





A Type is Worth a Thousand Tests

Manuel M T Chakravarty
*Applicative &
Tweag I/O*

 [mchakravarty](#)
 [TacticalGrace](#)
 [justtesting.org](#)
 [haskellformac.com](#)

Let's talk about...

Let's talk about...



Swift

Let's talk about...

Language
Design



Swift

Let's talk about...

Language
Design



Ecosystem

Swift

Let's talk about...

Why is it

Language

Ecosystem

Design
extraordinary

?

Swift

Unique Event

Java

C#

Objective-C

Java

Scala

Kotlin

C#

Objective-C

Java

Scala

Kotlin

C#

F#

Objective-C

Java

Scala

Kotlin

C#

F#

Objective-C

Swift

Why do languages become popular?

Why do languages become popular?

Platforms

Why do languages become popular?

JavaScript
Web

Objective-C
Cocoa

Python/Ruby
Cloud

C/C++
Unix

Java
JVM

C#
.NET

Platforms

Why do languages become popular?

JavaScript
Web

Swift
Cocoa

Python/Ruby
Cloud

C/C++
Unix

Java
JVM

C#
.NET

Platforms

Why will Swift be popular?

Why will Swift be popular?



Why will Swift be popular?

Challenge: Use the AND, OR, and NOT operators to navigate Byte through the world.

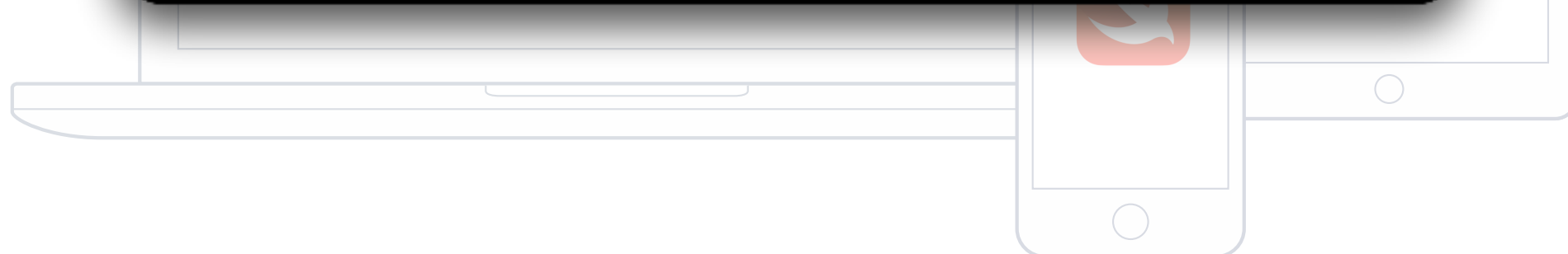
Each of these operators influences the way your conditional code runs:

- The **NOT operator (!)** inverts a **Boolean** value, saying, "if NOT this condition, do this".
- The **AND operator (&&)** combines two conditions and runs the code only if *both* are true.
- The **OR operator (||)** combines two conditions and runs the code if *at least one* is true.

Solve the challenge by choosing the operators that will work best so that Byte collects all the gems and toggles open the switches.

```
for i in 1...6 {
  moveForward()
  if isOnClosedSwitch && isBlocked {
    toggleSwitch()
    turnLeft()
    moveForward()
  }
}
```

Run My Code Hint

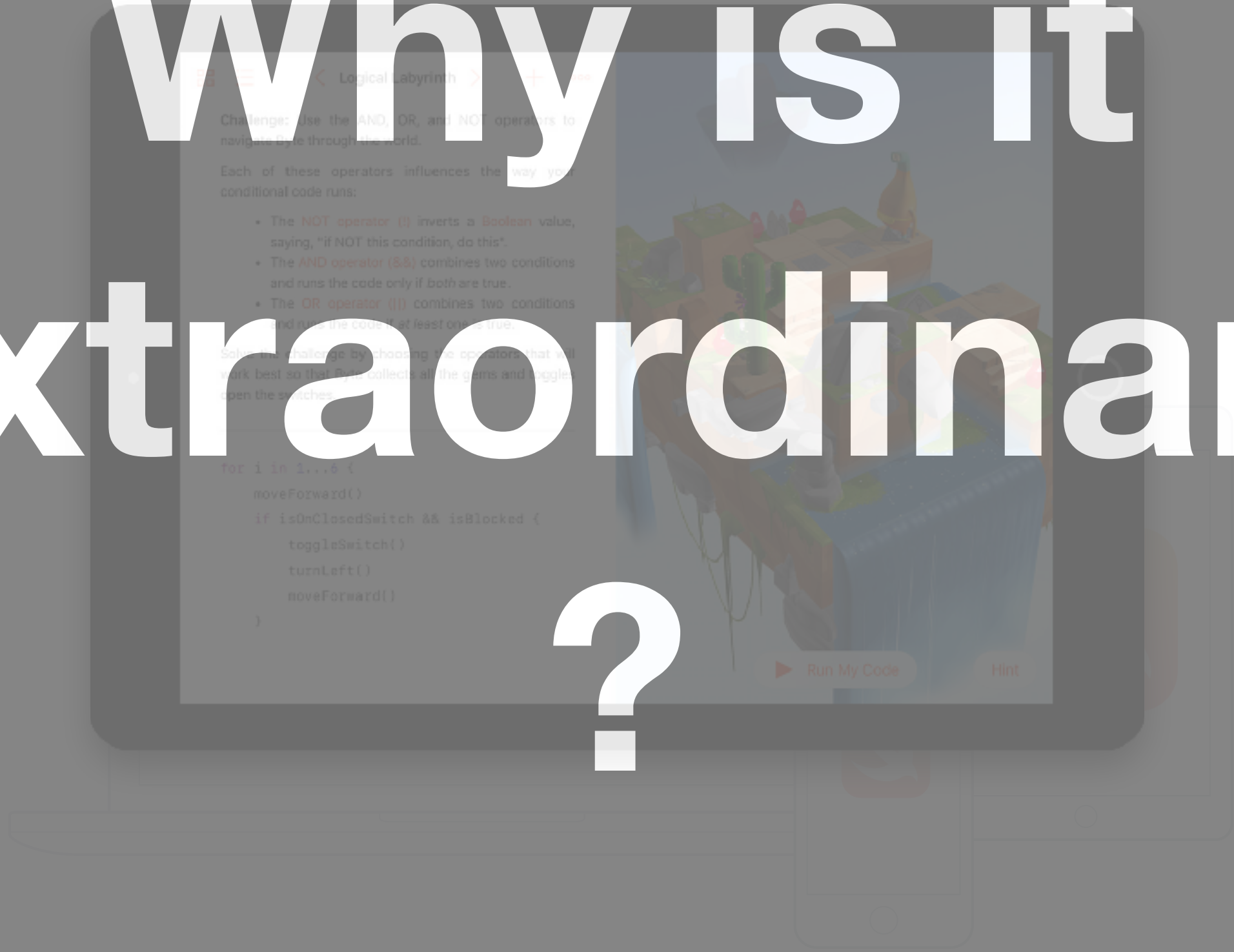


Why will Swift be popular?

Why is it

extraordinary

?



Functional Programming

From Objective-C to Swift

From Objective-C to Swift

Value types
(products & sums)

Advanced
type system

Explicit control of
mutability

Explicit
optionals

Generics &
protocols with
associated types

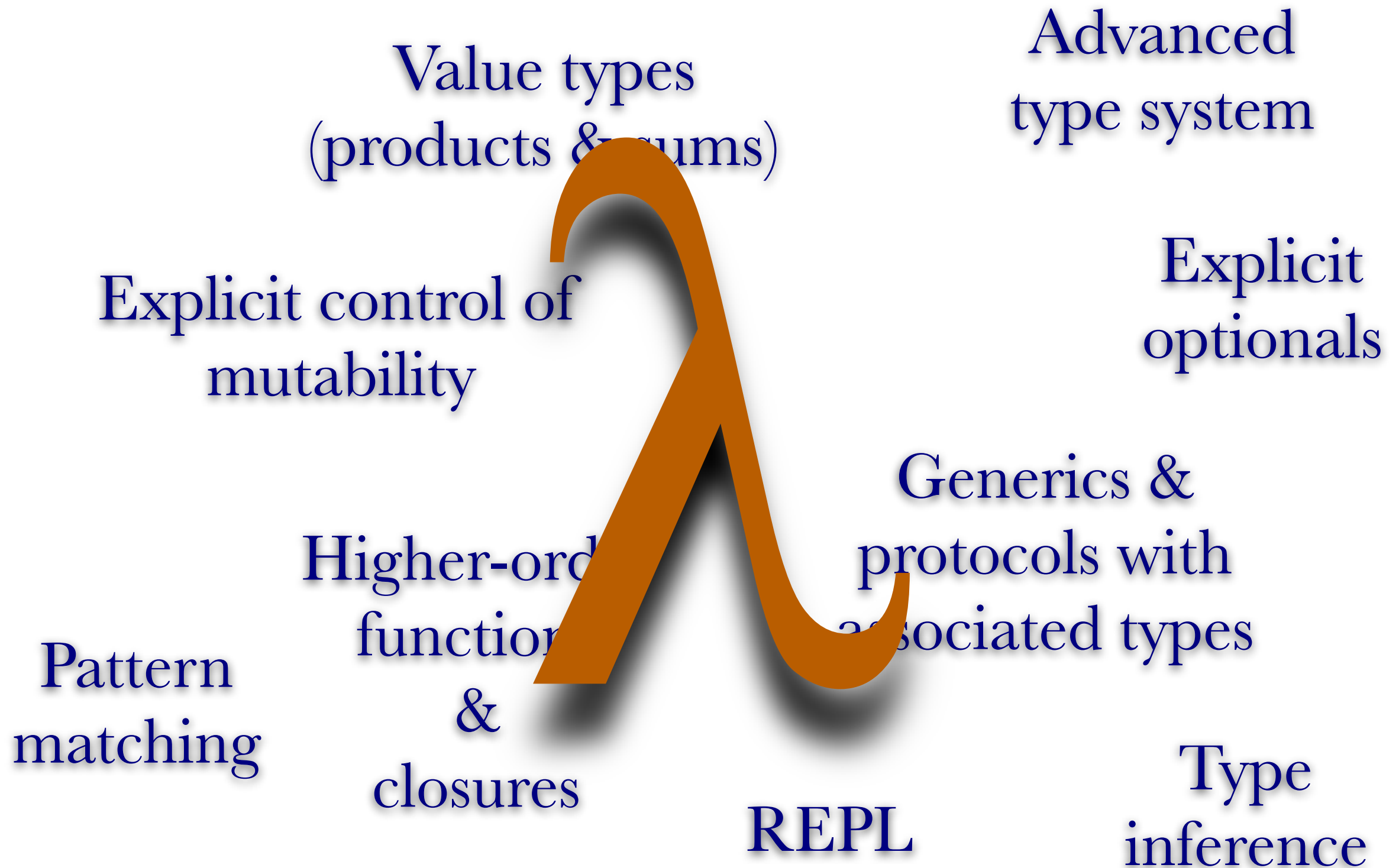
Pattern
matching

Higher-order
functions
&
closures

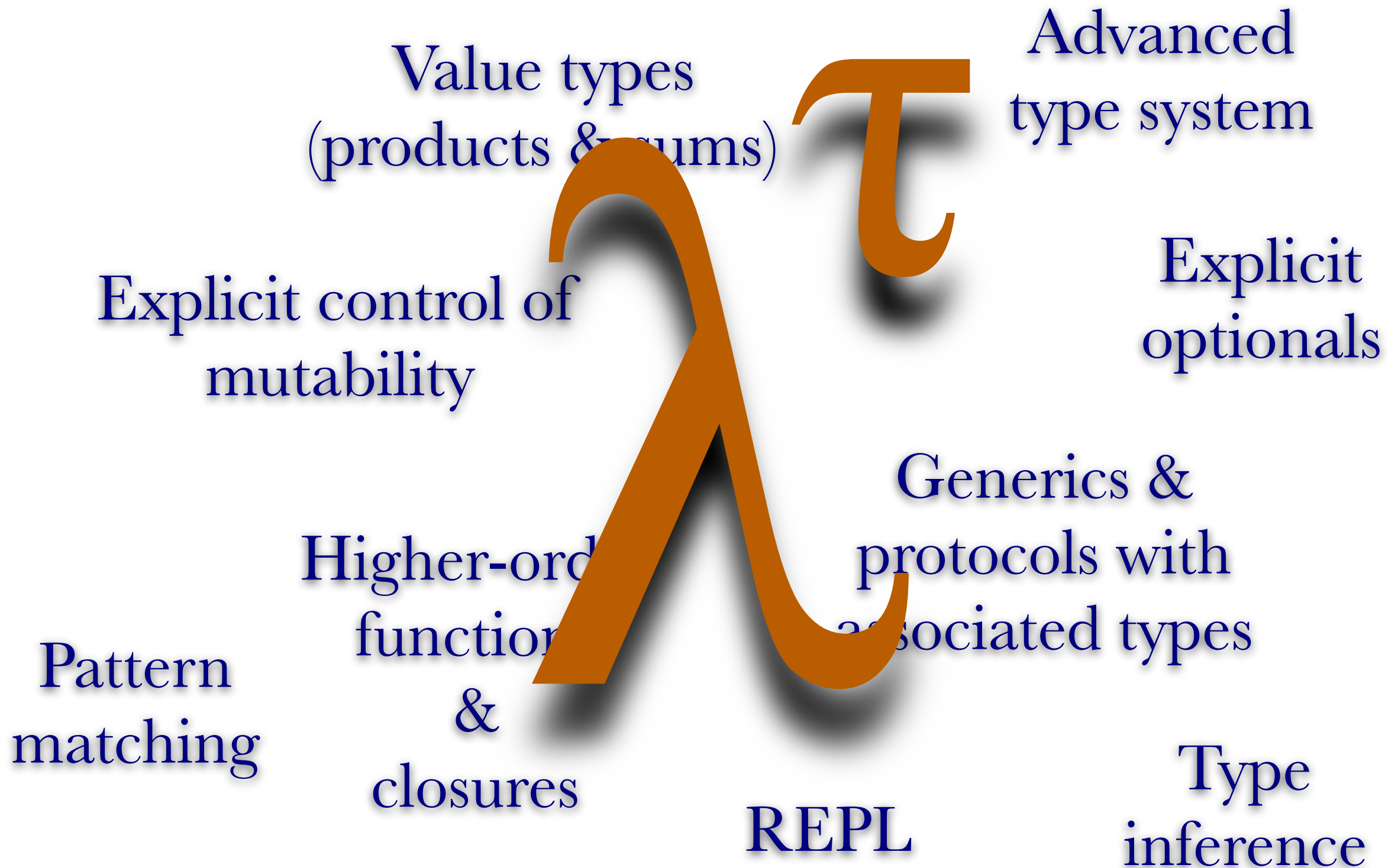
REPL

Type
inference

From Objective-C to Swift



From Objective-C to Swift



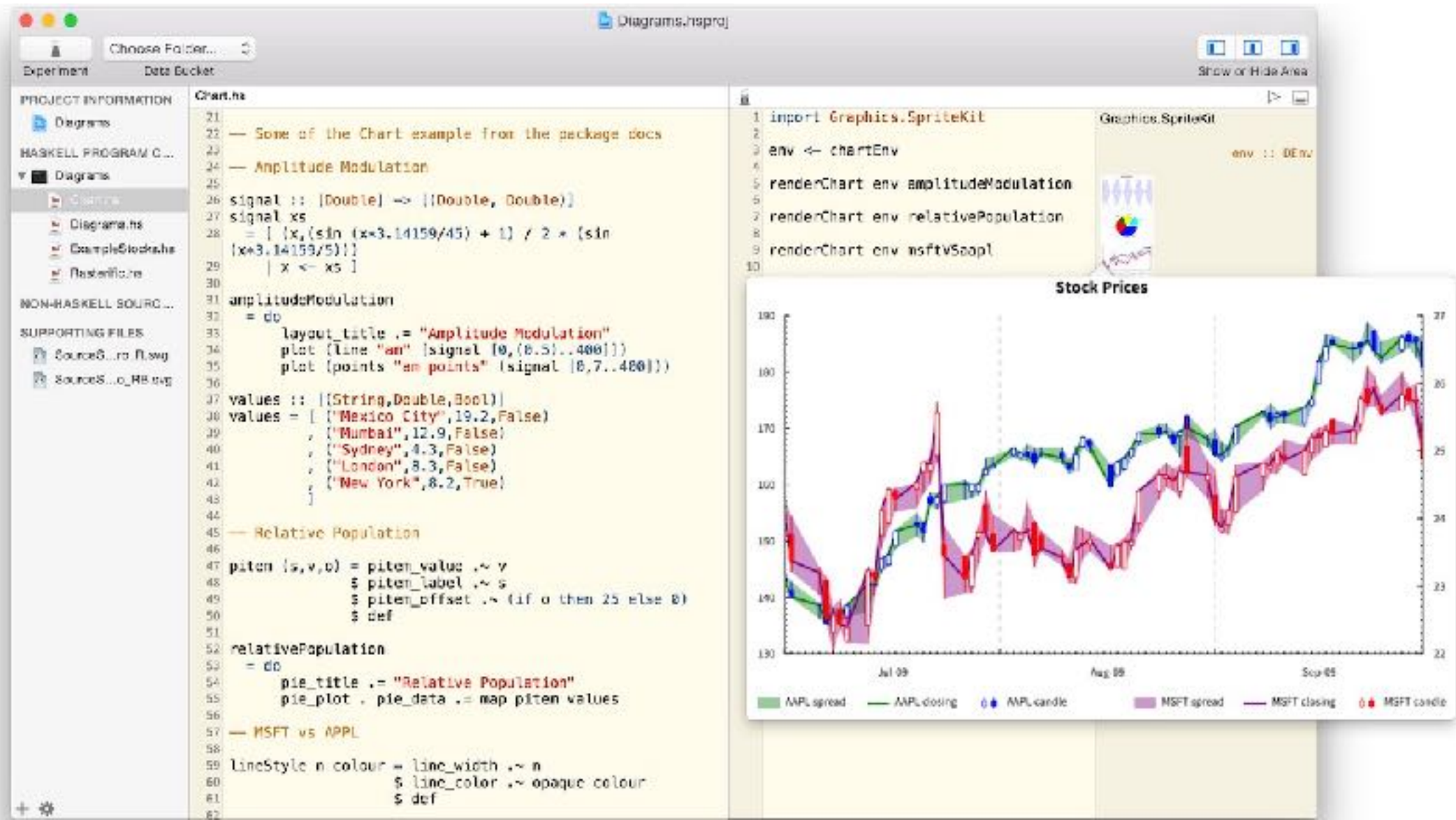
“Swift encourages typed
functional programming in a
mainstream language.”

