RxSwift at momondo

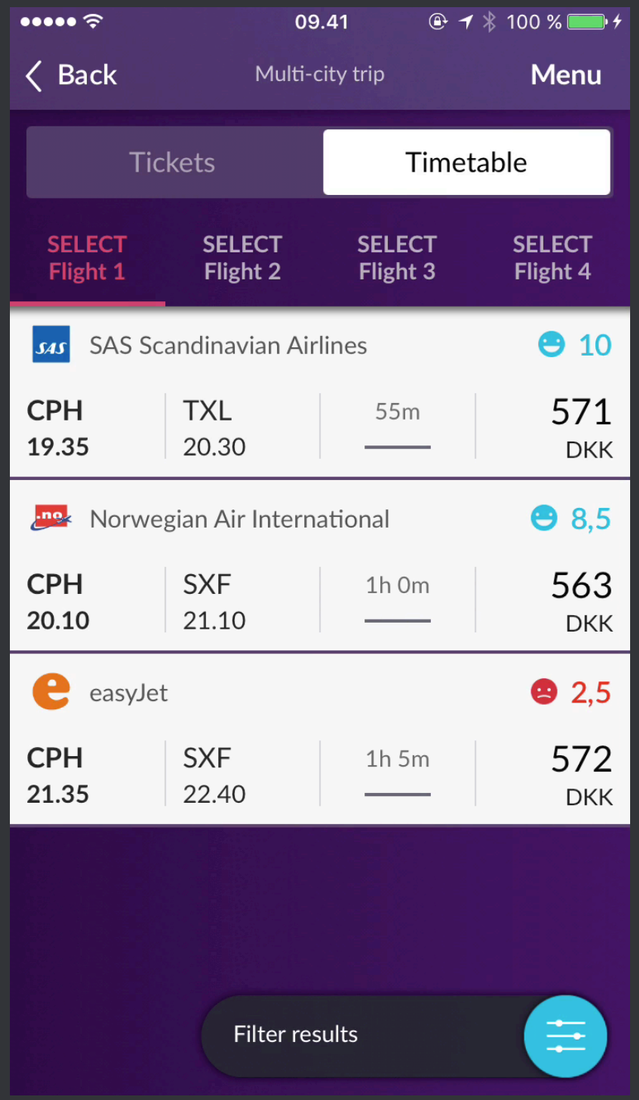
Copenhagen Cocoa Meetup
Dennis Torbichuk @dtrukr
28 February 2017

RxSwift at momondo

Focus areas for this presentation:

- Problem of "state" in typical applications
- Brief introduction into RxSwift and how it helps to minimize state
- Practical examples of using RxSwift at momondo

Timetable feature in momondo app



Search | Save as | Share | Export | Tools

text ~ "builder" AND issuetype = Bug AND project = MIS | Basic

Order by

- MIS-1473 Tablet: Ticket builder selection line is missing…
- MIS-240 Crah when closing segment details page in ti…
- MIS-236 Crash when selecting departure for one-way …
- MIS-1043 Reproducible crash in ticket builder
- MIS-1465 Crash when tapping second leg tab for onew…
- MIS-1474 Second leg result list in ticket builder is empty
- MIS-1469 Tablet: Return leg is incorrectly displayed in ti…
- MIS-1463 Ticket builder result list for second leg is not …
- MIS-1462 It is not possible to open best fit from ticket b…
- MIS-247 Filters are available in view for selected depa…

1 2

iOS Search / MIS-240

Crah when closing segment details page in ticket builder

2 of 61

61 bugs

Edit | Comment | Assign | More | Reopen | QA Review | Export

Details

Type:	Bug
Status:	DONE (View Workflow)
Priority:	Critical
Resolution:	Done
Affects Version/s:	None
Fix Version/s:	None
Component/s:	None
Labels:	flights
Environment:	Branch: feature/12_ticket_builder App version: 5.5.1 (1463) Device: iPhone 6+ OS version: iOS 8.3 Language: English US
Epic Link:	Ticket builder (Phone)
Sprint:	Sprint 21

People

Assignee: Unassigned

Assign to me

Reporter: Jakob Hansen

Votes: 0 Vote for this issue

Watchers: 1 Start watching this issue

Dates

Created: 20/Apr/15 16:23

Updated: 21/Apr/15 10:27

All systems have
essential **complexity**
State also adds
incidental **complexity**

See Moseley and Marks' paper, "Out of the Tar Pit"

state

Your stored values at any given time

State

State of a button:

- **`var visible`** → 2 states
- **`var enabled`** → 4 states
- **`var selected`** → 8 states!
- **`var highlighted`** → 16 states!!

State is

easy

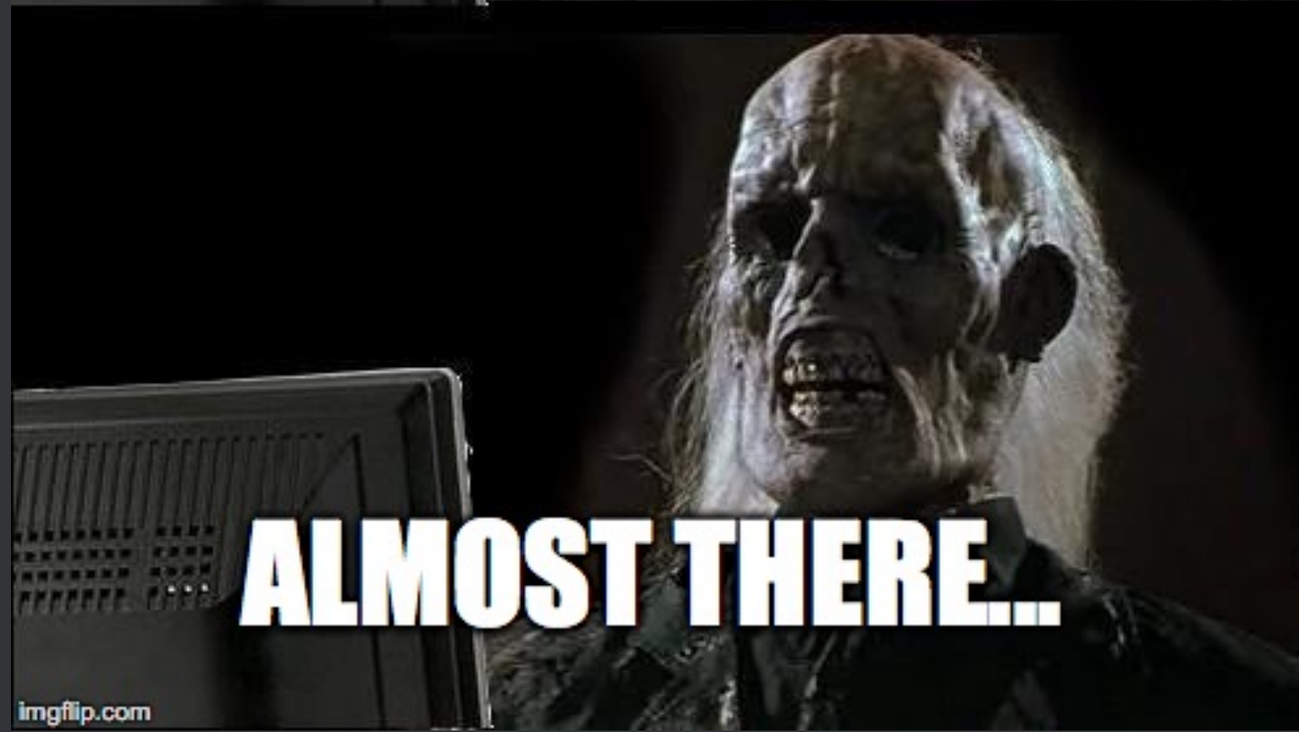
```
10 INPUT "What is your name: ", U$
20 PRINT "Hello "; U$
30 INPUT "How many stars do you want: ", N
40 S$ = ""
50 FOR I = 1 TO N
60 S$ = S$ + "*"
70 NEXT I
80 PRINT S$
90 INPUT "Do you want more stars? ", A$
100 IF LEN(A$) = 0 THEN GOTO 90
110 A$ = LEFT$(A$, 1)
120 IF A$ = "Y" OR A$ = "y" THEN GOTO 30
130 PRINT "Goodbye "; U$
140 END
```

