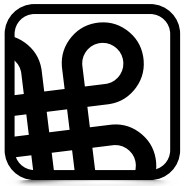


Macoun

RxReSwift

— Betriebsunfälle & Lehrgeld —

Christian Tietze
@ctietze

Ablauf

1. **RxSwift**
2. **ReSwift**

Motivation zu Trendthemen

- Neue Jobs
- Denken schulen
- Gehaltserhöhung verlangen

NARRATION
STATT
DIDAKTIK

RxSwift

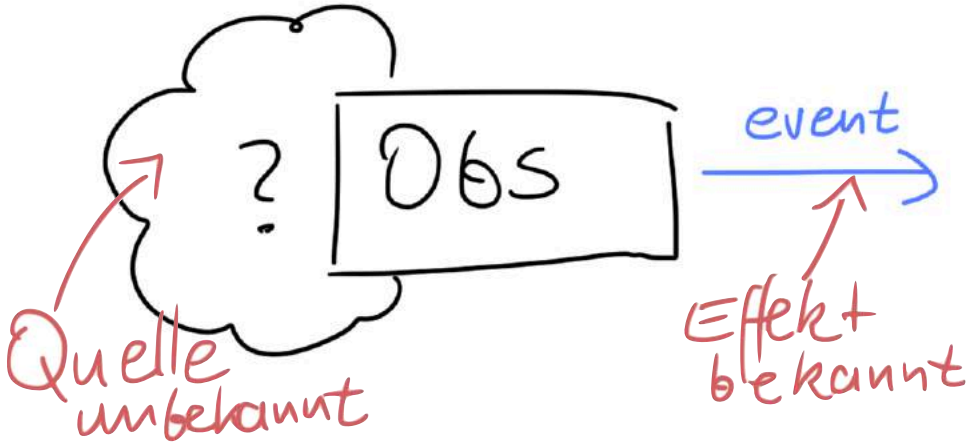
RxSwift Themen

1. Rx Vokabular klären
2. Wie baut man reaktive »Streams« ein?
3. *Event*-Produzenten testen
4. Downgraden: Reactive → Imperativ

RxSwift Themen

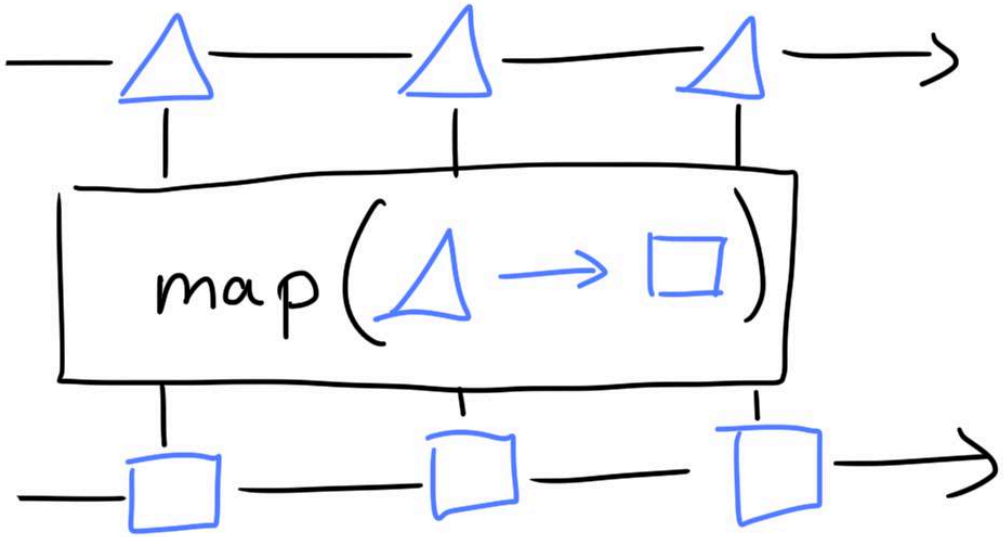
1. Rx Vokabular klären
2. Wie baut man reaktive »Streams« ein?
3. *Event*-Produzenten testen
4. Downgraden: Reactive → Imperativ

