From Properties to Tests

COS 326
David Walker

The lazy programmer's guide to writing 1000's of tests

An introduction to property based testing

@ScottWlaschin
fsharpforfunandprofit.com

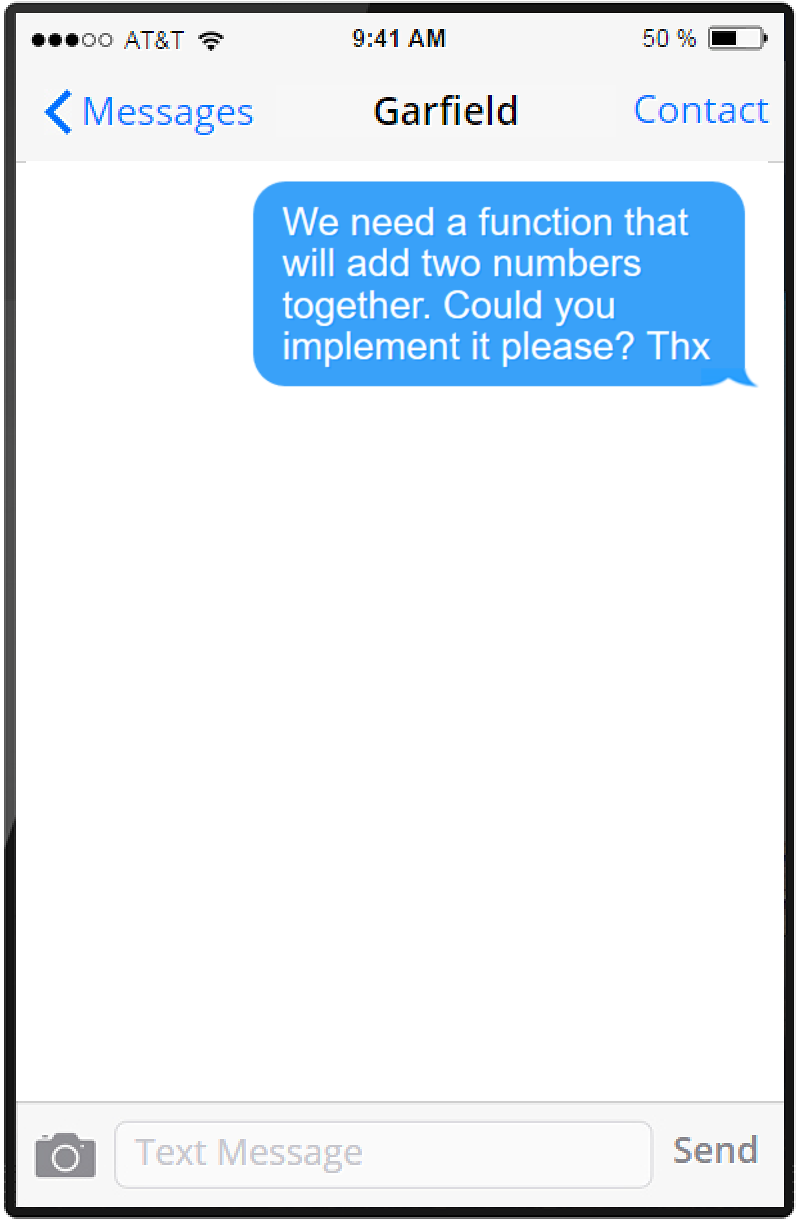
https://fsharp.org/

F# is a OCaml grafted onto C#, which is Microsoft's version of Java

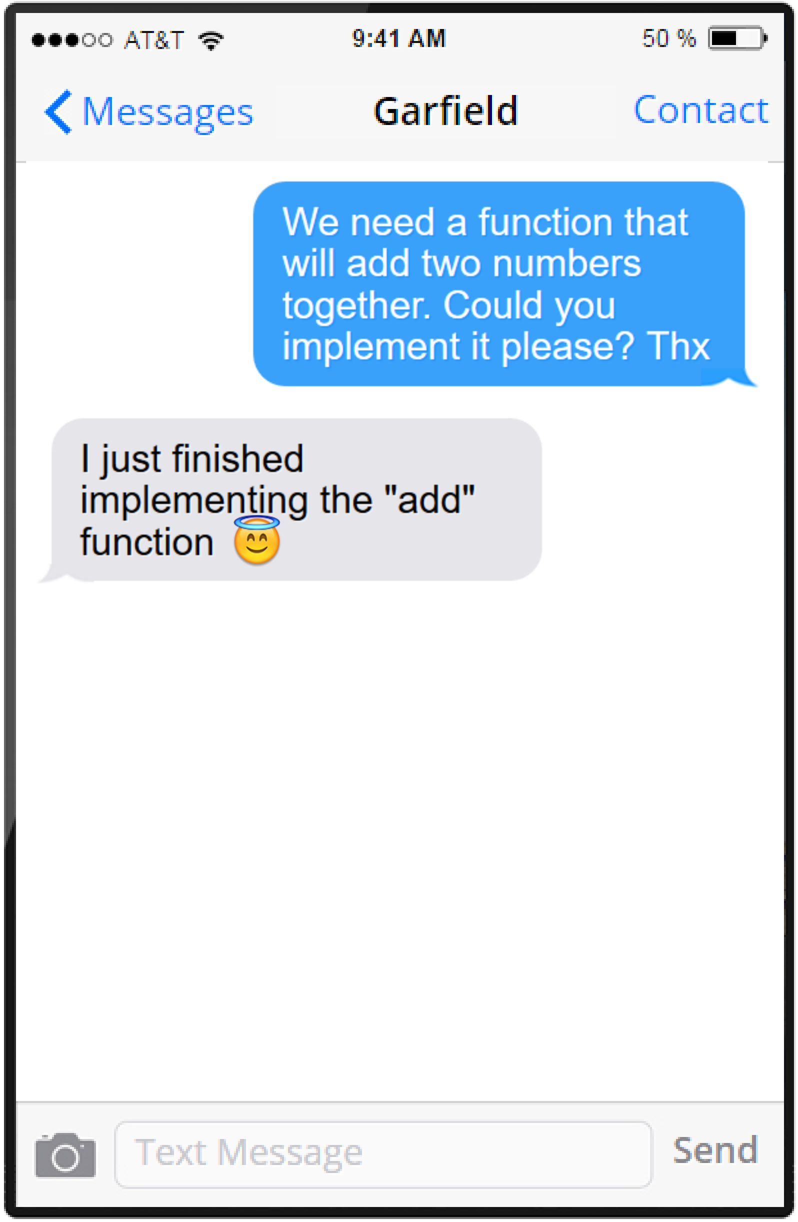
Part 1:
In which I have a conversation with a remote developer

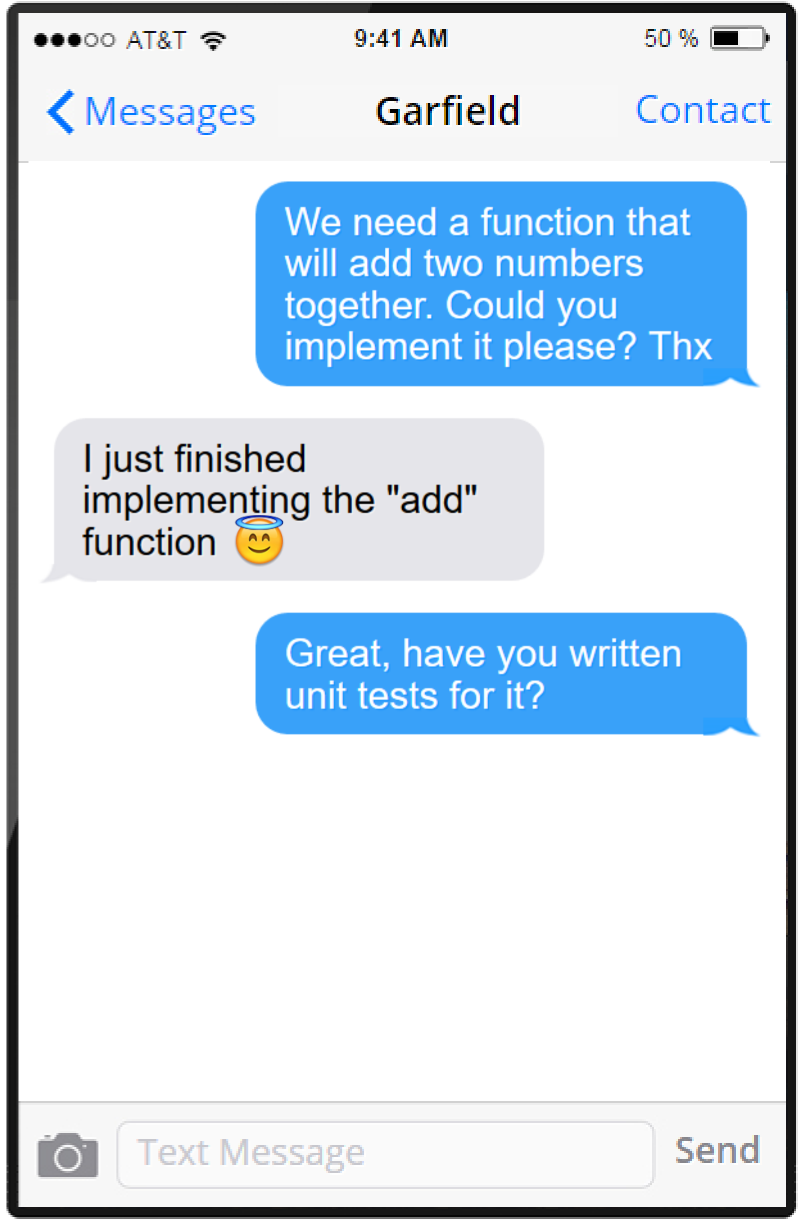
This was a project from a long time ago,
in a galaxy far far away

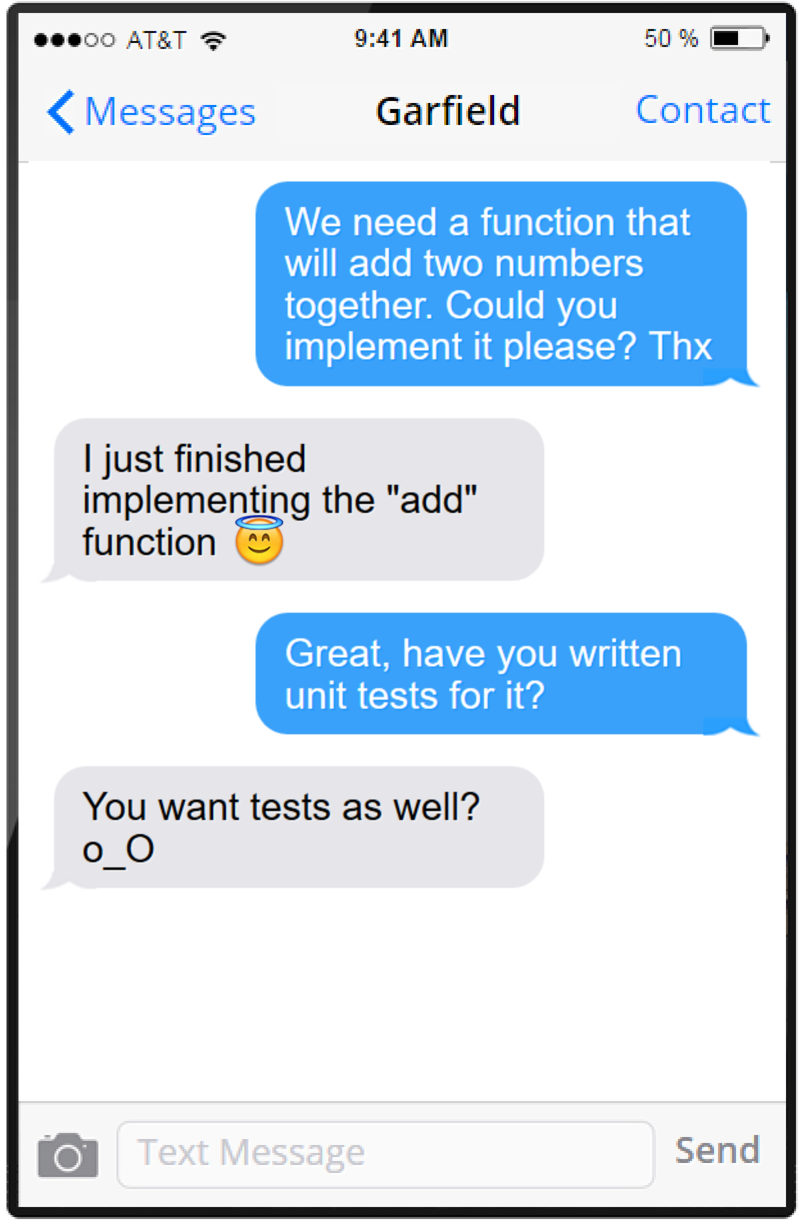
For some reason we needed a custom "add" function

