авиасейлс

Functional Core Imperative Shell

iOS разработчик из ✈️ Аааааавиасейлс ✈️

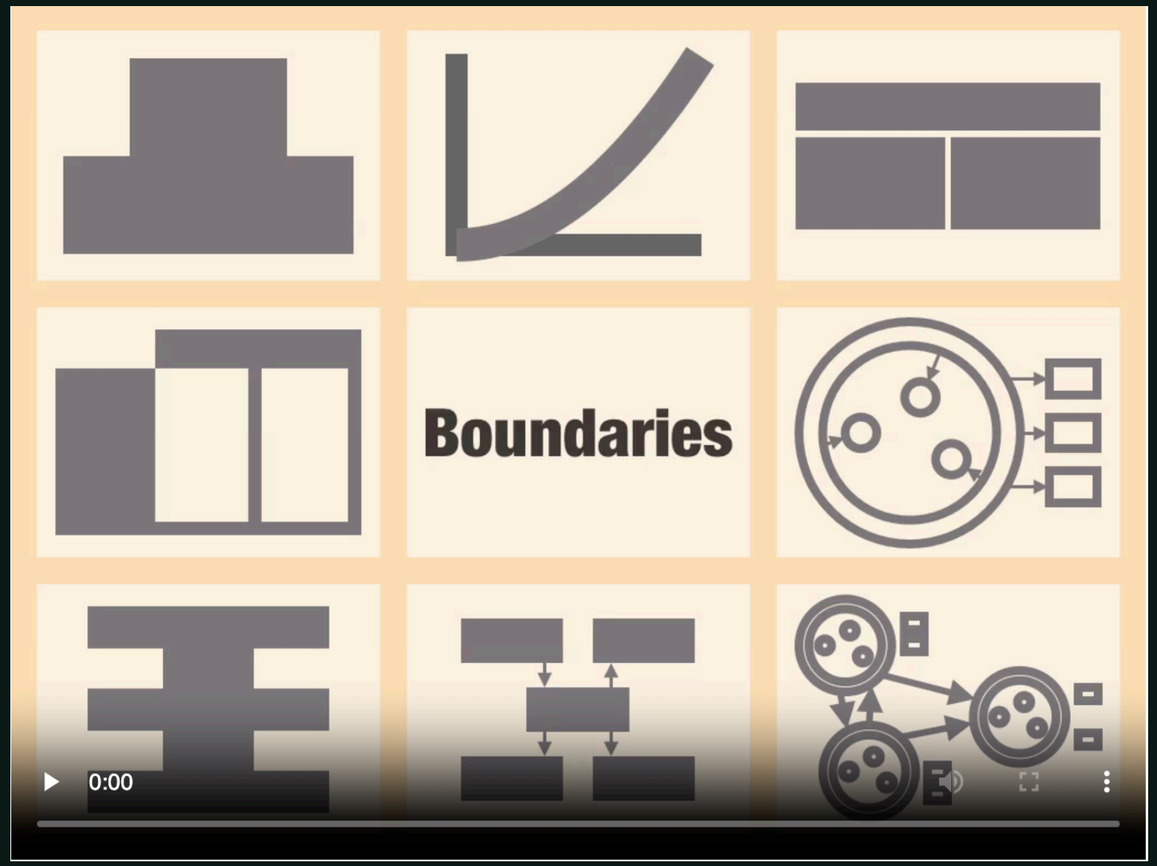
github.com/AgapovOne

→ 8 лет в Swift

→ Пришел в Авиасейлс из Контура

→ Делаю Guides, Короче в Авиасейлс

Gary Bernhardt



Алексей Озун

План

- → Проблема
- → Теория по функциональному программированию
- → Тестирование и стратегия
- → Паттерны фп. Decisions / Executor
- → Функциональное ядро и императивная оболочка
- → Немного про зависимости
- → Пример на MVVM с тестами

Проблема

MVVM

```
enum Event { case userDidTapButton }

enum State: Equatable {
    case loading
    case loaded(String)
}

struct Deps {
    let track: (Event) → Void
    let showSnackbar: (String) → Void
    let log: (Any ... ) → ()
    let fetchFact: () async throws → Data
}

func handle(_ event: Event) {}
```

MVVM

```
func handle(_ event: Event) {
    state = .loading
    deps.track(event)
    Task {
          do {
            let fact = try await deps.fetchFact()
            deps.log(fact)
            state = .loaded(fact.text)
        } catch {
            deps.log(error)
            deps.showSnackbar("Went wrong")
        }
    }
}
```

MVVM тесты

```
let expectation = expectation(description: "fact loading")
let viewModel = SystemViewModel(
    deps: .init(
        // ...
        fetchFact: { Fact(text: "some funny fact") }
    )
)

viewModel.handle(.userDidTapButton)
XCTAssertEqual(viewModel.state, .loading)

wait(for: [expectation])
XCTAssertEqual(viewModel.state, .loaded("some funny fact"))
```

ООП

ООП

Объекты из реального мира.