RxSwift.Observable



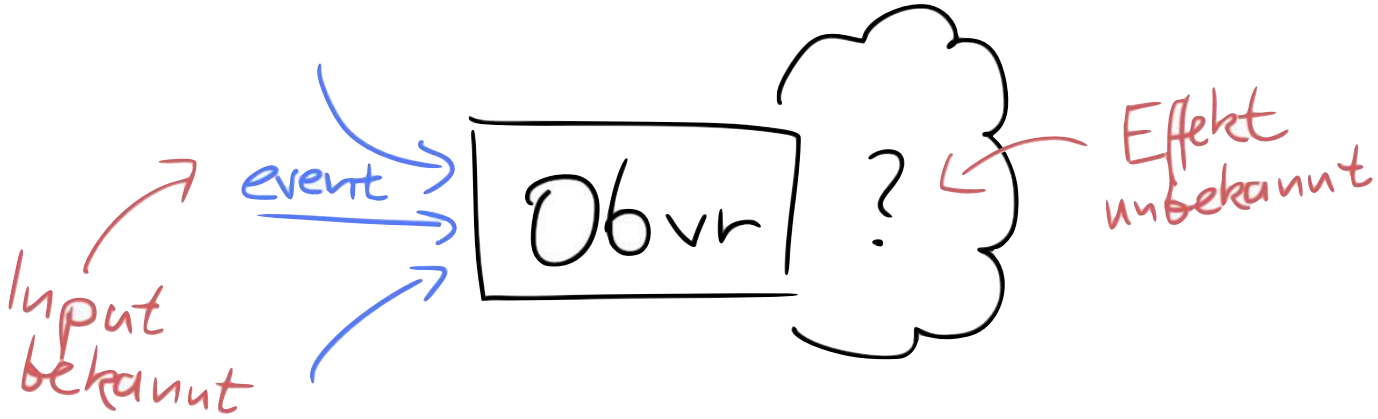
- Mentales Modell: `Sequence` Hurenkind
- *Event*-Emitter
- Observable abonnieren mit `.subscribe() -> Disposable`

Observables Operatoren




```
let bananas: Observable<Banana> = // ...
let peeledBananas: Observable<PeeledBanana> = bananas.map { $0.peel }
```

RxSwift.Observer



- Mentales Modell: Event Handler Schusterjunge
- Zentraler callback: `.on(_ event: Event<T>)`
- Reagiert auf `Event.next(T)`, `.error(Error)`, `.completed`

Rx: **R**eactive E**x**tensions

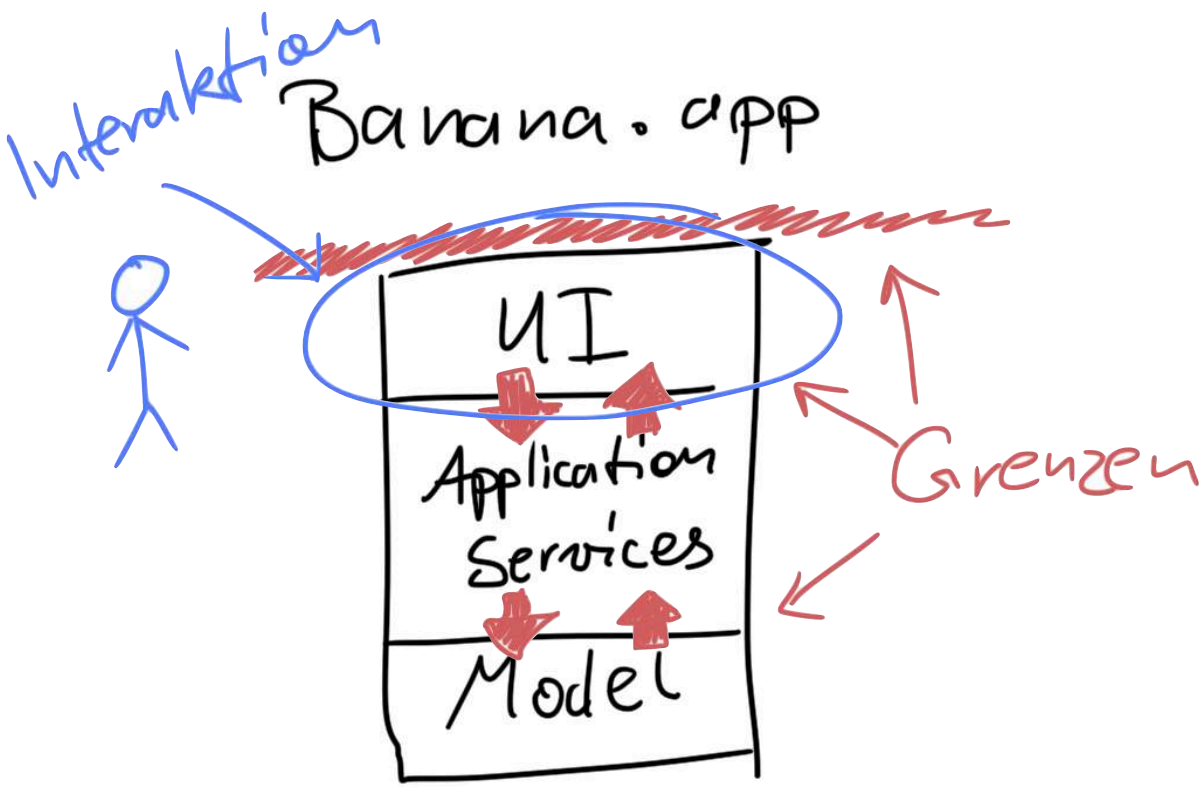
```
extension Reactive where Base: MyVeryAmazingButton {
    var amazement: Observable<Void> { return .never() }
}
```

```
let button = MyVeryAmazingButton()
button.rx.amazement
    .map { Void in "Wow!" }
```