Haskell for Mac

Adopting Swift right out of the gate

Haskell for Mac

Adopting Swift right out of the gate



The screenshot shows a Haskell IDE window titled "Diagrams.hsproj". The left pane displays the project structure, including "Diagrams.hs" and "ExampleStocks.hs". The main editor shows the source code for "Chart.hs", which defines functions for amplitude modulation, relative population, and MSFT vs APPL. The right pane shows the execution output, including the "Graphics.SpriteKit" environment and a "Stock Prices" chart. The chart displays stock prices for APPL and MSFT from July 2009 to September 2009, with a legend at the bottom identifying the data series.

```
21
22 — Some of the Chart example from the package docs
23
24 — Amplitude Modulation
25
26 signal :: [Double] -> [(Double, Double)]
27 signal xs
28 = [ (x, (sin (x*3.14159/45) + 1) / 2 * (sin
29     | x <- xs ]
30
31 amplitudeModulation
32 = do
33     layout_title .= "Amplitude Modulation"
34     plot [line "am" |signal [0,(0.5)..400]]
35     plot [points "am points" |signal [0,7..400]]
36
37 values :: [(String,Double,Bool)]
38 values = [ ("Mexico City",19.2,False)
39           , ("Mumbai",12.9,False)
40           , ("Sydney",4.3,False)
41           , ("London",8.3,False)
42           , ("New York",8.2,True)
43           ]
44
45 — Relative Population
46
47 piten (s,v,b) = piten_value .~ v
48               $ piten_label .~ s
49               $ piten_offset .~ (if 0 then 25 else 0)
50               $ def
51
52 relativePopulation
53 = do
54     pie_title .= "Relative Population"
55     pie_plot . pie_data .= map piten values
56
57 — MSFT vs APPL
58
59LineStyle n colour = line_width .~ n
60                  $ line_color .~ opaque colour
61                  $ def
62
```

```
1 import Graphics.SpriteKit
2
3 env ← chartEnv
4
5 renderChart env amplitudeModulation
6
7 renderChart env relativePopulation
8
9 renderChart env msftVsAppl
10
```

Graphics.SpriteKit
env :: DEnv

Stock Prices

APPL spread APPL closing APPL candle MSFT spread MSFT closing MSFT candle

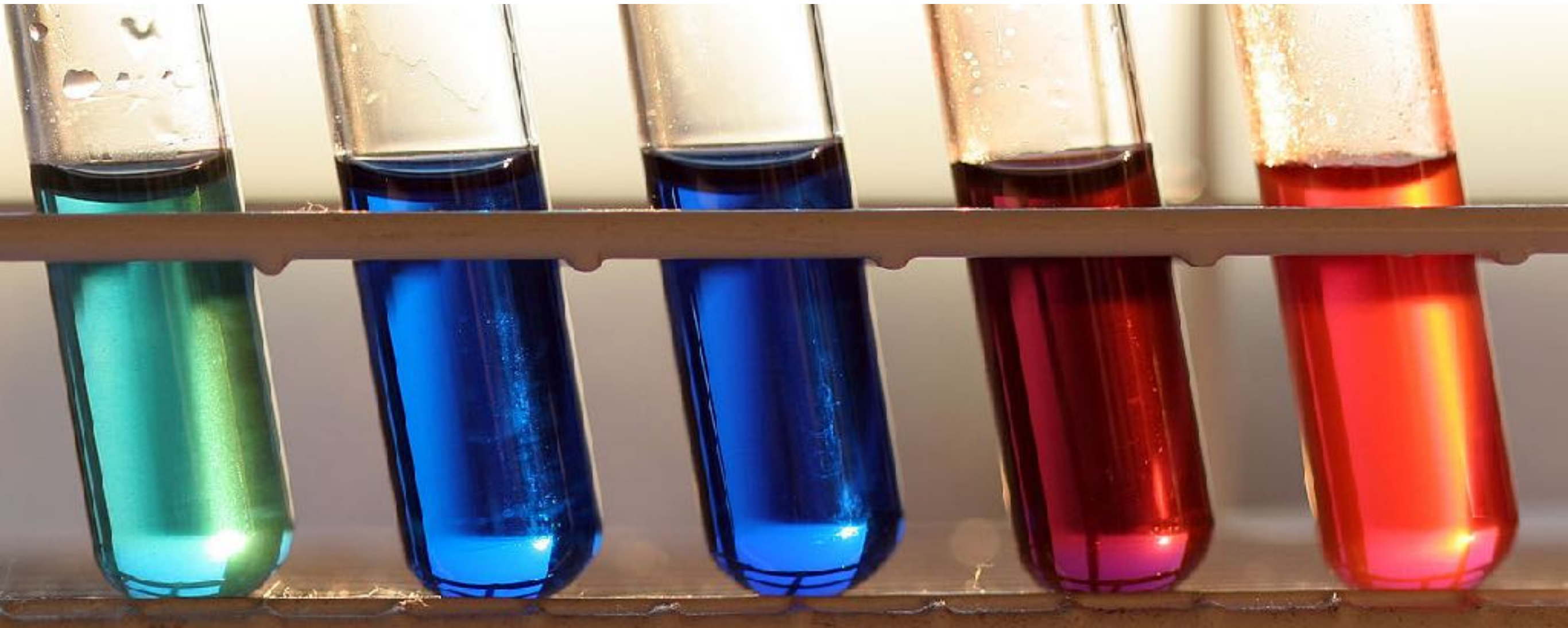
“Swift encourages **typed** functional programming in a mainstream language.”

Why Type Systems?

Catch bugs & prevent crashes

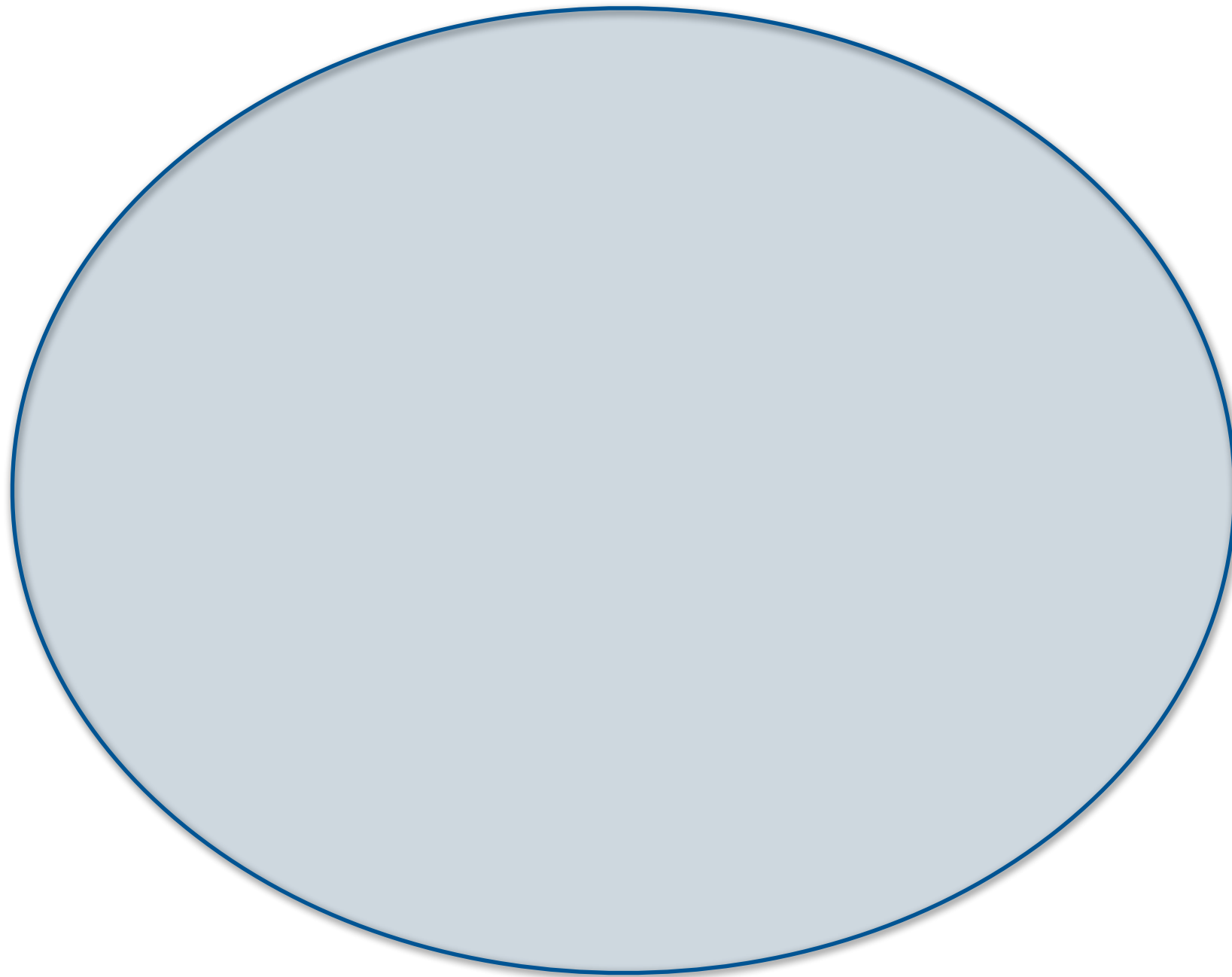


Less testing!

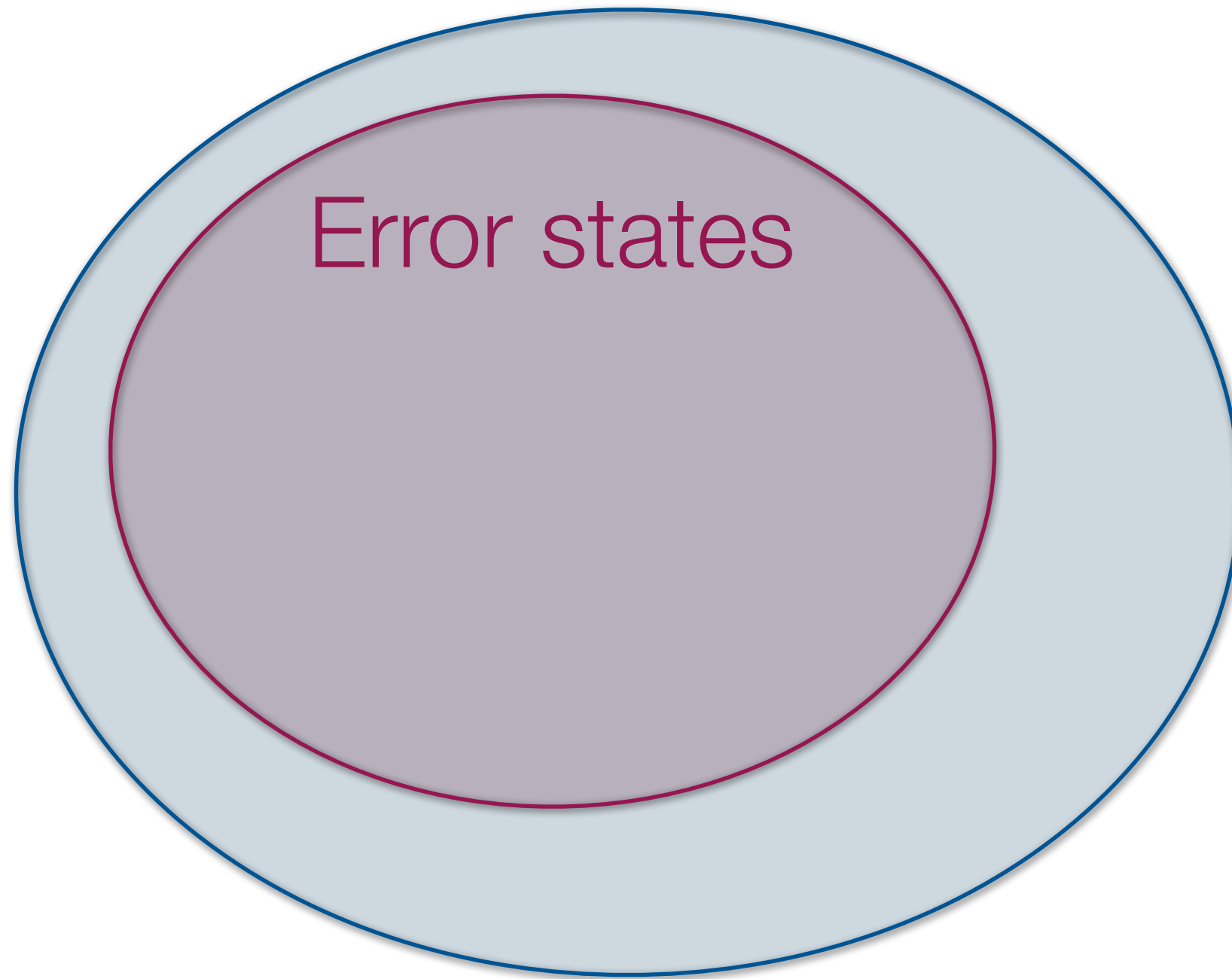


“What if you could gain the same confidence in correctness with fewer or simpler tests?”