Классы, наследование, инкапсуляция, полиморфизм.

Поведение принадлежит объекту.

```
class Athlete {
    var trainingHours: Float
    func train(_ hours: Float) { trainingHours += hours }
}
```


ФП

ФП

```
struct Athlete {
    let trainingHours: Float
}

func train(
    athlete: Athlete,
    hours: Float
) → Athlete {
    Athlete(trainingHours: athlete.trainingHours + hours)
}
```

ФП

Данные

Entity, Value Object, struct, data class

```
struct Athlete {
    let trainingHours: Float
    var hasTrainedEnoughForOlympics: Bool { trainingHours > 1000 }
}
```

ФП

Функции

→ Поведение

→ Бизнес логика

ФП

Эффекты

→ Общение с внешним миром
→ Мутации
→ Состояние
→ Интерфейс пользователя

Функции

🫧 **Pure** functions

1. Детерминированы
2. Без сайд-эффектов
3. (Изолированы)

🤡 **Impure** functions

Чистые функции

```
func sum(_ one: Int, _ two: Int) → Int {
    one + two
}
```

1. Детерминирована
2. Без сайд-эффектов
3. (Изолирована)

Деление

```
func divide(_ num: Double, by divisor: Double) → Double
```

Чистая функция?

Общение между функциями

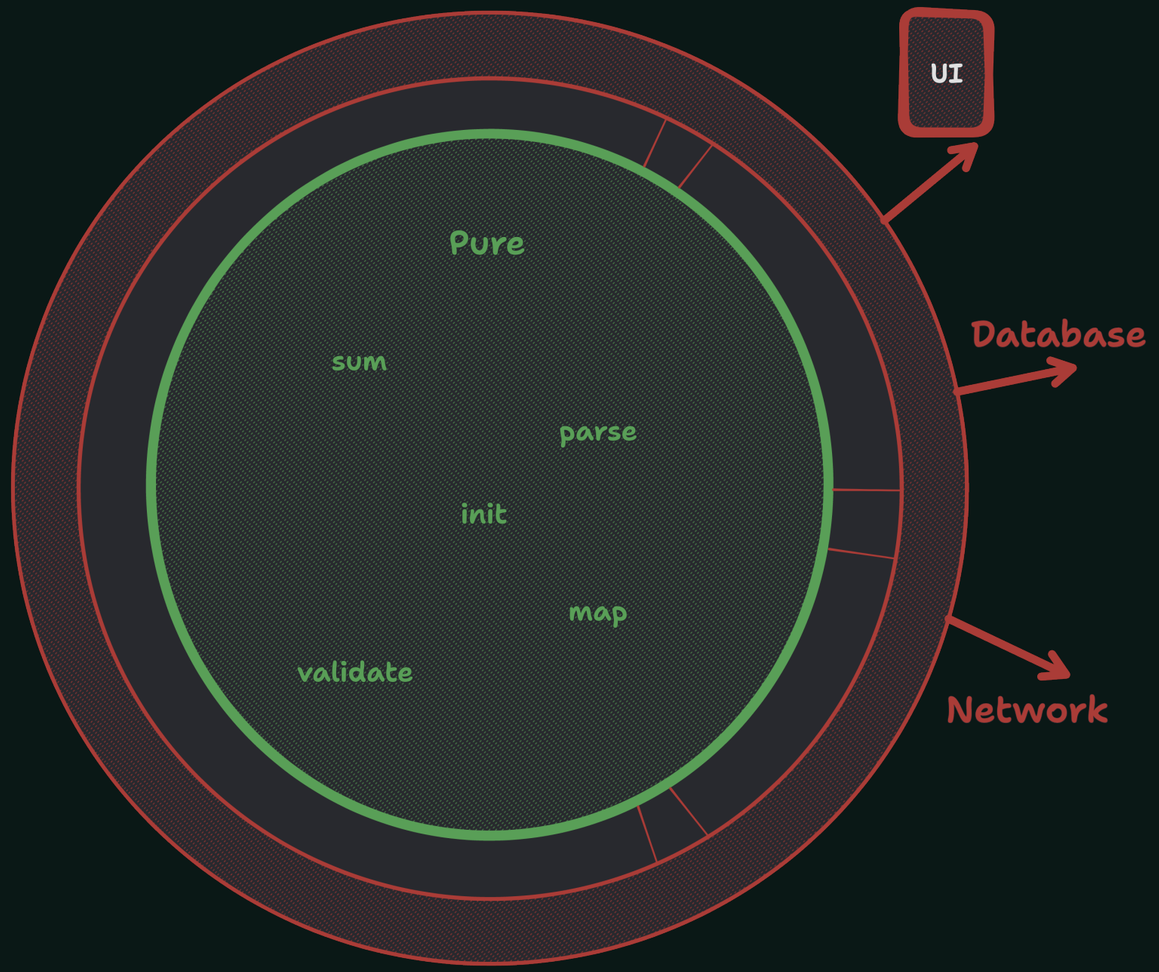
```
func pure() {
    pure()
    impure() // 🛑
}
```

```
func impure() {
    pure()
    impure()
}
```