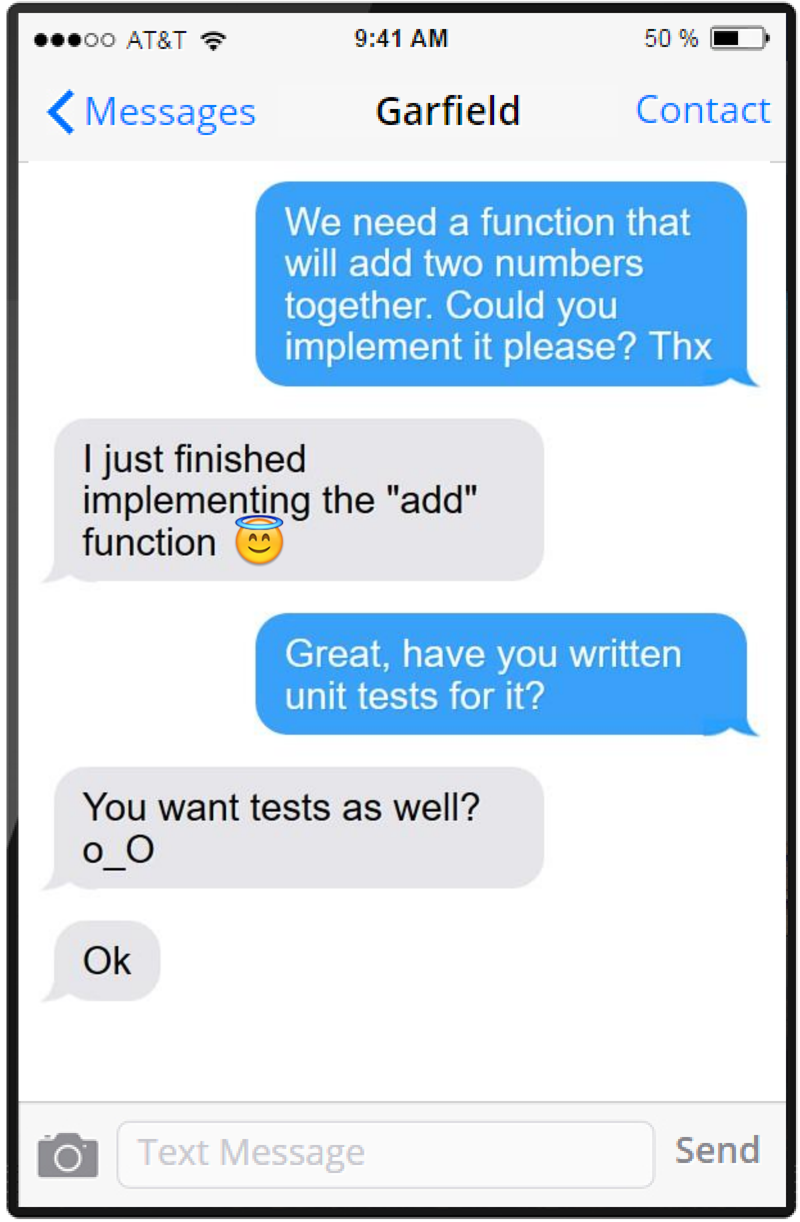


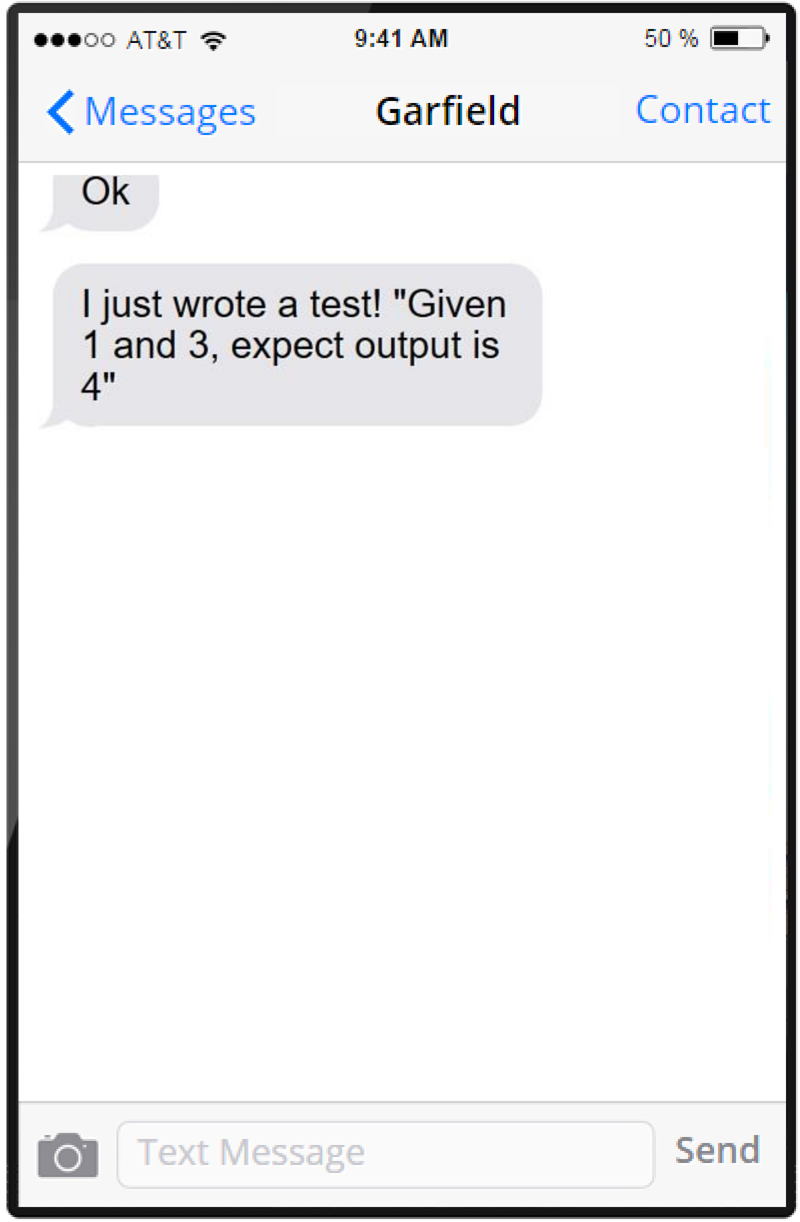
...some time later

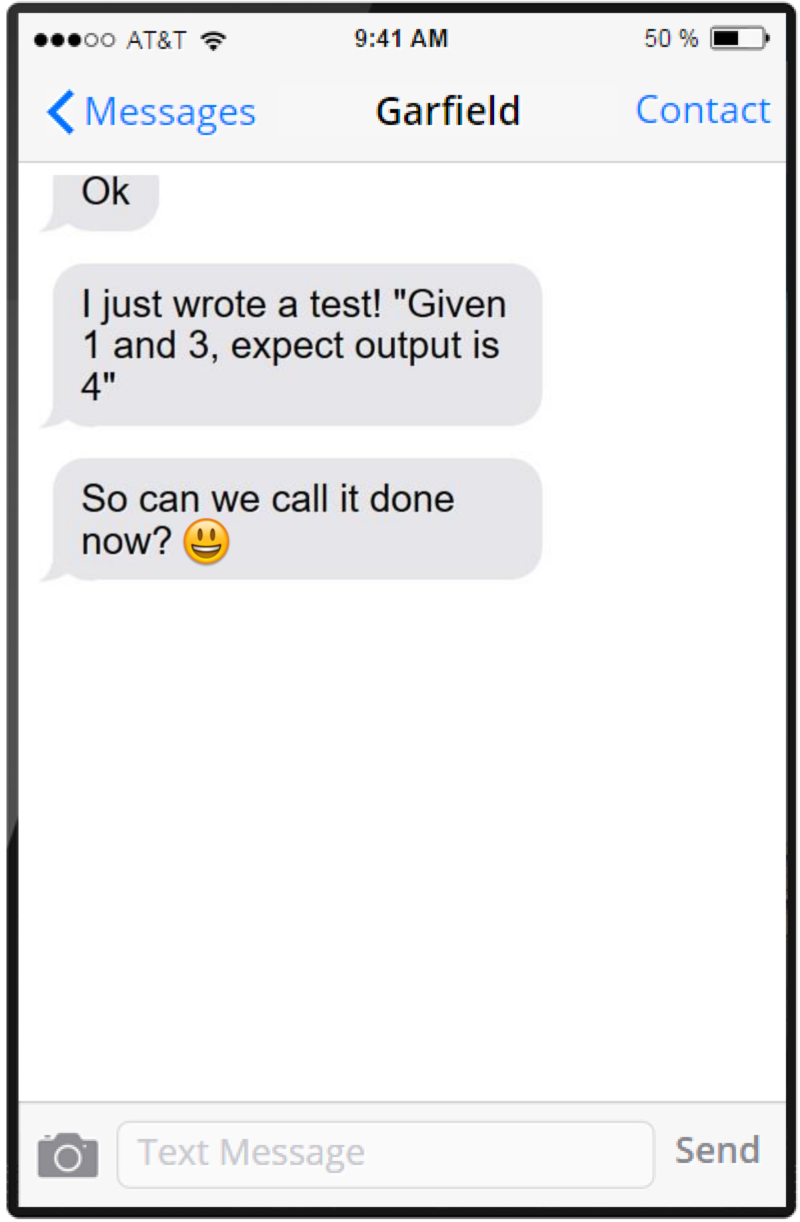



●●●○○ AT&T
9:41 AM
50 %
Messages
Garfield
Contact
We need a function that will add two numbers together. Could you implement it please? Thx
I just finished implementing the "add" function 😇
Great, have you written unit tests for it?
Text Message
Send

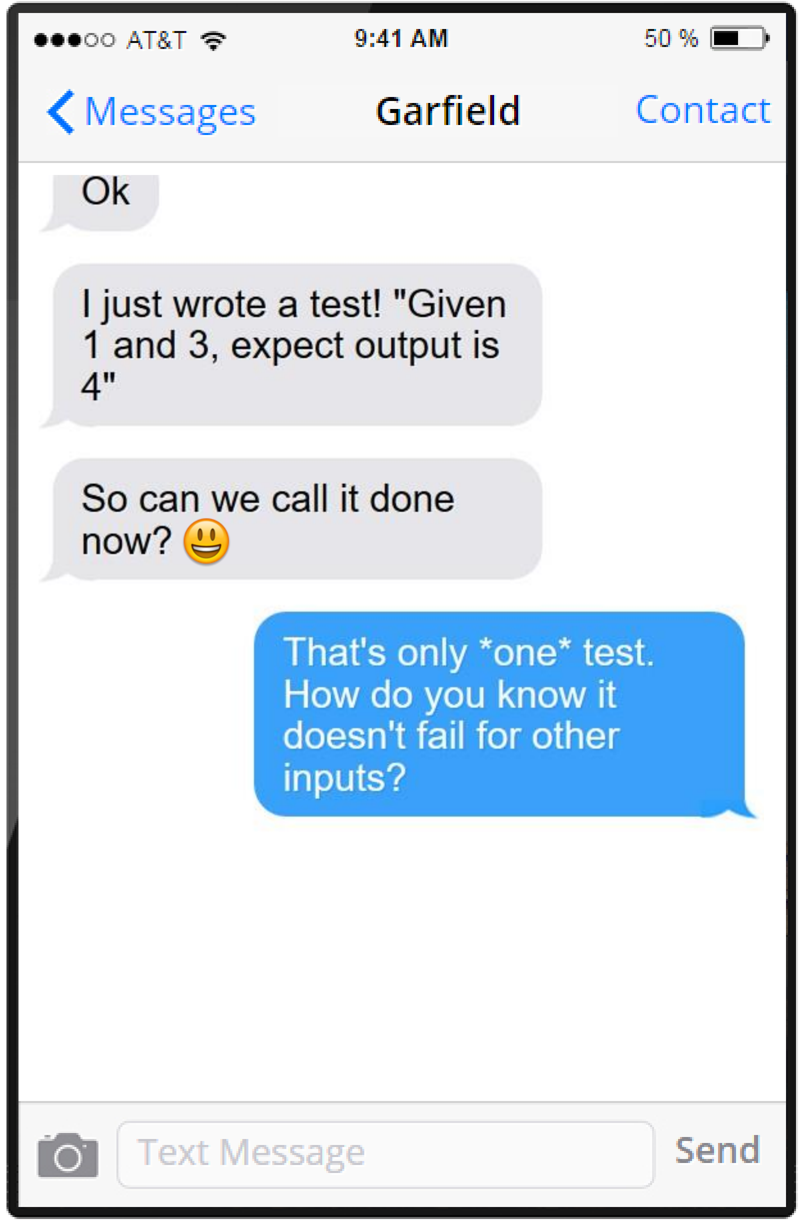

●●●○○ AT&T
9:41 AM
50 %
Messages
Garfield
Contact
We need a function that will add two numbers together. Could you implement it please? Thx
I just finished implementing the "add" function 😇
Great, have you written unit tests for it?
You want tests as well? o_O
Text Message
Send