RxSwift Themen

1. Rx Vokabular klären
2. Wie baut man reaktive »Streams« ein?
3. *Event*-Produzenten testen
4. Downgraden: Reactive → Imperativ

Mit UI starten



Grenzen ziehen

UI Komponenten sind an der Außengrenze der App:

```
NSButton().rx.tap       // ControlEvent<Void>
    .asObservable()     // Observable<Void>
```

Grenzen ziehen

UI Komponenten sind an der Außengrenze der App:

```
NSTextField().rx.text    // ControlProperty<String?>
    .asObservable()      // Observable<String?>
    .do(onNext: { print("> ", $0) })

NSTextField().rx.text    // ControlProperty<String?>
    .asObserver()        // Observer<String?>
    .on(.next("Hi"))
```

```
> nil
```

Grenzen ziehen

UI Komponenten sind an der Außengrenze der App:

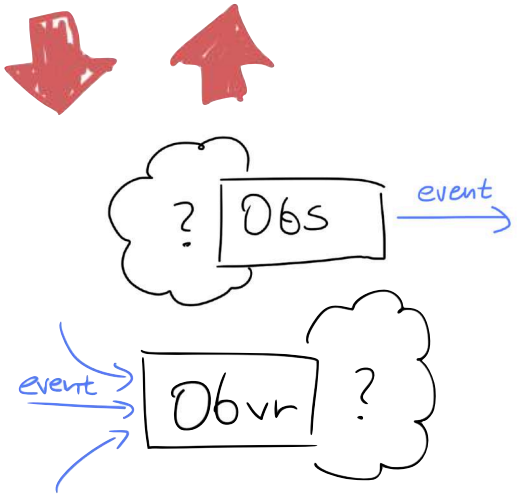
```
class MyViewController: NSViewController {
    override func viewDidLoad() {
        textField.text.asObservable()
    }
}
```

Grenzen ziehen

UI Komponenten sind an der Außengrenze der App:

Christians Vorgehen

- ✓ Ports definieren
- ✓ Ports verwenden
- ✓ Ports abonnieren



Ports Definieren

Verpasse deinen Komponenten `Observables`:

```
public class BananaViewController: NSViewController {
    // ...
    private let _bananaCreation = PublishSubject<Banana>()
    public var bananaCreation: Observable<Banana> {
        return _bananaCreation.asObservable()
    }
}
```

Ports Verwenden

Leite Signale von Basis-Komponenten an den Port weiter:

```
public class BananaViewController: NSViewController {
    // ...
    fileprivate let disposeBag = DisposeBag()
    public override viewDidLoad() {
        let bananas = originTextField.rx.text.asObservable()
            .map(Banana.init(origin:))
        createButton.rx.tap.asObservable()
                    .flatMapLatest(bananas)
                    .subscribe(_bananaCreation)
                    .disposed(by: disposeBag)
    }
}
```

Ports Abonnieren

Selbstverständlich werden Ports in Protokolle extrahiert! 😌

```
protocol BananaEventSource {
    var bananaCreation: Observable<Banana> { get }
}
extension BananaViewController: BananaEventSource { }
```

```
public class BananaEventHandler {
    private let subscription: Disposable
    public init(source: BananaEventSource) {
        self.subscription = source.bananaCreation
            .subscribe(onNext: { print($0.origin) })
    }
}
```

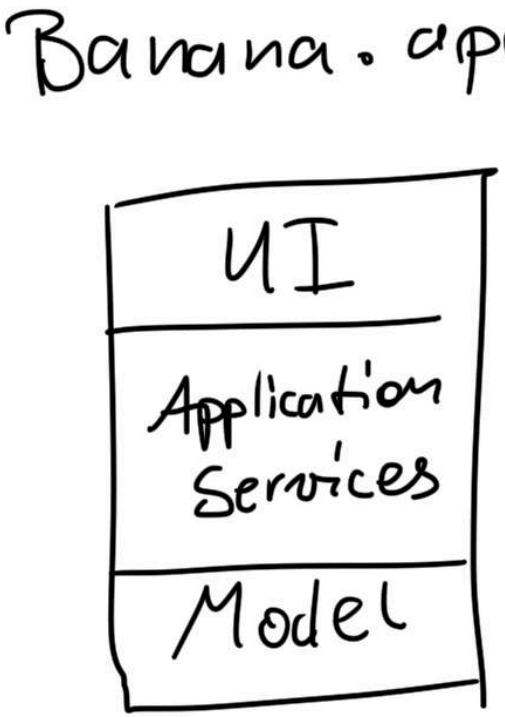
Ports Abonnieren

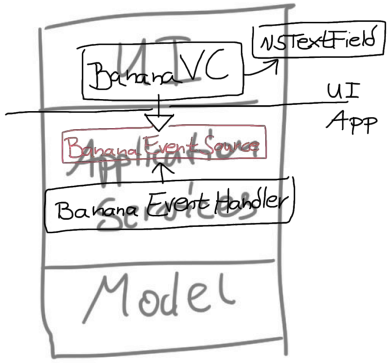
Selbstverständlich werden Ports in Protokolle extrahiert! 😎

```
protocol BananaEventSource { var bananaCreation: Observable<Banana> { get } }
extension BananaViewController: BananaEventSource { }
```

```
public class BananaPrinter: ObserverType {
    public on(_ event: Event<Banana>) {
        guard case .next(let banana) = event else { return }
        print(banana.origin)
    }
}

let subscription = source.bananaCreation.subscribe(BananaPrinter())
```

