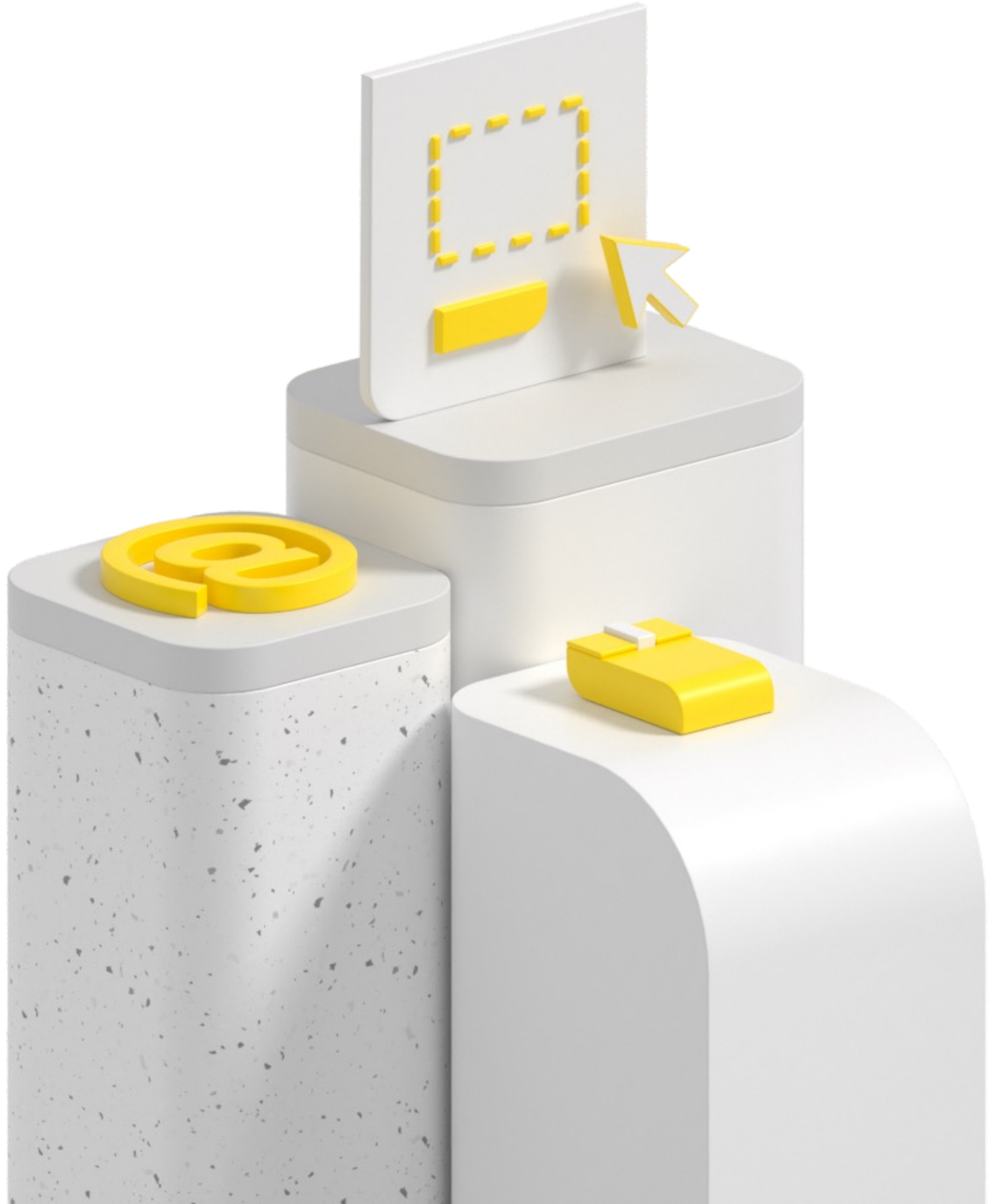
IT's TINKOFF

О дивный новый мир со Swift Concurrency



IT's TINKOFF

Володин Кирилл

volodin.kirill.a@gmail.com

Telegram: @leoniknik

План

1. **Мотивация**
2. Разработка
3. Тестирование
4. Reactive
5. Стратегия перехода на Swift Concurrency