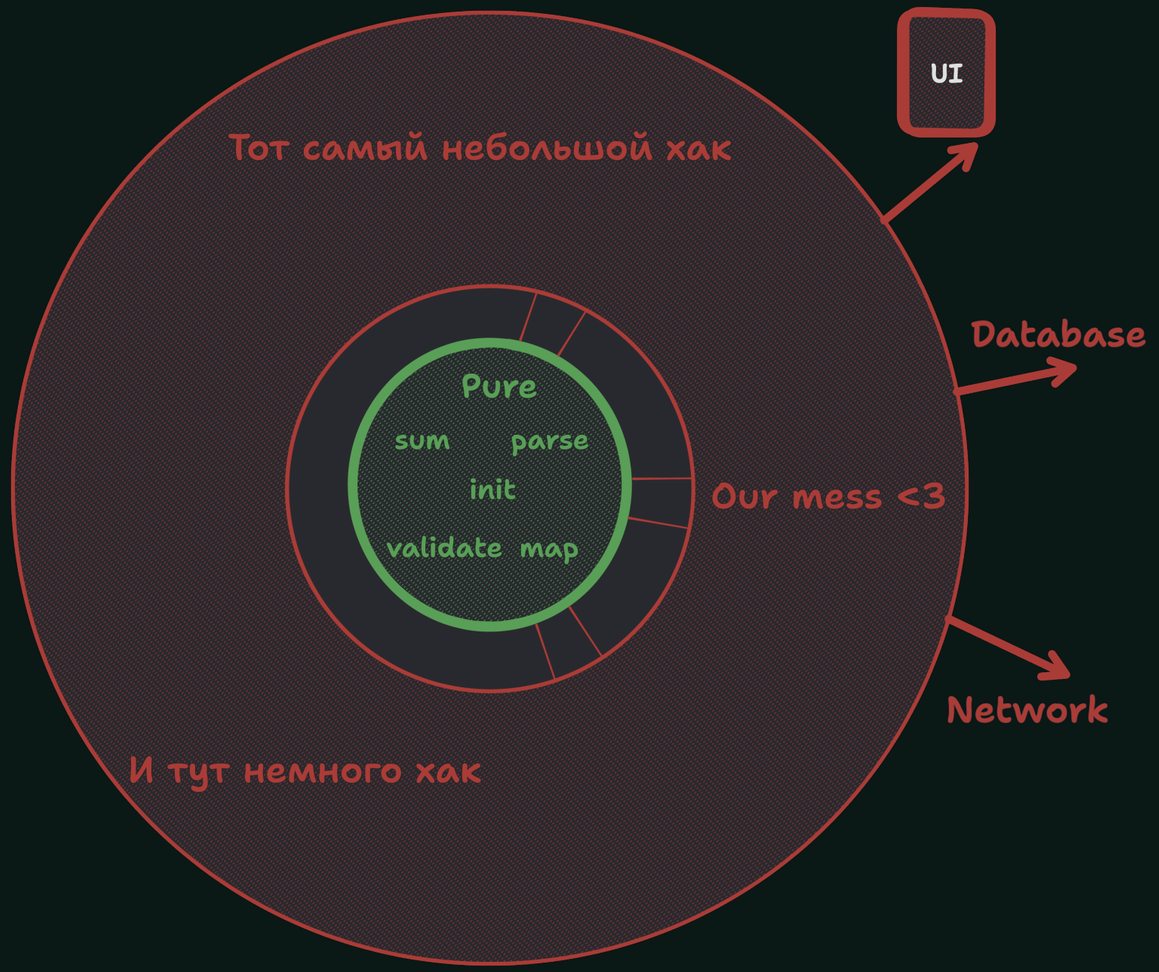
Больше чистых функций

1. Изолированы
2. Легко тестировать
3. Ожидаемо ведут себя

Стремимся

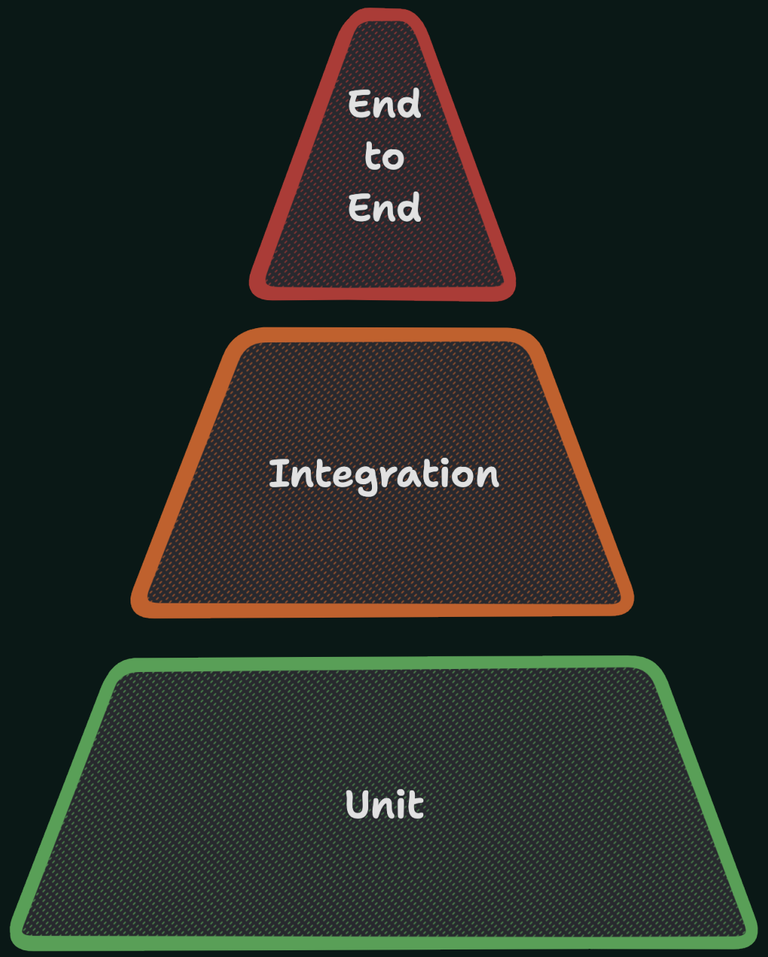


Получаем

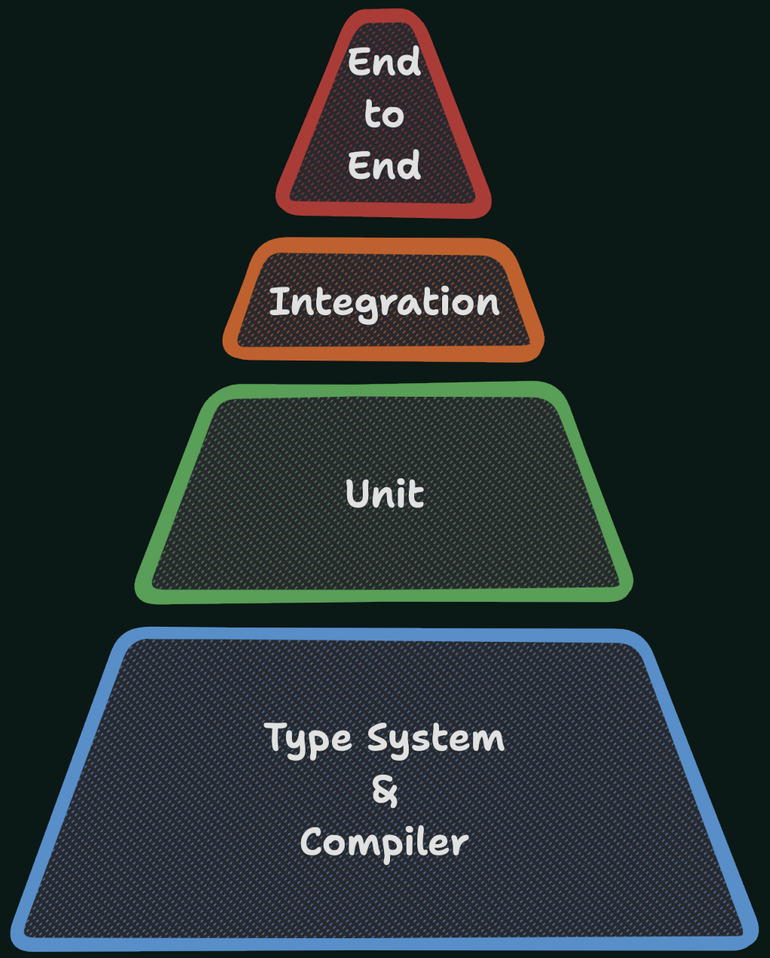

UI
Тот самый небольшой хак
Database
Pure
sum
parse
init
validate
map
Our mess <3
Network
И тут немного хак

Тесты

Тесты



Тесты



Юниты и интеграционные

```
func test_unit() {
    let actual = 2 + 2
    XCTAssertEqual(actual, 4)
}

func test_integration() {
    service.mockedResult = "Mocked"
    ViewModel(service).runLogic()
    XCTAssertEqual(viewModel.state, "Mocked")
}
```

Integration tests are a scam

📹 **Boundaries, Gary Bernhardt**

Юнит тесты!