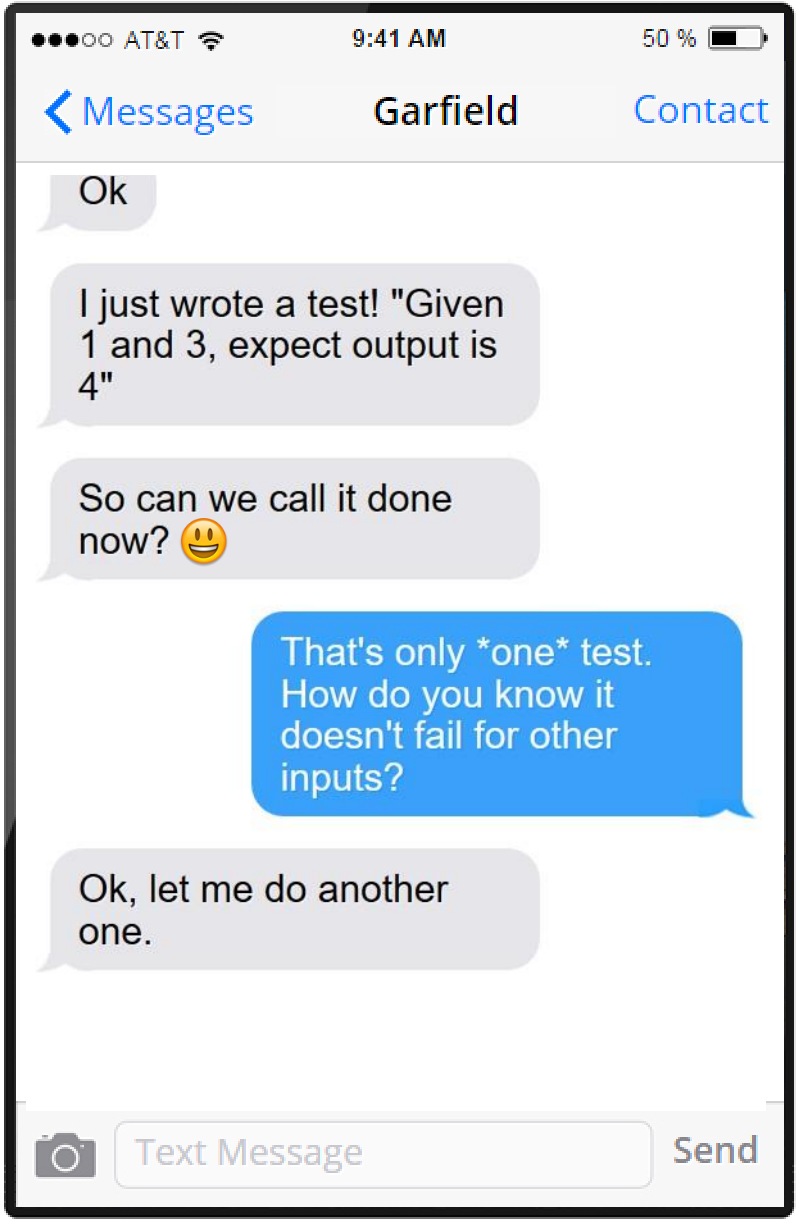

●●●○○ AT&T
9:41 AM
50 %
Messages
Garfield
Contact
We need a function that will add two numbers together. Could you implement it please? Thx
I just finished implementing the "add" function 😇
Great, have you written unit tests for it?
You want tests as well? o_O
Ok
Text Message
Send

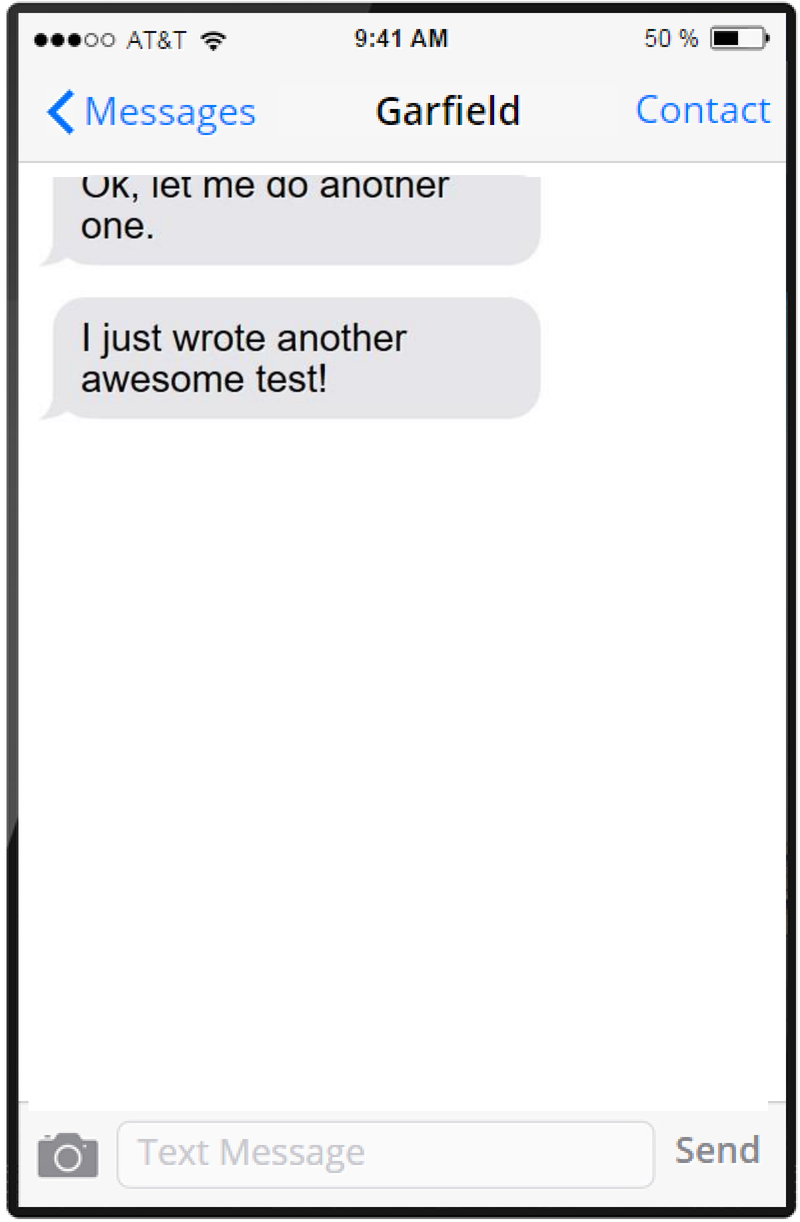

●●●○○ AT&T
9:41 AM
50 %
Messages
Garfield
Contact
Ok
I just wrote a test! "Given 1 and 3, expect output is 4"
Text Message
Send

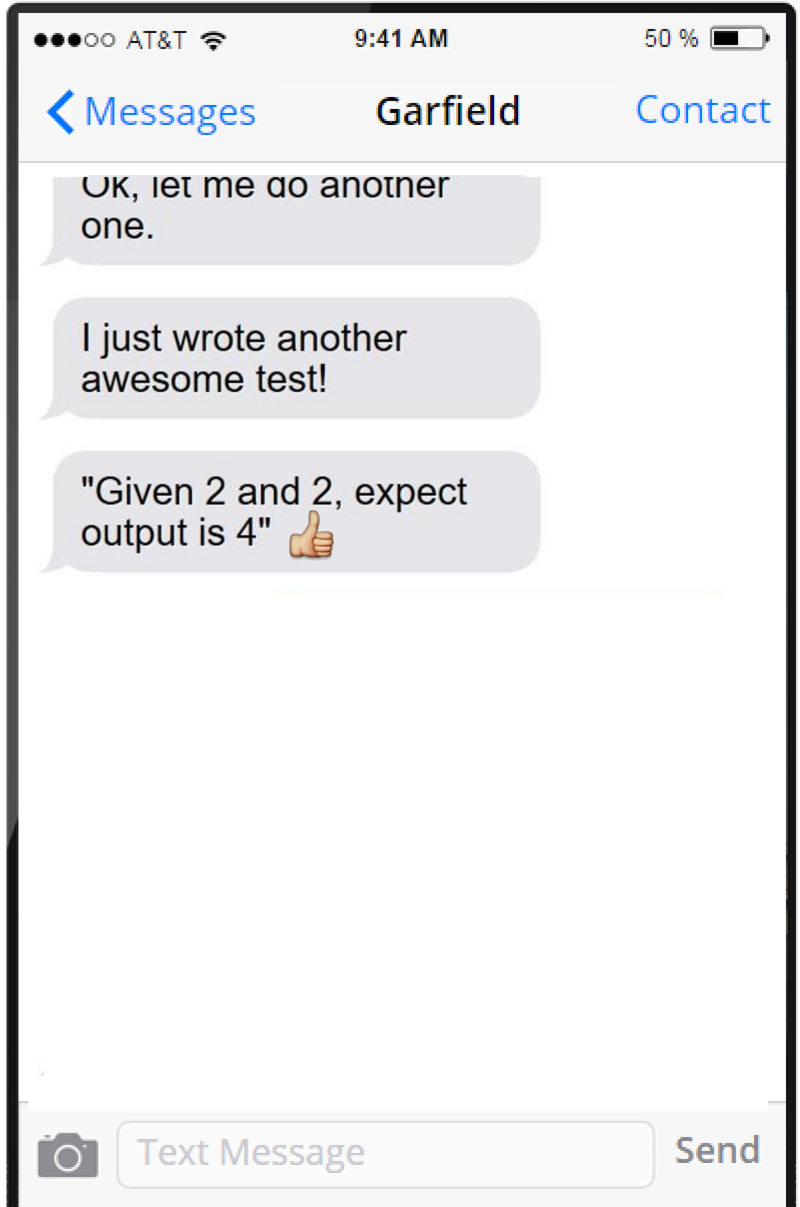


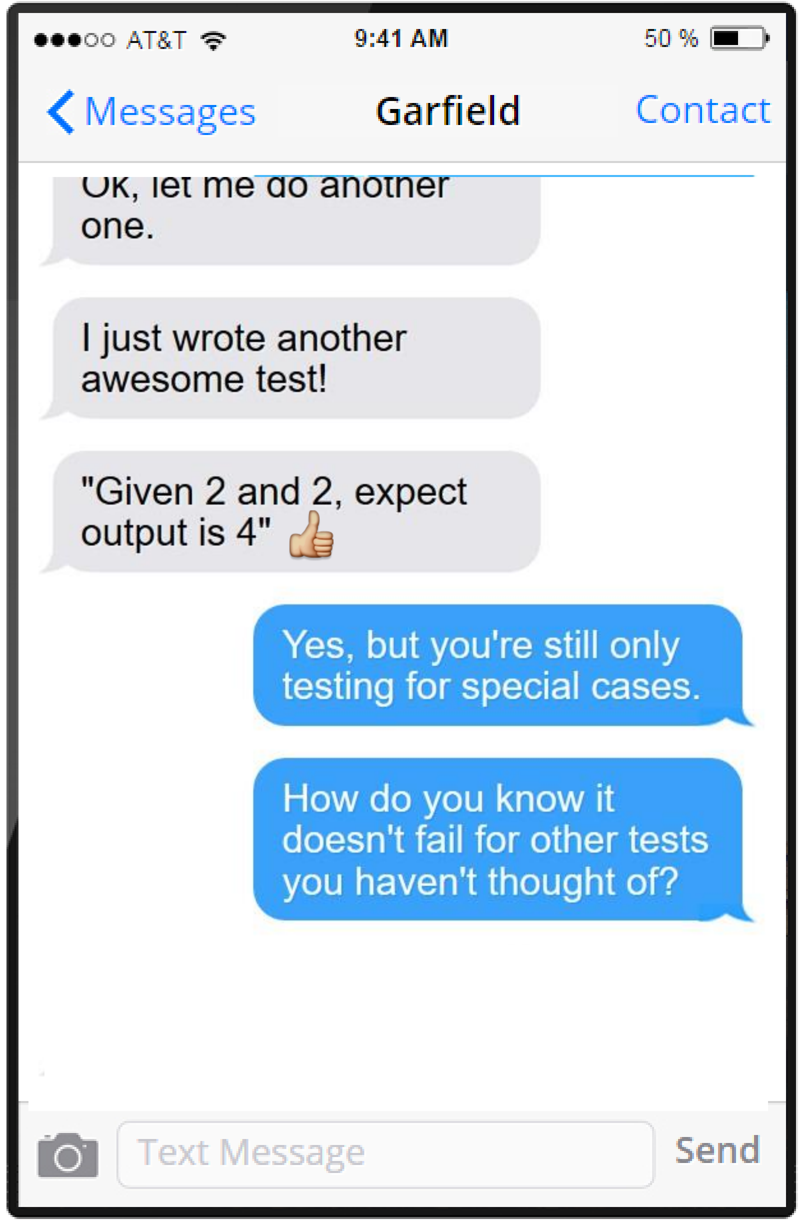
AT&T
9:41 AM
50 %
Messages
Garfield
Contact
Ok
I just wrote a test! "Given 1 and 3, expect output is 4"
So can we call it done now?
Text Message
Send