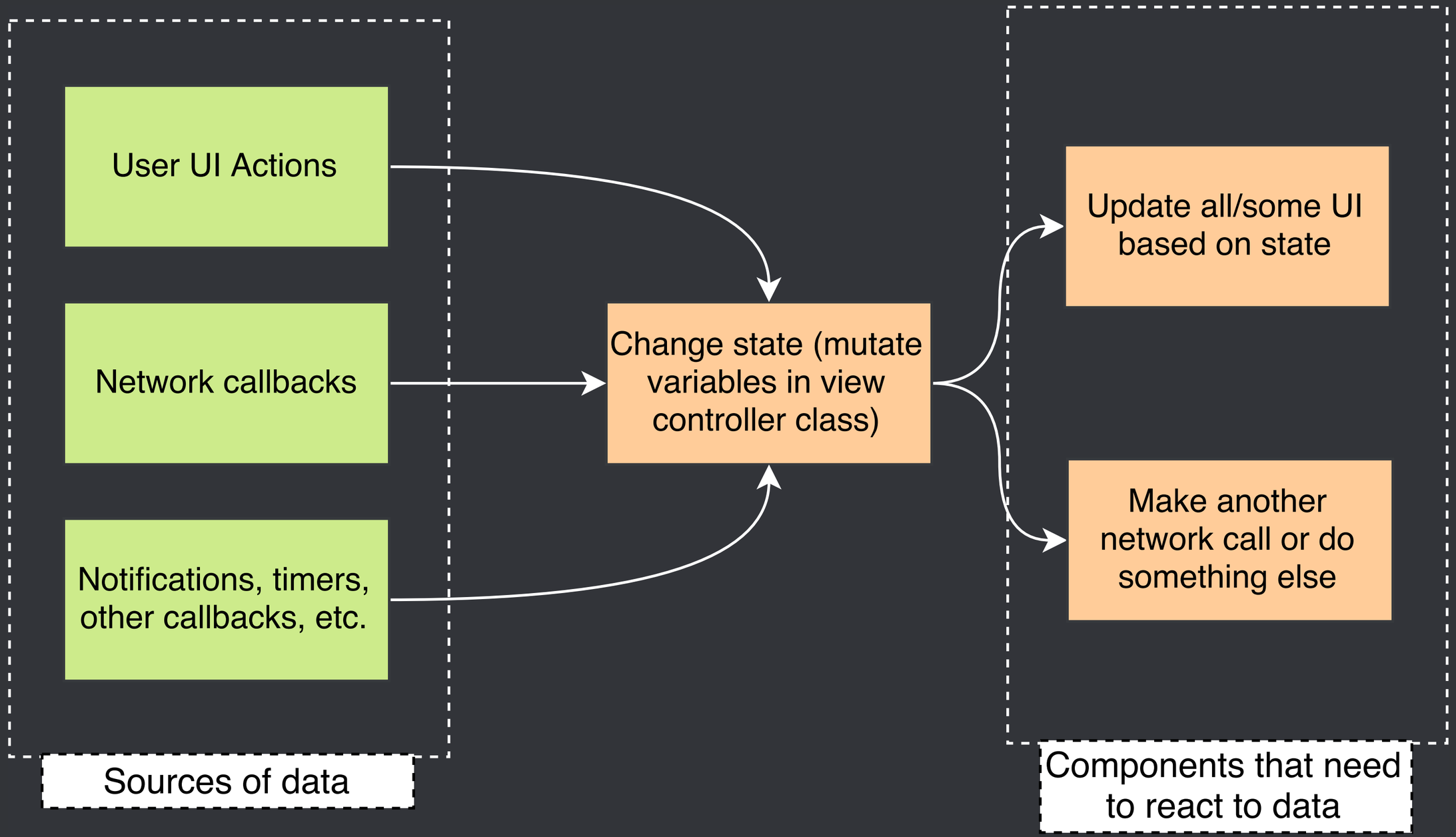
State is unpredictable

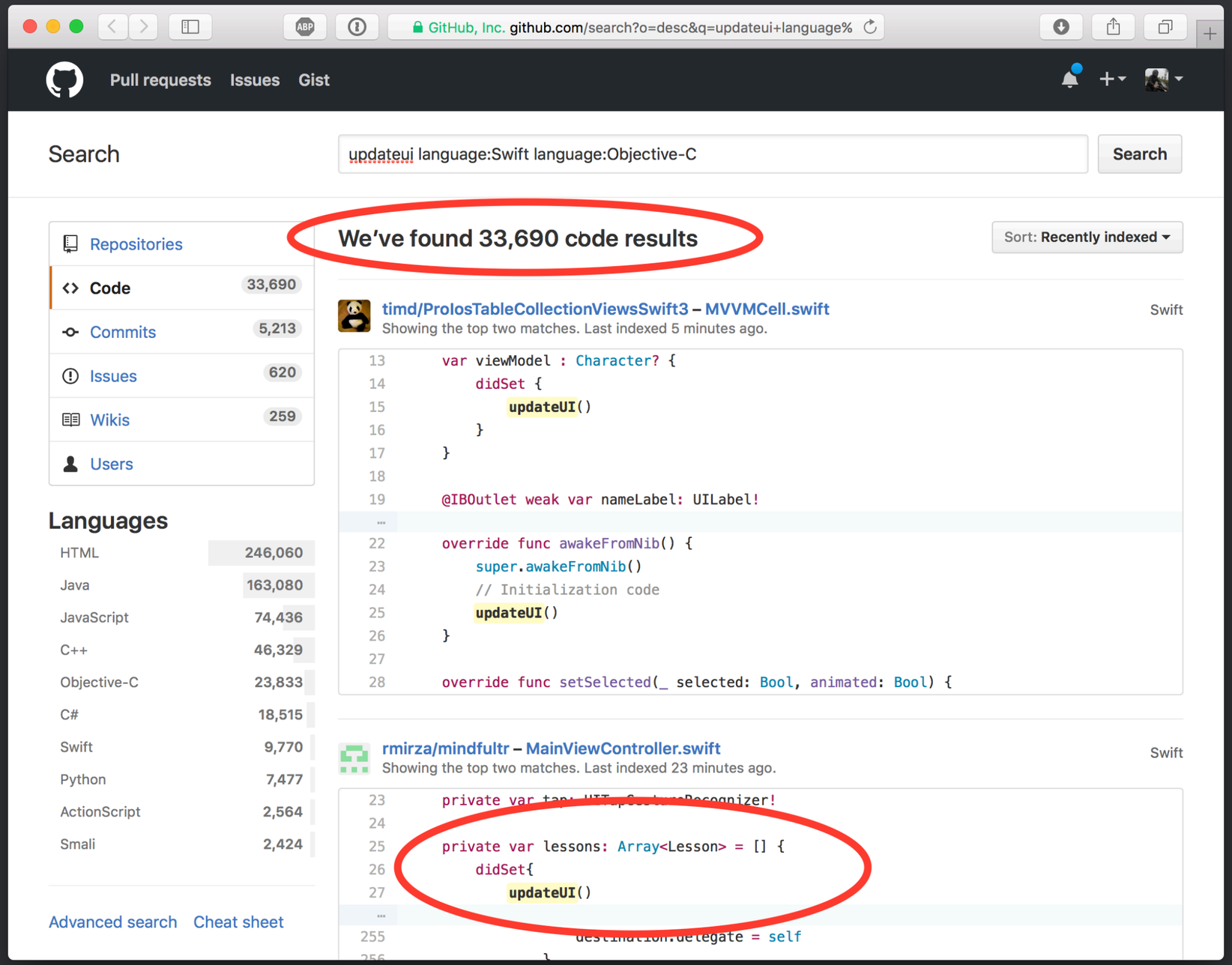
The challenge with state is that it actually needs to be handled which becomes a non-trivial problem