→ Быстрые
→ Изолированные
→ Надежные
→ Описательные

ФП паттерны

Паттерны

→ Single Responsibility principle
→ Open/Closed principle
→ Interface Segregation principle
→ Dependency Inversion principle
→ Factory
→ Strategy
→ Decorator

Паттерны

- → Single Responsibility principle
- → Open/Closed principle
- → Interface Segregation principle
- → Dependency Inversion principle
- → Factory
- → Strategy
- → Decorator

- → Functions
- → Functions?
- → Functions!
- → ...
- → you know

Принципы дизайна в ФП

- → Функции, типы, композиция
- → Функции как параметры
  - → Функции как интерфейсы
- → Частичное применение и Dependency injection
- → Pipe, chaining

ФП паттерны и дизайн. Монады? Видео, доклады по F#.

📹 **Scott Wlaschin**

Decisions & Executor

Decisions & Executor

```
func makeDecision(input: T) → Decision?
func makeDecisions(input: T) → [Decision]
// &
func execute(decision: Decision) {}
```

Decisions & Executor

```
enum Event { case userDidTapButton }
enum Decision { case showSnackbar }

enum State { case loading, failed }

func decision(event: Event, state: State) → Decision? { // 🫧 pure
    if event == .userDidTapButton {
        switch state {
            case .failed: return .showSnackbar("Everything went wrong")
            case .loading: return nil
        }
    }
}
```

Decisions & Executor

```
func execute(event: Event) { // 🤡 impure
    let decision = decision(event: Event, state: self.state)
    switch decision {
        case .showSnackbar:
            deps.showSnackbar($0)
        case nil:
            break
    }
}
```

Decision == Action == Effect == Intent

&

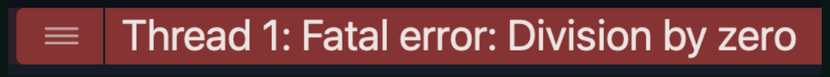
Executor == Performer == Handler

Деление

```
func divide(_ num: Double, by divisor: Double) → Double
```

Чистая функция?

Деление




```
// Impure!
func divide(_ num: Double, by divisor: Double) → Double {
    if divisor == 0 {
        fatalError("Division by zero")
    }
    return num / divisor
}
```

Деление чистое

```
// Pure?
func divide(
    _ num: Double,
    by divisor: Double,
    onDivisionByZero: () → Void
)  → Double? {
    if divisor == 0 {
        onDivisionByZero()
        return nil
    }
    return num / divisor
}
```

Деление чистое

```
enum DivisionResult {
    case divisionByZero
    case success(Double)
}
// Pure!
func divide(
    _ num: Double,
    by divisor: Double
) → DivisionResult {
    divisor == 0
        ? .divisionByZero
        : .success(num / divisor)
}
```

Деление чистое

```
func calculate() {
    switch divide(10, by: 2) {
        case .success(let result):
            self.calculationReslut = result
        case .divisionByZero:
            deps.destroyDevice()
    }
}

// In Tests
func test_division() {
    XCTAsertEqual(divide(10, by: 2), .success(5))
    XCTAsertEqual(divide(10, by: 0), .divisionByZero)
}
```

Functional Core
Imperative Shell

До

```
func sendToOlympics() async → [Athlete] {
    let athletes = await Database.athletes
    return athletes.filter { athlete in
        if athlete.trainingHours > 1000 && !athlete.isInjured {
            await Aviasales.buyCheapTickets(athlete)
            return athlete
        }
        return nil
    }
}
```

Core & Shell

```
func sendToOlympics() async → [Athlete] {
    findPrepared(await Database.athletes).forEach { athlete in
        Aviasales.buyCheapTickets(athlete)
    }
}

func findPrepared(_ athletes: [Athlete]) → [PreparedAthlete] {
    athletes.filter { try PreparedAthlete($0) }
}
```

Core & Shell

```
func sendToOlympics() async → [Athlete] { // 🐚 Shell
    findPrepared(Database.athletes).forEach { athlete in
        Aviasales.buyCheapTickets(athlete)
    }
}

func findPrepared(_ athletes: [Athlete]) → [PreparedAthlete] { // 🤯 Core
    athletes.filter { try PreparedAthlete($0) }
}
```