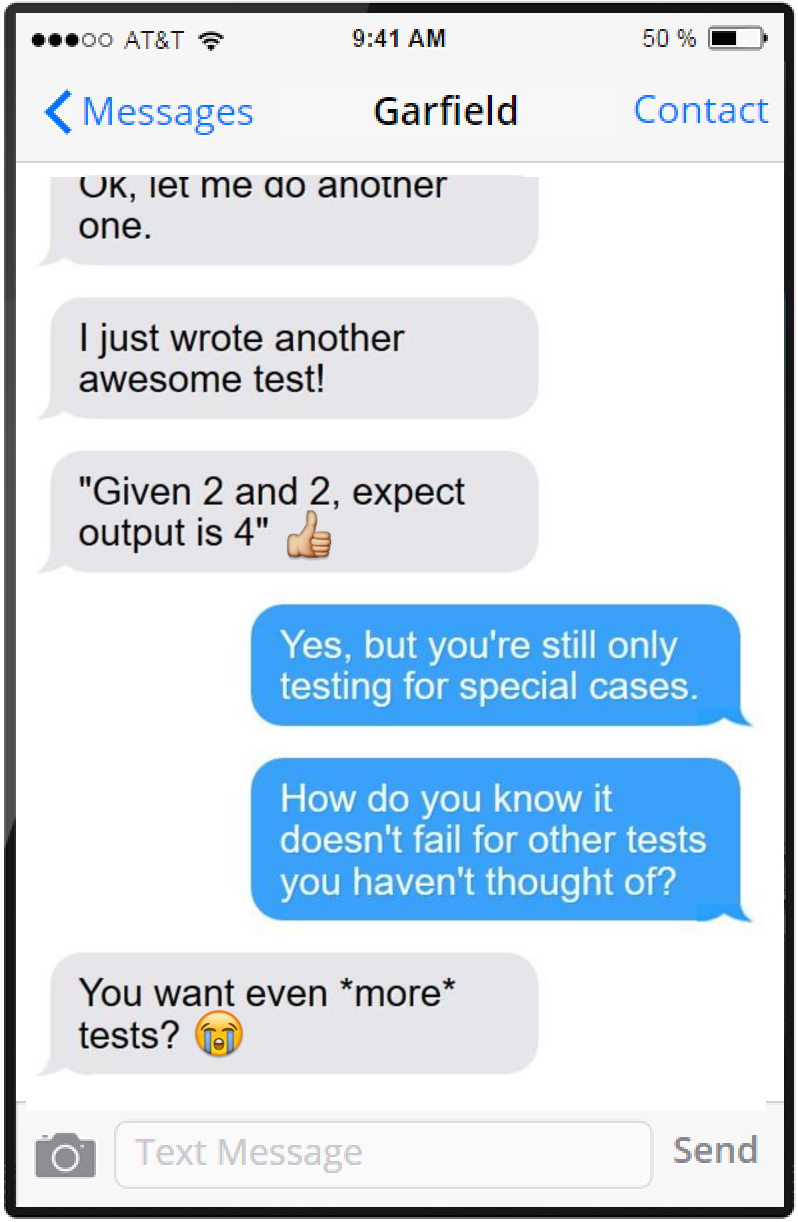

●●●○○ AT&T
9:41 AM
50 %
Messages
Garfield
Contact
Ok
I just wrote a test! "Given 1 and 3, expect output is 4"
So can we call it done now? 😃
That's only *one* test. How do you know it doesn't fail for other inputs?
Text Message
Send


●●●○○ AT&T
9:41 AM
50 %
Messages
Garfield
Contact
Ok
I just wrote a test! "Given 1 and 3, expect output is 4"
So can we call it done now? 😃
That's only *one* test. How do you know it doesn't fail for other inputs?
Ok, let me do another one.
Text Message
Send


●●●○○ AT&T
9:41 AM
50 %
Messages
Garfield
Contact
Ok, let me do another one.
I just wrote another awesome test!
Text Message
Send


●●●○○ AT&T
9:41 AM
50 %
Messages
Garfield
Contact
Ok, let me do another one.
I just wrote another awesome test!
"Given 2 and 2, expect output is 4" 👍
Text Message
Send


●●●○○ AT&T
9:41 AM
50 %
Messages
Garfield
Contact
Ok, let me do another one.
I just wrote another awesome test!
"Given 2 and 2, expect output is 4" 👍
Yes, but you're still only testing for special cases.
How do you know it doesn't fail for other tests you haven't thought of?
Text Message
Send



Seriously, how *do* you know that you have enough tests?

Seriously, how *do* you know that you have enough tests?

So I decide to start writing the unit tests myself

First, I had a look at the existing tests...

```
[<Test>]
let ``When I add 1 + 3, I expect 4``()=
  let result = add(1,3)
  Assert.AreEqual(4,result)
```

```
[<Test>]
let ``When I add 2 + 2, I expect 4``()=
  let result = add(2,2)
  Assert.AreEqual(4,result)
```

prints out stuff when test fails

Ok, now for my first new test...

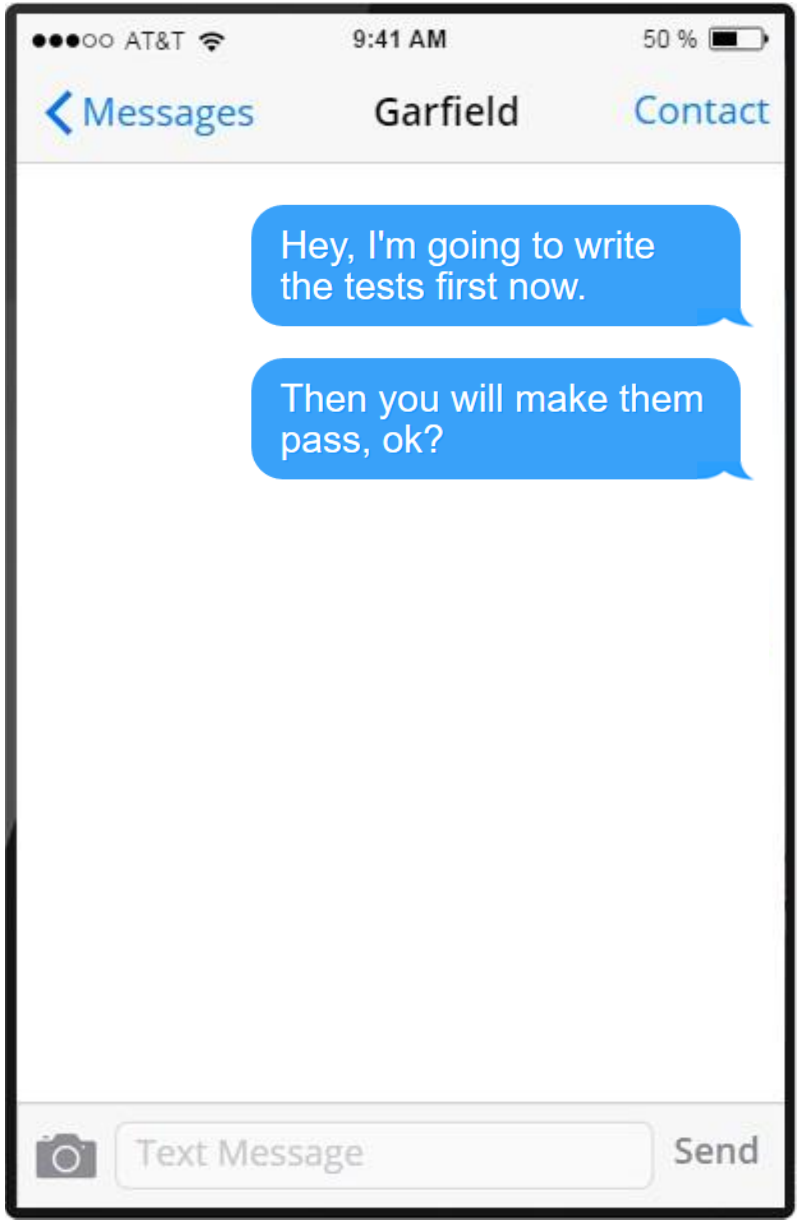
```
[<Test>]
let ``When I add -1 + 3, I expect 2``() =
  let result = add(-1,3)
  Assert.AreEqual(2,result)
```

That's funny..

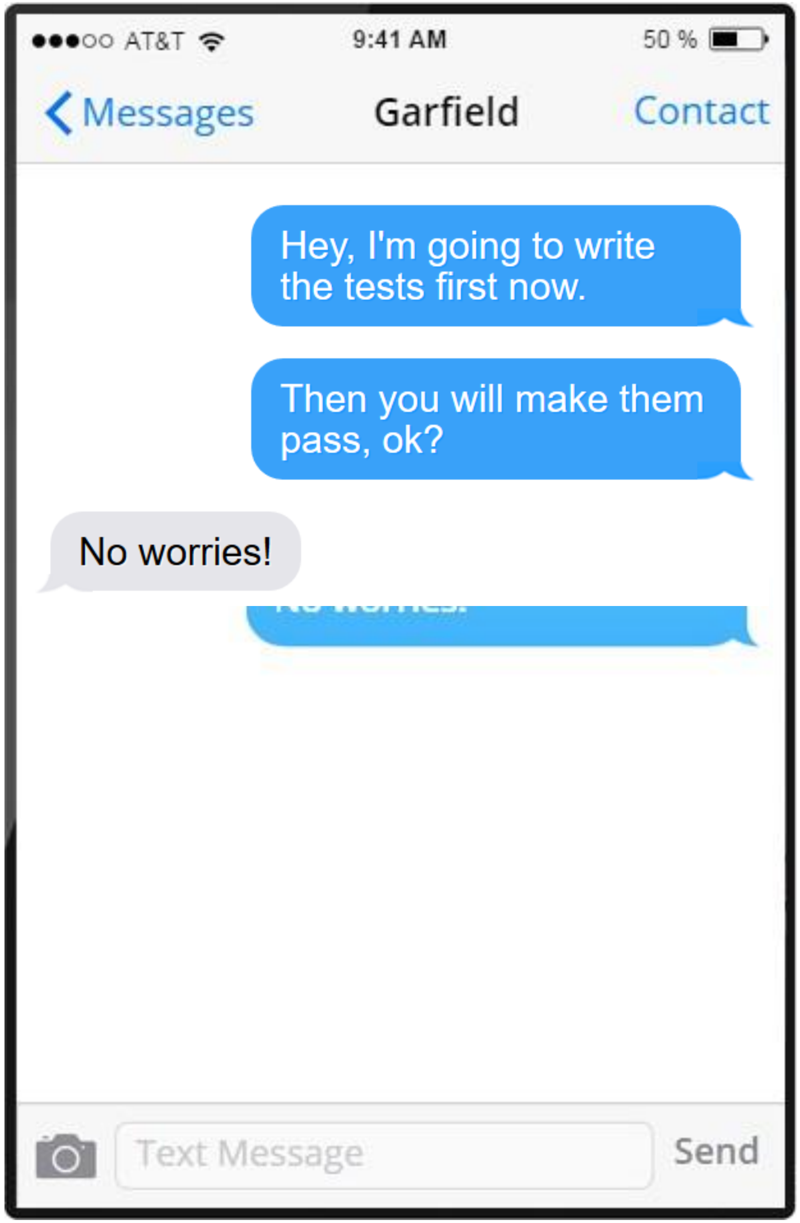
Hmm… let's look at the implementation…

```
let add(x,y) =
   4
```

wtf!