Ports Abonnieren

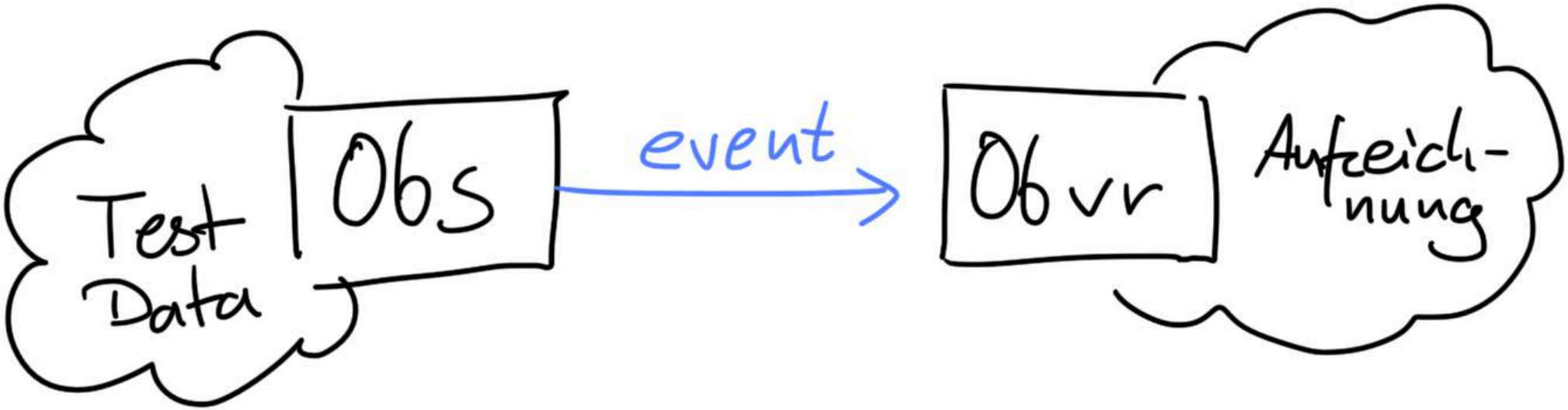




RxSwift Themen

1. Rx Vokabular klären
2. Wie baut man reaktive »Streams« ein?
3. *Event*-Produzenten testen
4. Downgraden: Reactive → Imperativ

Manuell testen



RxTest: Produzenten testen

1. `TestScheduler` steuert zeitliche Abfolge
2. `TestObserver` zeichnet Events und ihre Zeit auf
3. `Recorded<Event<T>>` unterstützt `==`

RxTest: Produzenten testen

1/4 Test-Einstiegspunkt hinzufügen

```
extension BananaCreationViewController {
    func changeOrigin(_ origin: String) {
        // Programmatic changes do not trigger `.rx.text`
        originTextField.string = origin

        // HACKY QUICKFIX: fire change event to delegate
        // ... originTextField.endEditing() ...
    }
```

RxTest: Produzenten testen

2/4 `TestScheduler` verwenden

```
let bananaCreationVC = BananaCreationViewController()
let scheduler = TestScheduler(initialClock: 0)
scheduler.scheduleAt(300) { bananaCreationVC.changeOrigin("Myanmar") }
scheduler.scheduleAt(400) { bananaCreationVC.changeOrigin("Honduras") }
scheduler.scheduleAt(500) { bananaCreationVC.createButtonClicked(nil) }
scheduler.scheduleAt(600) { bananaCreationVC.changeOrigin("Columbia") }
scheduler.scheduleAt(700) { bananaCreationVC.createButtonClicked(nil) }
```

RxTest: Produzenten testen

3/4 `TestObserver` anlegen

```
// TestScheduler.start: normalerweise zu T=200
let results: TestObserver<Banana>
    = scheduler.start { bananaCreationVC.bananaCreation }
```

RxTest: Produzenten testen

4/4 `TestObserver`-Aufzeichnung prüfen

```
// extension Banana: Equatable {}
let expectedEvents: [Recorder<Event<Banana>>] = [
    next(500, Banana(origin: "Honduras"),
    next(700, Banana(origin: "Columbia")
]
XCTAssertEqual(results.events, expectedEvents)
```