Core & Shell

🤯 Core

→ Мало зависимостей

→ Много веток

→ Изолирован

🐚 Shell

→ Много зависимостей

→ Мало веток

→ Интегрирован во внешний мир

```
struct PreparedAthlete {
    let trainingHours: Float

    init(_ athlete: ) throws {
        if athlete.isInjured { throw AthleteCreationError.injured }
        if athlete.trainingHours < 1000 { throw AthleteError.untrained }
    }
}
```

Parse, don't validate

→ Данные, которые уберут невозможное состояние

→ Парсинг как можно раньше

`Result<Success, Err> → Validated<Success, [Err]>`

Parse, don't validate

 **Alexis King**

FCIS в приложении

MVVM

```
enum Event { case userDidTapButton }

enum State: Equatable {
    case loading
    case loaded(String)
}

struct Deps {
    let track: (Event) → Void
    let showSnackbar: (String) → Void
    let log: (Any ... ) → ()
    let fetchFact: () async throws → Data
}

func handle(_ event: Event) {}
```

MVVM

```
func handle(_ event: Event) {
    state = .loading
    deps.track(event)
    Task {
          do {
            let fact = try await deps.fetchFact()
            deps.log(fact)
            state = .loaded(fact.text)
        } catch {
            deps.log(error)
            deps.showSnackbar("Went wrong")
        }
    }
}
```

MVVM тесты

```
let expectation = expectation(description: "fact loading")
let viewModel = SystemViewModel(
    deps: .init(
        // ...
        fetchFact: { Fact(text: "some funny fact") }
    )
)

viewModel.handle(.userDidTapButton)
XCTAssertEqual(viewModel.state, .loading)

wait(for: [expectation])
XCTAssertEqual(viewModel.state, .loaded("some funny fact"))
```

FCIS

```
// Before, impure
func handle(_ event: Event) {
    state = .loading
    deps.track(event)
    Task {
          do {
            let fact = try await deps.fetchFact()
            deps.log(fact)
            state = .loaded(fact.text)
        } catch {
            deps.log(error)
            deps.showSnackbar("Went wrong")
        }
    }
}
```

FCIS

```
// After, still impure
func handle(_ event: Event) {
    let decisions = Self.makeDecisions(event, &state)
    decisions.forEach {
        switch $0 {
            case .load: Task {
                do {
                    let fact = try await deps.fetchFact()
                    handle(.model(.finishLoading(fact)))
                } catch {
                    handle(.model(.failedLoading(error.localizedDescription)))
            } }
            case .log(let something): deps.log(something)
            case .track(let event): deps.track(event)
            case .showSnackbar(let message): deps.showSnackbar(message)
        }
    }
}
```

FCIS

```
enum Core {
  static func makeDecisions(
      event: Event,
      state: State
  ) → (State, [Decision]) {
      switch event {
          case .view(.userDidTapButton):
              return (.loading, [.track(event), .load])
          case .model(.failedLoading(let error)):
              return (state, [.log(error), .showSnackbar("Went wrong")])
          case .model(.finishLoading(let fact)):
              return (.loaded(fact.text), [.log(fact.text)])
      }
  }
}
```

FCIS

```
enum Core {
  static func makeDecisions(
      event: Event,
      state: inout State
  ) → [Decision] {
      switch event {
          case .view(.userDidTapButton):
              state = .loading
              return [.track(event), .load]
          case .model(.failedLoading(let error)):
              return [.log(error), .showSnackbar("Went wrong")]
          case .model(.finishLoading(let fact)):
              state = .loaded(fact.text)
              return [.log(fact.text)]
      }
  }
}
```

FCIS

```
enum Event: Hashable {
    enum ViewEvent: Hashable {
        case userDidTapButton
    }

    enum ViewModelEvent: Hashable {
        case finishLoading(Fact)
        case failedLoading(String)
    }

    case view(ViewEvent)
    case model(ViewModelEvent)
}
```

```
enum Decision: Hashable {
    case load
    case track(Event)
    case log(String)
    case showSnackbar(String)
}
```