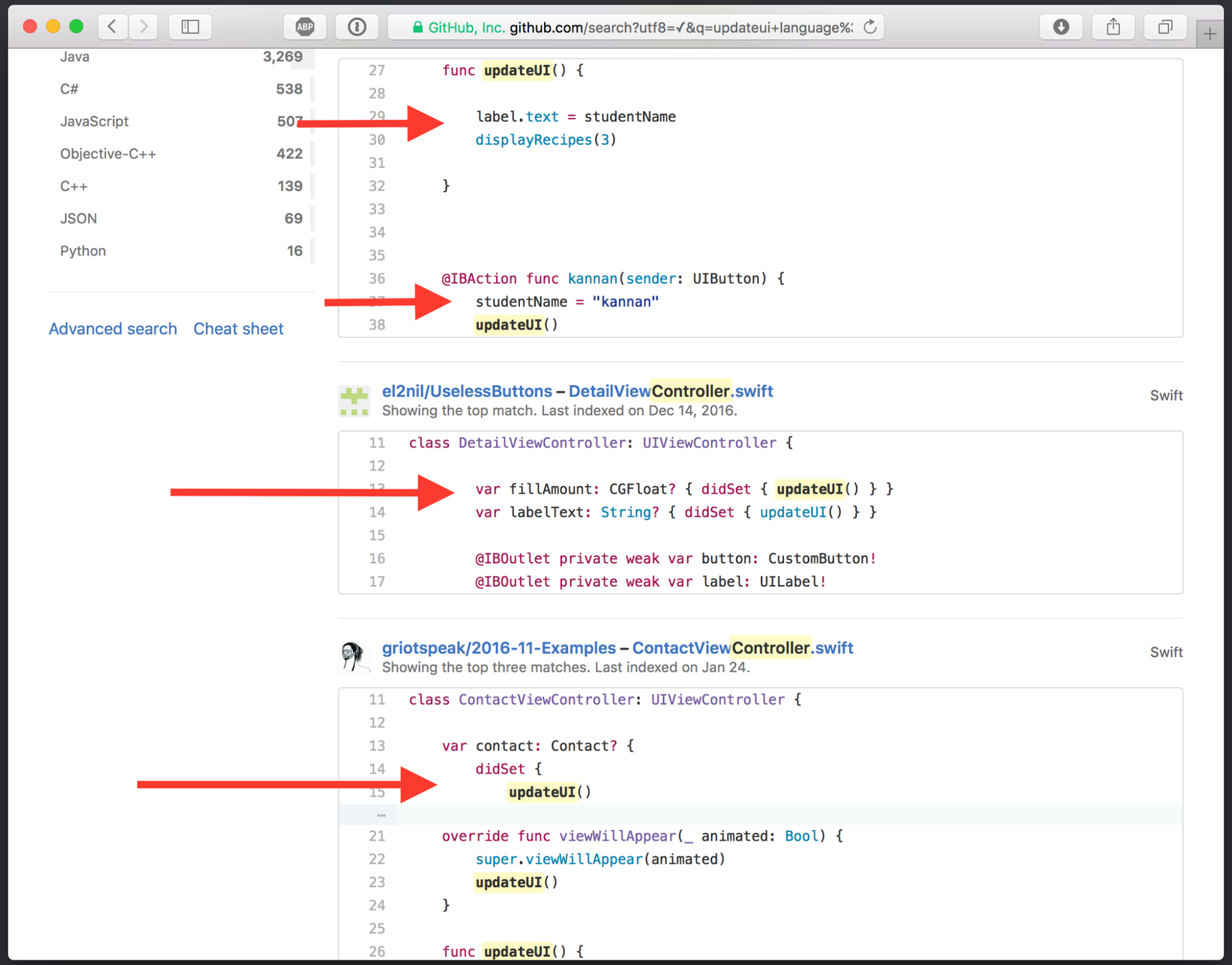
- `var visible` → 2 states
- `var enabled` → 4 states
- `var selected` → 8 states!
- `var highlighted` → 16 states!!