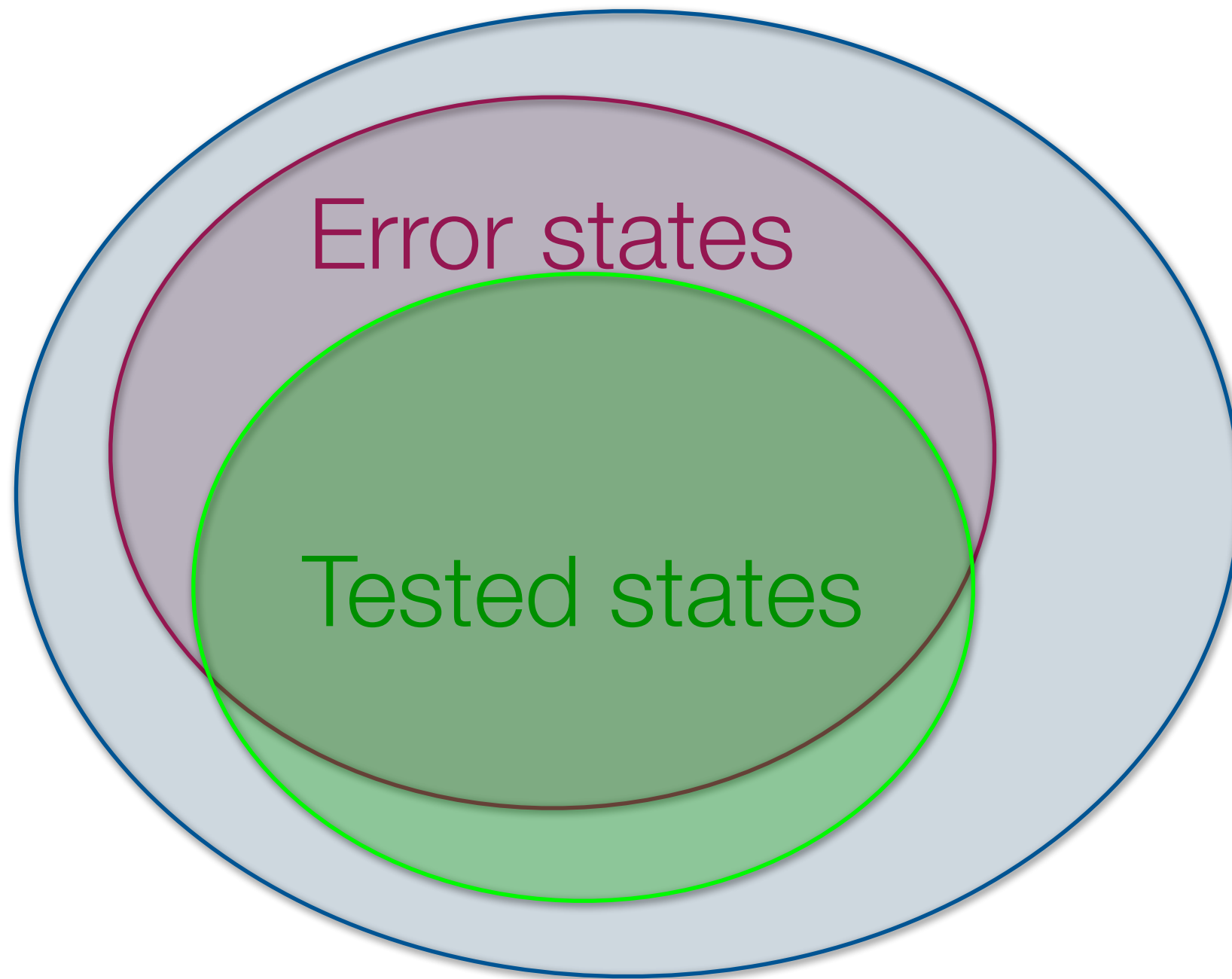
Possible states of your application



Possible states of your application

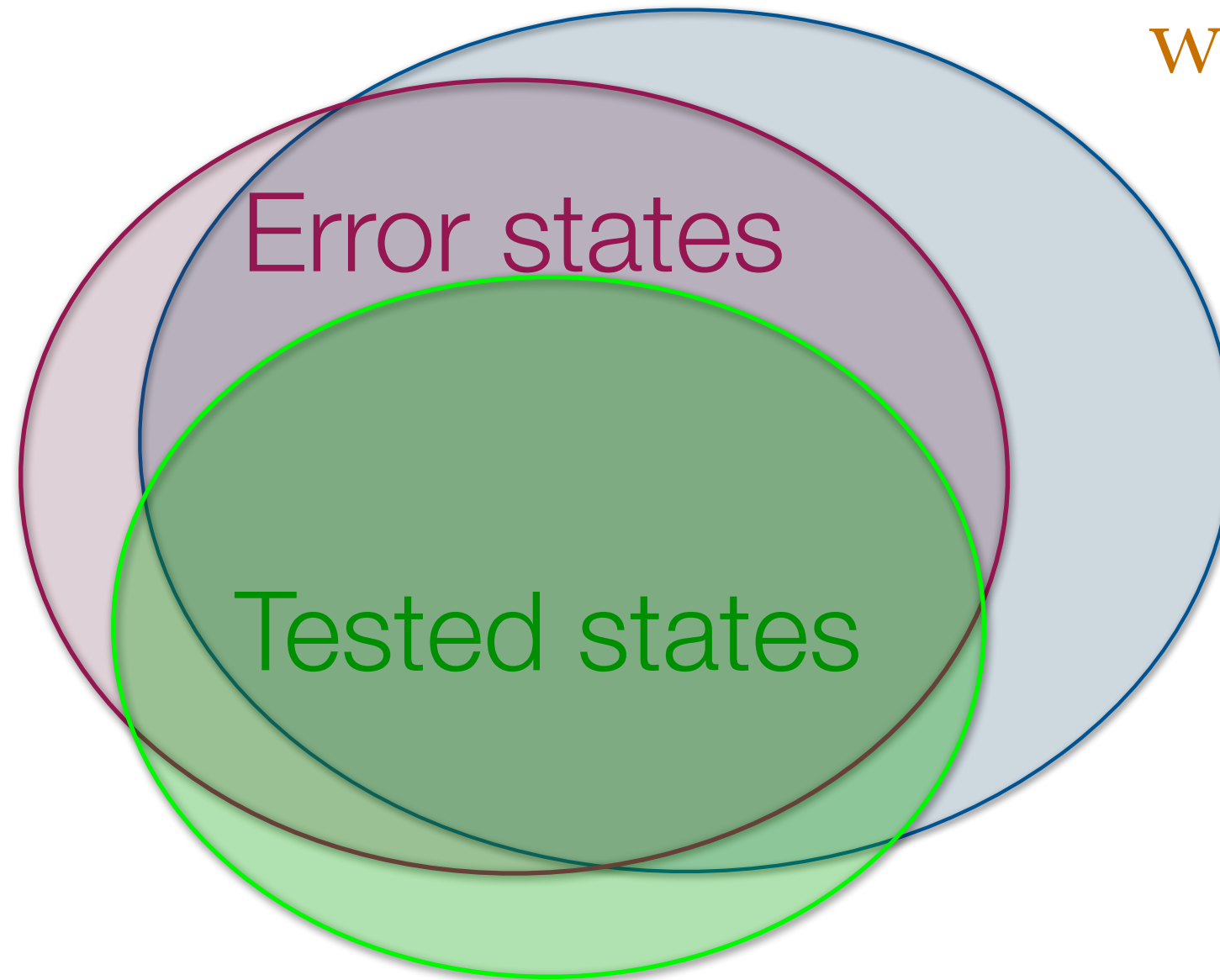


Possible states of your application



Possible states of your application

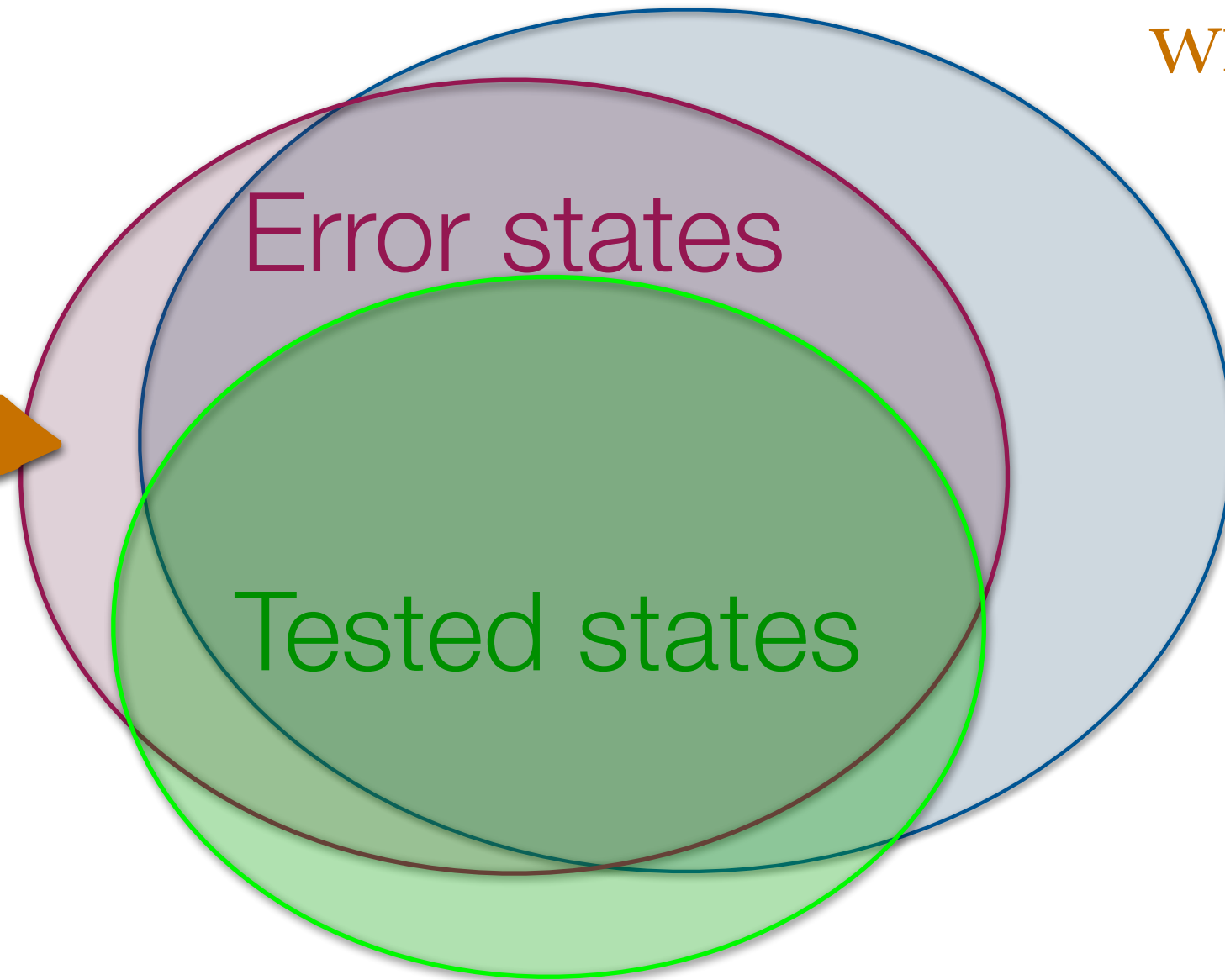
with types



Possible states of your application

with types

eliminate
bugs

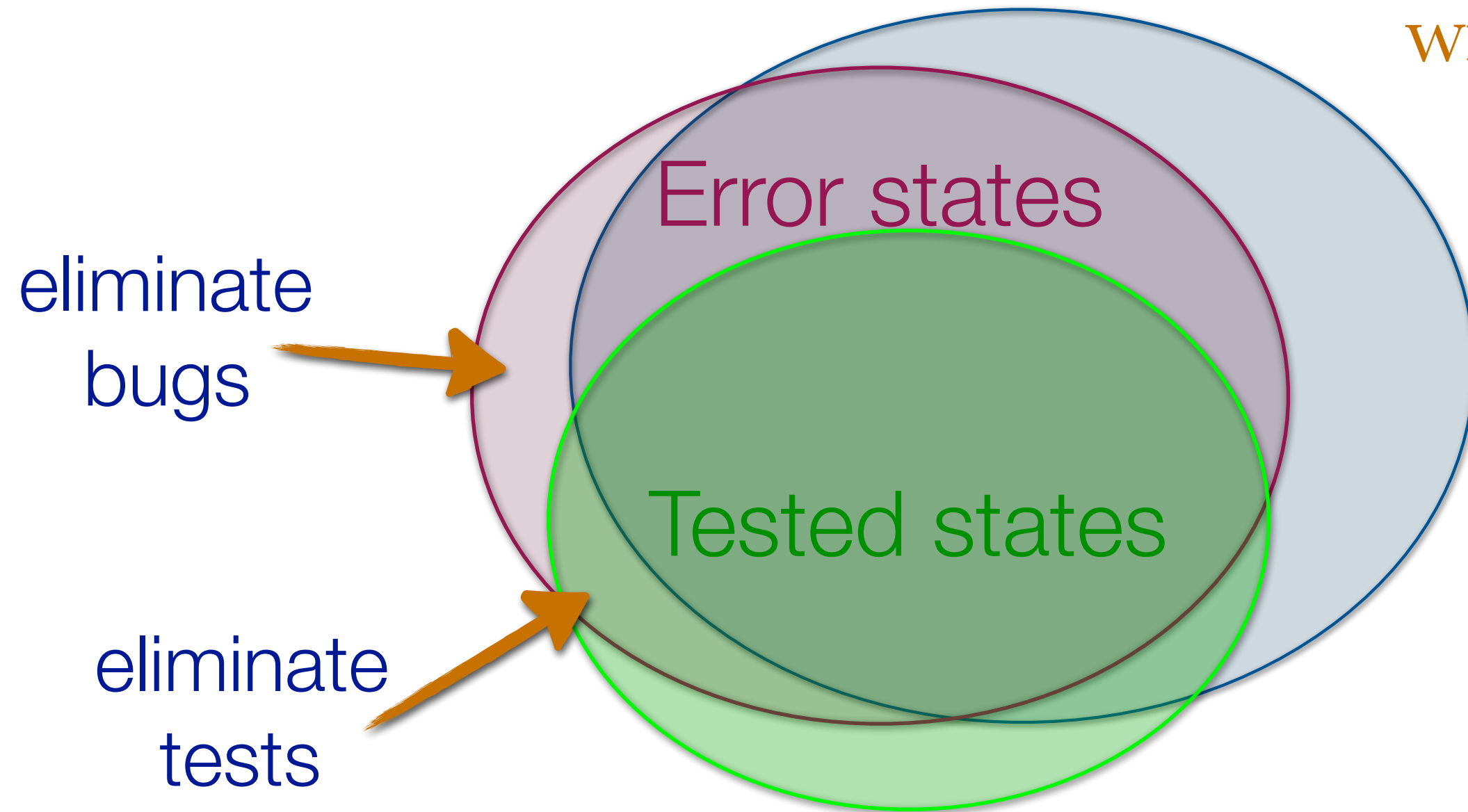


Error states

Tested states

Possible states of your application

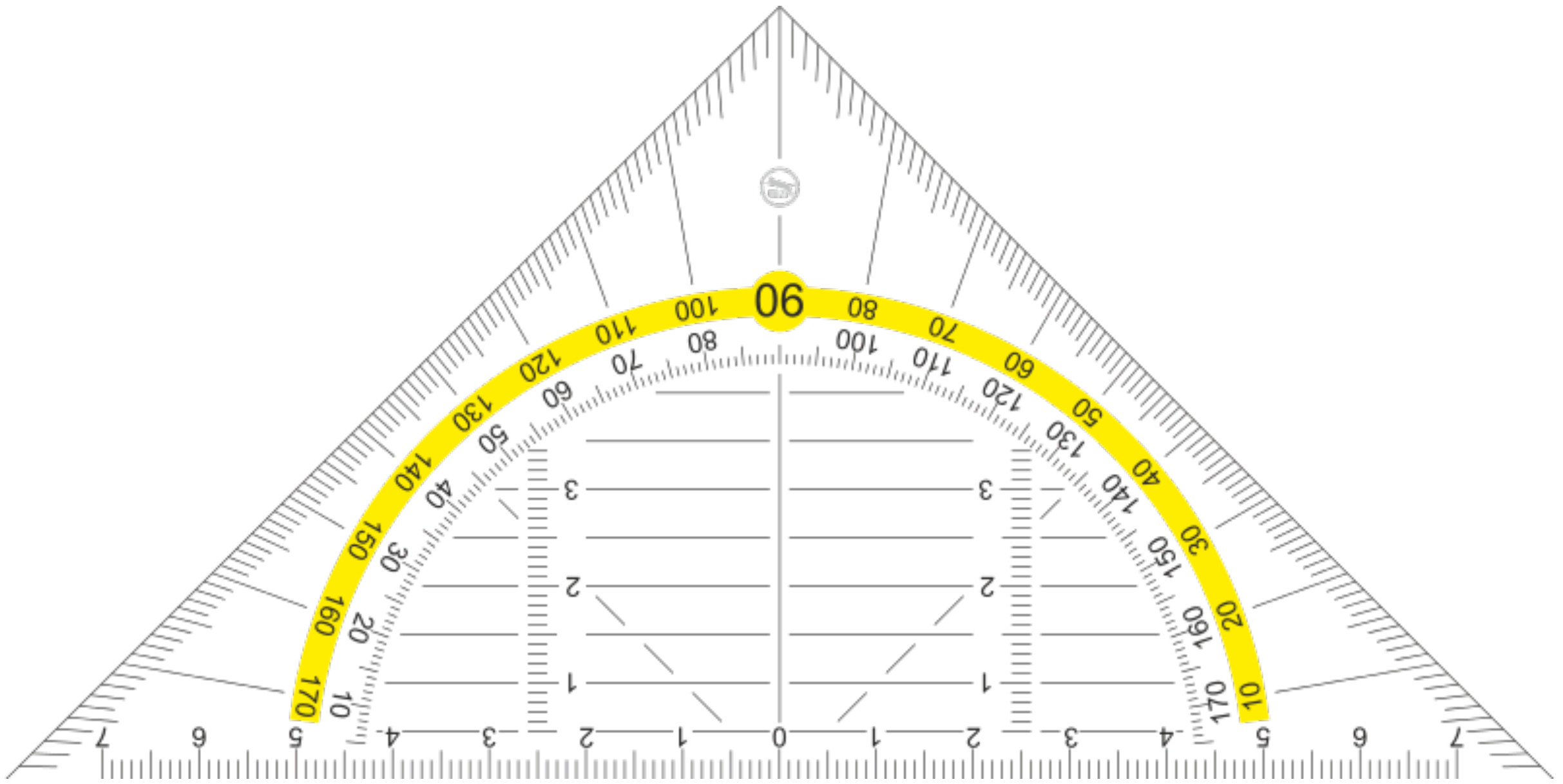
with types

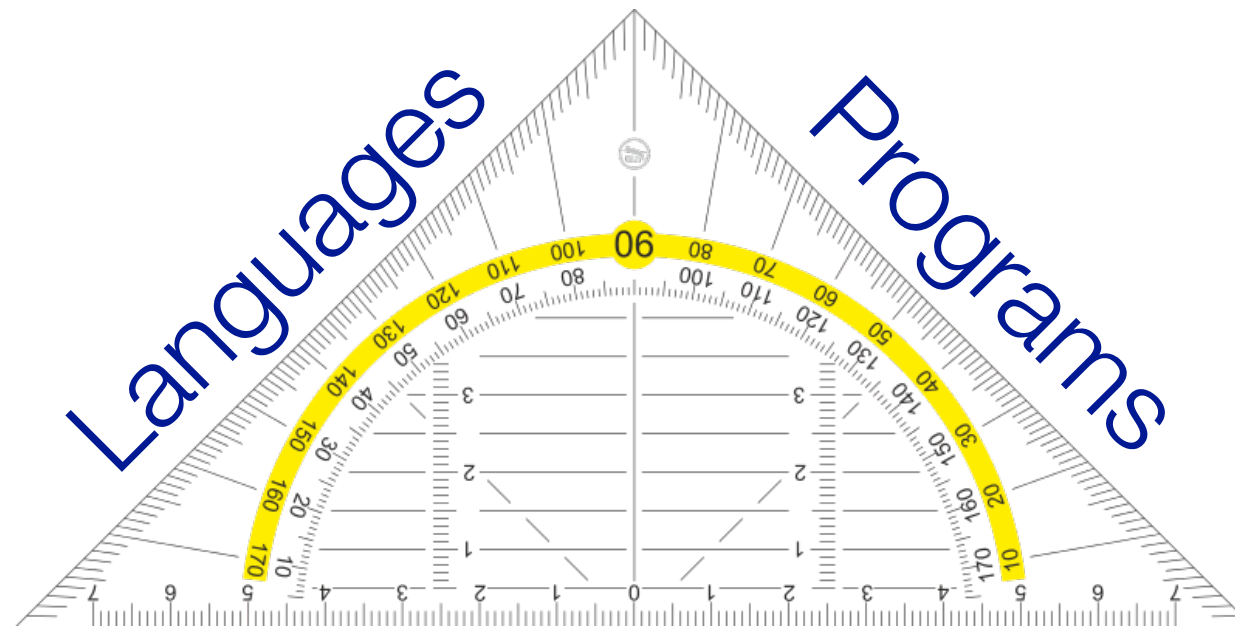


Are types restraints?

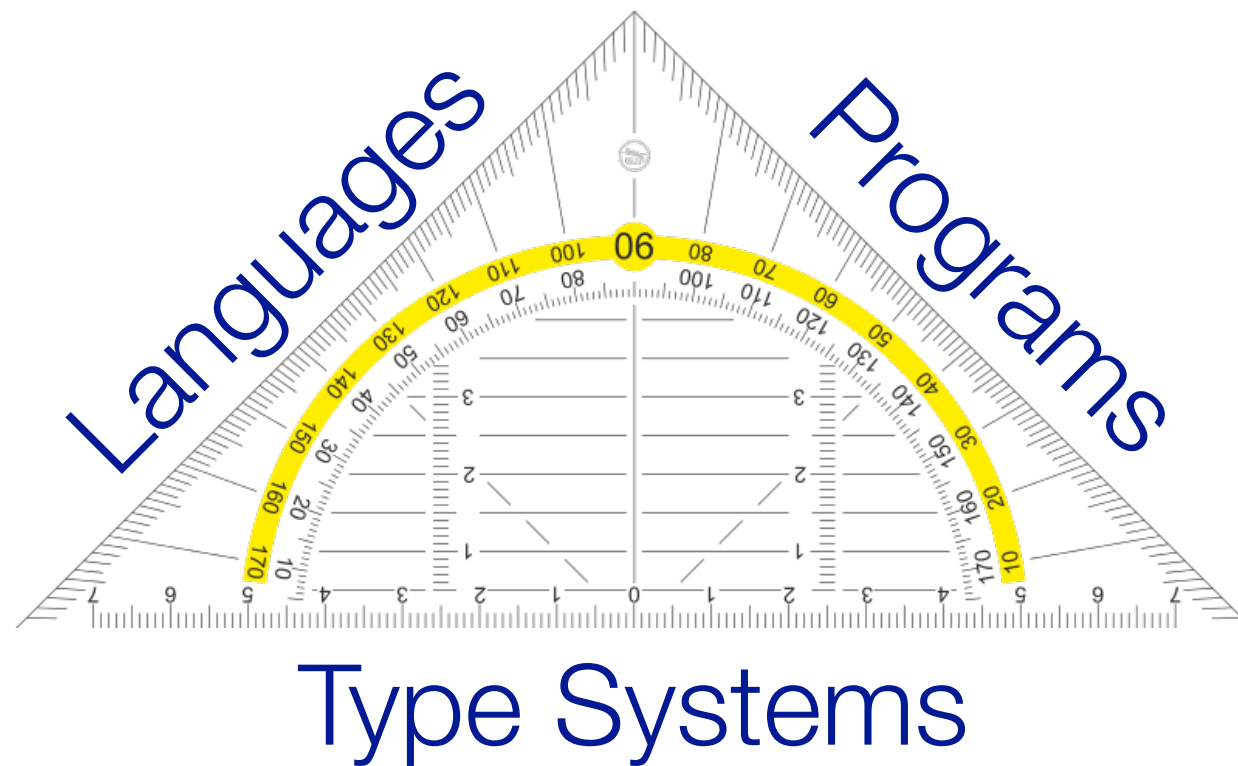


Types are a design tool!





Type Systems



“Make **undesirable** states
unrepresentable.”

“Swift encourages **typed functional programming** in a mainstream language.”

“Swift encourages **typed functional programming** in a mainstream language.”

“Swift encourages **typed functional programming** in a mainstream language.”

Value types

Immutable model & UI state machines



“Swift encourages **typed functional programming** in a mainstream language.”