RxSwift Themen

1. Rx Vokabular klären
2. Wie baut man reaktive »Streams« ein?
3. *Event*-Produzenten testen
4. Downgraden: Reactive → Imperativ

Szenario

- `TableViewController.display(_: Changes)` erhält zugleich `listData: [Stuff]?` und `selection: Stuff?`
- Wenn `listData` geändert wird,
  lade das Tabellenmodel neu
- Nach `NSTableView.reloadData()`,
  stell die alte Auswahl wieder her
- Ändere die `NSTableView.selectedRow`,
  wenn `selection` übergeben wird

Rx-Ansatz

Lade die Tabelle neu, wenn das Model sich ändert

```
class ViewController: NSViewController, NSTableViewDataSource {
    let tableModel = Variable<[Stuff]>([])
    override func viewDidLoad() {
        tableModel.asObservable().subscribe(onNext: { _ in
                self.tableView.reloadData()
            }).disposed(by: disposeBag)
    }
    func tableView(_:viewFor:row:) -> NSView? {
        let rowModel = tableModel.value[row]
        // ...
    }
}
```

Rx-Ansatz

Stell die Auswahl wieder her

```
// ...
override func viewDidLoad() {
    tableModel.asObservable().subscribe(onNext: { _ in
            let oldSelection = self.tableView.selectedRange()
            self.tableView.reloadData()
            self.tableView.setSelectedRange(oldSelection)
        }).disposed(by: disposeBag)
}
// ...
```

Rx-Ansatz

Aktualisiere die Auswahl bei Änderungen

```
let selectionModel = Variable<Int>(-1)
override func viewDidLoad() {
    // ...
    selectionModel.asObservable().subscribe(onNext: { row in
            guard row >= 0 else { return  }
            self.tableView.selectRowIndexes(IndexSet(integer: row),
                byExtendingSelection: false)
        }).disposed(by: disposeBag)
}
// ...
```

Rx-Ansatz

Ändere die Daten!!!1 1eins

```
func display(changes: Changes) {
    if let modelChange = changes.listData {
        tableModel.value = modelChange }
    if let selectionChange = changes.selection {
        selectionModel.value = selectionChange }
}
```

Rx-Ansatz

... und jetzt stell sicher, dass zuerst die Tabelle geladen wird, bevor die Auswahl sich verändert!