16 states need to be handled




User UI Actions
Network callbacks
Notifications, timers, other callbacks, etc.
Change state (mutate variables in view controller class)
Update all/some UI based on state
Make another network call or do something else
Sources of data
Components that need to react to data


GitHub, Inc. github.com/search?o=desc&q=updateui+language%
Pull requests
Issues
Gist
Search
updateui language:Swift language:Objective-C
Search
Repositories
Code 33,690
Commits 5,213
Issues 620
Wikis 259
Users
Languages
HTML 246,060
Java 163,080
JavaScript 74,436
C++ 46,329
Objective-C 23,833
C# 18,515
Swift 9,770
Python 7,477
ActionScript 2,564
Smali 2,424
Advanced search
Cheat sheet
We've found 33,690 code results
Sort: Recently indexed
timd/ProIosTableCollectionViewsSwift3 – MVVMCell.swift
Swift
Showing the top two matches. Last indexed 5 minutes ago.
13 var viewModel : Character? {
14 didSet {
15 updateUI()
16 }
17 }
18
19 @IBOutlet weak var nameLabel: UILabel!
22 override func awakeFromNib() {
23 super.awakeFromNib()
24 // Initialization code
25 updateUI()
26 }
27
28 override func setSelected(_ selected: Bool, animated: Bool) {
rmirza/mindfultr – MainViewController.swift
Swift
Showing the top two matches. Last indexed 23 minutes ago.
23 private var tap: UITapGestureRecognizer!
24
25 private var lessons: Array<Lesson> = [] {
26 didSet{
27 updateUI()
255 destination.delegate = self


GitHub, Inc. github.com/search?utf8=✓&q=updateui+language%3
Java 3,269
C# 538
JavaScript 507
Objective-C++ 422
C++ 139
JSON 69
Python 16
Advanced search
Cheat sheet
27 func updateUI() {
29 label.text = studentName
30 displayRecipes(3)
32 }
36 @IBAction func kannan(sender: UIButton) {
38 updateUI()
el2nil/UselessButtons – DetailViewController.swift
Showing the top match. Last indexed on Dec 14, 2016.
Swift
11 class DetailViewController: UIViewController {
14 var labelText: String? { didSet { updateUI() } }
16 @IBOutlet private weak var button: CustomButton!
17 @IBOutlet private weak var label: UILabel!
griotspeak/2016-11-Examples – ContactViewController.swift
Showing the top three matches. Last indexed on Jan 24.
Swift
11 class ContactViewController: UIViewController {
13 var contact: Contact? {
14 didSet {
15 updateUI()
21 override func viewWillAppear(_ animated: Bool) {
22 super.viewWillAppear(animated)
23 updateUI()
24 }
26 func updateUI() {

Timetable state

```
var segment:TicketBuilderSegment = .Departure ▶ 2 states
var direction:TicketBuilderSegment = .Departure ▶ 4 states
var departureSorting:FlightSorting = .DepartureSegmentDeparture {} ▶ 8 states
var returnSorting:FlightSorting = .ReturnSegmentDeparture {} ▶ 16 states
var state:TicketBuilderState ▶ 32 states
```

More state-like variables with side effects

```
private var selectedDeparturePrice:CurrencyValue?
private var selectedReturnPrice:CurrencyValue?
private var selectedPrice:CurrencyValue?
private var selectedDetailFlight:Flight?
private(set) var oneway = false
private var flights:[Flight] { return segment == .Departure ? departureFlights : returnFlights }
private var sorting:FlightSorting { return segment == .Departure ? departureSorting : returnSorting }
private var selectedDepartureFlight:Flight? {}
private var selectedReturnFlight:Flight? {}
override var result:FlightResult? {}
```


State! State! State! State!

48+ states! Handling:

```
if state == .FirstStep {
    ...
}

if state == .SecondStep {
    if indexPath.row == 0 {
        ...
    }
    if indexPath.row == 1 {
        if showAllFlights {
            ...
        } else {

            if selectedOutboundFlight != nil {

                if picker == .Departure {
                    ...
                } else {
                    ...
                }
                state = .FirstStep
```

```
            } else {

                if picker == .Return {
                    ...
                } else {
                    ...
                }
                state = .SecondStep
            }

            updateUI()
            delegateUpstream?.updatedFilter()
            hideHeaderSortingViewAndFilter()
            updateCheckmarkImage()
            tableView.reloadData()
        }
    } else {
        ...
    }
}
if state == .ThirdStep {
    if indexPath.row == 0 {
        ...
    }
    if indexPath.row == 1 {
        state = .FirstStep
        if picker == .Return {
            ...
        } else {
            ..
        }
        selectedOutboundFlight = nil
        selectedHomeboundFlight = nil
        updateUI()
    }
    if indexPath.row == 2 {
        ...
    }
}
```

Minimize state
Minimize complexity

Reactive programming principle

Any “getter” for mutable state causes problems. Instead of using getters, any calculated, generated, loaded or received state values should be immediately sent into a channel and any part of the program that wants access to these values must subscribe to the channel.
— Matt Gallagher