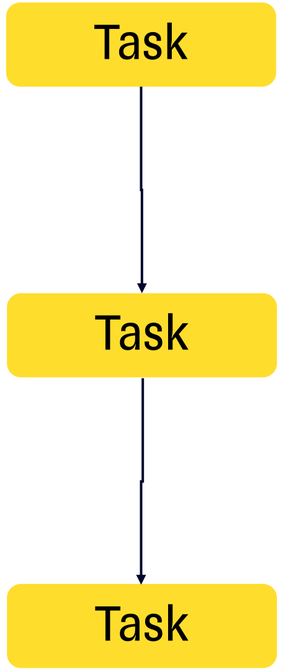
Callback hell

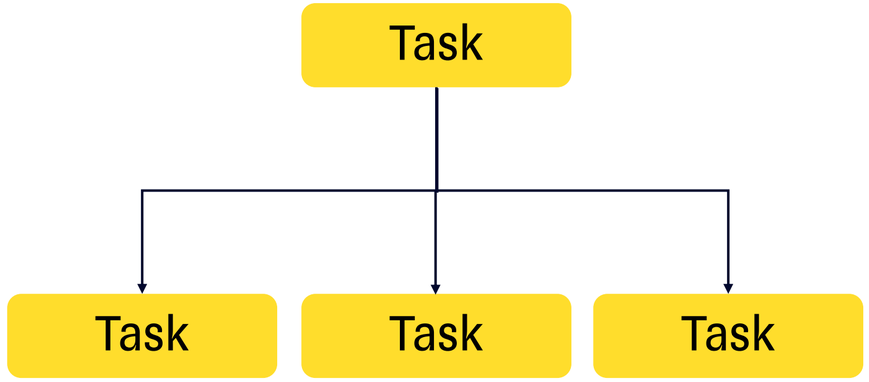
```
service.simpleFuncWithCompletion { [weak self] in
    self?.service.simpleFuncWithCompletion { [weak self] in
        self?.service.simpleFuncWithCompletion { [weak self] in
            print("success")
        }
    }
}
```

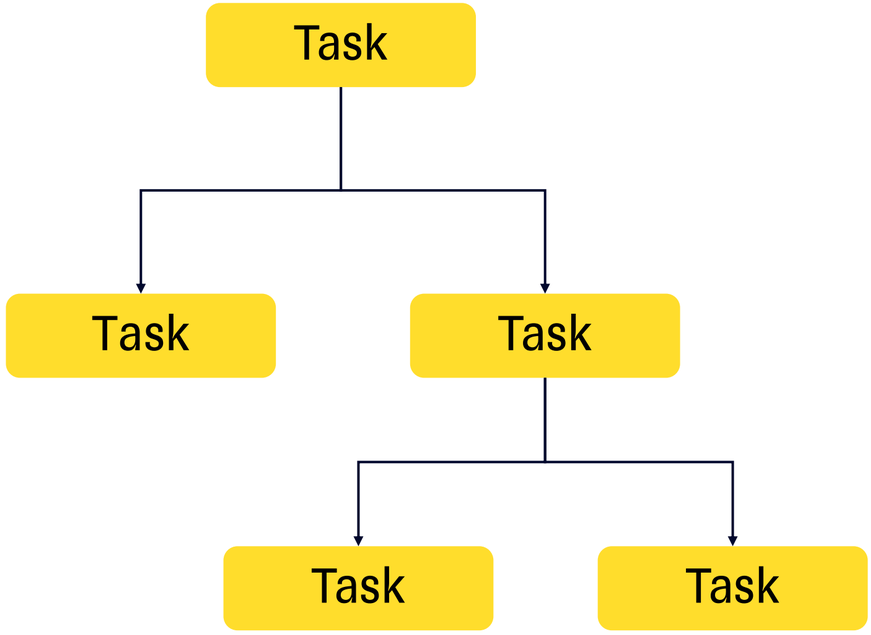
Error handling

```
service.funcWithResult { [weak self] result in
    switch result {
    case .success(_):
        self?.service.funcWithResultAndErrorCallback { [weak self] result, error in
            if let error = error {
                //.. handle error 2
            } else {
                self?.service.funcWithResult { result in
                    switch result {
                    case .success(_):
                        // ...
                    case .failure(_):
                        // handle error 3
                    }
                }
            }
    case .error(_):
        // .. handle error 1
    }
}
```