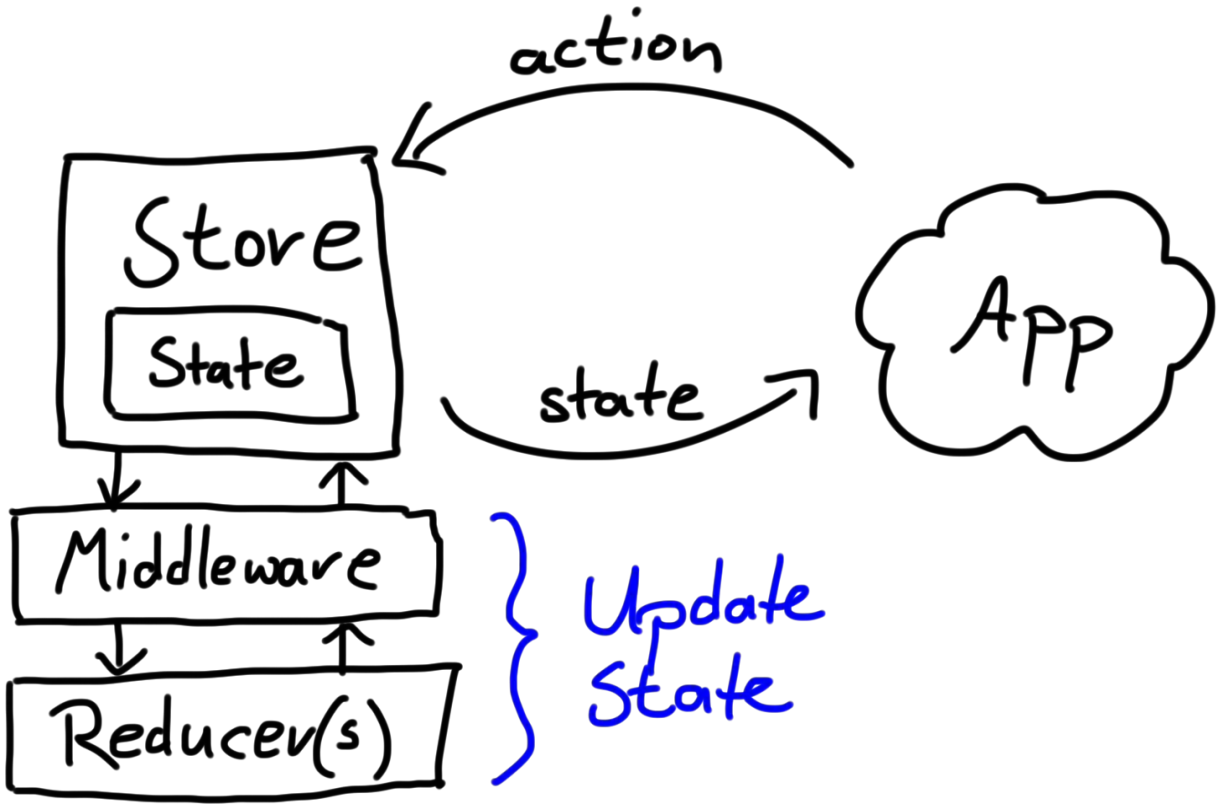
```
// ... ?
```

¬Rx

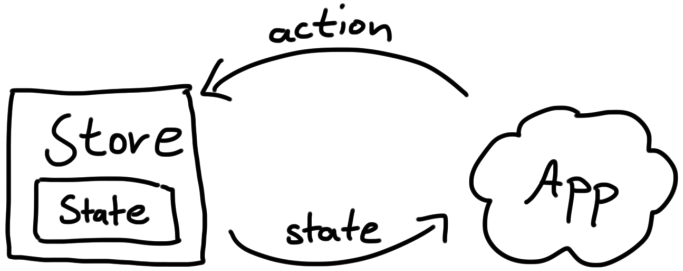
```
func display(changes: Changes) {
    var shouldReload = false
    if /* new data */ {
        self.tableModel = // ...
        shouldReload = true }

    // langlebiger Cache der Auswahl
    if let selectionChange = changes.selection {
        self.selectionModel.value = selectionChange }

    if shouldReload {
        let selection = self.selectionModel.value
        self.tableView.reloadData()
        self.select(selection) }           // re-select nach reload
}                                          // oder gar erste Auswahl
```

ReSwift

ReSwift Basics

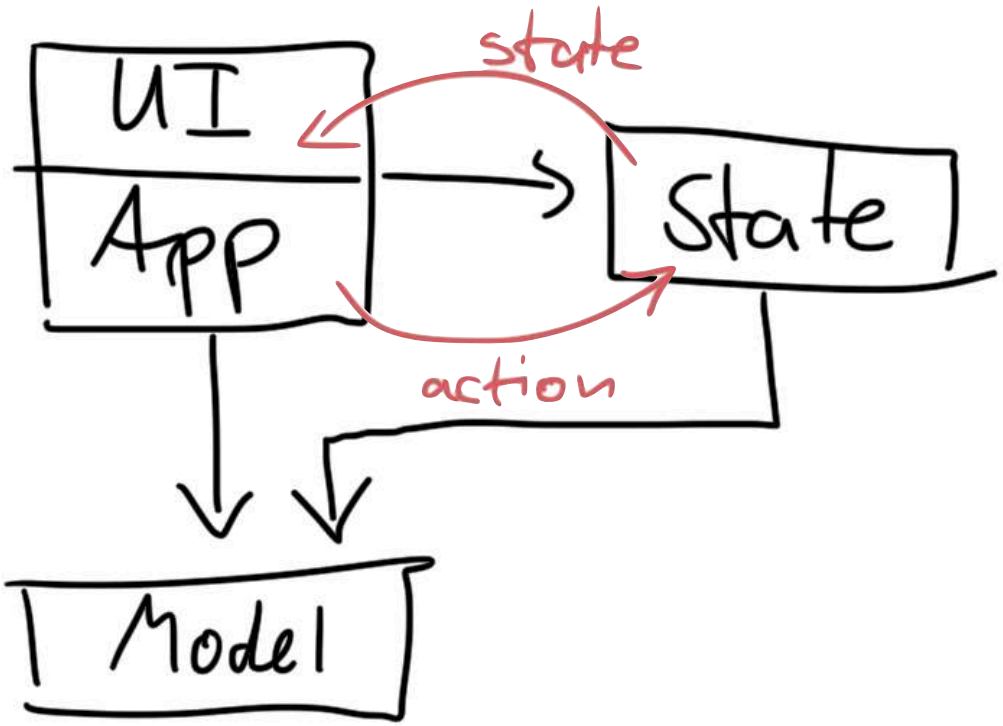


ReSwift Types



- `StateType` ist das Protokoll für den App-Zustand
- `Store`-Instanz wird konfiguriert
- `Action`-Protokoll markiert, dass etwas im Store geschehen soll
- `store.dispatch(_:Action)` ist der Einstiegspunkt

State Modul



State Modul

- ★ Isoliert testen
- ★ Isoliert denken
- ★ Kompiliert unabhängig 

- ☆ cross-platform reuse

Optimistische UI

What happens in UI
stays in UI.