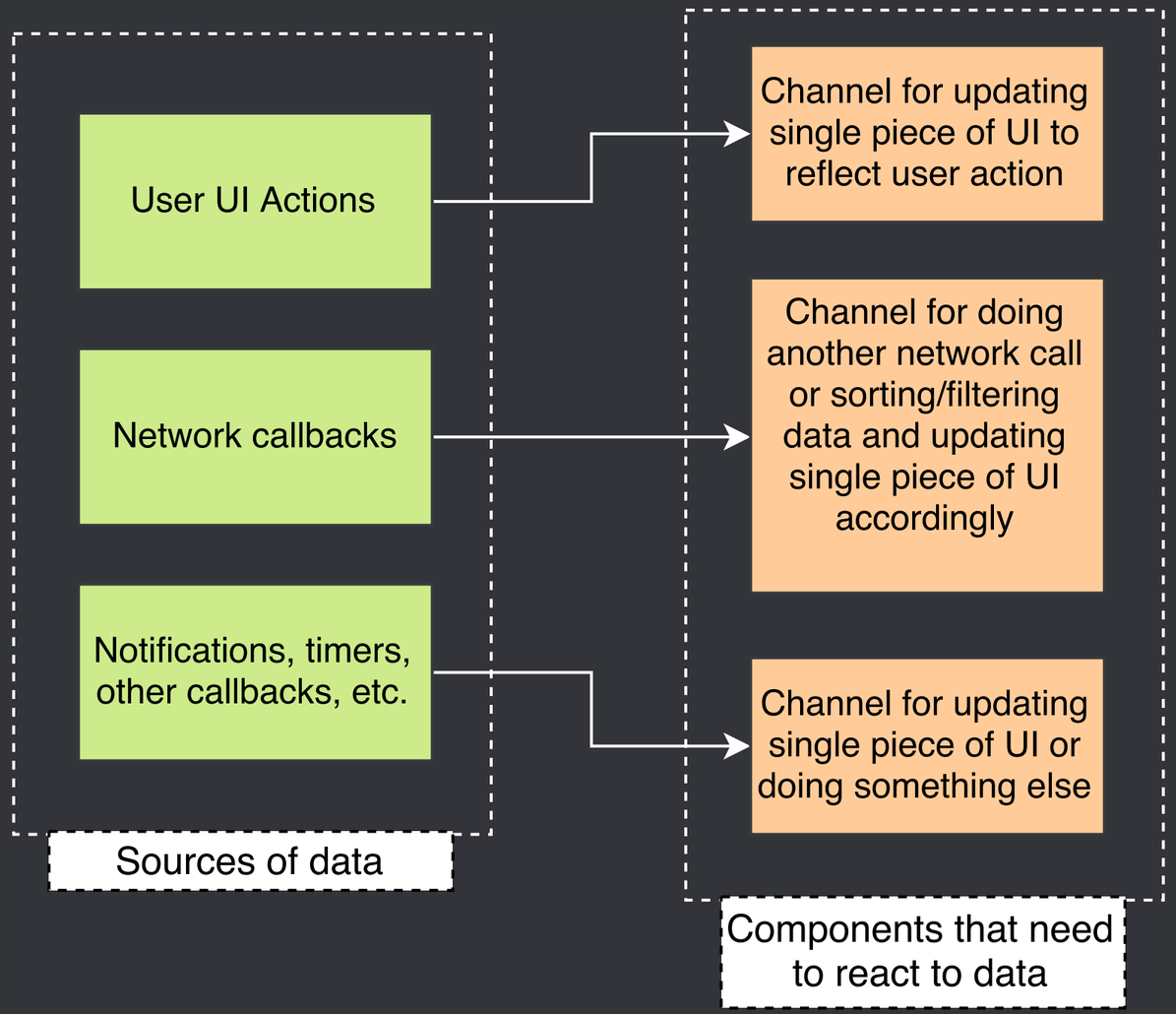
Reactive programming

manages asynchronous data flows

between sources of data

and components that need to react to that data.

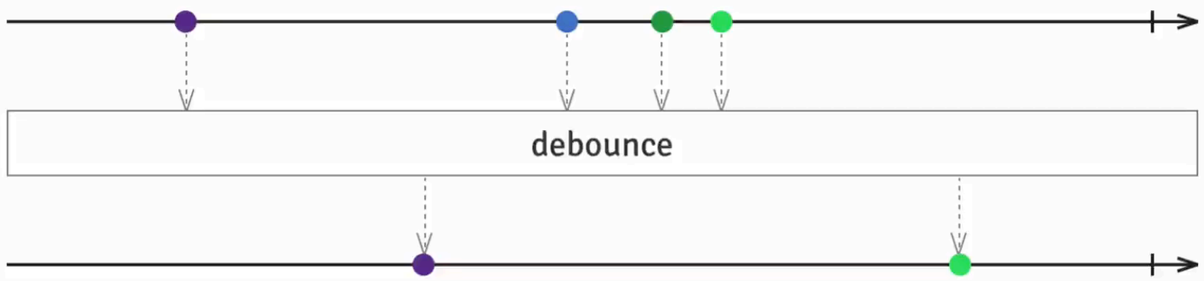
Minimising state and thus reducing complexity



RxSwift

The Observer pattern done right

ReactiveX is a combination of the best ideas from
the Observer pattern, the Iterator pattern, and functional programming



CREATE

Easily create event streams or data streams.

COMBINE

Compose and transform streams with query-like operators.

LISTEN

Subscribe to any observable stream to perform side effects.

Observables are a representation of

any collection of values over
any amount of time

- They can send anything, just like an array can hold anything.
- They’re very familiar, and very close to arrays in that they hold stuff.

Differences:

- You can always look into an array and get whatever it has.
- If you’re not listening to an observable, then you miss it.

Examples of observables

- Arrays of data
- Network I/O
- UI Events: button taps, any kind of UI interaction
- Notifications
- Delegate callbacks
- KVO, etc.


Everything is a stream

Observables in Rx

Observables are sequences in Rx and described by a push interface

```
enum Event<Element>  {
    case Next(Element)        // next element of a sequence
    case Error(ErrorType)     // sequence failed with error
    case Completed            // sequence terminated successfully
}

class Observable<Element> {
    func subscribe(observer: Observer<Element>) -> Disposable
}
```

Observables are lazy

There is one crucial thing to understand about observables.

- When an observable is created, it doesn't perform any work simply because it has been created.
- If you just call a method that returns an **`Observable`**, no sequence generation is performed, and there are no side effects. Sequence generation starts when **`subscribe`** method is called

E.g. Let's say you have a method with similar prototype:

```
func searchWikipedia(searchTerm: String) -> Observable<Results> {}

let searchForMe = searchWikipedia("me")

// no requests are performed, no work is being done, no URL requests were fired

let cancel = searchForMe
  // sequence generation starts now, URL requests are fired
  .subscribeNext { results in
      print(results)
  }
```

Transforming Observables

- buffer
- flatMap
- map
- window
- ...

Filtering Observables

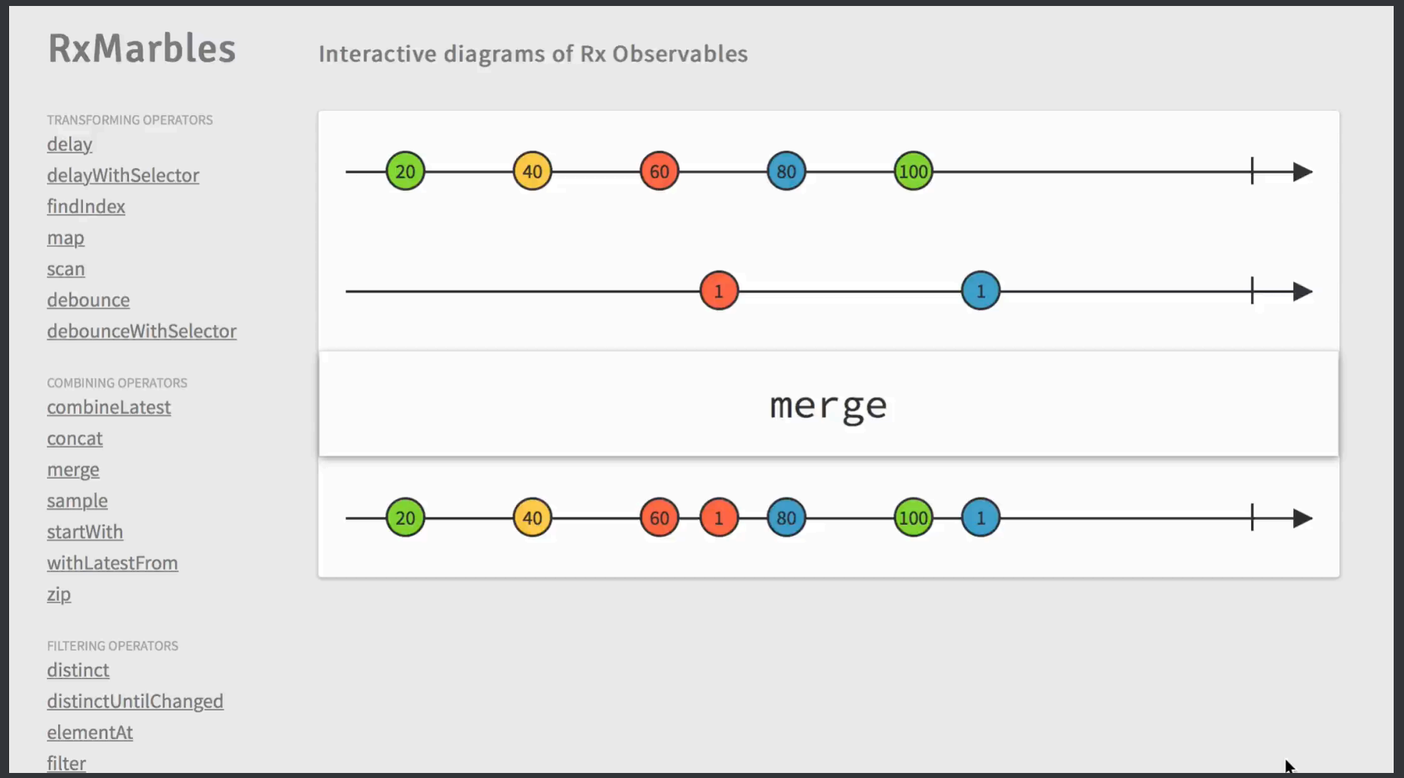
- debounce / throttle
- distinctUntilChanged
- elementAt
- filter
- skip
- ...