Value types

Immutable model & UI state machines



Protocols with associated types

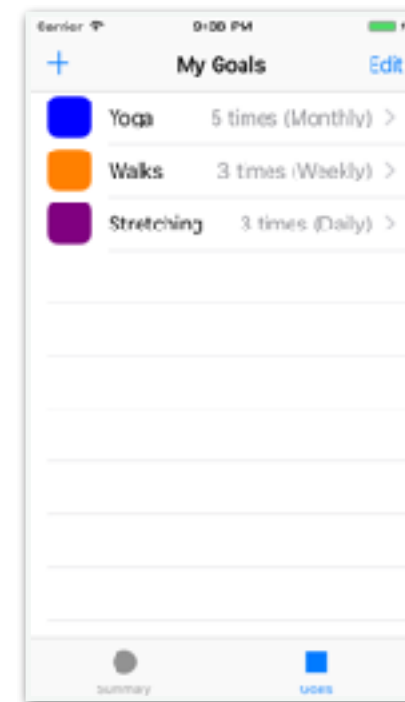
Generic change propagation



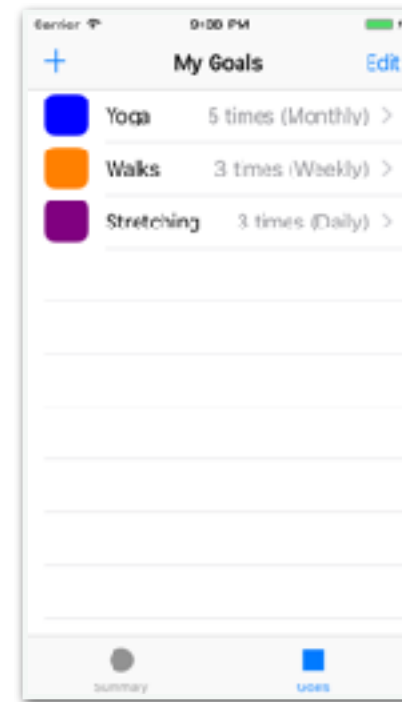
Goals.app



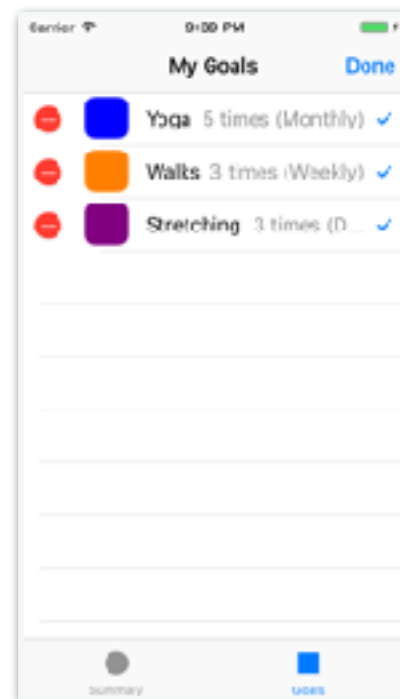
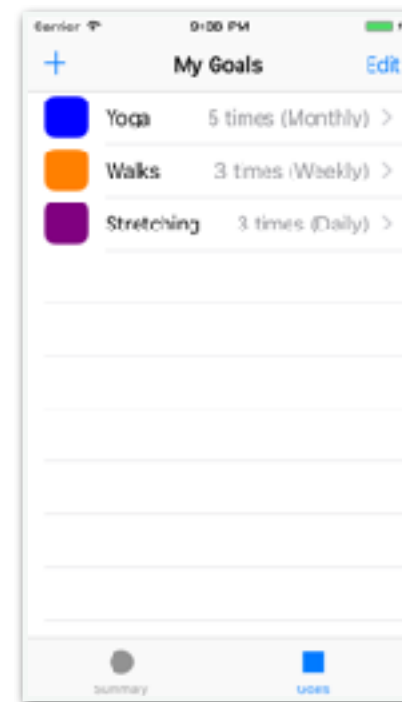
Goals.app



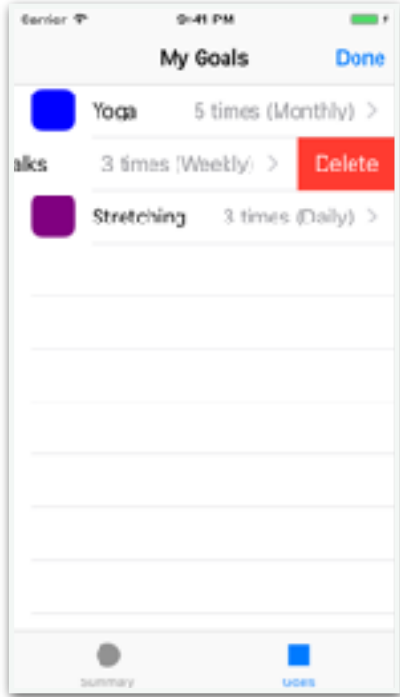
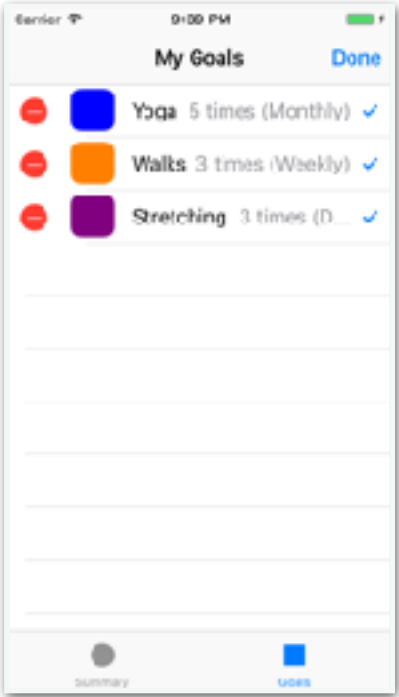
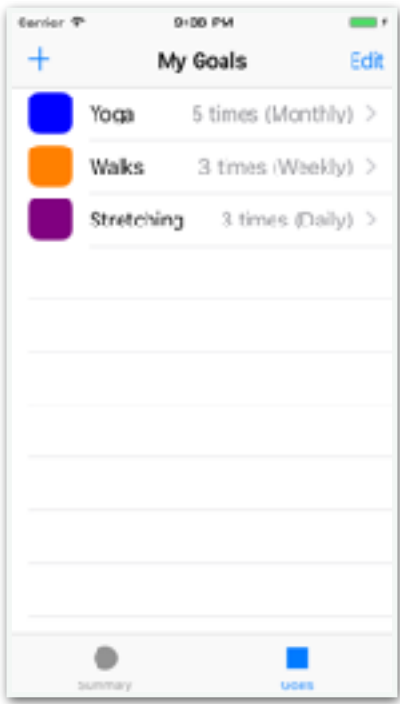
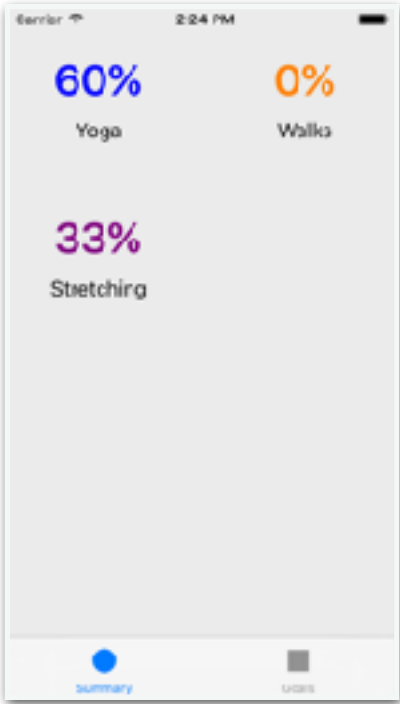
Goals.app



Goals.app



Goals.app





Value Types

Localise & structure change

Value types in Swift

Value types in Swift

struct

```
struct Point {  
    var x: Double  
    var y: Double  
}
```

Value types in Swift

struct

```
struct Point {  
    var x: Double  
    var y: Double  
}
```

enum with associated values

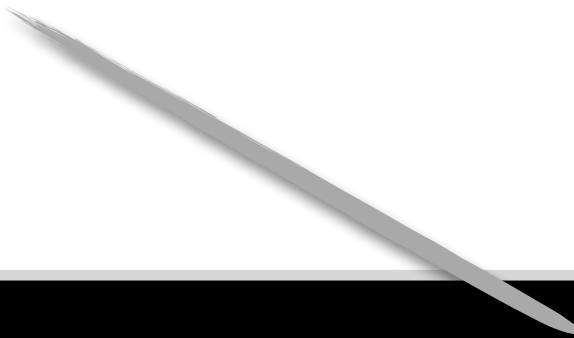
```
enum Result<ResultType> {  
    case result(value: ResultType)  
    case error(err: NSError)  
}
```


Value types in Swift

struct

```
struct Point {  
    var x: Double  
    var y: Double  
}
```

generic type
parameter



enum with associated values

```
enum Result<ResultType> {  
    case result(value: ResultType)  
    case error(err: NSError)  
}
```


Kotlin Edition

```
// Data class (similar to struct)
data class Point(var x: Double, var y: Double)

// Sealed class (similar to enum)
sealed class Result {
    data class Success<T>(val value: T) : Result()
    data class Error(val err: Exception) : Result()
}
```

Reference types versus value types

```
var v: RefPnt
```

```
var w: RefPnt
```

```
class RefPnt { var x, y: Double ... }
```

Reference types versus value types

`var v: RefPnt` 

`x: 0, y: 0`

`var w: RefPnt`

```
class RefPnt { var x, y: Double ... }  
v = RefPnt(x: 0, y: 0)
```

Reference types versus value types

`var v: RefPnt` 

`x: 0, y: 0`

`var w: RefPnt` 

```
class RefPnt { var x, y: Double ... }  
  
v = RefPnt(x: 0, y: 0)  
w = v
```

Reference types versus value types

`var v: RefPnt` 

`x: 10, y: 0`

`var w: RefPnt` 

```
class RefPnt { var x, y: Double ... }  
  
v = RefPnt(x: 0, y: 0)  
w = v  
w.x = 10
```

Reference types versus value types

```
var v: Point
```

```
var w: Point
```

```
struct Point { var x, y: Double }
```

Reference types versus value types

`var v: Point` → `x: 0, y: 0`

`var w: Point`

```
struct Point { var x, y: Double }  
v = Point(x: 0, y: 0)
```

Reference types versus value types

`var v: Point` → `x: 0, y: 0`

`var w: Point` → `x: 0, y: 0`

```
struct Point { var x, y: Double }  
  
v = Point(x: 0, y: 0)  
w = v
```


Reference types versus value types

`var v: Point` → `x: 0, y: 0`

`var w: Point` → `x: 10, y: 0`

```
struct Point { var x, y: Double }  
  
v = Point(x: 0, y: 0)  
w = v  
w.x = 10
```

Value types

`var v: Point` → `x: 0, y: 0`

`var w: Point` → `x: 10, y: 0`

Value types

`var v: Point` → `x: 0, y: 0`

`var w: Point` → `x: 10, y: 0`

Localise change

Value types

`var v: Point` → `x: 0, y: 0`

`var w: Point` → `x: 10, y: 0`

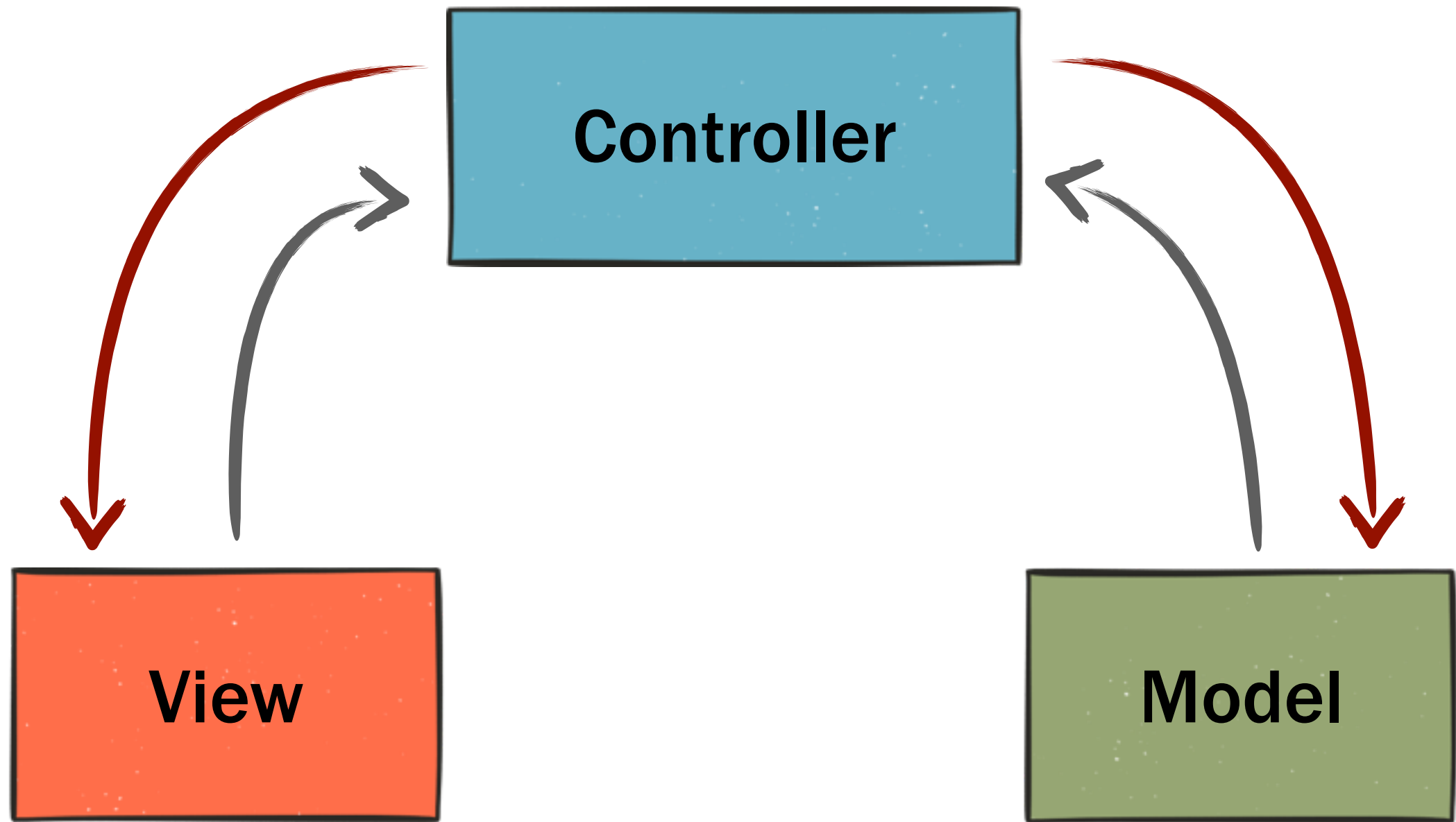
Localise change

Facilitate local reasoning

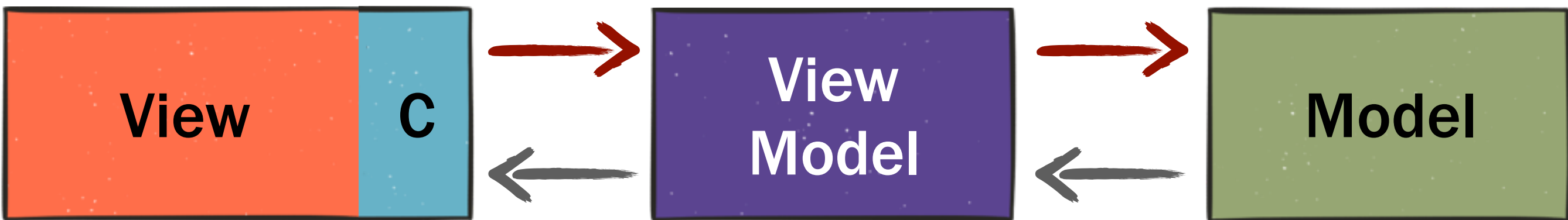
Value Types

Immutable Model

MVC & MVVM Architectures

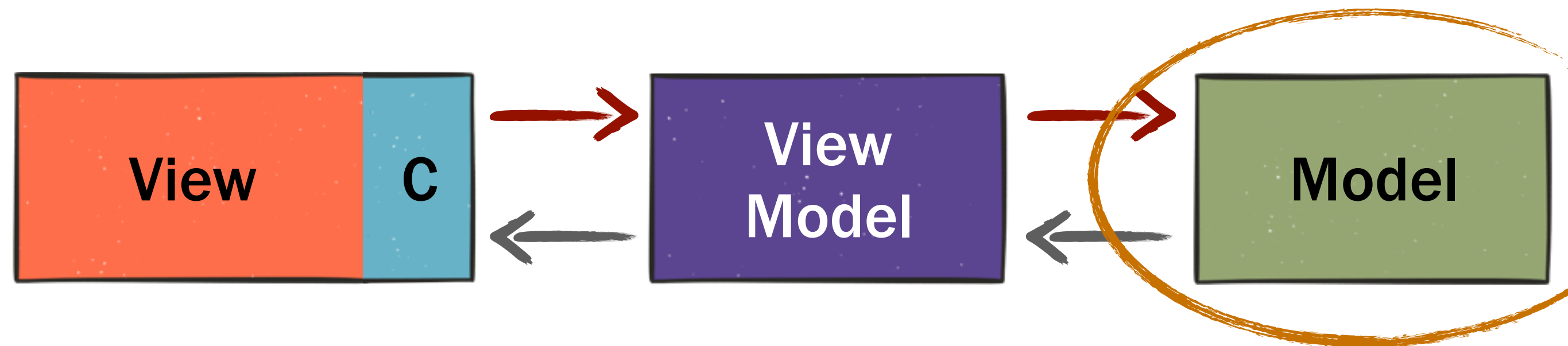


MVC & MVVM Architectures



MVC & MVVM Architectures

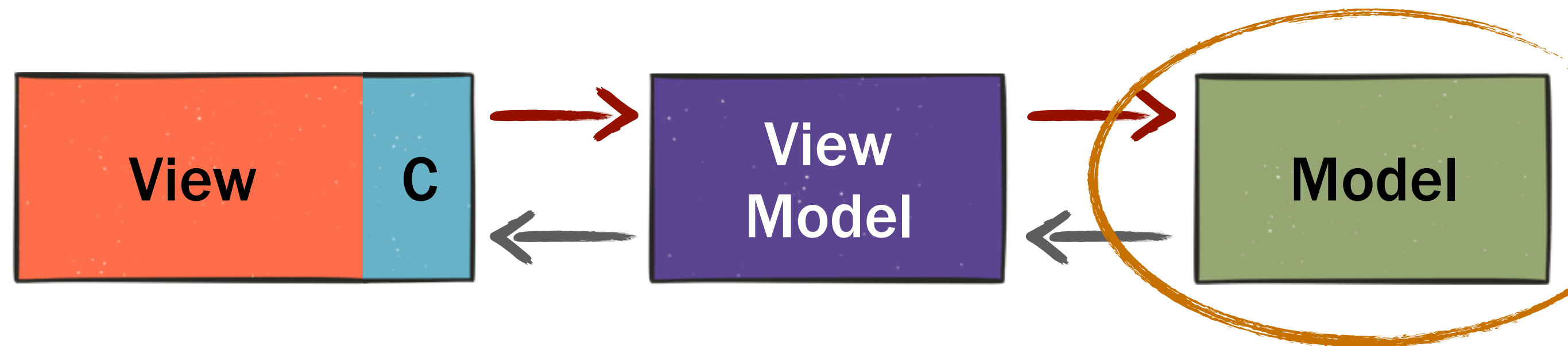
passed around as
reference type



MVC & MVVM Architectures

✗ Accidental changes

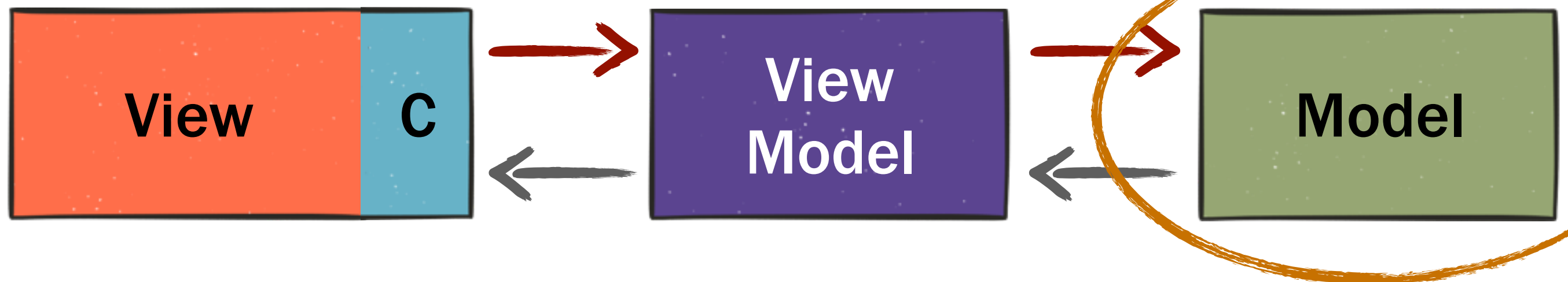
passed around as
reference type



MVC & MVVM Architectures

- ✗ Accidental changes
- ✗ Changes in wrong order

passed around as
reference type



Immutable Goals

```
struct Goal {  
  let uuid:      UUID           // fast equality  
  var colour:   UIColor  
  var title:    String  
  var interval: GoalInterval  
  var frequency: Int  
}
```