TDD = "Test-Driven Development"


●●●○○ AT&T
9:41 AM
50 %
Messages
Garfield
Contact
Hey, I'm going to write the tests first now.
Then you will make them pass, ok?
No worries!
Text Message
Send

Time for some more tests...

```
[<Test>]
let ``When I add 2 + 3, I expect 5``()=
  let result = add(2,3)
  Assert.AreEqual(5,result)
```

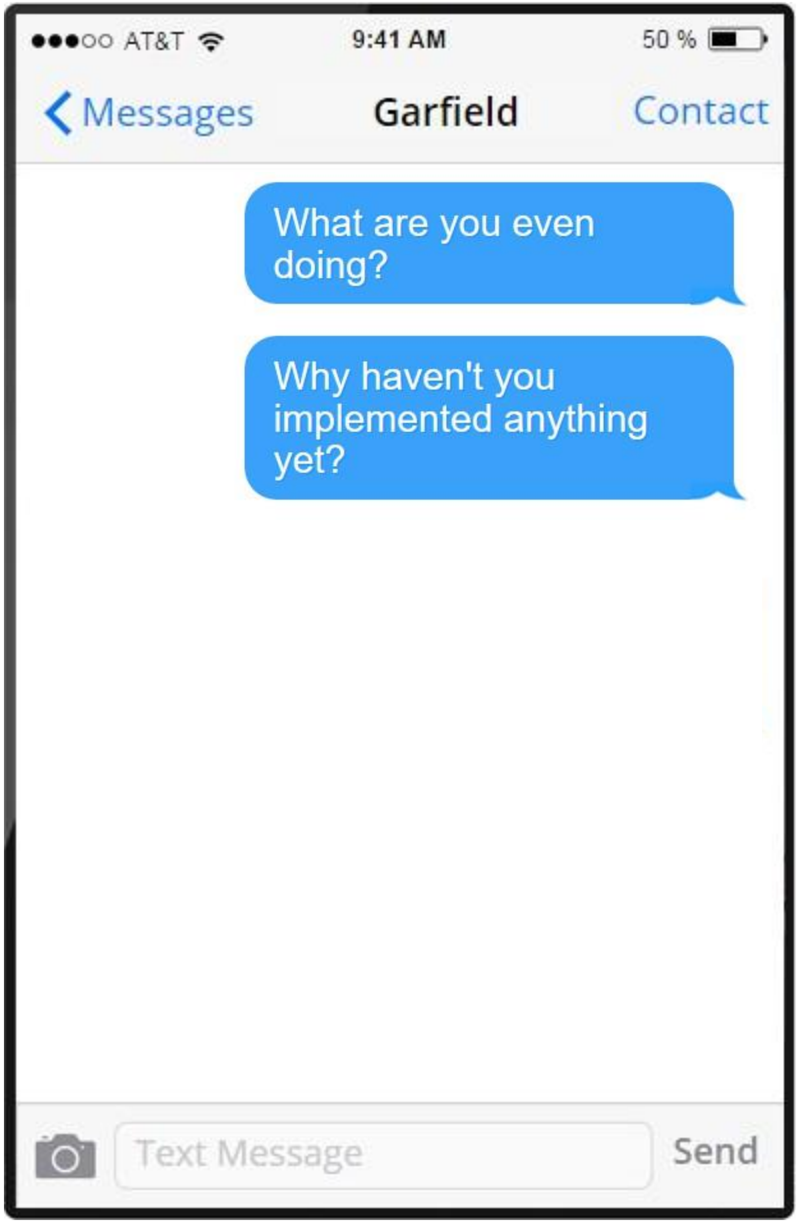
```
[<Test>]
let ``When I add 1 + 41, I expect 42``()=
  let result = add(1,41)
  Assert.AreEqual(42,result)
```

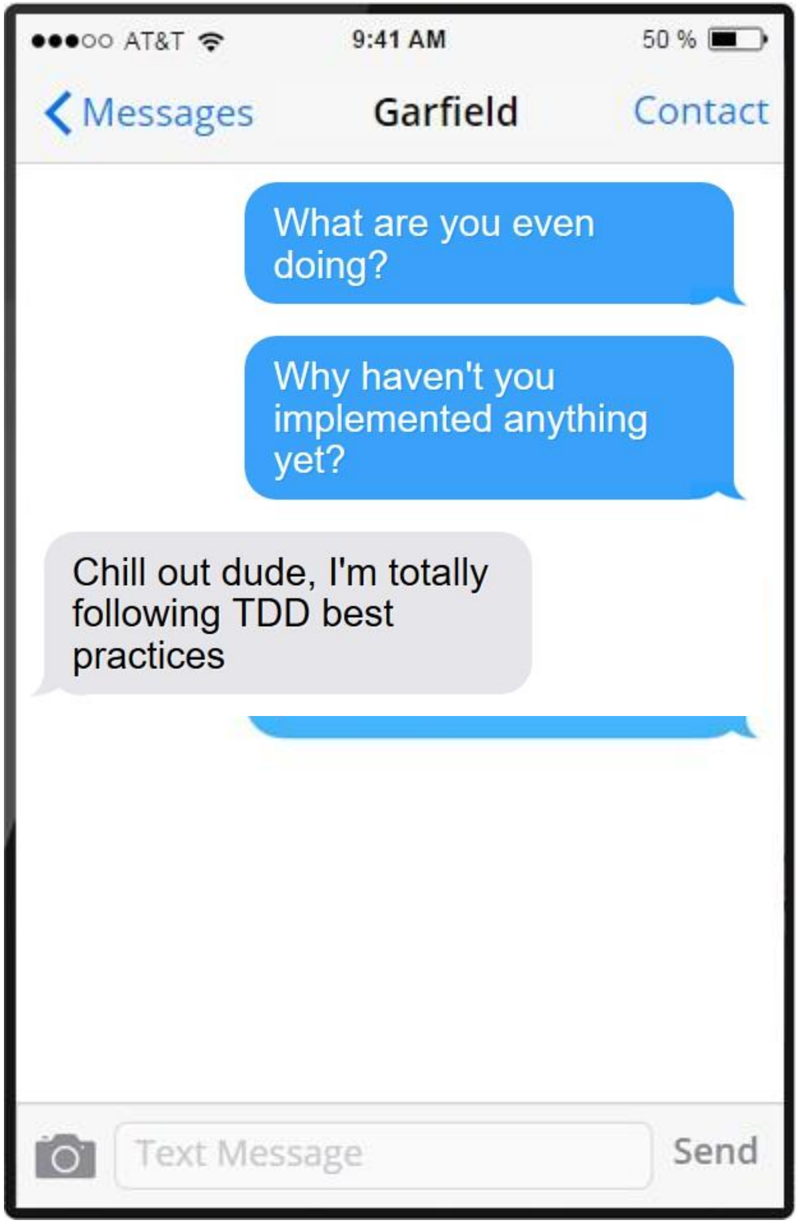
Okay, the tests pass.
That looks good.

But let's just check the implementation again...

```
let add(x,y) =
   match (x,y) with
   | (2,3) -> 5
   | (1,41) -> 42
   | (_,_) -> 4    // all other cases
```

wtf wtf wtf


●●●○○ AT&T
9:41 AM
50 %
Messages
Garfield
Contact
What are you even doing?
Why haven't you implemented anything yet?
Text Message
Send


●●●○○ AT&T
9:41 AM
50 %
Messages
Garfield
Contact
What are you even doing?
Why haven't you implemented anything yet?
Chill out dude, I'm totally following TDD best practices
Text Message
Send

TDD best practices

Write only enough code to make the failing unit test pass.

http://www.javiersaldana.com/articles/tech/refactoring-the-three-laws-of-tdd

Another attempt at a test

```
[<Test>]
let ``When I add two numbers,
      I expect to get their sum``()=

   let testData = [
      (1,2,3)
      (2,2,4)
      (3,5,8)
      (27,15,42)
      ]
   for (x,y,expected) in testData do
      let actual = add(x,y)
      Assert.AreEqual(expected,actual)
```

Let's check the implementation one more time....

```
let add(x,y) =
   match (x,y) with
   | (1,2) -> 3
   | (2,3) -> 5
   | (3,5) -> 8
   | (1,41) -> 42
   | (27,15) -> 42
   | (_,_) -> 4     // all other cases
```

It dawned on me who I was dealing with...

...the legendary burned-out, always lazy and often malicious programmer called...

The Enterprise Developer From Hell

(EDFH)

Rethinking the approach

The EDFH will always make specific examples pass, no matter what I do...

So let's not use specific examples!

Let's use random numbers instead…

```
[<Test>]
let ``When I add two random numbers,
       I expect their sum to be correct``()=

       let x = randInt()
       let y = randInt()
       let actual = add(x,y)
       let expected = …   // hmmm. need add(x,y)
       Assert.AreEqual(expected, actual)
```

Let's use random numbers instead…

```
[<Test>]
let ``When I add two random numbers,
       I expect their sum to be correct``()=

       let x = randInt()
       let y = randInt()
       let actual = add(x,y)
       let expected = …   // hmmm. need add(x,y)
       Assert.AreEqual(expected, actual)
```

If I reimplement add(x,y):

- I might make the same mistakes as I made in the "real" implementation.

- It is a lot of work if "add" is complicated

Questions for everyone:

How would you write a test for an "add" function?

But without re-implementing "add"

And without using specific examples

Proofs about Programs

a program

```
let reverse (xs: int list) : int list =
  let rec aux input output =
    match input with
      [] -> output
    | hd::tl -> aux tl (hd::output)
  in
  aux xs []
```

a theorem (or *specification*)

for all xs : list. reverse (reverse xs) = xs

Can we use a spec as a test? Yes!

Randomly generate universally quantified objects (xs)
and **check the property** (ie: reverse (reverse xs) = xs)

From Equations to Properties

Our specs have been "equational:"

(1) `for all x : t. e1 = e2` `(x probably appears in e1 and e2)`

A special case:

(2) `for all x : t. prop(e) = true`

Here "prop" is a function s -> bool
- a "property" or a "predicate" about values with type s.

We often write (2) like this:

(3) `for all x : t. prop(e)`

From Properties to Tests

`Given a property about a program like this:`

```
for all x : t. prop(e)
```

We can *prove it*

- pro: high reliability: know it is true for all x
- con: costly: takes time and expertise

skills at testing increase the reliability component

We can *test it*

- con: weaker reliability: know it is true for some x
- pro: cheap: takes less time and less expertise

Conjecture: Understanding and being good at specs and proofs makes you a better programmer. It also makes you a better tester.

Part II:
Property based testing

What are the "requirements" for the "add" function?

Requirements for the "add" function?

- It's often hard to know where to get started
- Tests *distinguish* good implementations from bad ones: So think of a bad one
- More generally: compare your implementation with an implementation of something different...
  - E.g. How does "add" differ from "subtract"?

Requirements for the "add" function?

- Addition vs. subtraction:
  - For subtraction, the order of the parameters makes a difference
  - For addition it doesn't.

For subtraction, the order of the parameters makes a difference, while for addition it doesn't.

```
[<Test>]
let ``When I add two numbers, the result
      should not depend on parameter order``()=

   for _ in [1..100] do
     let x = randInt()
     let y = randInt()
     let result1 = add(x,y)
     let result2 = add(y,x)
     Assert.AreEqual(result1,result2)
```

reversed params

It doesn't depend on addition, and it eliminates a whole class of incorrect implementations!

The EDFH responds with this implementation:

```
let add(x,y) =
   x * y
```

How about using the difference between addition and multiplication?

For example:

- adding one twice is the same as adding two
- multiplying by one twice is NOT the same as multiplying by two

Test: two "add 1"s is the same as one "add 2".

```
[<Test>]
let ``Adding 1 twice is the same as adding 2``()=

   for _ in [1..100] do
     let x = randInt()
     let result1 = add(add(x,1),1)
     let result2 = add(x,2)
     Assert.AreEqual(result1,result2)
```

The EDFH responds with:

```
let add(x,y) =
   x - y
```

TEST: ``Adding 1 twice is the same as adding 2`` ✓

But luckily we have the previous test as well!

TEST: ``When I add two numbers, the result should not depend on parameter order`` ✗

Ha! Gotcha, EDFH!

The EDFH responds with another implementation:

```
let add(x,y) =
   0
```

TEST: ``Adding 1 twice is the same as adding 2``

TEST: ``When I add two numbers, the result
should not depend on parameter order``

Aarrghh! Where did our approach go wrong?

Requirements for the "add" function

- We need to check that the result is somehow connected to the input!
- Is there a trivial property of "add" that we know the answer to?
  - (without reimplementing our own version)
- Yes! Adding *zero* is the same as doing nothing

We have to check that the result is somehow connected to the input.

Adding zero is the same as doing nothing

```
[<Test>]
let ``Adding zero is the same as doing nothing``()=

   for _ in [1..100] do
     let x = randInt()
     let result1 = add(x,0)
     let result2 = x
     Assert.AreEqual(result1,result2)
```

Finally, the EDFH is defeated...

TEST: ``Adding 1 twice is the same as adding 2``

TEST: ``When I add two numbers, the result should not depend on parameter order``

TEST: ``Adding zero is the same as doing nothing``

If these are all true we MUST have a correct implementation*

* not quite true

Refactoring

Let's extract the shared code...