Combining Observables

- merge
- combineLatest
- zip
- ...

Learn operators

A website for experimenting with diagrams of Rx Observables, for learning purposes: http://rxmarbles.com

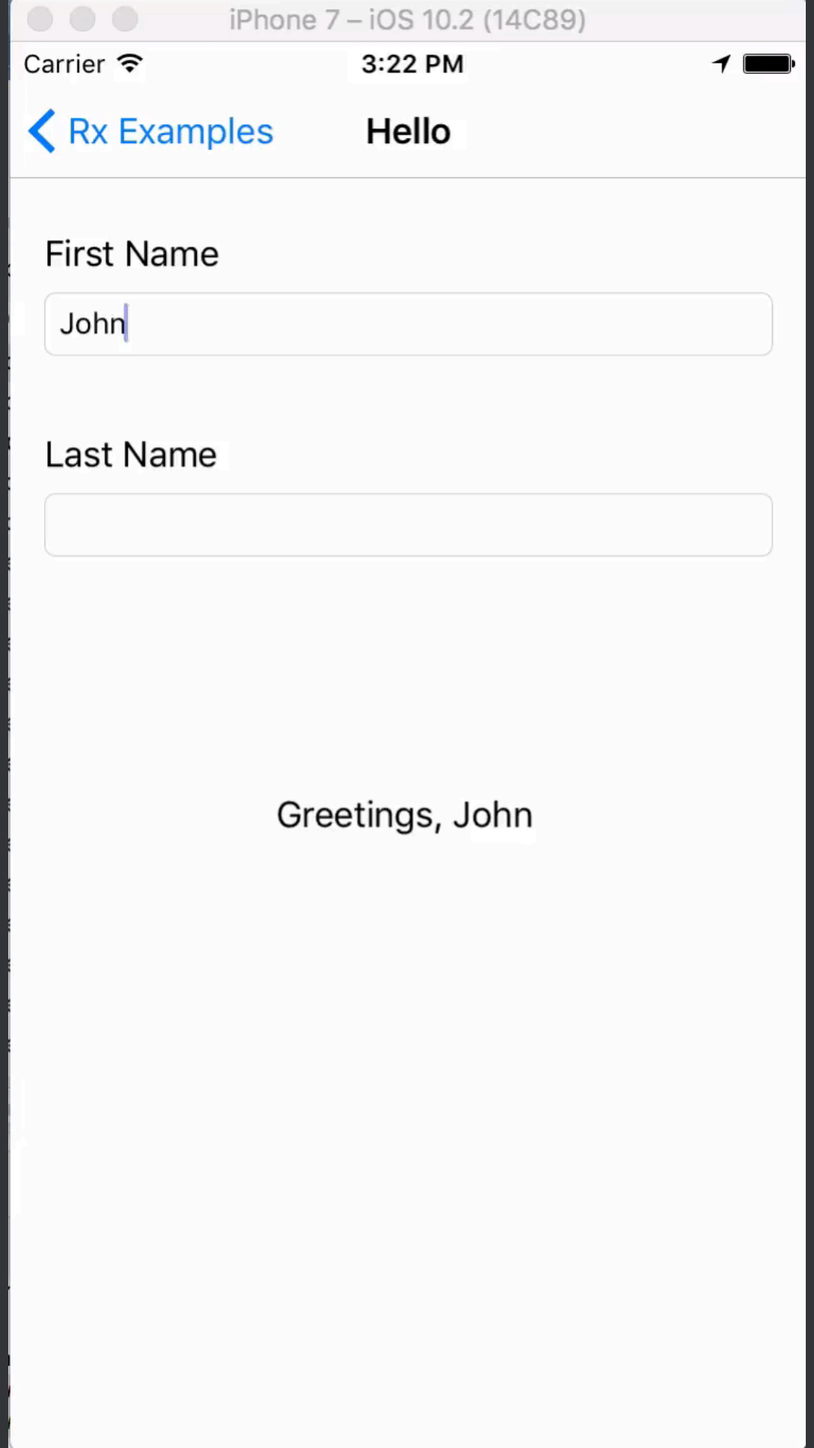


Bindings

```
Observable.combineLatest(firstName.rx.text, lastName.rx.text)
    { $0 + " " + $1 }
    .map { "Greetings, \($0)" }
    .bindTo(greetingLabel.rx.text)
```

This also works with **`UITableView`**s and **`UICollectionView`**s.

```
viewModel
    .rows
    .bindTo(tableView.rx.items(cellId: "Cell", cellType: Cell.self))
    { (_, viewModel, cell) in
        cell.title = viewModel.title
        cell.url = viewModel.url
    }
    .disposed(by: disposeBag)
```



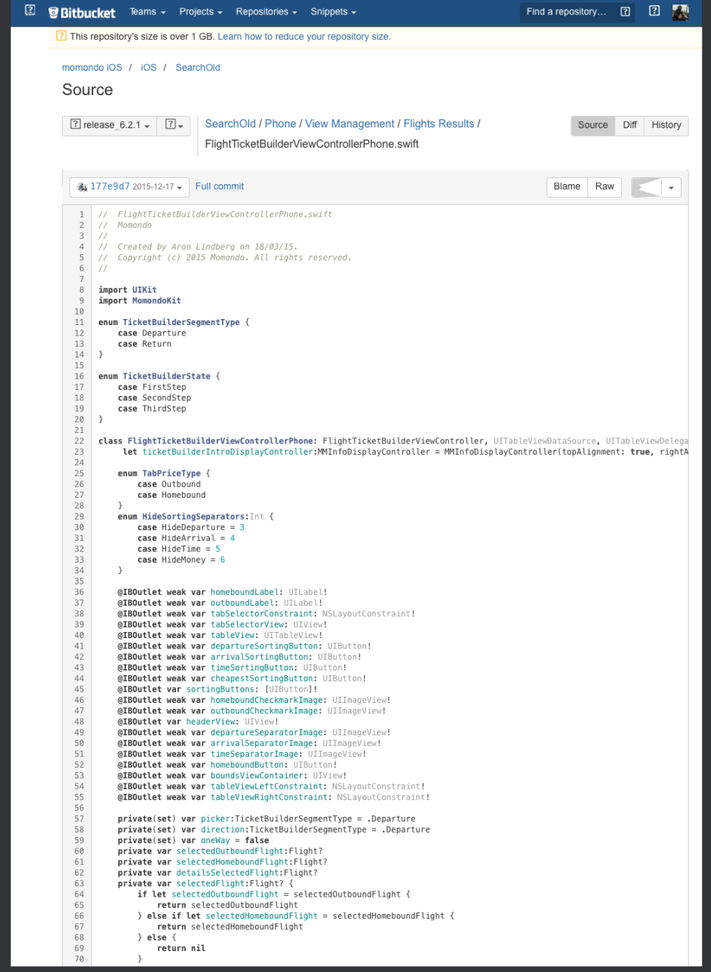
Before implementing RxSwift: 15 state-like variables in TimeTable VC and many lines of code to handle those states.