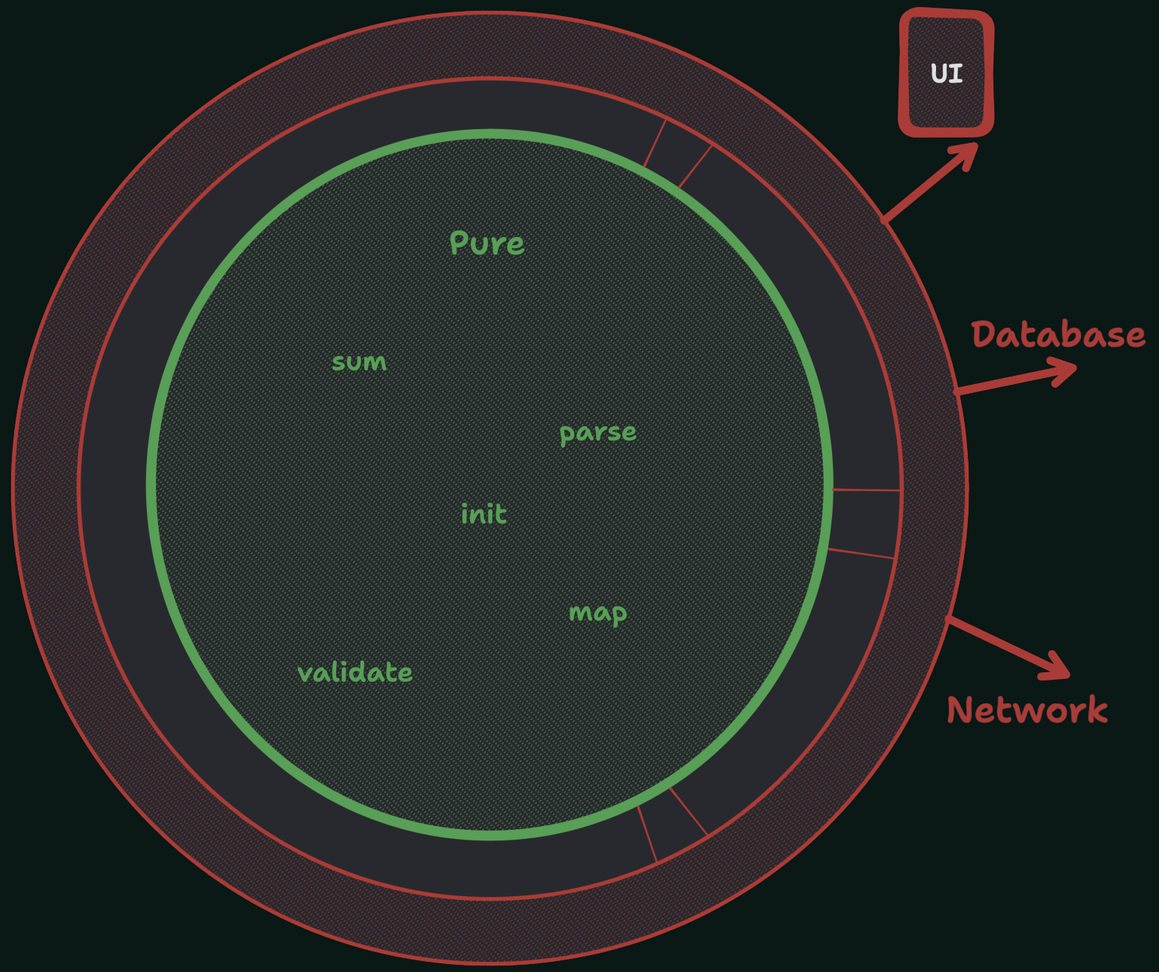
Тестирование и FCIS

```
func test_effects_onUserTap_startsLoading() async throws {
    var state = State.initial
    let decisions = FcisVM.makeDecisions(.view(.userDidTapButton), &state)

    XCTAssertEqual(state, .loading)
    XCTAssertTrue(decisions.contains(.load))  // Isolated match only on one function of our system
}

func test_effects_onFinishLoading_setsFact() async throws {
    var state = State.loading
    let decisions = FcisVM.makeDecisions(.model(.finishLoading(Fact(text: "some fact"))), &state)

    XCTAssertEqual(state, .loaded("some fact"))
    XCTAssertEqual(decisions, [.log("some fact")]) // Complete match
}
```