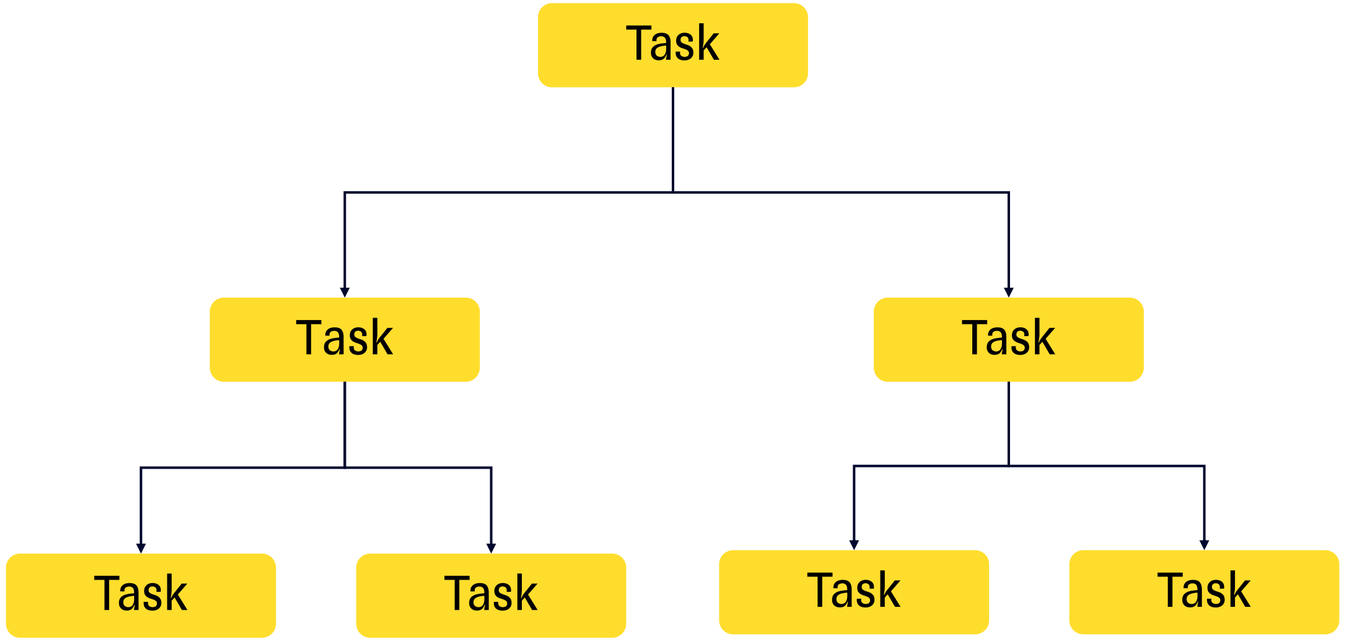
Отмена операций







Отмена операций



Неструктурность

```
func someWork(completion: @escaping () -> ()) {
    DispatchQueue.global(qos: .utility).async {
        DispatchQueue.main.async {
            completion()
        }
    }
}
```

Классические проблемы многопоточности

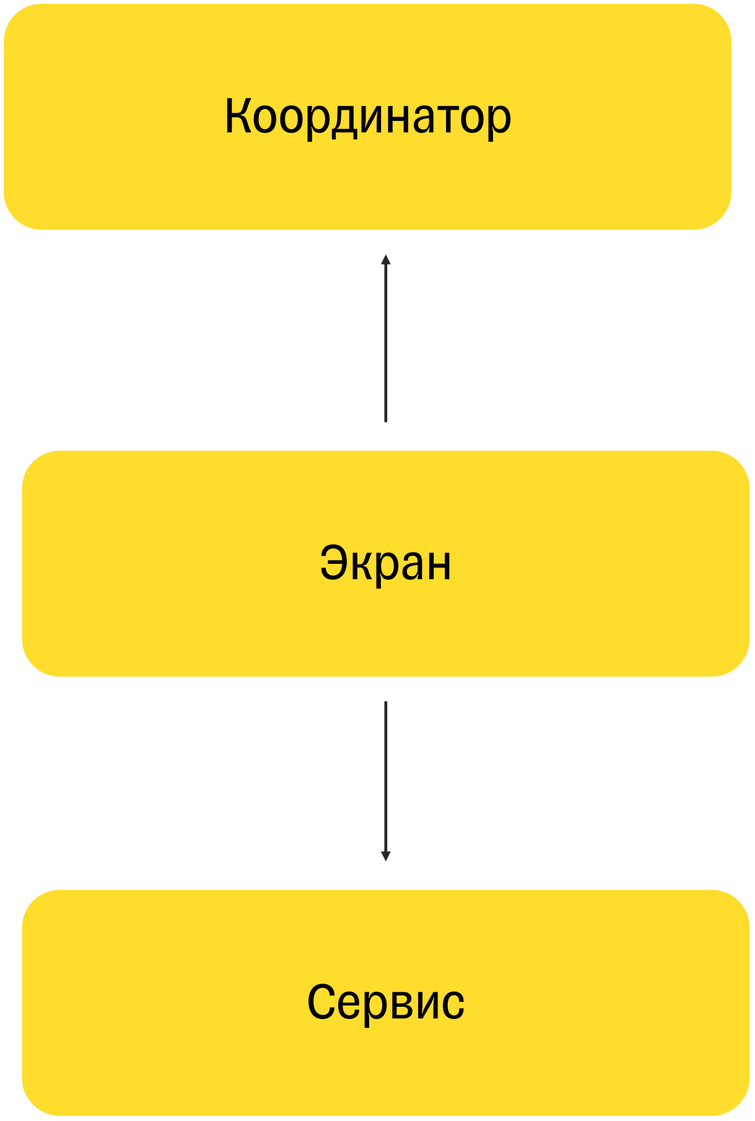
- Race condition
- Deadlock
- Priority inversion
- UI Thread corruption
- Thread explosion
- Context switching overhead
- Starvation