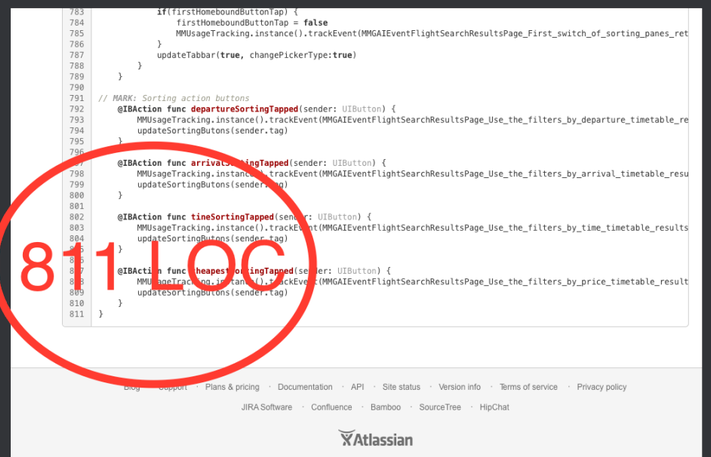
After implementing RxSwift: 3 Observables to handle all possible states

```
let flightsObservable = ReplaySubject<[[TimetableSegment]]>.create(bufferSize: 1)
let tabbarObservable = ReplaySubject<Int>(0).create(bufferSize: 1)
let selectedSegmentObservable = ReplaySubject<(Int, [TimetableSegment?])>.create(bufferSize: 1)
```

- flightsObservable: network request results, sorting and filtering results
- tabbarObservable: tabbar user actions
- selectedSegmentObservable: selecting a particular flight user action

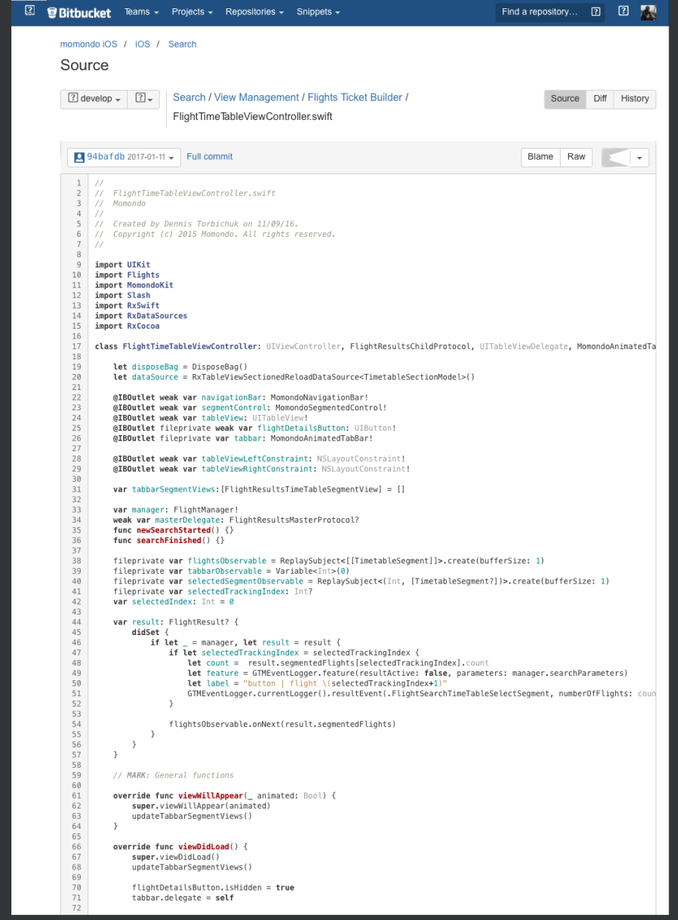
```
//This observable flow will fire up whenever there is a new FlightResult OR tabbar index changed
Observable.combineLatest(flightsObservable.asObservable(), tabbarObservable.asObservable()) {($0.0, $0.1)}
.map {(segmentedFlights, segmentIndex) -> [TimetableSectionModel] in

    let sections: [TimetableSectionModel] = [
    .flightSection(items: segmentedFlights[segmentIndex]
        .map({ (item) -> TimetableSectionItem in
            .flightItem(item)
    }))]

    return sections

}
    .asDriver(onErrorJustReturn: [])
    .drive(self.tableView.rx.items(dataSource: self.dataSource))

  //This observable flow will fire up whenever there is at least one segment selected
  selectedSegmentObservable.asObservable()
      .filter{$0.1.flatMap({$0}).count > 0}
      .subscribe(onNext: { [weak self] _ in
          self?.masterDelegate?.hideFilter()
      }).addDisposableTo(disposeBag)
```

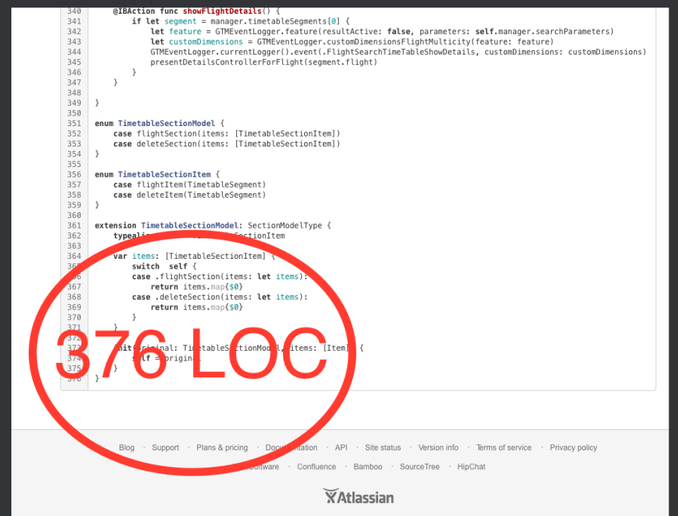




Timetable view controller before changes

- 811 lines of code
- 15 state-like variables with unclear side-effects
- Missing handling of some states
- 61 bugs reported by QA - most relate to poor state handling
- Confusing code, nobody has clear picture of what's going on
- Tight coupling between components in view controller
- Hard to extend functionality





Timetable view controller after changes

- 376 lines of code: 2.15x less code
- 0 state-like variables, instead clearly defined channels or flows
- Very few bugs reported by QA after extending functionality to support up-to 4 flights
- Concise code that clearly demonstrates relationship between sources of data and components that need to react to changes of this data
- No coupling between components in view controller
- Other features implemented using Rx from the beginning, like new destination

Before following reactive programming principles

- Brief discussion with peers on technical implementation
- Developer goes into crunch mode adding UI components one by one, adding data sources, state variables and connecting everything together

After we started following reactive programming principles

- Team discussion to identify all possible inputs and components that need to react to changes
- Identify and mockup channels or flows which consist of sources of data, observable chains and subscribers that react to changes.
- Discuss the flows with the team/QA to ensure everything is covered
- Implement flows

9:41 AM 100%

momondo

Welcome to Copenhagen

Every place has a mood ...