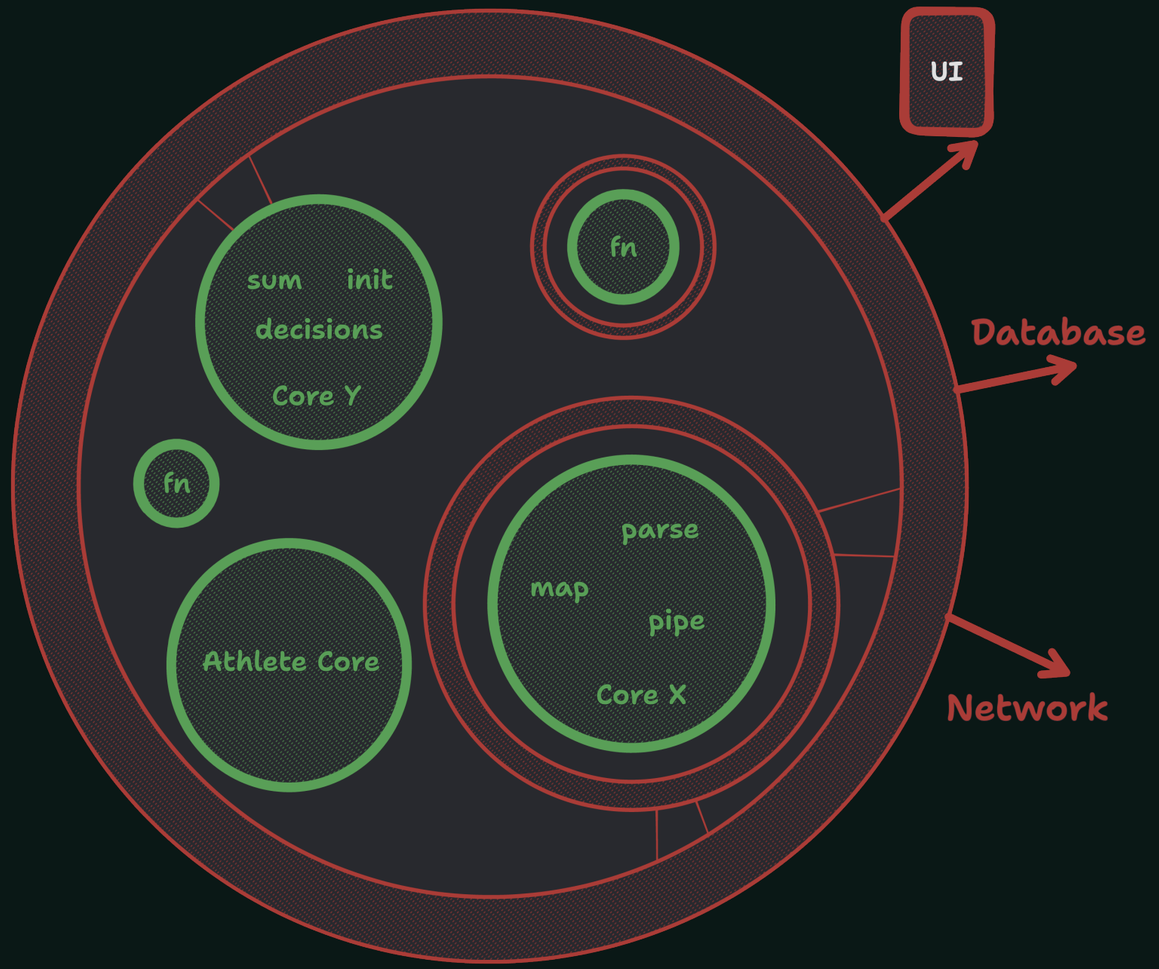
Как разделить на Core & Shell

1. Выделяем чистые функции
2. Выносим в них поведение, которое можно протестировать
3. Используем паттерн Decision Producer / Performer
4. Связываем ядро и оболочку из сайд-эффектов

FCIS



FCIS


UI
Database
Network
fn
sum init
decisions
Core Y
fn
Athlete Core
parse
map
pipe
Core X

Как тестировать в парадигме FCIS

Юнит тесты на **ядро**

Интеграционные:

1. Немного. Мало веток, много связей.
2. Не тестировать 🐚 Shell.

Спасибо

iOS разработчик из ✈️ Аааааавиасейлс ✈️

github.com/AgapovOne

Зависимости

Зависимости через протоколы

```
protocol CartRepository {
    func items() → [Item]
}

class DefaultCartRepository: CartRepository {
    func items() → [Item] { } // from DB
}

class MockCartRepository: CartRepository {
    func items() → [Item] {} // fake
}
```

Зависимости и использование

```
func test_x() {
    let repo: CartRepository
    repo = MockCartRepository(items: [])
    assert(repo.items() == [])
}

// ViewModel
let repo: CartRepository = DefaultCartRepository()
func load() {
    repo.items()
}
```

Функции!

Зависимости на функциях

```
struct Deps {
    var track: (Event) → Void
    var showSnackbar: (String) → Void
    var log: (Any ... ) → Void
    var call: () async throws → Data
}
```

Зависимости и их подмена

```
// Global? Module? Class?
var deps = Deps(
    track: { event in AnalyticsService.shared.track(event) },
    showSnackbar: { AlertManager.show($0) },
    log: { print($0) },
    getCartItems: { try await Network.cartItems() }
)

// in test:
deps.getCartItems = { [] }
```

Зависимости в локальных функциях

```
// Dependency for one method
func load(getCartItems: () async throws → [Item]) {
    try await getCartItems()
}
```

Зависимости

How to Control the World

 **Pointfree**

Take control of your dependencies

📹 **Brandon Williams (youtube)**