Immutable Goals

```
struct Goal {  
  let uuid:      UUID           // fast equality  
  var colour:   UIColor  
  var title:    String  
  var interval: GoalInterval  
  var frequency: Int  
}
```

```
 typealias GoalProgress = (goal: Goal, count: Int?)
```

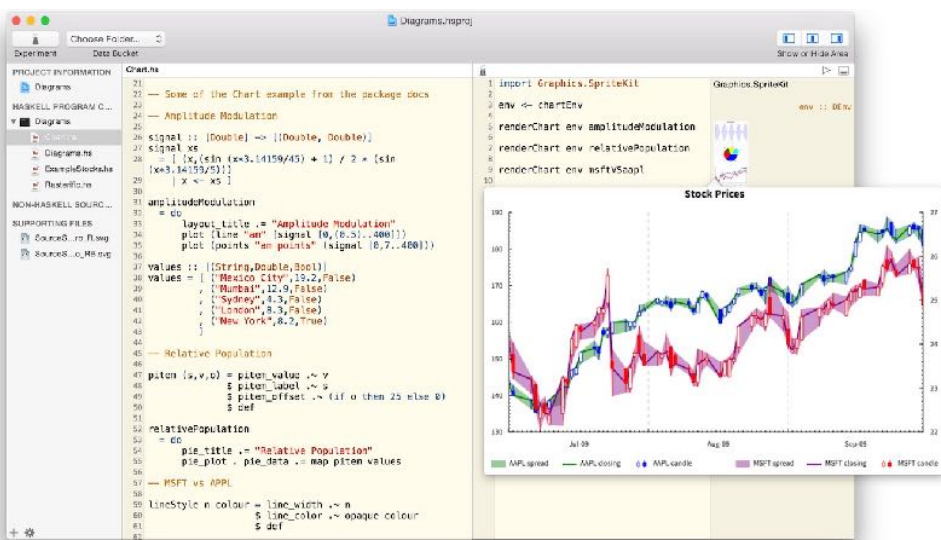
Immutable Goals

```
struct Goal {  
  let uuid:      UUID           // fast equality  
  var colour:    UIColor  
  var title:     String  
  var interval:  GoalInterval  
  var frequency: Int  
}  
  
typealias GoalProgress = (goal: Goal, count: Int?)  
  
typealias Goals = [GoalProgress] // array
```

Kolin Edition

```
data class Goal(  
    val uuid: UUID,  
    var colour: Color,  
    var title: String,  
    var interval: GoalInterval,  
    var frequency: Int)  
  
typealias GoalProgress = Pair<Goal, Int?>  
  
typealias Goals = Array<GoalProgress>
```

Immutable Models Do Scale!

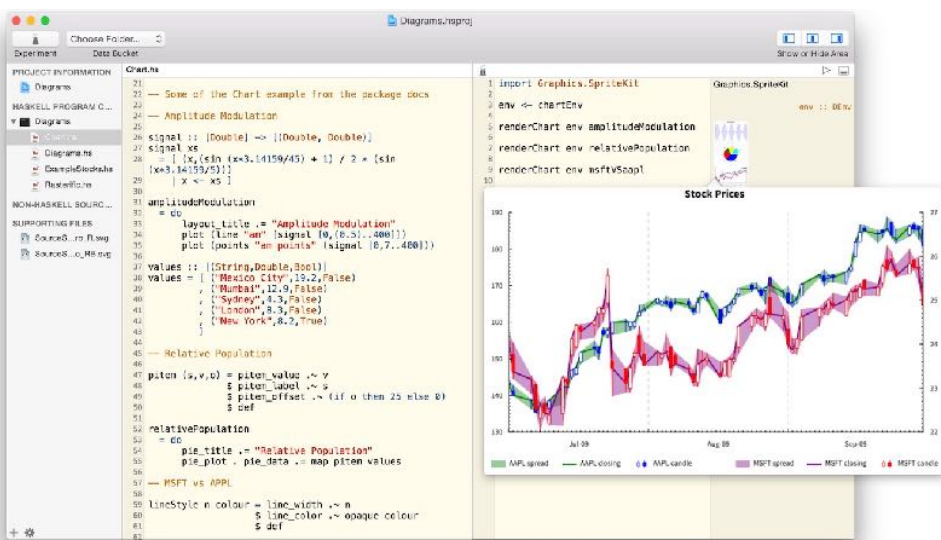


Immutable Models Do Scale!

Haskell for Mac

Uses an immutable model

Adopts Cocoa Document Architecture



Immutable Models Do Scale!

Haskell for Mac

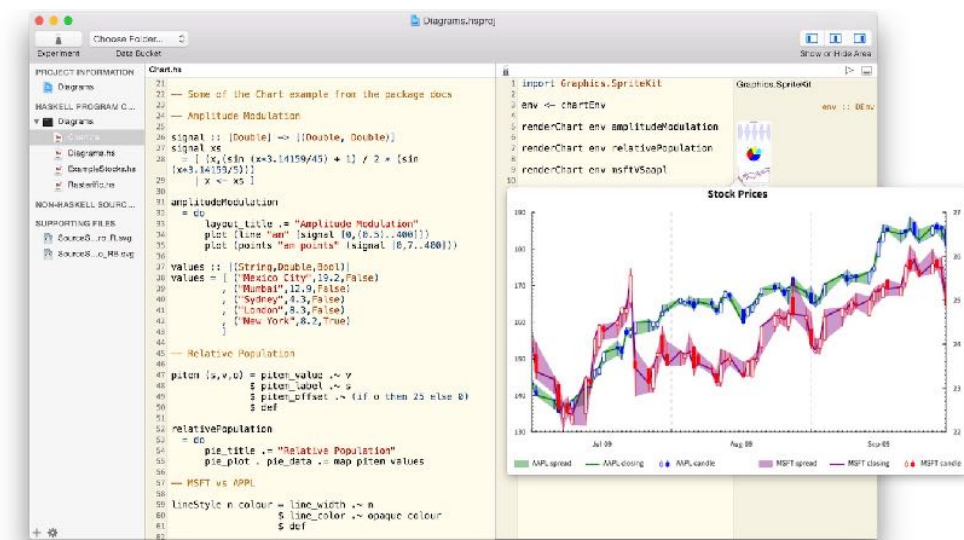
Uses an immutable model

Adopts Cocoa Document Architecture

“Functional Programming in a Stateful World”

YOW! Lambda Jam 2015

(on speakerdeck.com)



But what if...

But what if...

“Use an **immutable wrapper API** to wrap mutable structures.”

But what if...

“Use an **immutable wrapper API** to wrap mutable structures.”

✓ **React (Native)**

Virtual DOM wraps DOM

But what if...

“Use an **immutable wrapper API** to wrap mutable structures.”

✓ React (Native)

Virtual DOM wraps DOM

✓ Haskell SpriteKit — Haskell library binding

Algebraic data type wraps SpriteKit's mutable object graph

Replacing Tests

Replacing Tests

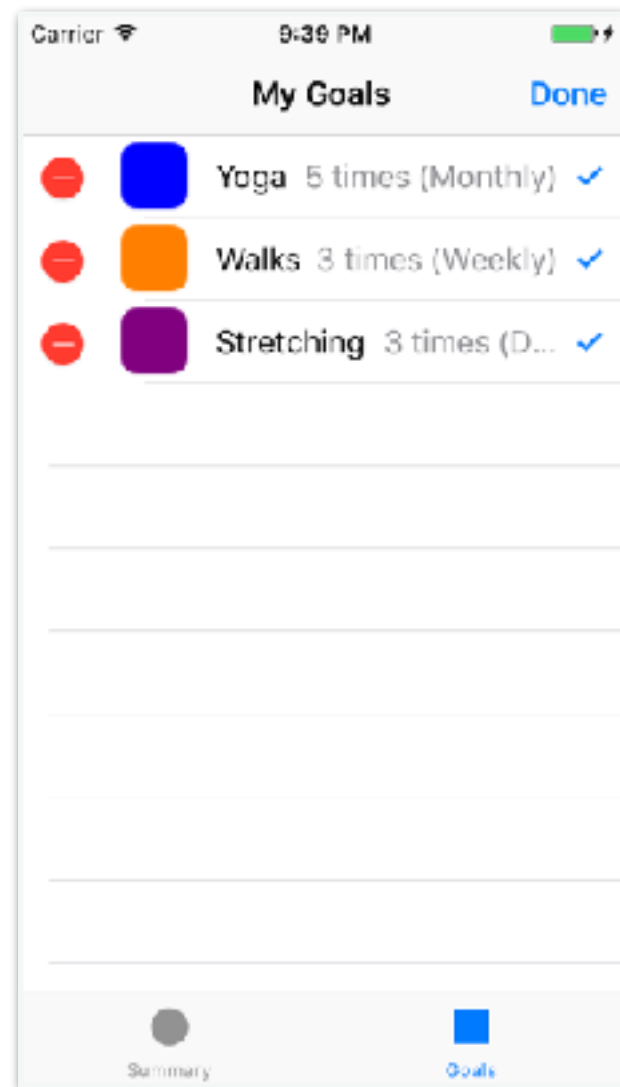
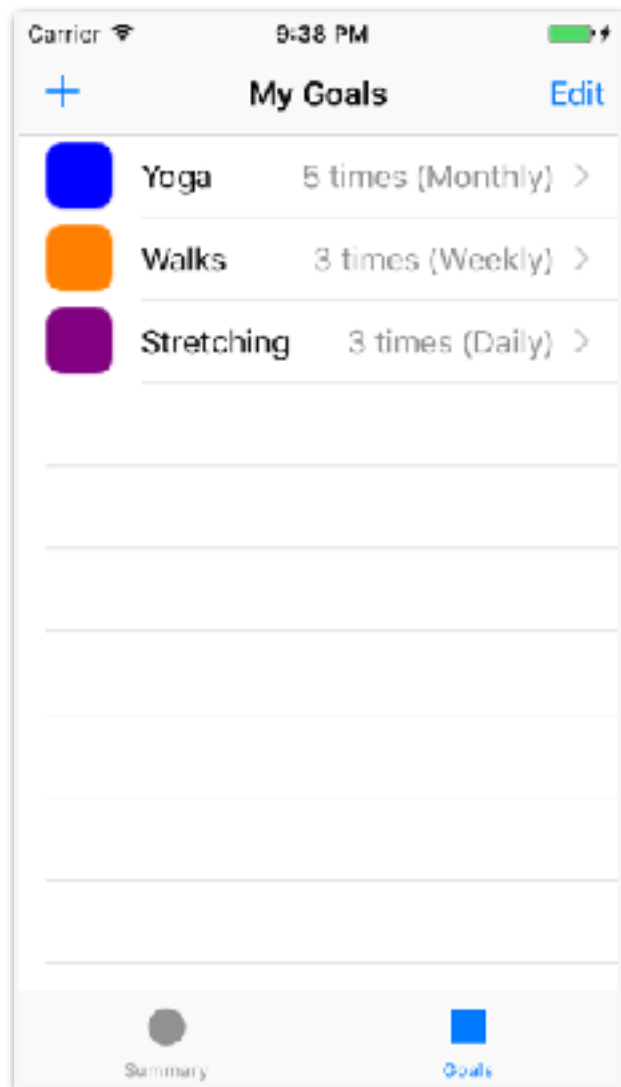
- ✓ Avoid accidental changes of model state
Replaces integration tests

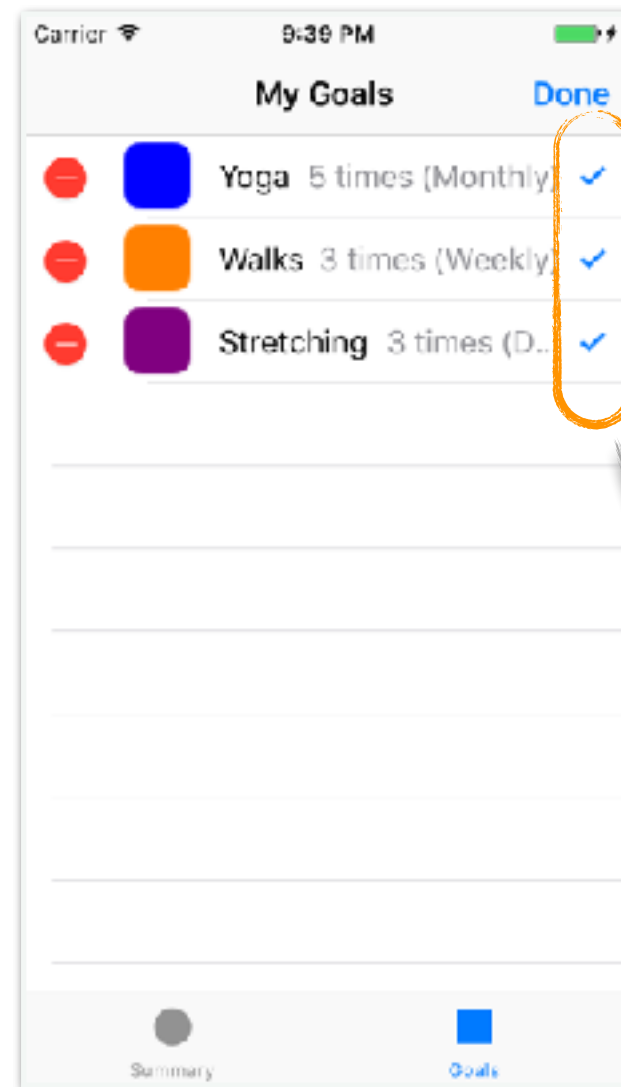
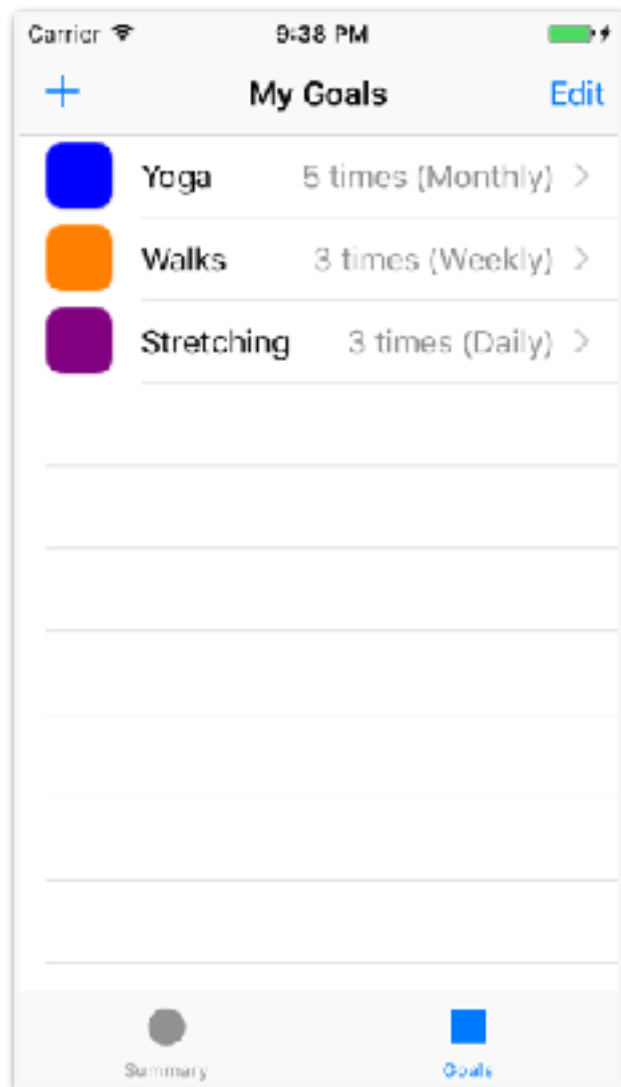
Replacing Tests