Swift Concurrency

План

1. Мотивация
2. **Разработка**
3. Тестирование
4. Reactive
5. Стратегия перехода на Swift Concurrency

Разработка



```
class FirstPresenter {
    // ...
}

extension FirstPresenter: IFirstPresenter {
    func viewDidLoad() {
        view?.show(isLoading: true)
        DispatchQueue.global().async { [weak self] in
            self?.service.load { result in
                self?.value = result
                DispatchQueue.main.async {
                    self?.view?.show(isLoading: false)
                }
            }
        }
    }
}
```

```
class FirstPresenter {
    // ...
}

extension FirstPresenter: IFirstPresenter {
    func viewDidLoad() {
        view?.show(isLoading: true)
        DispatchQueue.global().async { [weak self] in
            self?.service.load { result in
                self?.value = result
                DispatchQueue.main.async {
                    self?.view?.show(isLoading: false)
                }
            }
        }
    }
}
```

```
class FirstPresenter {
    // ...
}

extension FirstPresenter: IFirstPresenter {
    func viewDidLoad() {
        view?.show(isLoading: true)
        DispatchQueue.global().async { [weak self] in
            self?.service.load { result in
                self?.value = result
                DispatchQueue.main.async {
                    self?.view?.show(isLoading: false)
                }
            }
        }
    }
}
```

```
class FirstPresenter {
    // ...
}

extension FirstPresenter: IFirstPresenter {
    func viewDidLoad() {
        view?.show(isLoading: true)
        DispatchQueue.global().async { [weak self] in
            self?.service.load { result in
                self?.value = result
                DispatchQueue.main.async {
                    self?.view?.show(isLoading: false)
                }
            }
        }
    }
}
```

```swift
extension Service: IService {

    func load(completion: @escaping (Int) -> Void) {
        let request = Request()
        networkService.process(request: request) { [weak self] data in
            guard let self = self else { return }
            let result = self.parse(data: data)
            completion(result)
        }
    }
}
```

```
extension Service: IService {

    func load(completion: @escaping (Int) -> Void) {
        let request = Request()
        networkService.process(request: request) { [weak self] data in
            guard let self = self else { return }
            let result = self.parse(data: data)
            completion(result)
        }
    }
}
```

```
extension Service: IService {

    func load(completion: @escaping (Int) -> Void) {
        let request = Request()
        networkService.process(request: request) { [weak self] data in
            guard let self = self else { return }
            let result = self.parse(data: data)
            completion(result)
        }
    }
}
```

```
func viewDidLoad() {
    view?.show(isLoading: true)
    DispatchQueue.global().async { [weak self] in
        self?.service.load { result in
            self?.value = result
            DispatchQueue.main.async {
                self?.view?.show(isLoading: false)
            }
        }
    }
}
```

```
func viewDidLoad() {
    view?.show(isLoading: true)
    DispatchQueue.global().async { [weak self] in
        self?.service.load { result in
            self?.value = result
            DispatchQueue.main.async {
                self?.view?.show(isLoading: false)
            }
        }
    }
}
```

```
func viewDidLoad() async {
    view?.show(isLoading: true)
    self?.service.load { result in
        self?.value = result
        DispatchQueue.main.async {
            self?.view?.show(isLoading: false)
        }
    }
}
```

```
func viewDidLoad() async {
    view?.show(isLoading: true)
    self?.service.load { result in
        self?.value = result
        DispatchQueue.main.async {
            self?.view?.show(isLoading: false)
        }
    }
}
```

```swift
@MainActor protocol IFirstViewController: AnyObject {
    func show(isLoading: Bool)
}

func viewDidLoad() async {
    await view?.show(isLoading: true)
    self?.service.load { result in
        self?.value = result
        await self?.view?.show(isLoading: false)
    }
}
```

```swift
@MainActor protocol IFirstViewController: AnyObject {
    func show(isLoading: Bool)
}

func viewDidLoad() async {
    await view?.show(isLoading: true)
    self?.service.load { result in
        self?.value = result
        await self?.view?.show(isLoading: false)
    }
}
```

```swift
func viewDidLoad() async {
    await view?.show(isLoading: true)
    self?.service.load { result in
        self?.value = result
        // ❌ Cannot pass function of type '(Int) async -> Void'
        // to parameter expecting synchronous function type
        await self?.view?.show(isLoading: false)
    }
}
```

```
func viewDidLoad() async {
    self.service.load { [weak self] result in
        self?.value = result
    }
}
```

```swift
func viewDidLoad() async {
    self.service.load { [weak self] result in
        self?.value = result // ❌ Опасно
    }
}
```

```
extension Service: IService {

    func load(completion: @escaping (Int) -> Void) {
        let request = Request()
        networkService.process(request: request) { [weak self] data in
            guard let self = self else { return }
            let result = self.parse(data: data)
            completion(result)
        }
    }
}
```

```swift
extension NetworkService: INetworkService {

    func process(request: Request, completion: @escaping (Data) -> Void) {

        DispatchQueue.global().async {
            let result = Data()
            completion(result)
        }
    }
}
```

```
func viewDidLoad() async {
    self.service.load { [weak self] result in
        self?.value = result // ❌ Опасно
    }
}
```