Pass in a "property"

```
let propertyCheck propertyFn =
   // property has type: (int,int) -> bool

   for _ in [1..100] do
     let x = randInt()
     let y = randInt()
     let result = propertyFn(x,y)
     Assert.IsTrue(result)
```

Check the property is true for random inputs

And the tests now look like:

```
let commutativeProperty(x,y) =
   let result1 = add(x,y)
   let result2 = add(y,x)
   result1 = result2
```

```
[<Test>]
let ``When I add two numbers, the result
      should not depend on parameter order``()=

   propertyCheck commutativeProperty
```

And the second property

```
let adding1TwiceIsAdding2OnceProperty(x,_) =
    let result1 = add(add(x,1),1)
    let result2 = add(x,2)
    result1 = result2
```

```
[<Test>]
let ``Adding 1 twice is the same as adding 2``()=

    propertyCheck adding1TwiceIsAdding2OnceProperty
```

This is really just a crude version of associativity!

And the third property

```
let identityProperty(x,_) =
    let result1 = add(x,0)
    result1 = x
```

```
[<Test>]
let ``Adding zero is the same as doing nothing``()=

    propertyCheck identityProperty
```

Review

Testing with properties

- The parameter order doesn't matter
- Doing "add 1" twice is the same as doing "add 2" once
- Adding zero does nothing

These properties apply to ALL inputs

We can generate arbitrarily many random examples, test the properties and see if they hold.

~~Testing~~ Specification with properties

- "Commutativity" property
- "Associativity" property
- "Identity" property

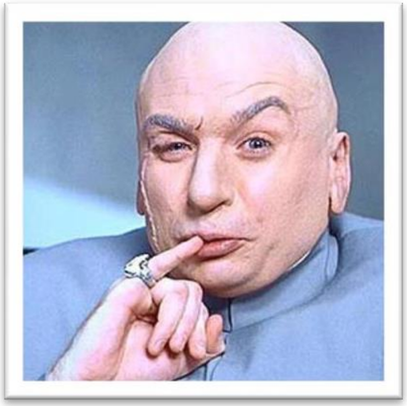
These properties define addition!

The EDFH can't create an incorrect implementation!

Bonus: By using specifications, we have understood the requirements in a deeper way.

Why bother with the EDFH?

Surely such a malicious programmer is unrealistic and over-the-top?



Evil

Stupid

Lazy

In practice,
no difference!

In my career, I've always had to deal with one stupid person in particular ☹

Me!

The real EDFH!

When I look at my old code, I almost always see something wrong!

I've often created flawed implementations, not out of evil intent, but out of unawareness and blindness

Part III:
QuickCheck and its ilk

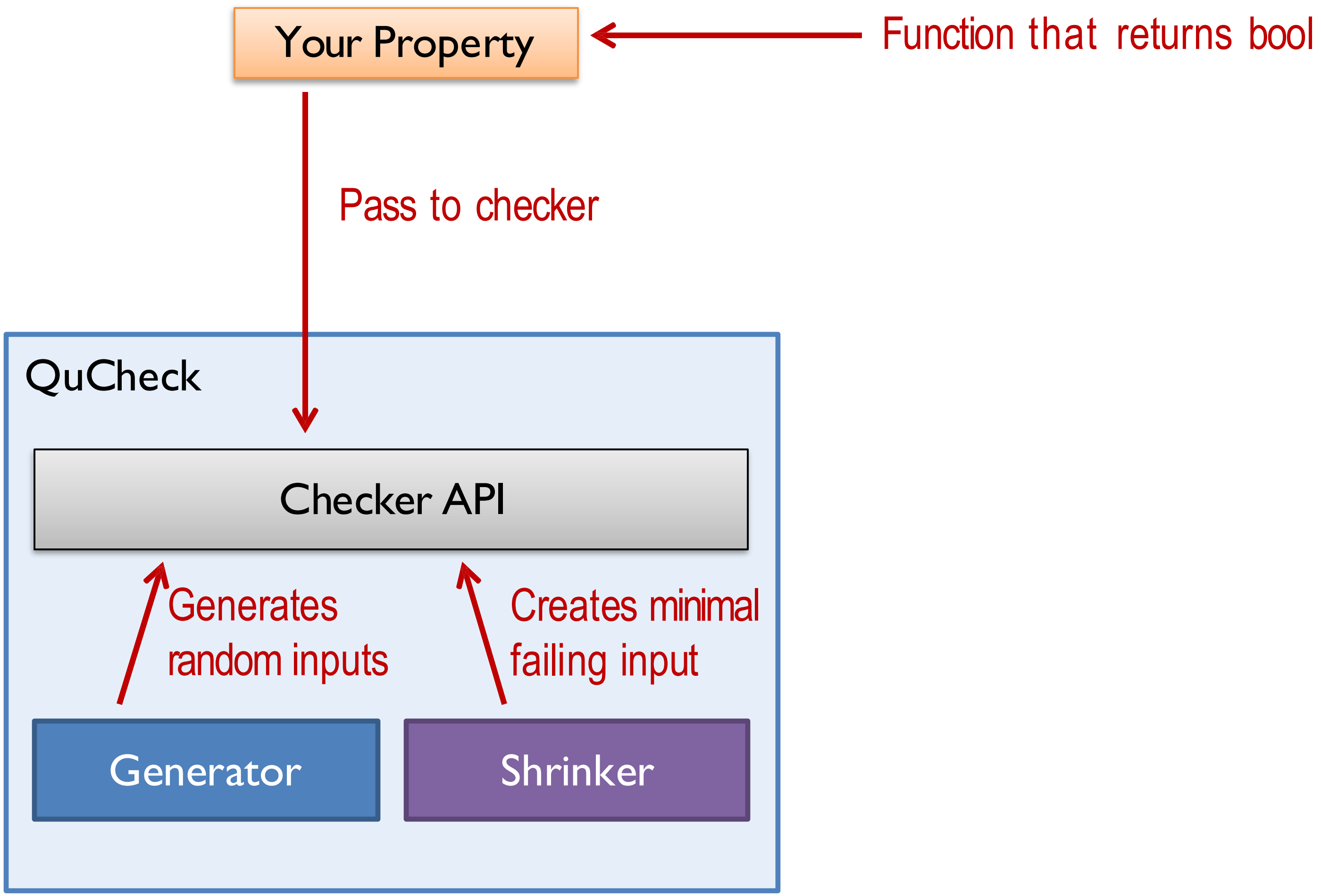
Wouldn't it be nice to have a toolkit for doing this?

The "QuickCheck" library was originally developed for Haskell by Koen Claessen and John Hughes, and has been ported to many other languages.

In OCaml: QCheck

opam install qcheck

See: https://o1-labs.github.io/ocamlbyexample/build-qcheck.html



Your Property
Function that returns bool
Pass to checker
QuCheck
Checker API
Generates random inputs
Creates minimal failing input
Generator
Shrinker

Using QCheck

dune

```
(executable
  (public_name qc)
  (name main)
  (libraries qcheck))
```

main.ml

```
let ....
```

Using QCheck

Write some properties: t -> bool

```
type iprop = int * int -> bool

let commutes f (x,y) = f x y = f y x
let add_commutes p = commutes (+) p
let sub_commutes p = commutes (-) p
```

Create a generator

```
module Q = Qcheck

type 'a gen = 'a Q.arbitrary

let ints : int gen = Q.int
let pairs : (int * int) gen = Q.tup2 ints ints
```

Using QCheck

Write some properties: t -> bool

```
type iprop = int * int -> bool

let commutes f (x,y) =
  f x y = f y x
let add_commutes p =
  commutes (+) p
let sub_commutes p =
  commutes (-) p
```

Create a generator

```
module Q = Qcheck

type 'a gen = 'a Q.arbitrary

let ints = Q.int
let pairs = Q.tup2 ints ints
```

Create a test

```
let t1 = Q.Test.make
            ~name:"add commutes"
            ~count:100
            pairs
            add_commutes
```

~n – OCaml optional argument

name of test

number of random tests

generator

property

Using QCheck

Write some properties: t -> bool

```
type iprop = int * int -> bool

let commutes f (x,y) =
  f x y = f y x
let add_commutes p =
  commutes (+) p
let sub_commutes p =
  commutes (-) p
```

Create a generator

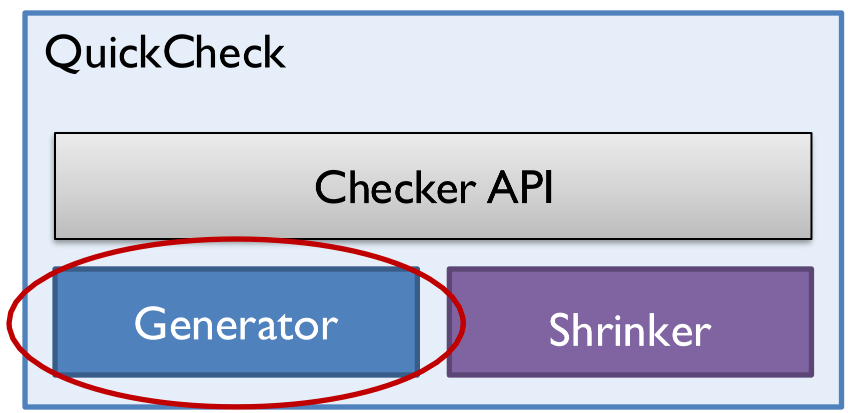
```
module Q = Qcheck

type 'a gen = 'a Q.arbitrary

let ints = Q.int
let pairs = Q.tup2 ints ints
```

Create a test

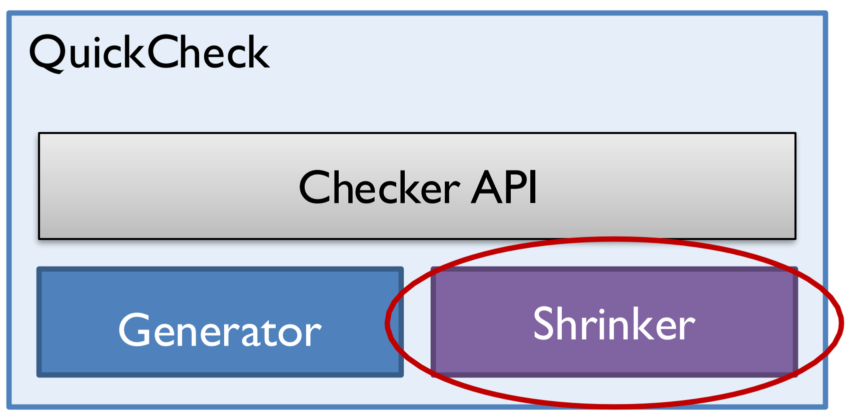
```
let t1 =
  Q.Test.make
       ~name:"add commutes"
       ~count:100
       pairs
       add_commutes
```

```
let _ =
  QCheck_base_runner.run_tests
    ~verbose:true
    [t1;t2]
```

Generators:
making random inputs



Shrinking: dealing with failure



How shrinking works

Property to test – we know it's gonna fail!

```
let smallerThan81Property x =
   x < 81
```

"int" generator

```
0, 1, 3, -2, 34, -65, 100
```

Fails at 100!

So 100 fails, but knowing that is not very helpful

Time to start shrinking!

How shrinking works

Given a value, a shrinker produces a sequence of values that are (in some way) smaller than the given value

```
let smallerThan81Property x =
   x < 81
```

Shrink list for 100

`0, 50, 75, 88, 94, 97, 99`

Fails at 88!

Generate a new sequence up to 100

Shrink again starting at 88

How shrinking works

Given a value, a shrinker produces a sequence of values that are (in some way) smaller than the given value

```
let smallerThan81Property x =
   x < 81
```

Shrink list for 88

0, 44, 66, 77, 83, 86, 87

Fails at 83!

Generate a new sequence up to 88

Shrink again starting at 83

How shrinking works

Given a value, a shrinker produces a sequence of values that are (in some way) smaller than the given value

```
let smallerThan81Property x =
   x < 81
```

Shrink list for 83

`0, 42, 63, 73, 78, 81, 82`

Fails at 81!

Generate a new sequence up to 83

Shrink again starting at 81

How shrinking works

Given a value, a shrinker produces a sequence of values that are (in some way) smaller than the given value

```
let smallerThan81Property x =
   x < 81
```

Shrink list for 81

```
0, 41, 61, 71, 76, 79, 80
```

All pass!

Generate a new sequence up to 81

Shrink has determined that 81 is the smallest failing input!