— *Redewendung*

Optimistische UI

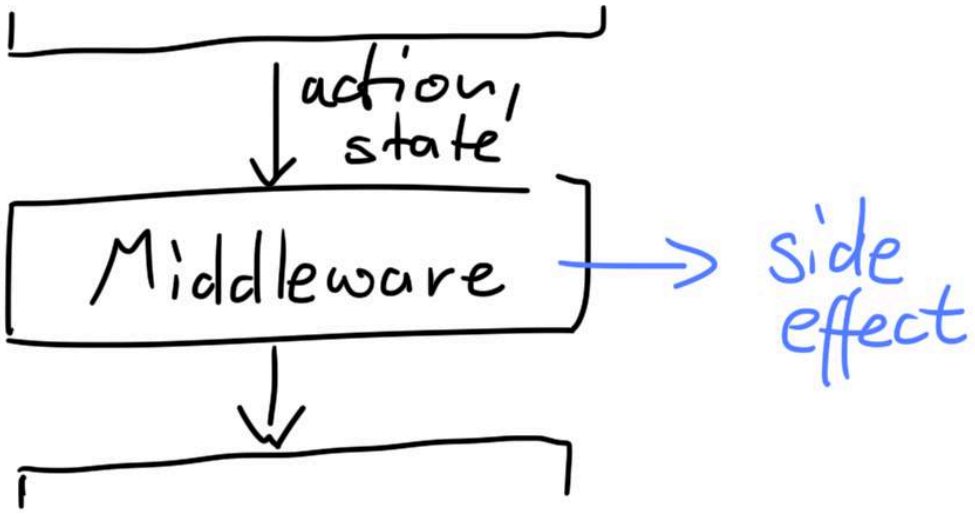
- `Actions` mit side-effects abpuffern (Netzwerk, Datei-I/O)
- Sofort Änderungen in der UI vorschlagen
- Update vom State liefert irgendwann **Die Wahrheit™**

= keine Konflikte!

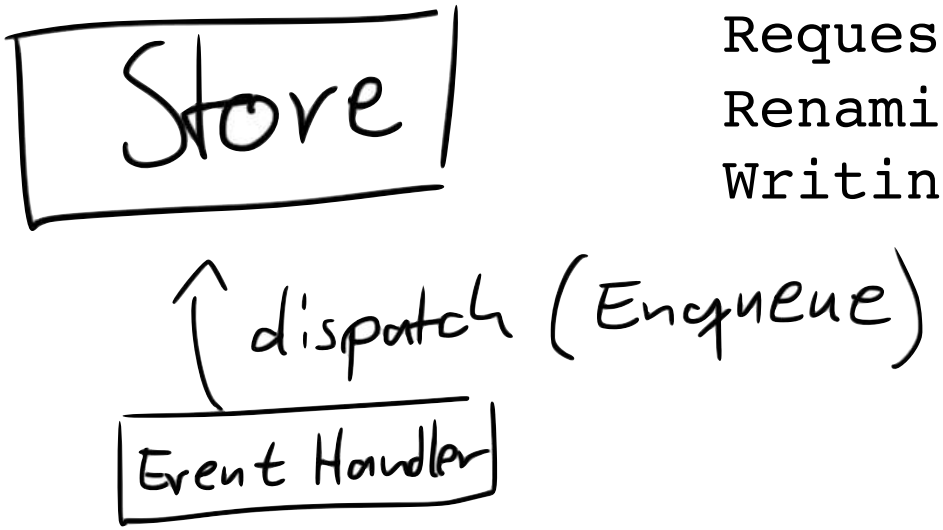
Middleware Anxiety

- 🚫 Keine Middleware für *side effects*
- 💞 EnqueueAction + Subscriber + DequeueAction