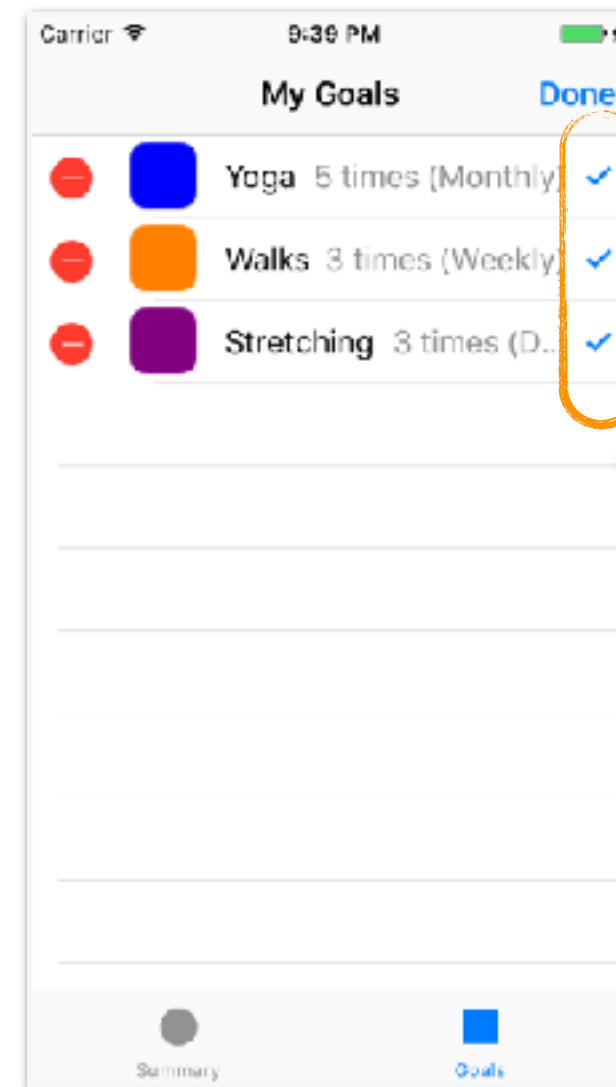
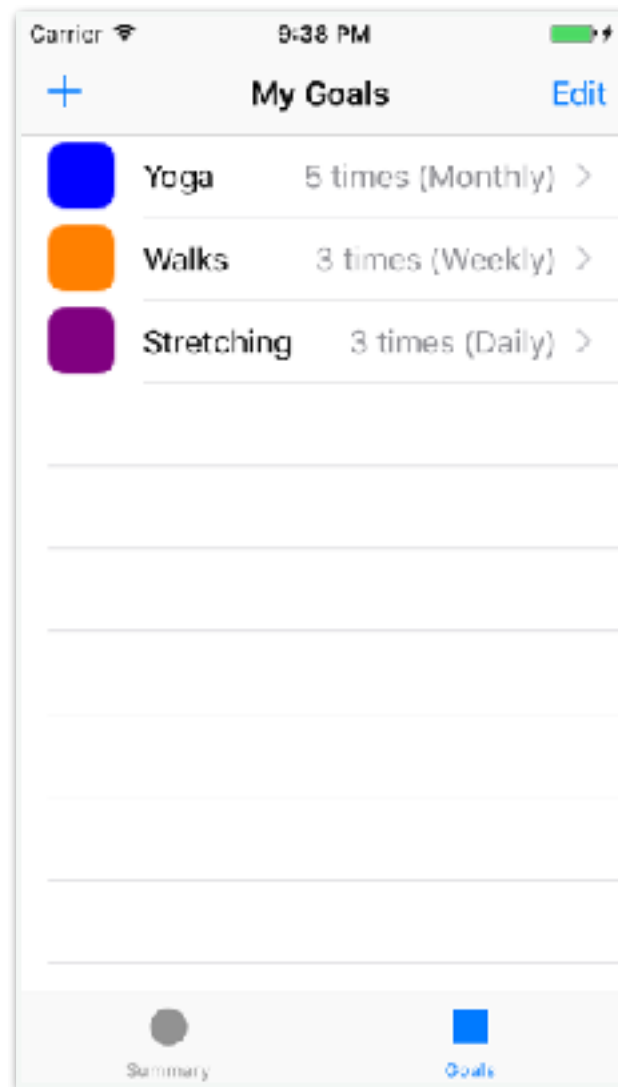
- ✓ Avoid accidental changes of model state
Replaces integration tests
- ✓ Avoid races on concurrent threads
Often not properly tested

Enums with Associated Values

UI State Machines

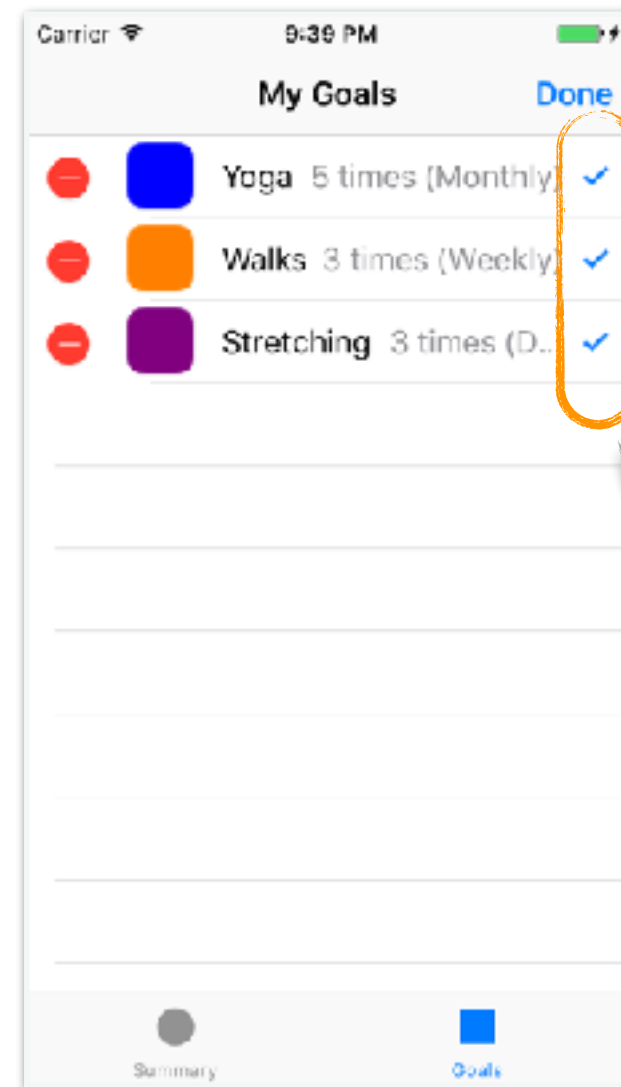
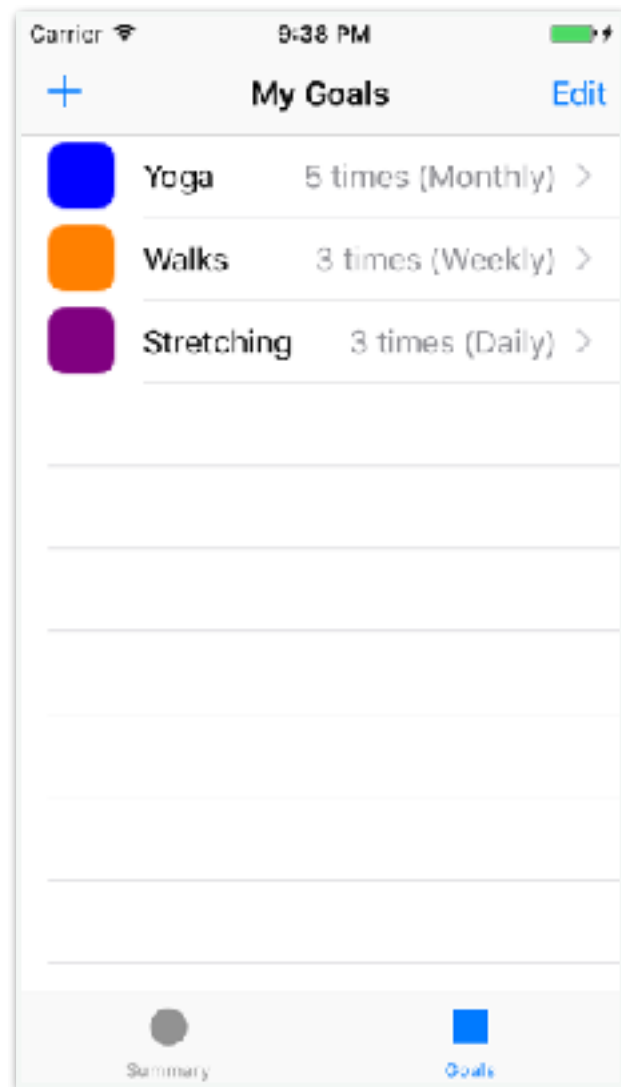






Is goal active?

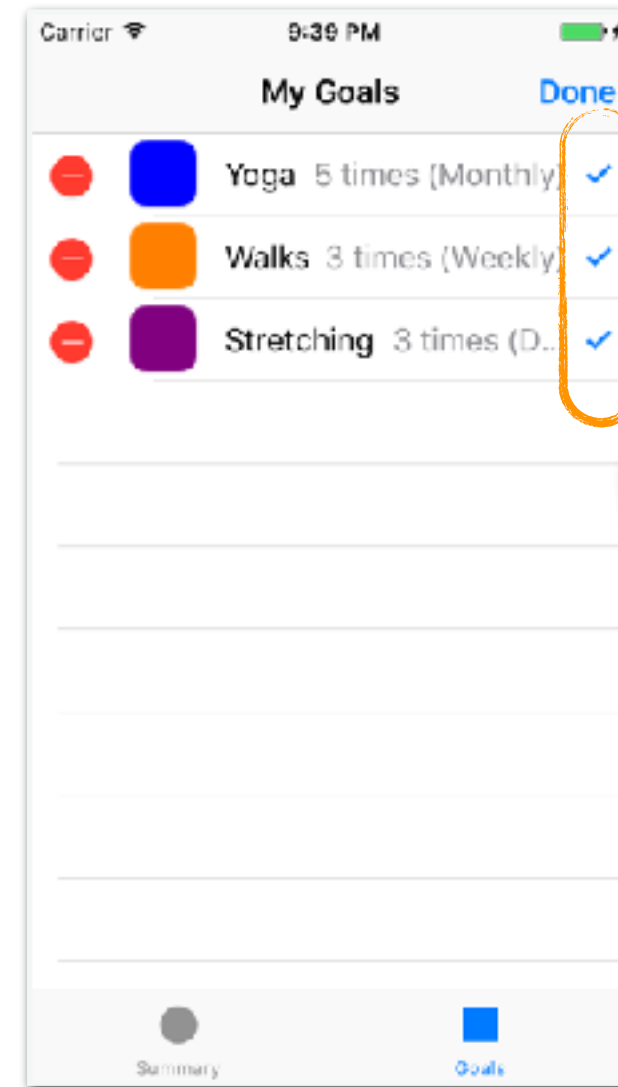
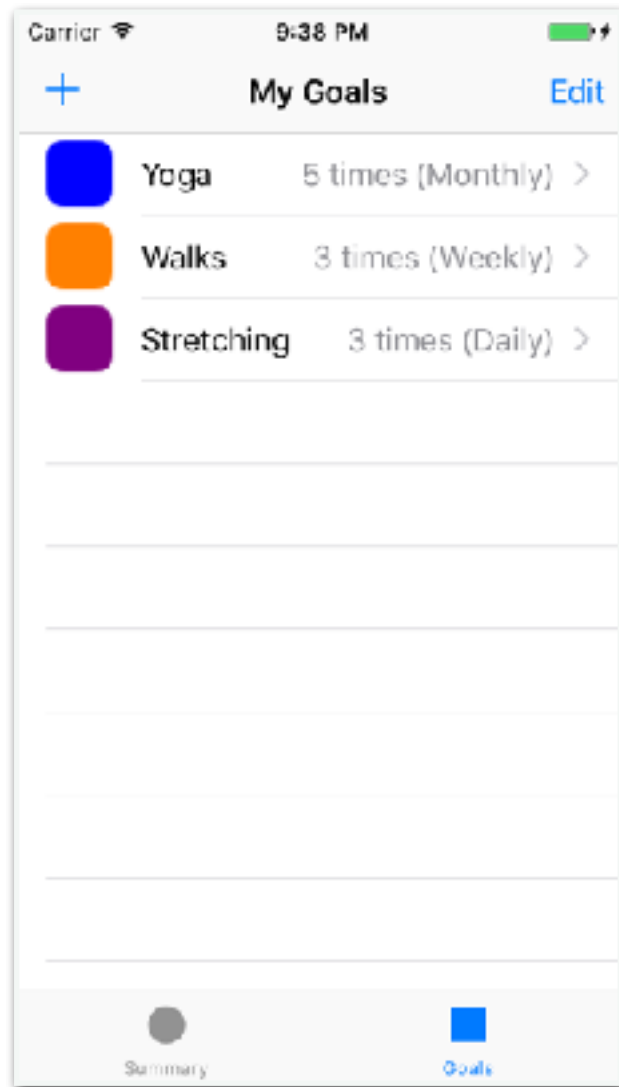
```
typealias GoalProgress = (goal: Goal, count: Int?)
typealias Goals = [GoalProgress]
```



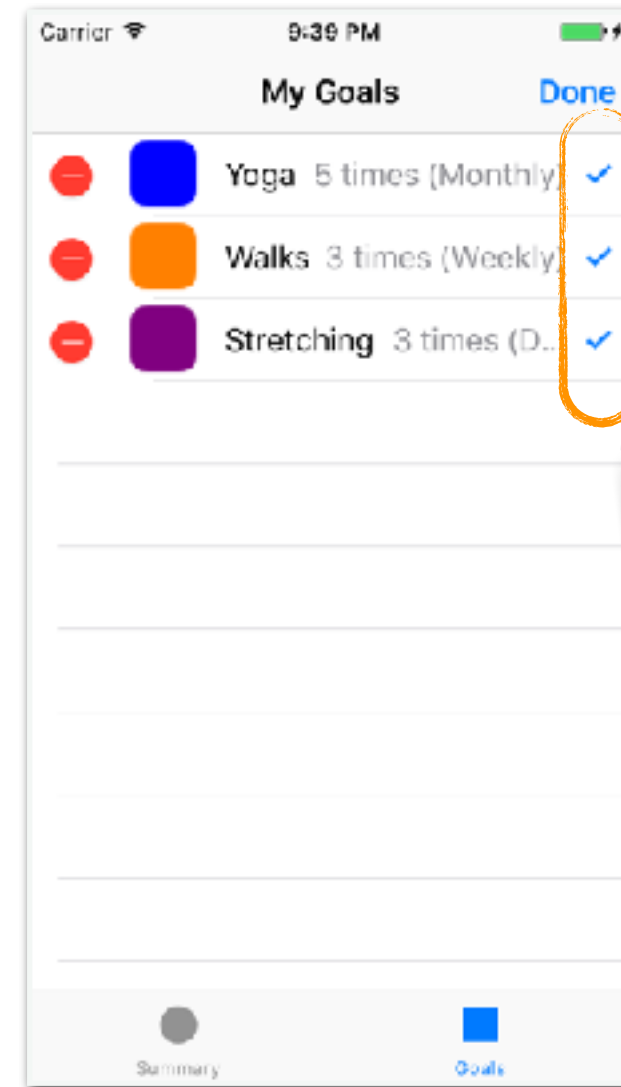
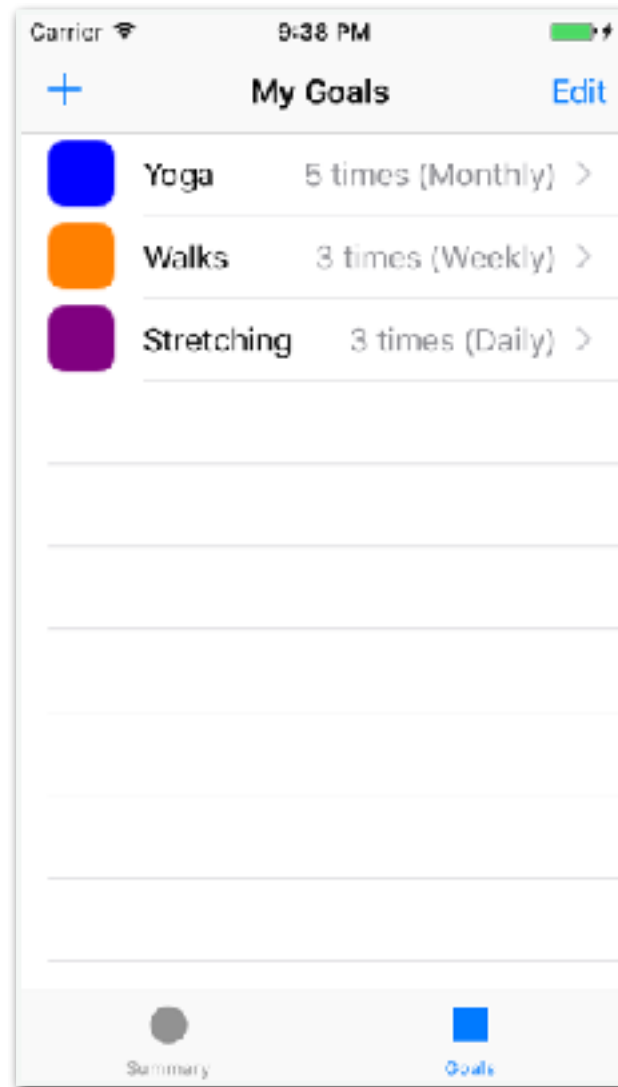
Is goal active?

```
typealias GoalProgress = (goal: Goal, count: Int?)  
typealias Goals = [GoalProgress]
```

Need to be careful not to lose progress info



```
var goalsActivity: [Bool]
```



```
var goalsActivity: [Bool]
```

Not valid outside of editing mode

```
enum GoalsEditState {  
  case displaying  
  case editing  
}
```

```
var goalsActivity: [Bool]
```



```
enum GoalsEditState {  
  case displaying  
  case editing(goalsActivity: [Bool])  
}
```

```
enum GoalsEditState {  
    case displaying  
    case editing(goalsActivity: [Bool])  
}
```

GoalsController: UITableViewController

```
var editState: GoalsEditState = .displaying
```

```
...
```

```
enum GoalsEditState {  
    case displaying  
    case editing(goalsActivity: [Bool])  
}
```

GoalsController: UITableViewController

```
var editState: GoalsEditState = .displaying  
...  
switch editState { // change mode
```



```
enum GoalsEditState {  
    case displaying  
    case editing(goalsActivity: [Bool])  
}
```

GoalsController: UITableViewController

```
var editState: GoalsEditState = .displaying  
...  
switch editState { // change mode  
    case .displaying: // alter UI state  
        navigationItem.setLeftBarButton(nil, ...)  
        editState = .editing(goalsActivity:  
                               dataSource.goalsActivity)  
  
    case .editing(let goalsActivity):  
        navigationItem.setLeftBarButton(addButton, ...)  
        dataSource.commitGoalsActivity(goalsActivity)  
        editState = .displaying }
```

```
enum GoalsEditState {  
  case displaying  
  case editing(goalsActivity: [Bool])  
}
```

GoalsController

```
guard case .editing(var goalsActivity) = editState  
  else { return }
```

```
enum GoalsEditState {  
  case displaying  
  case editing(goalsActivity: [Bool])  
}
```

access to mode-specific
state is guarded

GoalsController

```
guard case .editing(var goalsActivity) = editState  
else { return }
```

```
enum GoalsEditState {  
  case displaying  
  case editing(goalsActivity: [Bool])  
}
```

access to mode-specific
state is guarded

GoalsController

```
guard case .editing(var goalsActivity) = editState  
  else { return }  
  
let newIsActive = !goalsActivity[indexPath.item]  
cell.editingAccessoryType = newIsActive ? .checkmark  
  : .none
```


Kolin Edition

```
sealed class GoalsEditState {  
  data class Displaying : GoalsEditState()  
  data class Editing(val goalsActivity: Array<Boolean>)  
    : GoalsEditState()  
}
```

```
sealed class GoalsEditState {
    data class Displaying : GoalsEditState()
    data class Editing(val goalsActivity: Array<Boolean>)
        : GoalsEditState()
}
```

```
var editState: GoalsEditState
    = GoalsEditState.Displaying()
...
when(editState) {
    is GoalsEditState.Displaying -> {
        navigationItem.setLeftBarButton(null, ...)
        editState = GoalsEditState.Editing(
            goalsActivity = dataSource.goalsActivity)
    }
    is GoalsEditState.Editing -> {
        navigationItem.setLeftBarButton(addButton, ...)
        dataSource.commitGoalsActivity(goalsActivity)
        editState = GoalsEditState.Displaying()
    }
}
```

Replacing Tests

Replacing Tests

- ✓ Avoid inconsistent state changes
Replaces UI tests

Replacing Tests

- ✓ Avoid inconsistent state changes
Replaces UI tests
- ✓ Avoid non-exhaustive transitions
Replaces exhaustiveness tests

A close-up photograph of an Ethernet RJ45 connector. The connector is made of clear plastic and has eight gold-colored metal contacts visible. The wires are color-coded: green, blue, orange, and brown. The background is a plain, light-colored surface.

