.map(_ + 1).sorted == list.sorted.map(_ + 1)

Compose Methods

```
list2.reverse ++ list1.reverse
==
(list1 ++ list2).reverse
```

Test Oracle

Verify against another implementation

multithreaded result == single-threaded result

jsonLibX result == jsonLibY result

There are others

http://bit.ly/2o6DKsy

Examples

Math.abs

Math.abs

Determine the magnitude of a value by discarding the sign. Results are >= 0.

Diamond Kata

http://claysnow.co.uk/recycling-tests-in-tdd/

Given a capital letter, print a diamond starting with ‘A’.

Given a capital letter, print a diamond starting with ‘A’.

The supplied letter must be at the widest point.

Given a capital letter, print a diamond starting with ‘A’.

The supplied letter must be at the widest point.

Use ‘*’ to denote a space between characters and a ‘-‘ to denote a space around characters.

printDiamond(‘A’)

```
A
```

printDiamond(‘B’)

```
-A-
B*B
-A-
```

printDiamond(‘C’)

```
--A--
-B*B-
C***C
-B*B-
--A--
```

Diamond Kata

DiamondTest/DiamondProps

John Hughes - Testing the Hard Stuff and Staying Sane

Clojure/West

March 24-26 2014

The Palace Hotel San Francisco

factual

STAPLES

Before

- Files over 1GB?
- Rehashing?
- > 6 weeks of effort!

After

- Database with *one* record!
- 5—6 calls to reproduce
- < 1 day to fix

46:41 47:17

Summary

EBT

EBT

Easy to understand

EBT

Easy to understand
Quick to implement

EBT

Easy to understand
Quick to implement
Good for implementations with few combinations

EBT

Easy to understand
Quick to implement
Good for implementations with few combinations
Needed for regression

EBT

Easy to understand
Quick to implement
Good for implementations with few combinations
Needed for regression

EBT

Easy to understand
Quick to implement
Good for implementations with few combinations
Needed for regression

Limited by developer's imagination

EBT

Easy to understand
Quick to implement
Good for implementations with few combinations
Needed for regression

Limited by developer's imagination
Hard to test complex implementations

EBT

Easy to understand
Quick to implement
Good for implementations with few combinations
Needed for regression

Limited by developer's imagination
Hard to test complex implementations
Easy to miss edge cases

EBT

Easy to understand
Quick to implement
Good for implementations with few combinations
Needed for regression

Limited by developer's imagination
Hard to test complex implementations
Easy to miss edge cases
Boring to write

PBT

PBT

Edge cases for free

PBT

Edge cases for free
Hundreds of tests

PBT

Edge cases for free
Hundreds of tests
Reusable Generators

PBT

Edge cases for free
Hundreds of tests
Reusable Generators
More thinking involved

PBT

Edge cases for free
Hundreds of tests
Reusable Generators
More thinking involved
Good for complex implementations

PBT

Edge cases for free
Hundreds of tests
Reusable Generators
More thinking involved
Good for complex implementations

PBT

Edge cases for free
Hundreds of tests
Reusable Generators
More thinking involved
Good for complex implementations

Requires investment in learning techniques

PBT

Edge cases for free
Hundreds of tests
Reusable Generators
More thinking involved
Good for complex implementations

Requires investment in learning techniques
More thinking involved

PBT

Edge cases for free
Hundreds of tests
Reusable Generators
More thinking involved
Good for complex implementations

Requires investment in learning techniques
More thinking involved
Have to write Generators/Shrinkers

PBT

Edge cases for free
Hundreds of tests
Reusable Generators
More thinking involved
Good for complex implementations

Requires investment in learning techniques
More thinking involved
Have to write Generators/Shrinkers
Not good for regression

EBT + PBT = WIN

EBT

EBT

Basic cases

EBT

Basic cases
Regression (bugs)

EBT

Basic cases
Regression (bugs)

PBT

EBT

Basic cases
Regression (bugs)

PBT

Edge cases

EBT

Basic cases
Regression (bugs)

PBT

Edge cases
Complex outcomes

EBT

Basic cases
Regression (bugs)

PBT

Edge cases
Complex outcomes
Explore libraries

EBT

Basic cases
Regression (bugs)

PBT

Edge cases
Complex outcomes
Explore libraries
Deeper understand of the problem

Links

The lazy programmer's guide to writing 1000's of tests - Scott Wlaschin

I Dream of Gen'ning - Kelsey Gilmore-Innis

Practical Property-Based Testing - Charles O'Farrell

Property-Based Testing for Better Code - Jessica Kerr

Testing the Hard Stuff and Staying Sane - John Hughes

Thank You

Thank You!