Shrinking – final result

Shrinking is built into the arbitrary:

```
let ints : int gen = Q.int

// result: Falsifiable, after 23 tests (3 shrinks)
// 81
```

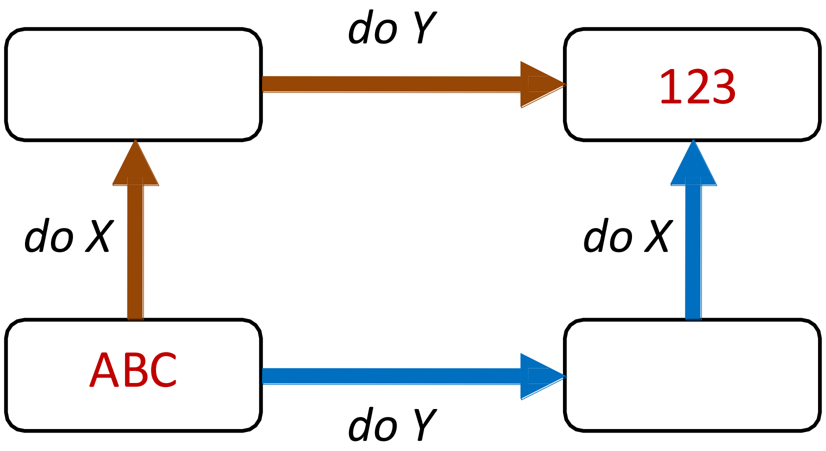
Shrinking is really helpful to show the boundaries where errors happen

Shrinking works with compound types too!

Part IV:
How to choose properties

What properties should I use? I can't think of any!

"Different paths, same destination"

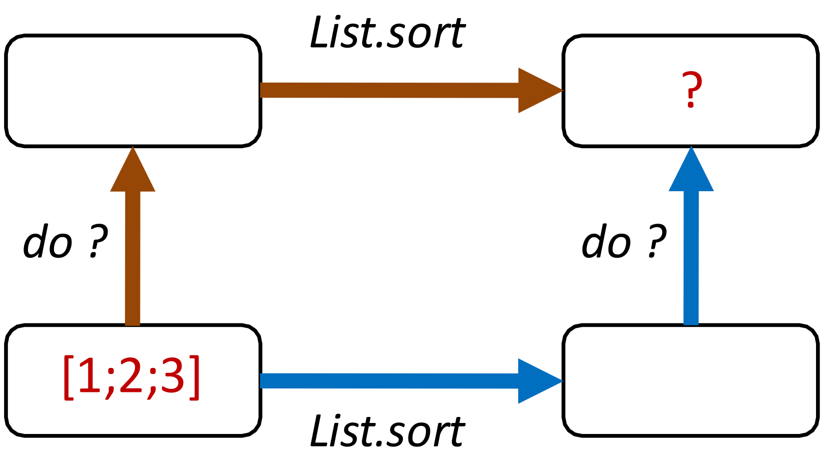


Examples:

- Commutivity
- Associativity
- Map f then Map g

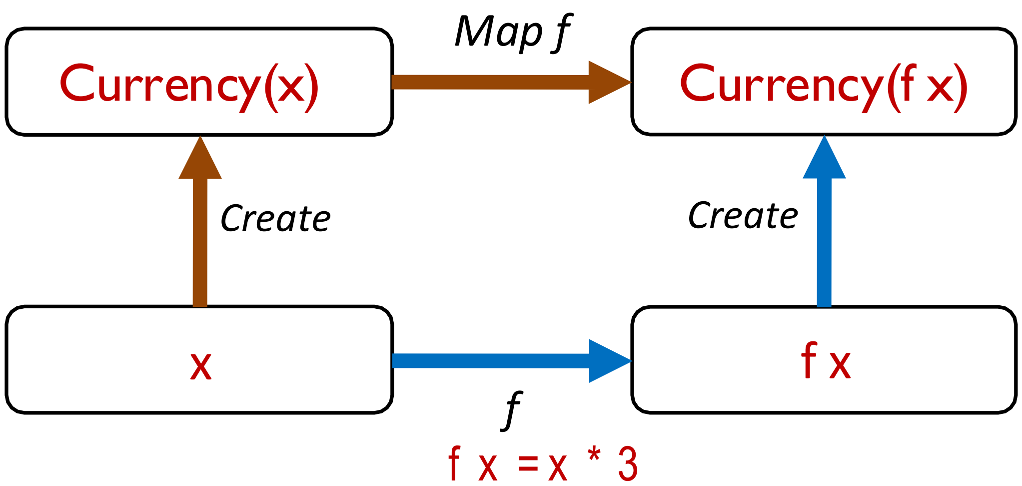
"Different paths, same destination"

Applied to a sort function



"Different paths, same destination"

Applied to a map function

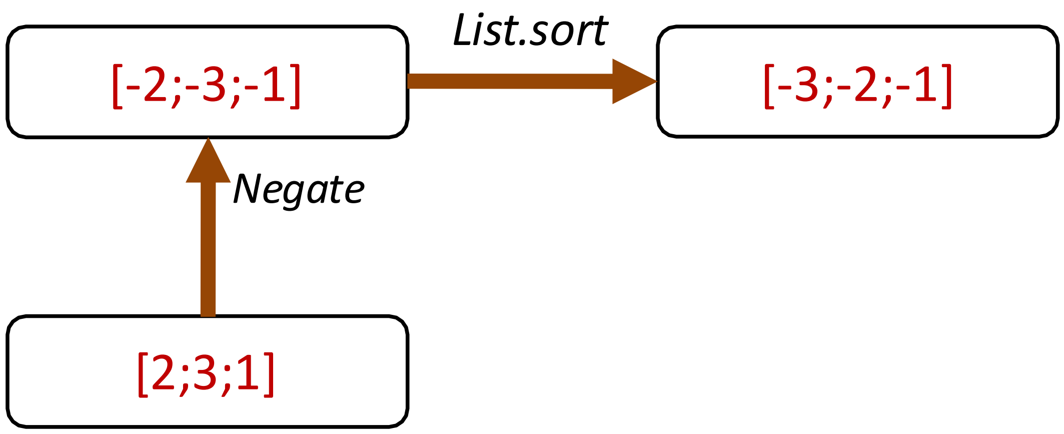



```
Currency(2)
   .Map(x => x * 3)
```

```
Currency(2 * 3)
```

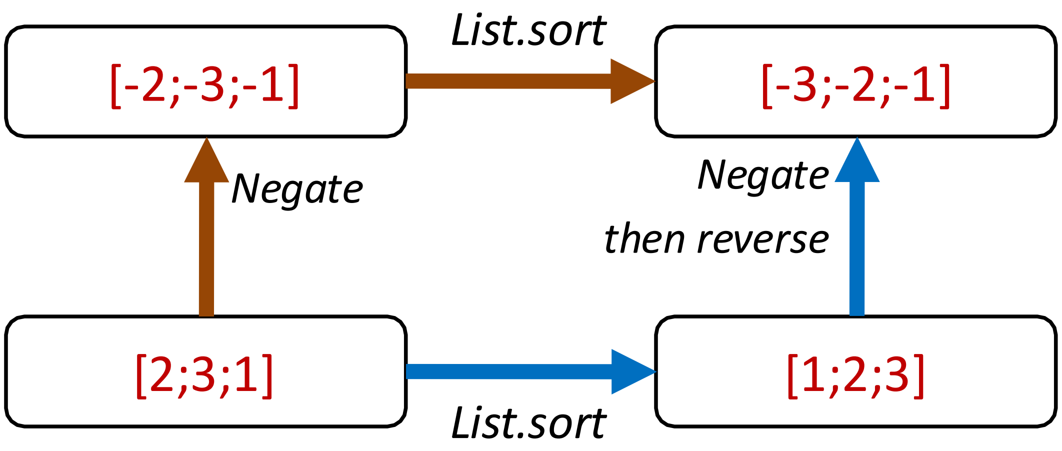
"Different paths, same destination"

Applied to a sort function

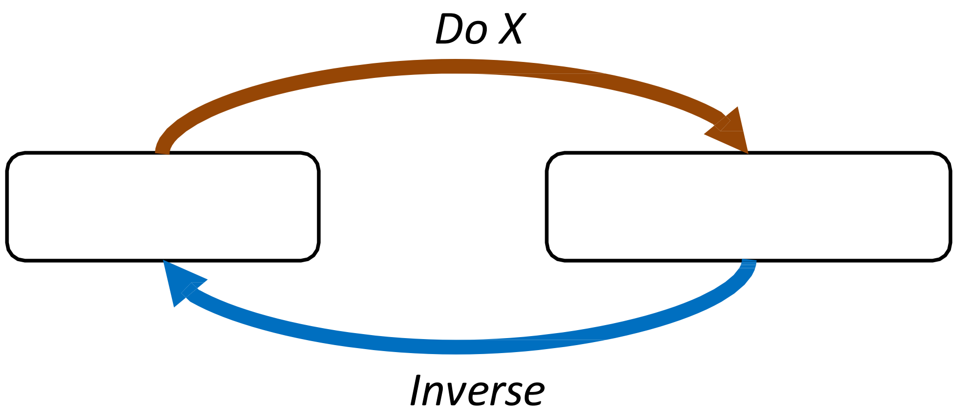


"Different paths, same destination"

Applied to a sort function



"There and back again"

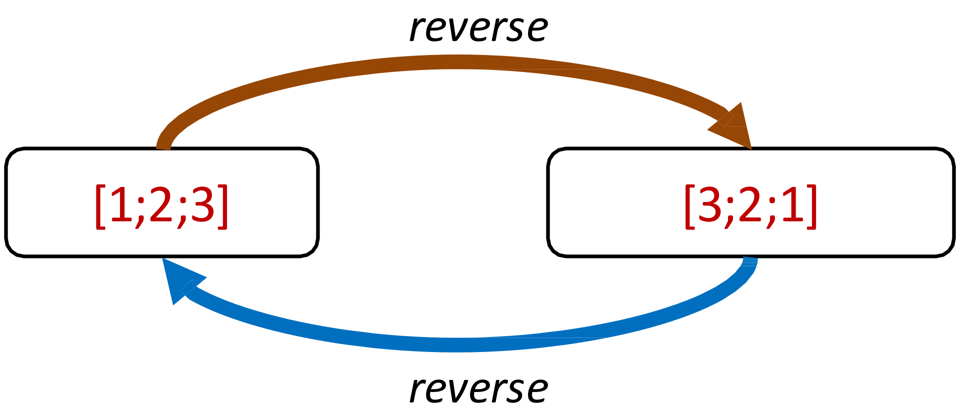


Examples:

- Serialization/Deserialization
- Addition/Subtraction
- Write/Read
- SetProperty/GetProperty

"There and back again"

Applied to a list reverse function



Pro tip:
We often need a combination of properties, not just one

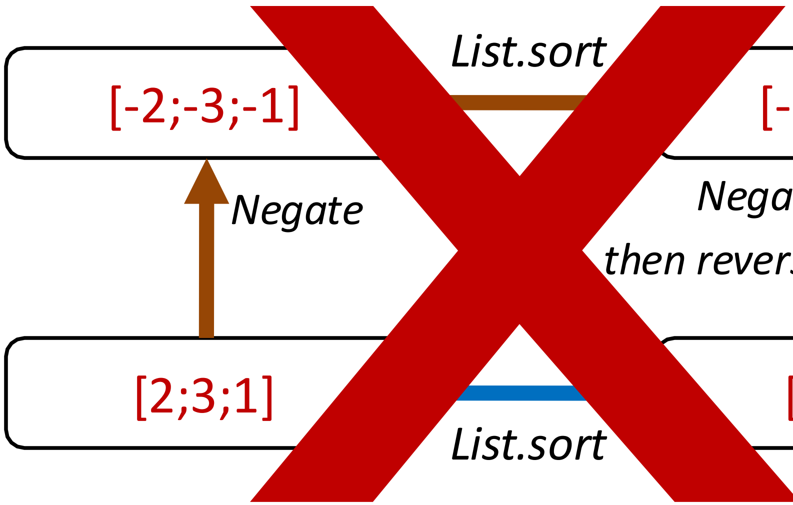
We needed three properties to define "add"

"Some things never change"

Examples:

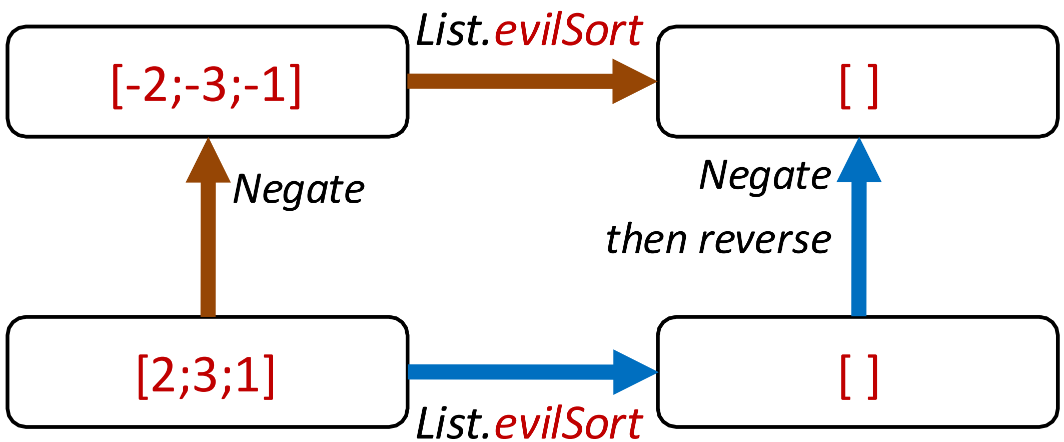
- Size of a collection
- Contents of a collection
- Balanced trees

The EDFH and List.Sort



The EDFH can beat this!