Protocols with associated types
Structured change propagation

“Now that the model layer is
immutable, how do we change
anything?”

$\Delta t: 0s$

25

Mutable Variable

$\Delta t: 1s$



Mutable Variable

$\Delta t: 2s$

11

Mutable Variable

$\Delta t: 3s$

73

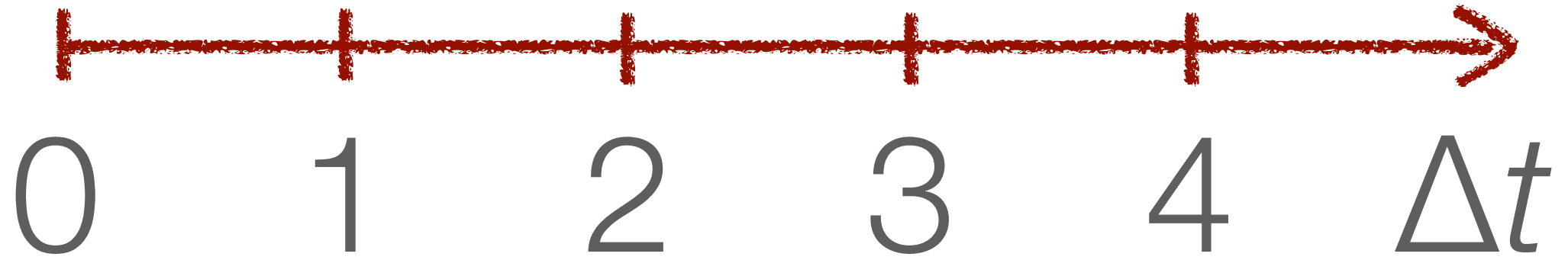
Mutable Variable

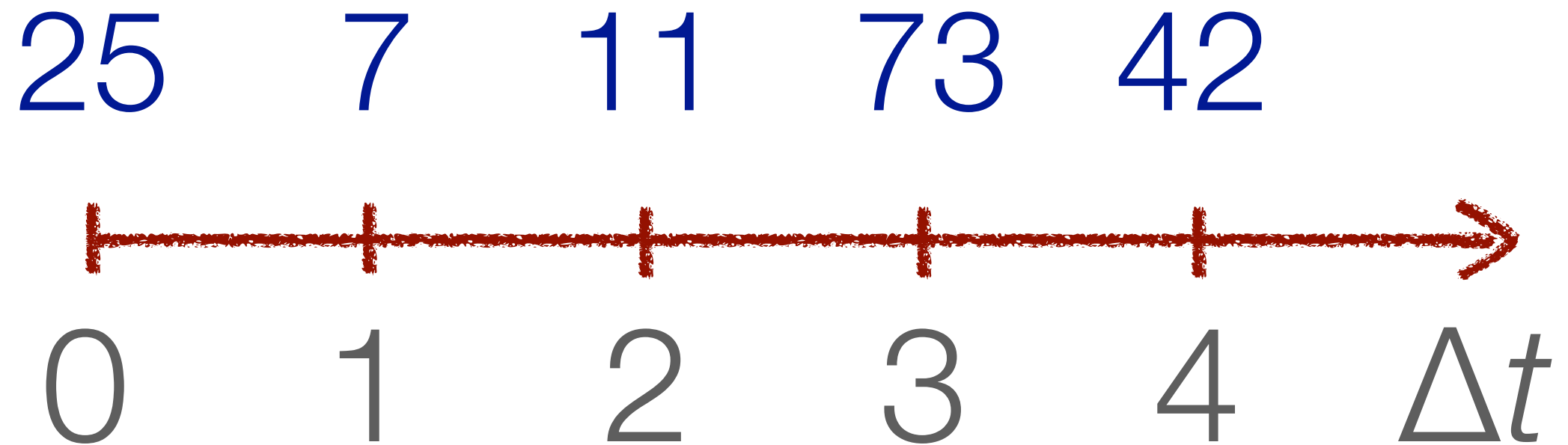
$\Delta t: 4s$

42

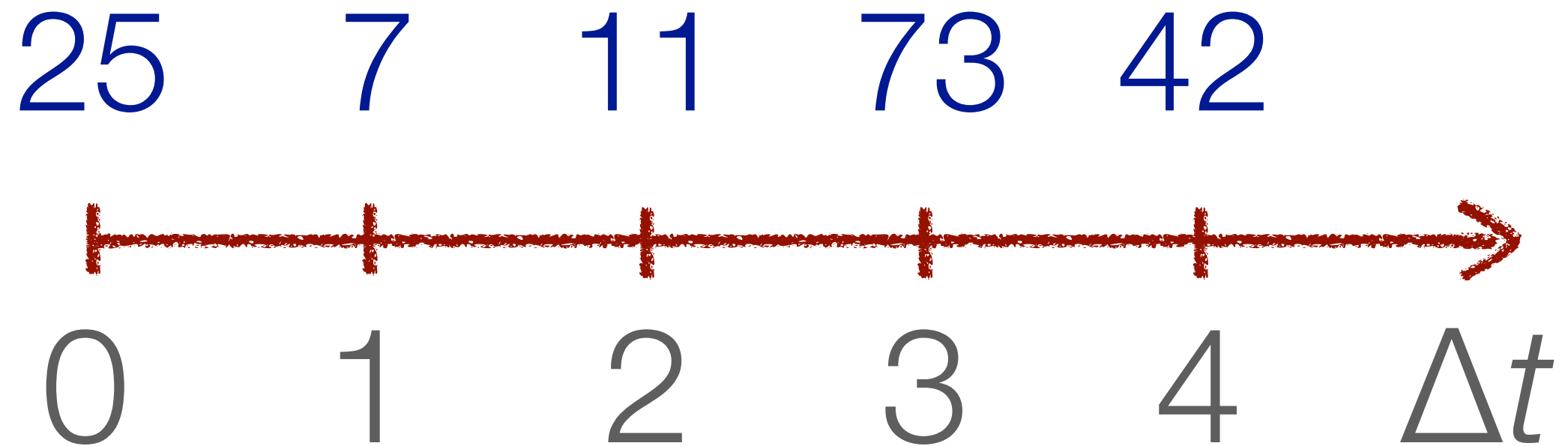
Mutable Variable

25 7 11 73 42





✓ Flow of changes explicit in code



- ✓ Flow of changes explicit in code
- ✓ Changes can be processed with combinators (map, fold, accumulate, and so on)

Changes API

```
protocol Observable: class {  
  associatedtype ObservedValue
```


Changes API

```
protocol Observable: class {
  associatedtype ObservedValue

  func observe<Context: AnyObject>(
    withContext context: Context,
    observer: (Context, ObservedValue) -> ()
  ) -> ()
}
```

registered observer
call back

Kolin Edition

```
interface Observable<ObservedValue> {  
    fun <Context : Any> observe(  
        context: Context,  
        observer: (Context, ObservedValue) -> Unit)  
}
```

Changes API

```
protocol Observable: class {
  associatedtype ObservedValue

  func observe<Context: AnyObject>(
    withContext context: Context,
    observer: (Context, ObservedValue) -> ()
  ) -> ()
}
```

registered observer
call back

```
protocol Observable: class {
  associatedtype ObservedValue

  func observe<Context: AnyObject>(
    withContext context: Context,
    observer: (Context, ObservedValue) -> ())
    -> ()
}
```

```
protocol Observable: class {
  associatedtype ObservedValue

  func observe<Context: AnyObject>(
    withContext context: Context,
    observer: (Context, ObservedValue) -> ())
    -> ()
}
```

Ephemeral stream of changes

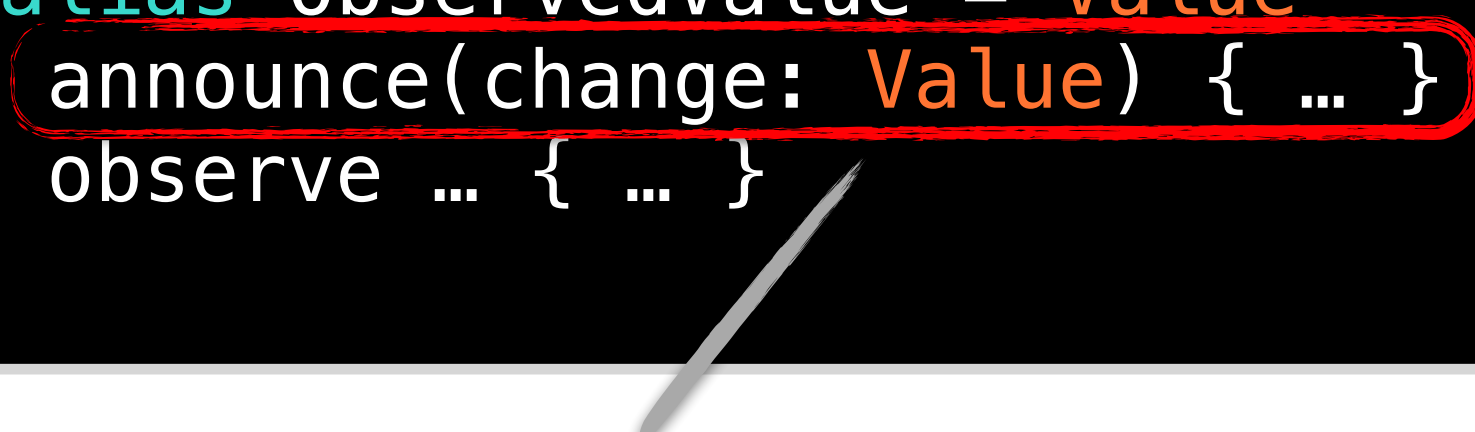
```
class Changing<Value>: Observable {
  typealias ObservedValue = Value
  func announce(change: Value) { ... }
  func observe ... { ... }
}
```

```
protocol Observable: class {
  associatedtype ObservedValue

  func observe<Context: AnyObject>(
    withContext context: Context,
    observer: (Context, ObservedValue) -> ())
    -> ()
}
```

Ephemeral stream of changes

```
class Changing<Value>: Observable {
  typealias ObservedValue = Value
  func announce(change: Value) { ... }
  func observe ... { ... }
}
```



Add another change value into the stream

```
interface Observable<ObservedValue> {  
    fun <Context : Any> observe(  
        context: Context,  
        observer: (Context, ObservedValue) -> Unit)  
}
```

```
class Changing<ObservedValue>  
    : Observable<ObservedValue> {  
    fun announce(change: ObservedValue) { ... }  
    fun observe ... { ... }  
}
```

```
protocol Observable: class {
  associatedtype ObservedValue

  func observe<Context: AnyObject>(
    withContext context: Context,
    observer: (Context, ObservedValue) -> ())
    -> ()
}
```

Ephemeral stream of changes

```
class Changing<Value>: Observable {
  typealias ObservedValue = Value
  func announce(change: Value) { ... }
  func observe ... { ... }
}
```

Add another change value into the stream


```

protocol Observable: class {
  associatedtype ObservedValue

  func observe<Context: AnyObject>(
    withContext context: Context,
    observer: (Context, ObservedValue) -> ())
    -> ()
}

```

Accumulating value

```

class Accumulating<Value, Accumulator>: Observable {
  typealias ObservedValue = Accumulator
  init <Observed: Observable where ...> (
    observing observed: Observed,
    startingFrom initial: Accumulator,
    accumulate: (Value, Accumulator) -> Accumulator)
  func observe ... { ... } }

```

```

protocol Observable: class {
  associatedtype ObservedValue

  func observe<Context: AnyObject>(
    withContext context: Context,
    observer: (Context, ObservedValue) -> ())
    -> ()
}

```

Accumulating value

```

class Accumulating<Value, Accumulator>: Observable {
  typealias ObservedValue = Accumulator
  init <Observed: Observable where ...> (
    observing observed: Observed,
    startingFrom initial: Accumulator,
    accumulate: (Value, Accumulator) -> Accumulator)
  func observe ... { ... } }

```

Accumulating change

```
interface Observable<ObservedValue> {  
    fun <Context : Any> observe(  
        context: Context,  
        observer: (Context, ObservedValue) -> Unit)  
}
```

```
class Accumulating<Value, Acc>(  
    val observed: Observed,  
    val initial: Acc,  
    val accumulate: (Value, Acc) -> Acc)  
    : Observable<Acc> {  
    fun observe ... { ... }  
}
```

extension Observable {

}

`extension Observable {`

`map` apply function to each observed value

`}`

`extension Observable {`

`map` apply function to each observed value

`merge` combine two streams of observations

`}`

extension Observable {

map apply function to each observed value

merge combine two streams of observations

accumulate

combine the values of a stream of observations using an accumulation function

...