Rotate the wheel to find
new experiences

Pan Gesture TableV

Rotation Gesture

Tap Gesture

Tap Gesture

Tap Gesture

Tap Gesture

HAY HOUSE 25m

Pan Gesture Up/Dn

Tap Gesture

09.41 100 %

momondo

Welcome to Copenhagen

Every place has a mood ...

Rotate the wheel to find
new experiences ...

ALL MOODS

A. C. PERCH'S THEHANDEL 80m

9:41 AM 100%

Tap Gesture

Pan Gesture Up/Dn

Tap Gesture

Pan Gesture Up/Dn

Pan Gesture Horizo

HAY HOUSE 25m

Copenhagen's City Centre

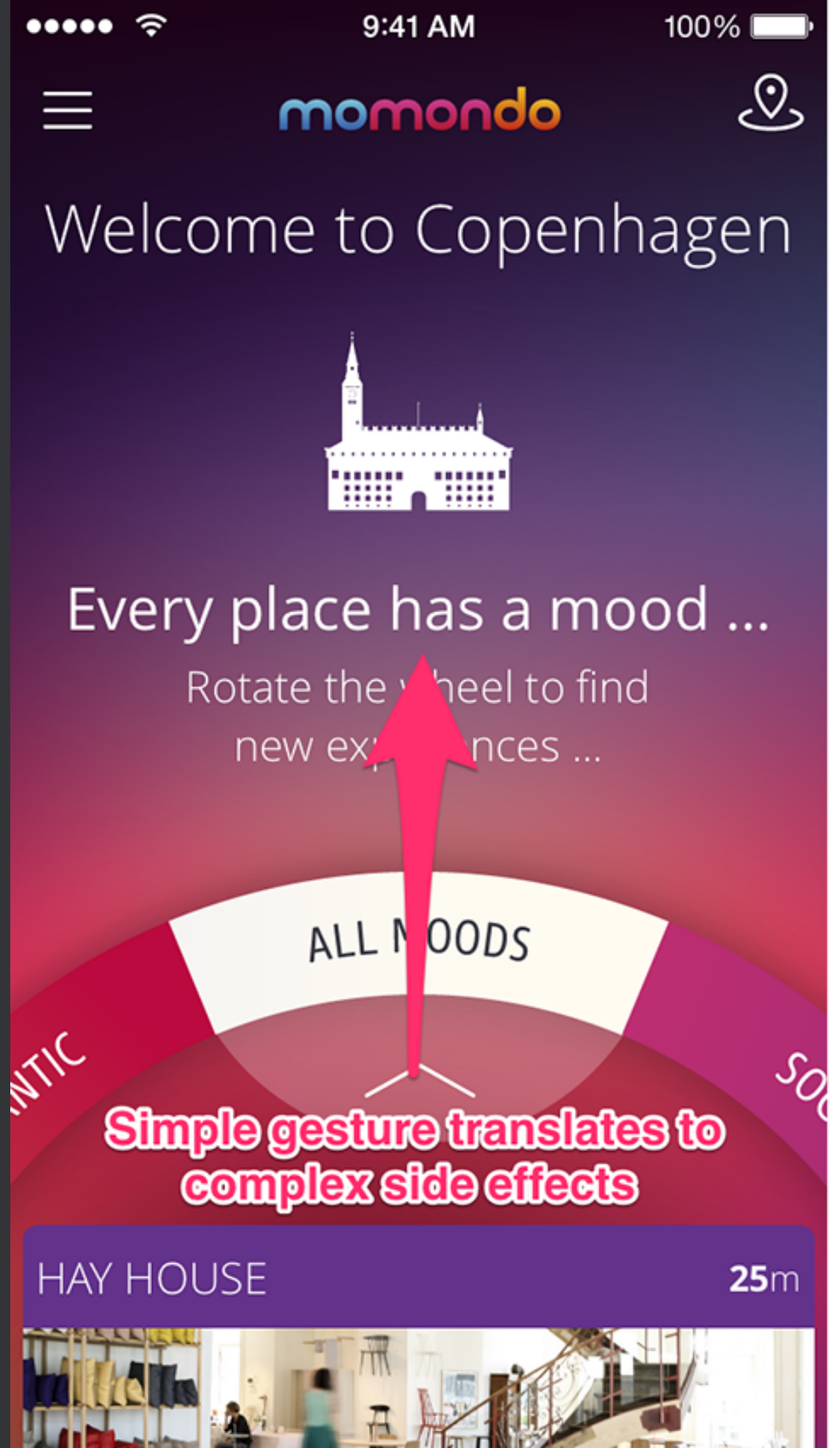
LA GALETTE 250m

Pan Gesture Up/Dn

Pan Gesture Horizo

Copenhagen's City Centre

UFORMEL 250m



What happens when you drag the wheel up?

1. Wheel moves up
2. Text on the wheel fades out
3. Wheel glow fades out
4. Wheel scales down
5. Wheel tab fades out
6. Introduction text moves up proportionally
7. Introduction text fades out
8. Onboarding arrow fades out
9. Logo fades out
10. Background image fades out
11. List of cards moves up
12. Specific gesture recognisers become enabled/disabled based on the wheel drag progress
13. Flights banner moves up (closes)

How all of these things placed on 4 different view controllers are kept in sync and move together?

Conclusion: Benefits of using Reactive Programming

In short, using Rx will make your code:

- Reusable <- Because it's composable
- Declarative <- Because definitions are immutable and only data changes
- Understandable and concise <- Raising the level of abstraction and removing transient states
- Less stateful <- Because of unidirectional data flows

Cons

- Steep learning curve
- Requires changing the way you think
- Requires mature development team
- Forces one to think more and code less
- Introducing 3rd party dependency
- Some operators and concepts can be confusing in the beginning.

References

Here are some links to material referenced in the talk:

- RxSwift and ReactiveX
- What is reactive programming and why should I use it? by Matt Gallagher
- Out of the Tar Pit by Moseley and Marks
- Mutability, aliasing, and the caches you didn't know you had by Andy Matuschak
- Enemy of the state by Justin Spahr-Summers
- "Advanced iOS Application Architecture and Patterns" from WWDC 2014 by Andy Matuschak and Colin Barrett

Thanks for not falling asleep. Questions?