The EDFH and List.Sort



EvilSort just returns an empty list!

This passes the "commutivity" test!

"Some things never change"

Used to ensure the sort function is good

[2;3;1] → [1;2;3]

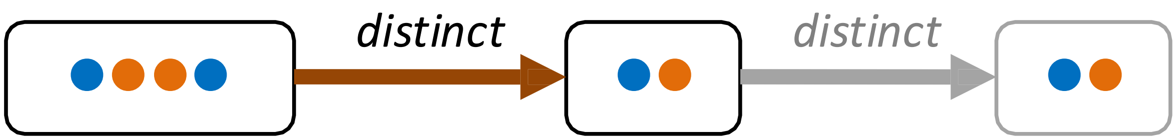
List.sort

Must be one of these permutations

[1; 2; 3]; [2; 1; 3]; [2; 3; 1];
[1; 3; 2]; [3; 1; 2]; [3; 2; 1]

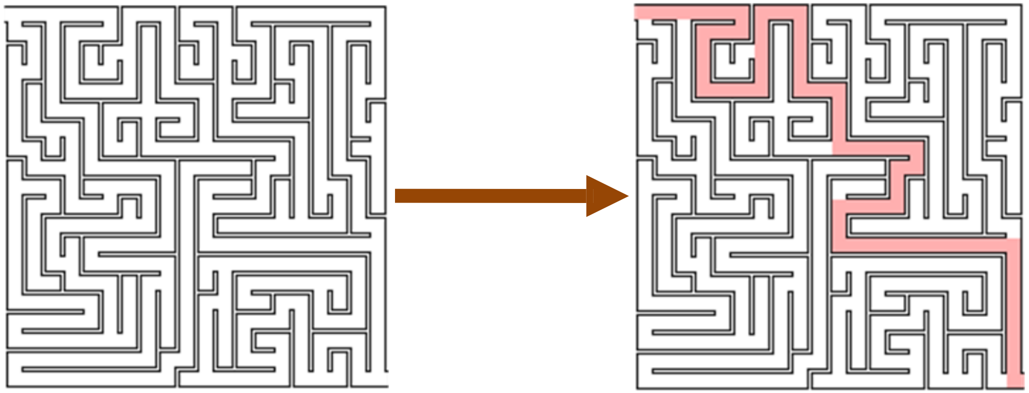
The EDFH is beaten now!

"The more things change, the more they stay the same"



Idempotence:
- Sort
- Filter
- Event processing
- Required for distributed designs

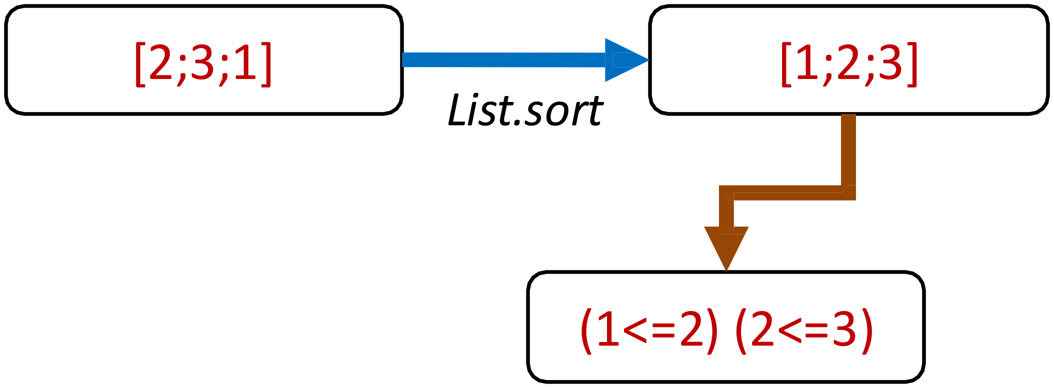
"Hard to prove, easy to verify"



- Prime number factorization
- Too many others to mention!

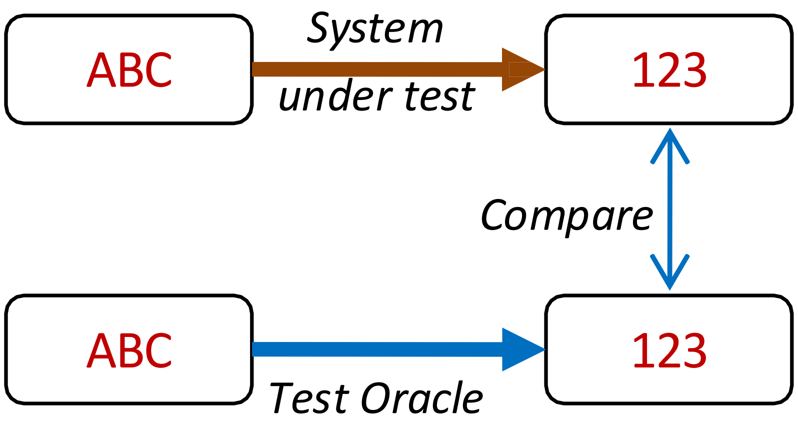
"Hard to prove, easy to verify"

Applied to a sort



To verify the sort,
check that each pair is ordered

"The test oracle"



- Compare optimized with slow brute-force version
- Compare parallel with single thread version
- Legacy system is the oracle for a replacement system

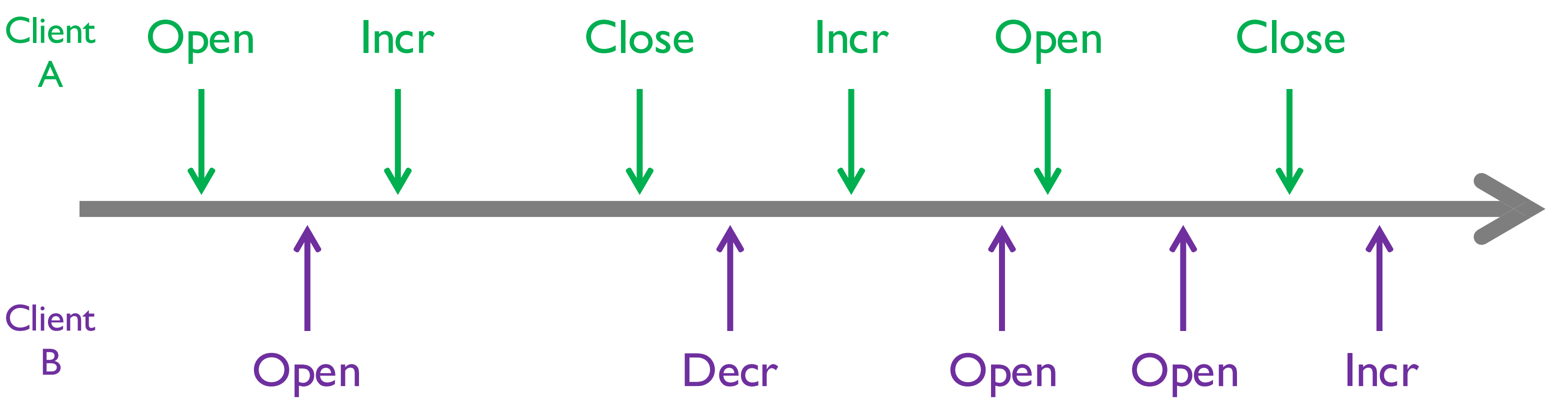
Part V:
Model based testing

Using the test oracle approach
for complex implementations

Testing a simple database

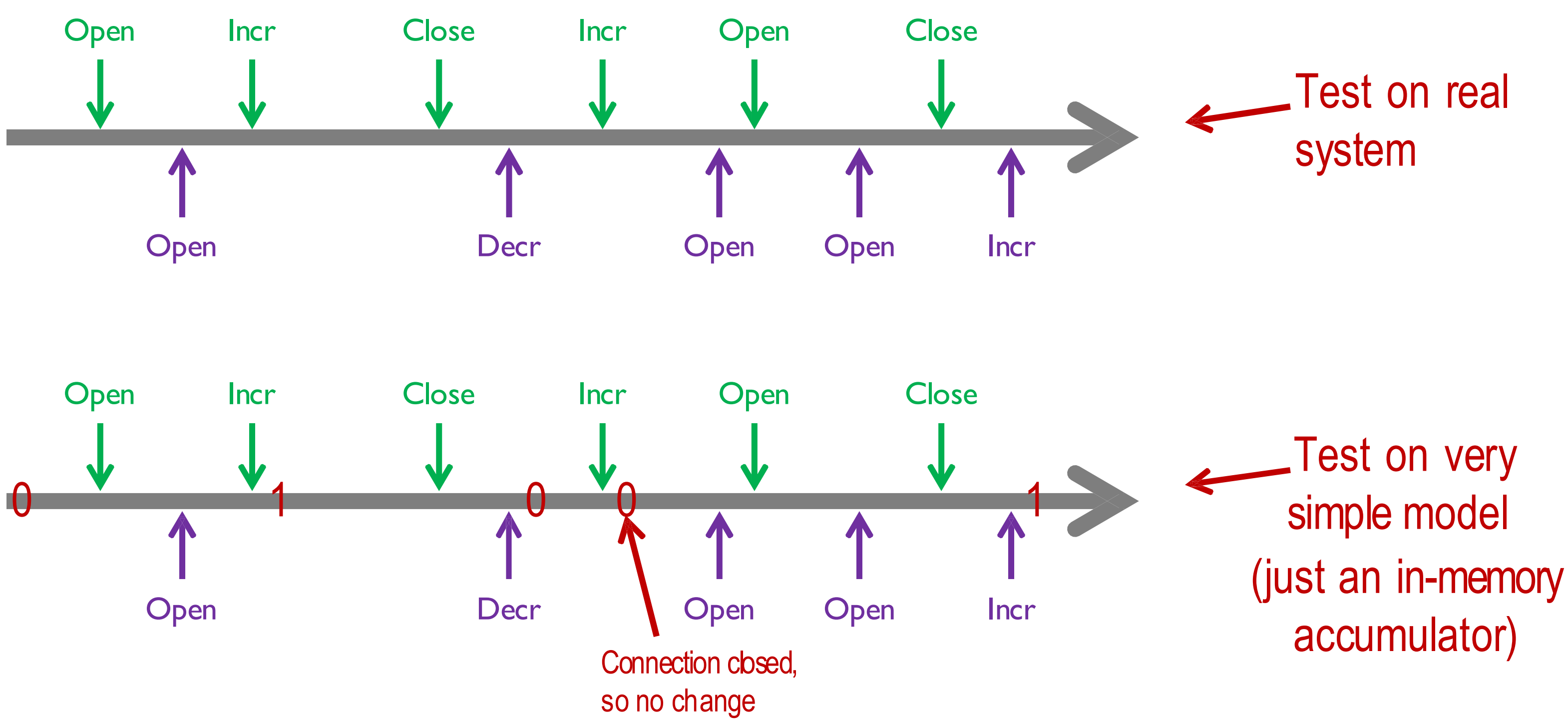
Four operations: Open, Close, Increment, Decrement
Two clients: Client A and Client B

Let QuickCheck generate a random list of these actions for each client



How do use this to check that our db works?

Testing a simple database



Compare model result with real system!

Example-based tests vs. Property-based tests

Example-based tests vs. Property-based tests

- PBTs are more general
  - One property-based test can replace many example-based tests.
- PBTs can reveal overlooked edge cases
  - Nulls, negative numbers, weird strings, etc.
- PBTs ensure deep understanding of requirements
  - Property-based tests force you to think! ☹
- PBTs can do shrinking to find the boundary cases!
- Example-based tests are still helpful though!
  - Less abstract, easier to understand

Summary

Be lazy! Don't write tests, generate them!

Use property-based thinking to gain deeper insight into the requirements

PBT Resources

Search for "property-based testing" and your language!

- Java: jqwik.net
- Python: https://hypothesis.readthedocs.io/en/latest/

Look for talks by John Hughes

- How to specify it! https://www.youtube.com/watch?v=zvRAyq5wj38
- Don't write tests! https://www.youtube.com/watch?v=hXnS_Xjwk2Y

And others may be good

- "Property-BasedTesting in a Screencast Editor" by OskarWickström
- "MetamorphicTesting" by HillelWayne