Middleware Anxiety



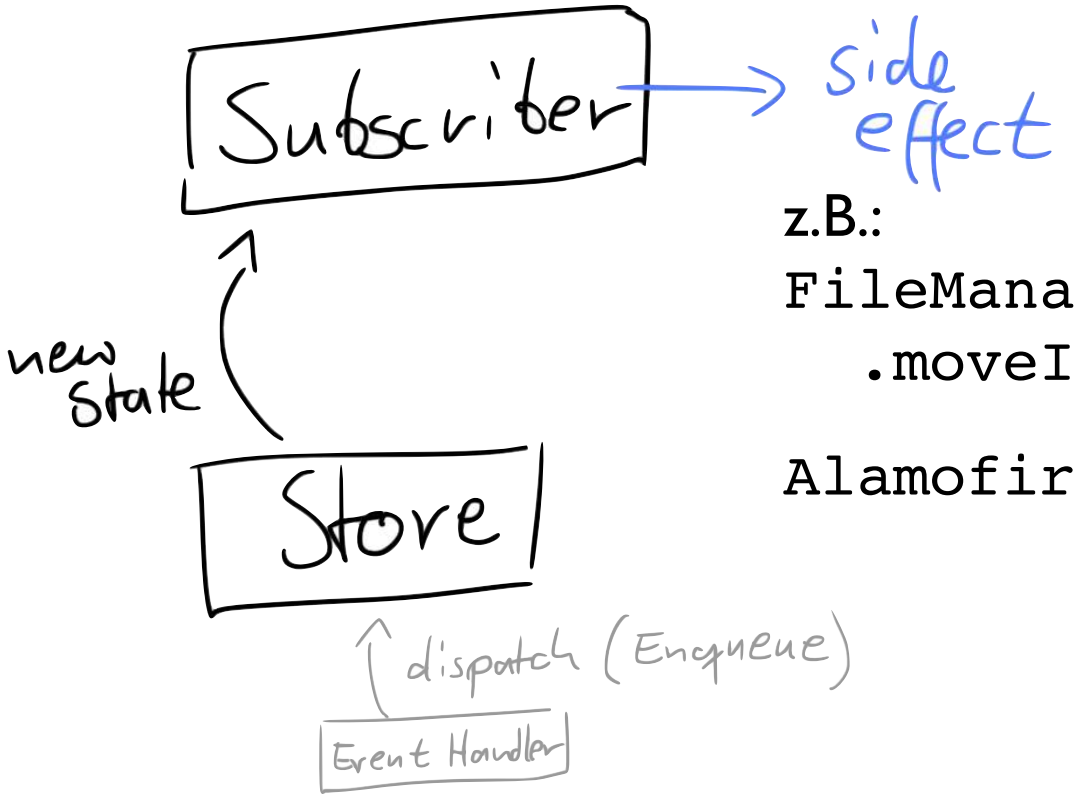
Middleware Anxiety



z.B.:

```
RequestingUsers
RenamingFile
WritingNoteChange
```

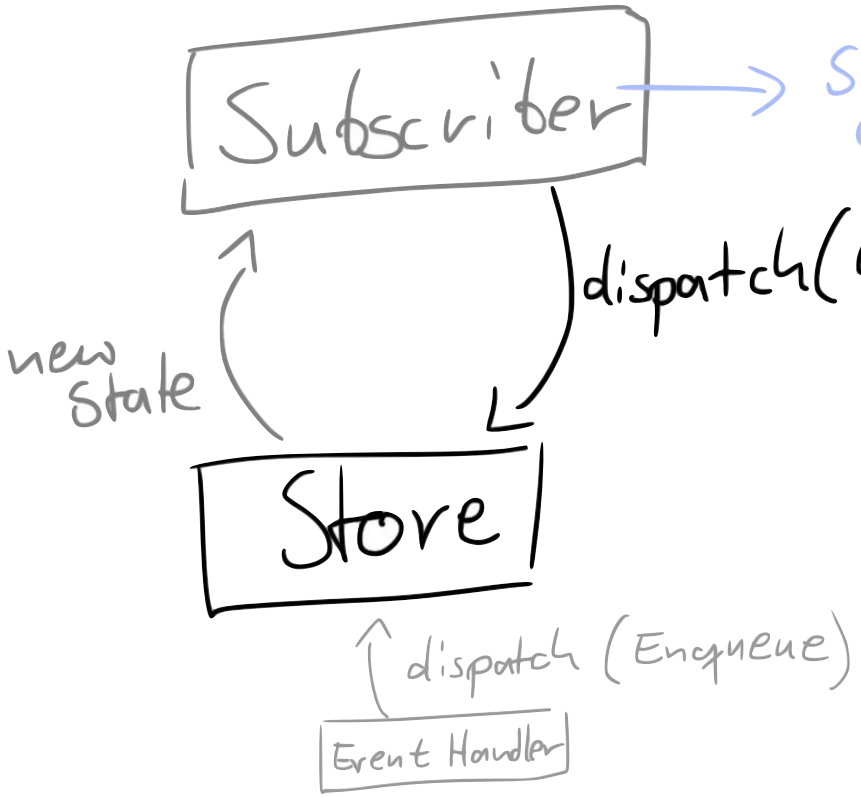
Middleware Anxiety



z.B.:

```
FileManager.default
   .moveItem(at:to:)

Alamofire.request("...")
```

Middleware Anxiety



z.B.:

```
CancelingRenameFile
CompletingUserRequest
FinishingNoteChange
```

BECOMING THE MYTHICAL

10x

PROGRAMMER

Middleware Anxiety

»Use Middleware to wrap services that perform any kind of I/O.«
(jemand im ReSwift Channel auf Gitter.im)

\+ In Subscribern zu denken ist leichter

Viel Zeremoniell:

− Substates für die *side effect queues* nerven

− Enqueue/Dequeue Actions nerven (Tests!)

= Subscriber schreiben ist ok

± Subscriber-Performance ist etwa gleich zu Middleware

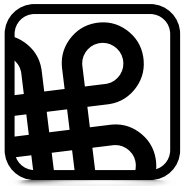
RxReSwift

```
let subscription = store.rx.state
    // wie gewohnt kann man Operatoren anwenden:
    .map { $0.substate }
    .distinctUntilChanged()

    // erstellt einen ReactiveStoreSubscriber:
    .subscribe(onNext: { substate in
        print("Incoming: ", substate)
    })
```

Vielen Dank

Fragen!

http://christiantietze.de — @ctietze



Macoun