}

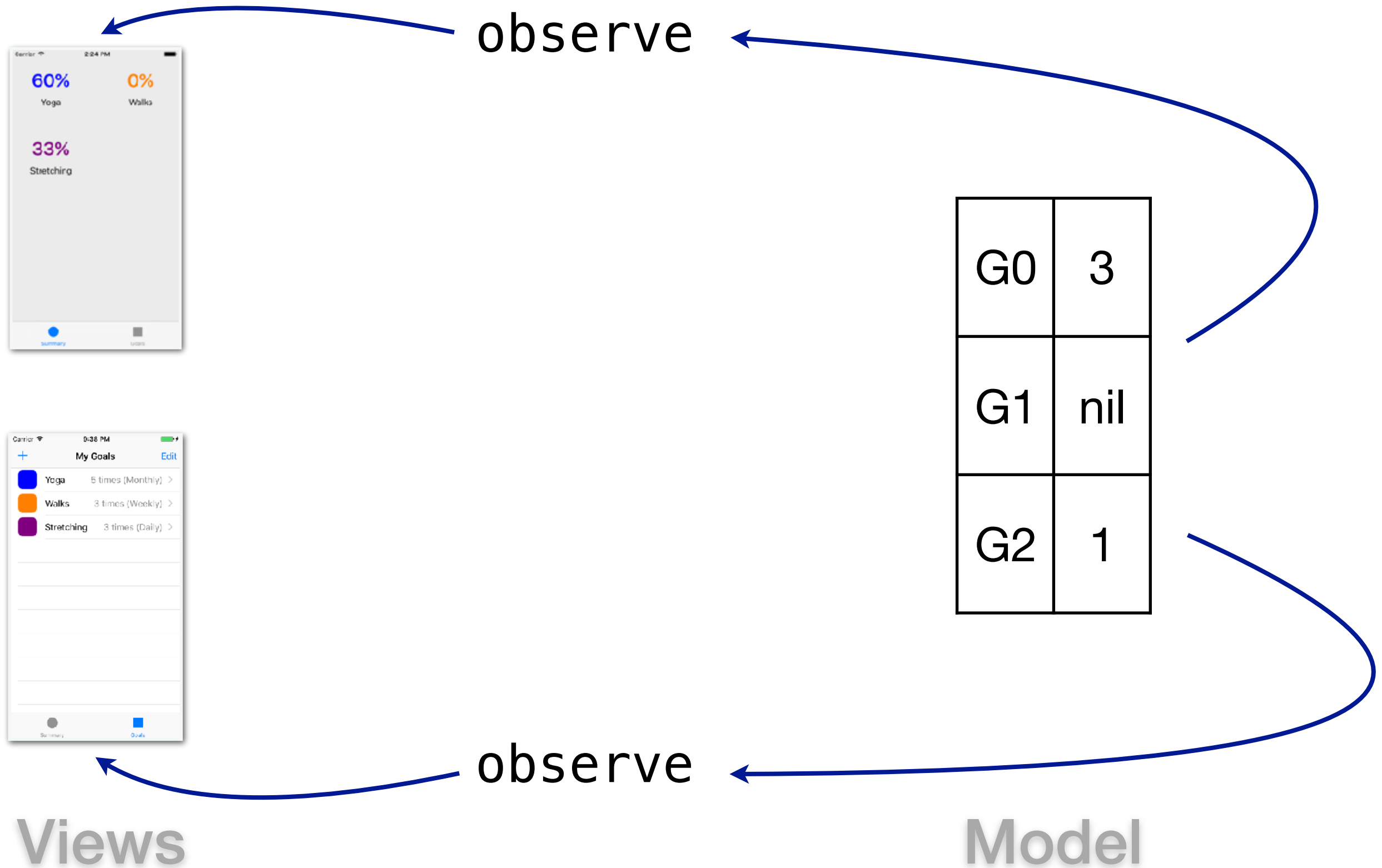
Change Propagation in Goals

G0	3
G1	nil
G2	1

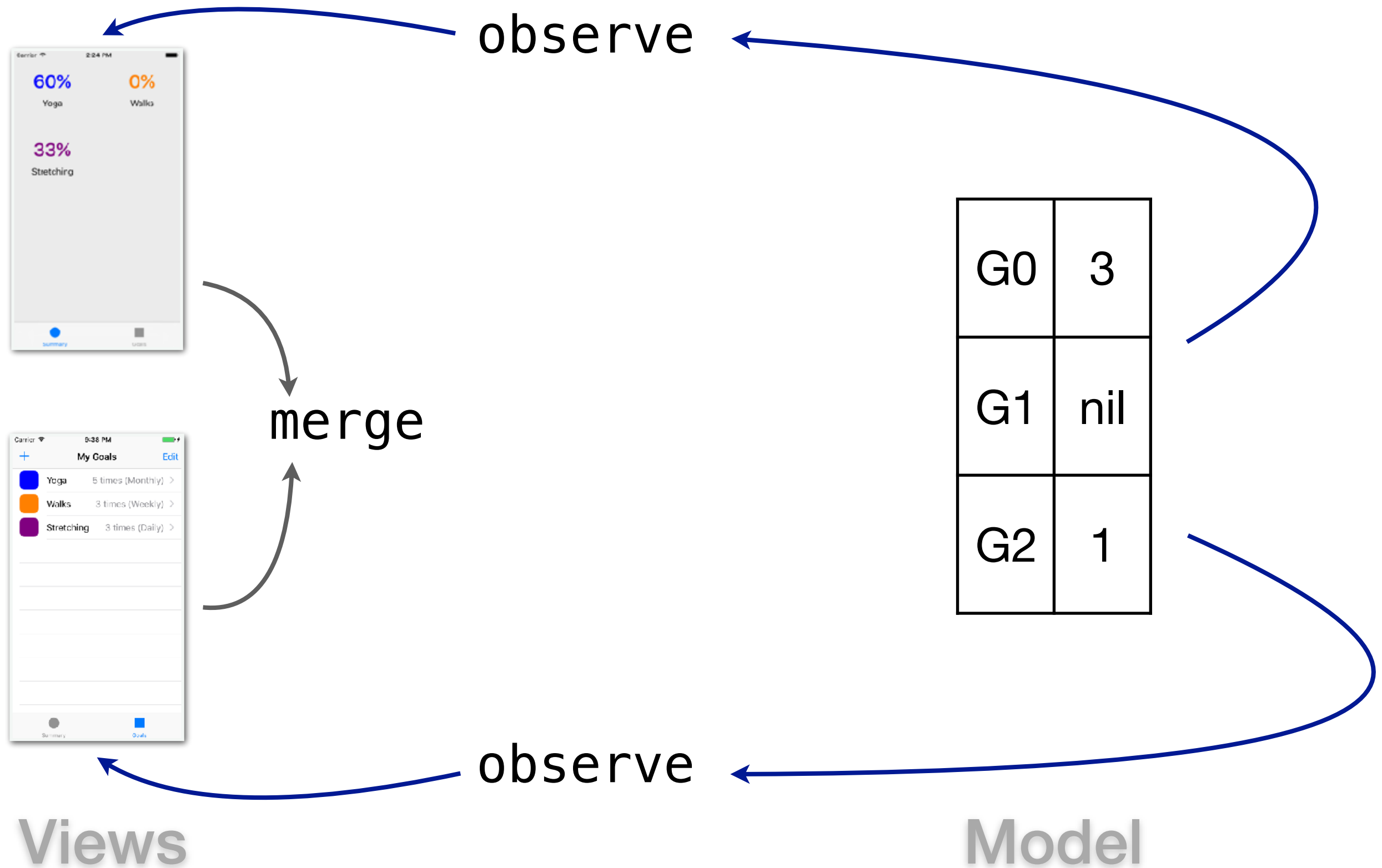
Views

Model

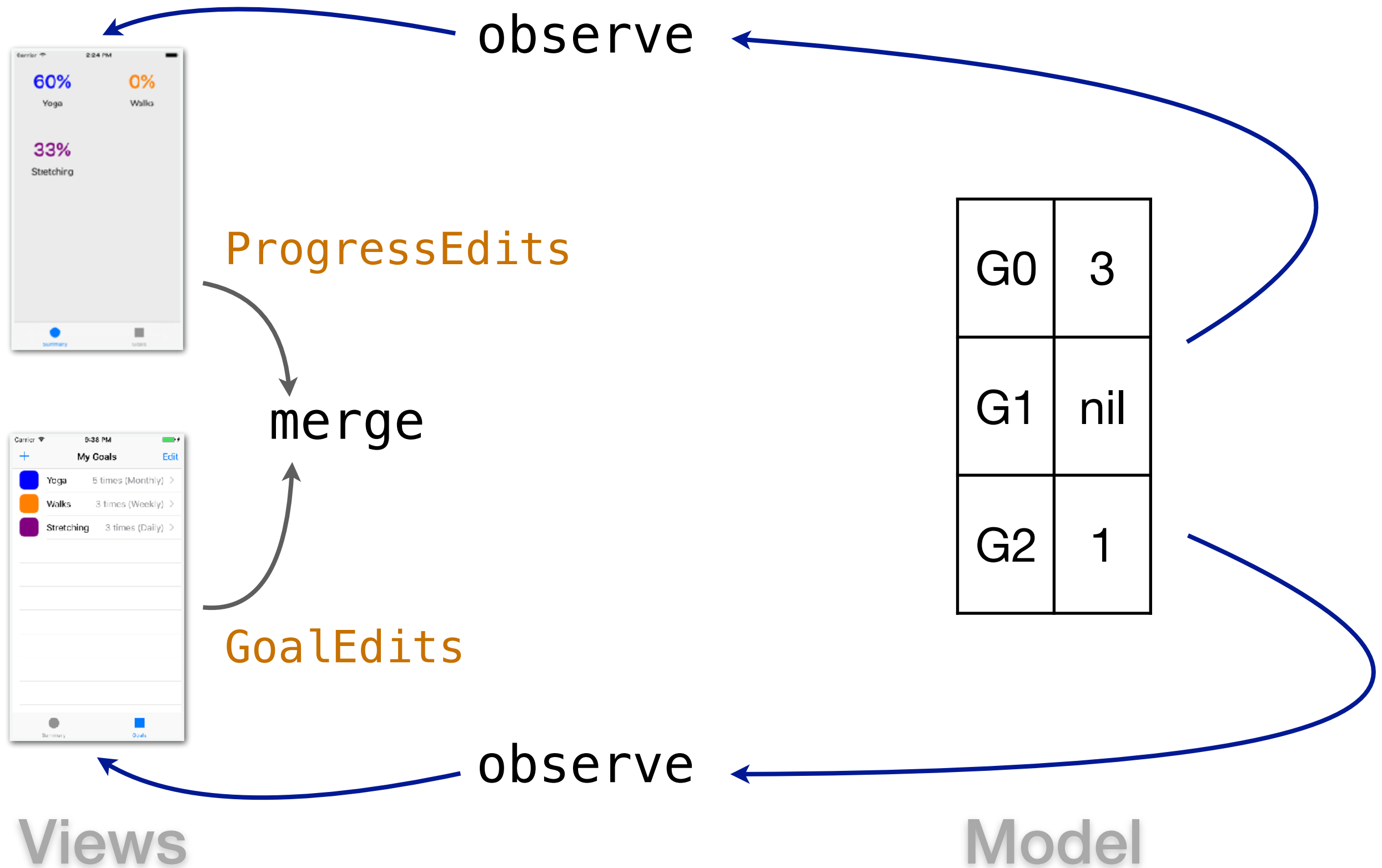
Change Propagation in Goals



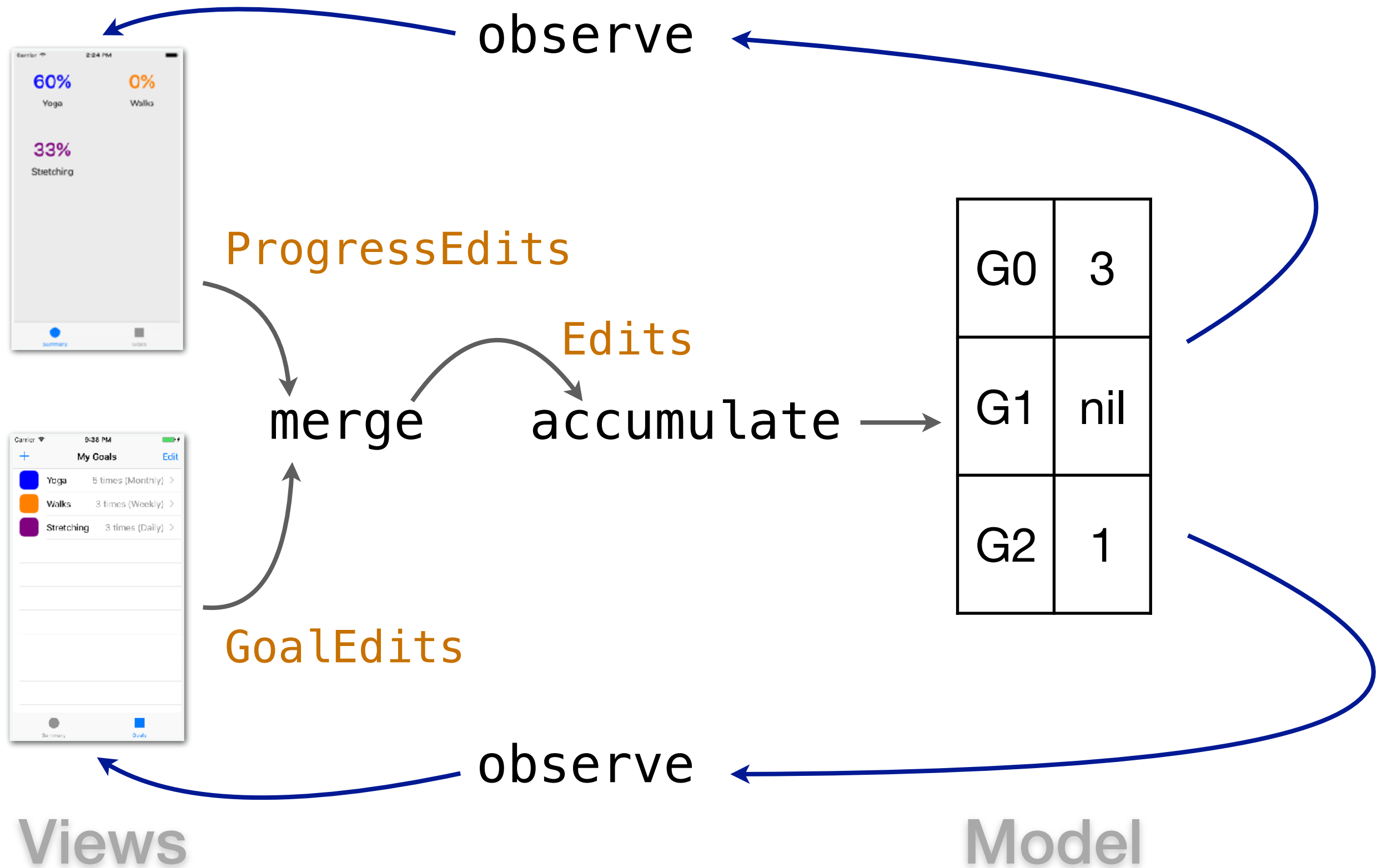
Change Propagation in Goals



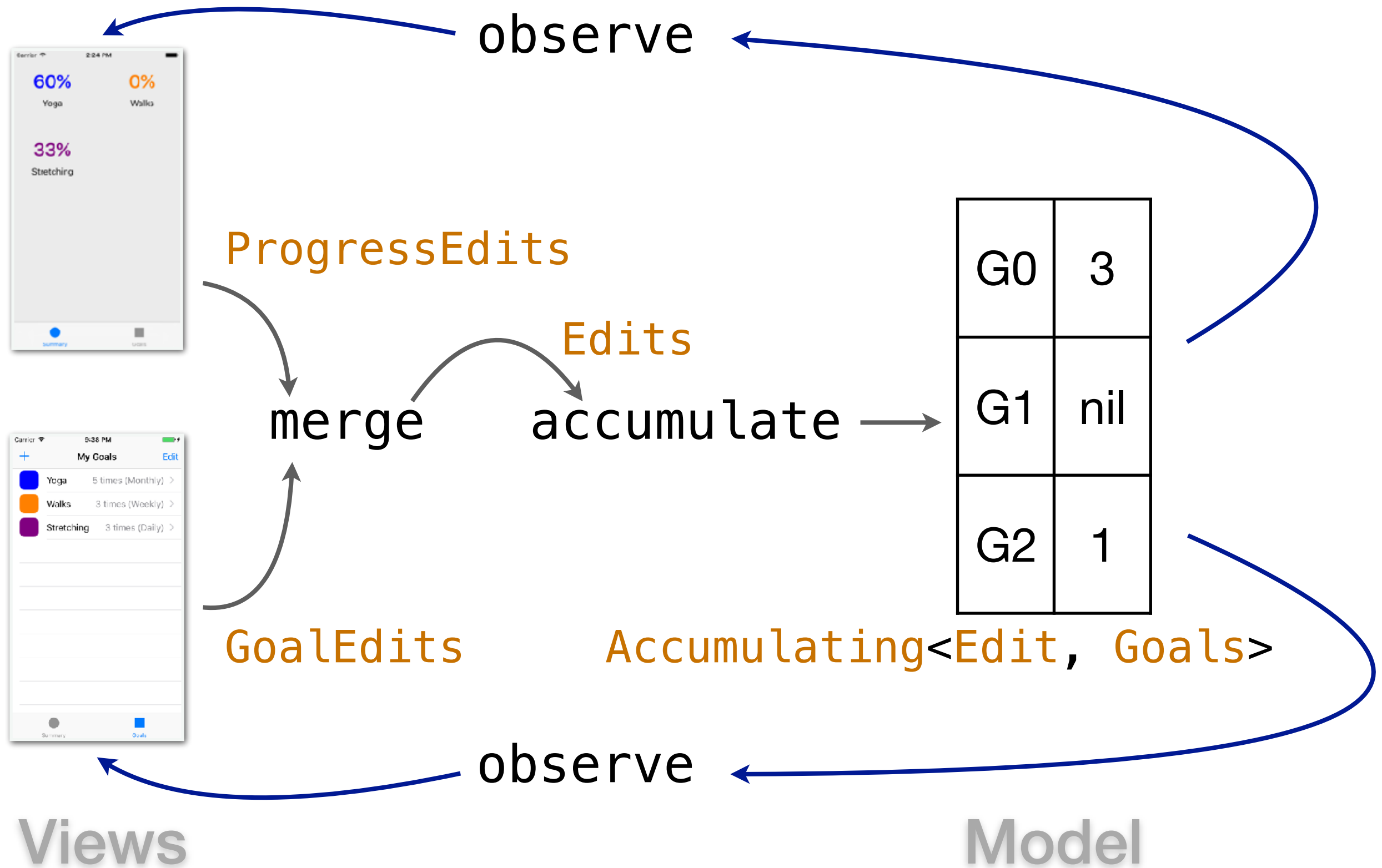
Change Propagation in Goals



Change Propagation in Goals



Change Propagation in Goals



GoalEdits

ProgressEdits

Edits

Accumulating<Edit, Goals>

```
typealias GoalEdits = Changing<GoalEdit>
```

ProgressEdits

Edits

Accumulating<Edit, Goals>

```
typealias GoalEdits = Changing<GoalEdit>
```

ProgressEdits

Edits

Accumulating<Edit, Goals>


```
typealias GoalEdits = Changing<GoalEdit>

enum GoalEdit {
  case add(goal: Goal)
  case delete(goal: Goal)
  case update(goal: Goal)
  case setActivity(activity: [Bool])
}
```

ProgressEdits

Edits

Accumulating<Edit, Goals>

```
typealias GoalEdits = Changing<GoalEdit>

enum GoalEdit {
    case add(goal: Goal)
    case delete(goal: Goal)
    case update(goal: Goal)
    case setActivity(activity: [Bool])
}

extension GoalEdit {
    func transform(_ goals: Goals) -> Goals {
        switch self {
            case .add(let newGoal):
                ...
        }
    }
}
```

ProgressEdits

Edits

Accumulating<Edit, Goals>

Kolin Edition

```
typealias GoalEdits = Changing<GoalEdit>

sealed class GoalEdit {
    data class Add(val goal: Goal) : GoalEdit()
    data class Delete(val goal: Goal) : GoalEdit()
    data class Update(val goal: Goal) : GoalEdit()
    data class SetActivity(val activity: Array<Boolean>)
        : GoalEdit()
}

fun GoalEdit.transform(goals: Goals): Goals {
    when(this) {
        is GoalEdit.Add -> { ... }
        ...
    }
}
```

```
typealias GoalEdits = Changing<GoalEdit>

enum GoalEdit {
    case add(goal: Goal)
    case delete(goal: Goal)
    case update(goal: Goal)
    case setActivity(activity: [Bool])
}

extension GoalEdit {
    func transform(_ goals: Goals) -> Goals {
        switch self {
            case .add(let newGoal):
                ...
        }
    }
}
```

ProgressEdits

Edits

Accumulating<Edit, Goals>

GoalEdits

```
typealias ProgressEdits = Changing<ProgressEdit>
```

Edits

```
Accumulating<Edit, Goals>
```

GoalEdits

```
typealias ProgressEdits = Changing<ProgressEdit>
```

Edits

```
Accumulating<Edit, Goals>
```

GoalEdits

```
typealias ProgressEdits = Changing<ProgressEdit>
enum ProgressEdit {
    case bump(goal: Goal)
}
```

Edits

Accumulating<Edit, Goals>

GoalEdits

```
typealias ProgressEdits = Changing<ProgressEdit>

enum ProgressEdit {
    case bump(goal: Goal)
}

extension ProgressEdit {
    func transform(_ goals: Goals) -> Goals { ... }
}
```

Edits

Accumulating<Edit, Goals>

GoalEdits

```
typealias ProgressEdits = Changing<ProgressEdit>

enum ProgressEdit {
    case bump(goal: Goal)
}

extension ProgressEdit {
    func transform(_ goals: Goals) -> Goals { ... }
}
```

```
typealias Edits = Changing<Edit>
```

Accumulating<Edit, Goals>

GoalEdits

```
typealias ProgressEdits = Changing<ProgressEdit>

enum ProgressEdit {
    case bump(goal: Goal)
}

extension ProgressEdit {
    func transform(_ goals: Goals) -> Goals { ... }
}
```

```
typealias Edits = Changing<Edit>

enum Edit {
    case goalEdit(edit: GoalEdit)
    case progressEdit(edit: ProgressEdit)
}
```

Accumulating<Edit, Goals>

GoalEdits

ProgressEdits

Edits

```
let model: Accumulating<Edit, Goals>
```

GoalEdits

ProgressEdits

Edits

```
let model: Accumulating<Edit, Goals>
```

GoalEdits

ProgressEdits

Edits

```
let model: Accumulating<Edit, Goals>  
  = edits.accumulate(startingFrom: initialGoals) {  
    edit, currentGoals in  
      return edit.transform(currentGoals)  
  }
```

Kolin Edition

```
val model: Accumulating<Edit, Goals>  
  = edits.accumulate(initial = initialGoals)  
    { edit, currentGoals ->  
      return edit.transform(currentGoals)  
    }
```

Replacing Tests

Replacing Tests

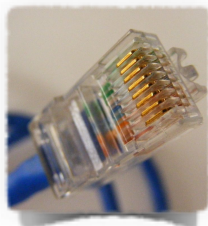
- ✓ Avoid changes from unexpected places
Replaces integration tests

Replacing Tests


- ✓ Avoid changes from unexpected places
Replaces integration tests
- ✓ Avoid conflicting state changes
Replaces integration tests

Check out the source code of Goals.app!

<https://github.com/mchakravarty/goalsapp>



 haskellformac.com

 [mchakravarty](https://github.com/mchakravarty)

 [TacticalGrace](https://twitter.com/TacticalGrace)

 justtesting.org

 [types](#)  [state](#)

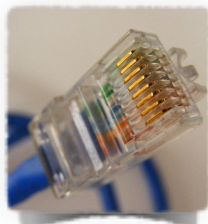
Check out the source code of Goals.app!

<https://github.com/mchakravarty/goalsapp>

Type systems are a design tool

Swift encourages typed functional programming

Types replace entire categories of tests



haskellformac.com



[mchakravarty](https://github.com/mchakravarty)



[TacticalGrace](https://twitter.com/TacticalGrace)



justtesting.org



types



state

Thank you!

Image Attribution

flickr

<https://www.flickr.com/photos/30998987@N03/5408763997>
<https://www.flickr.com/photos/provisions/7986149891/>
<https://www.flickr.com/photos/digitalcurrency/2438119267>
<https://www.flickr.com/photos/pviojoenchile/2760021279/>



Wikimedia

https://commons.wikimedia.org/wiki/File:Nile_blue_05.jpg
https://commons.wikimedia.org/wiki/File:Handcuffs01_2008-07-27.jpg
https://commons.wikimedia.org/wiki/File:Set_square_Geodreieck.svg

dribbble

<https://dribbble.com/shots/1667698-Free-vector-Macbook-Ipad-and-Iphone>

 **openclipart**

<https://openclipart.org/detail/463/heart>