@_unavailableFromAsync

```
class SomeService {
    func foo() async {
        print(Thread.current) ⚠️
        // Class property 'current' is unavailable from asynchronous contexts;
        // Thread.current cannot be used from async contexts.;
        // this is an error in Swift 6
    }
}
```

@_unavailableFromAsync

```
protocol IService {
    @_unavailableFromAsync func load(completion: @escaping (Int) -> Void)
}

func viewDidLoad() async {
    service.load { [weak self] result in
        // Instance method 'load' is unavailable from asynchronous contexts;
        // this is an error in Swift 6 ⚠️
        self?.value = result
    }
}
```

```swift
protocol IService {
    @_unavailableFromAsync func load(completion: @escaping (Int) -> Void)

    func load() async -> Int
}

extension Service: IService {

    func load() async -> Int {
        let request = Request()
        return await withCheckedContinuation { [weak self] continuation in

            self?.networkService.process(request: request) { data in
                let result = self?.parse(data: data) ?? .zero
                continuation.resume(returning: result)
            }
        }
    }
}
```

```swift
protocol IService {
    @_unavailableFromAsync func load(completion: @escaping (Int) -> Void)

    func load() async -> Int
}

extension Service: IService {

    func load() async -> Int {
        let request = Request()
        return await withCheckedContinuation { [weak self] continuation in

            self?.networkService.process(request: request) { data in
                let result = self?.parse(data: data) ?? .zero
                continuation.resume(returning: result)
            }
        }
    }
}
```

```swift
protocol IService {
    @_unavailableFromAsync func load(completion: @escaping (Int) -> Void)

    func load() async -> Int
}

extension Service: IService {

    func load() async -> Int {
        let request = Request()
        return await withCheckedContinuation { [weak self] continuation in

            self?.networkService.process(request: request) { data in
                let result = self?.parse(data: data) ?? .zero
                continuation.resume(returning: result)
            }
        }
    }
}
```

```swift
func viewDidLoad() async {
    await view?.show(isLoading: true)
    self.value = await service.load()
    await view?.show(isLoading: false)
}
```

```
actor MyActor {
    func foo() {
        print(Thread.current) ⚠️
    }
}
```

```
class FirstViewController: UIViewController {

    private let presenter: IFirstPresenter

    override func viewDidLoad() {
        super.viewDidLoad()

        Task {
            await presenter.viewDidLoad()
        }
    }
}
```

Capture list у Task

```
class ViewController: UIViewController {
    let service = Service()

    override func viewDidAppear(_ animated: Bool) {
        super.viewDidAppear(animated)

        Task { // тут нет [weak self]
            await service.someWork() // компилятор даже не просит написать self,
                                     // так как блок у Task @_implicitSelfCapture
        }
    }
}
```

Capture list y Task

```
class ViewController: UIViewController {
    let service = Service()

    var task: Task<Void, Never>?

    deinit {
        task?.cancel()
    }

    override func viewDidAppear(_ animated: Bool) {
        super.viewDidAppear(animated)

        task = Task {
            await service.someWork()
        }
    }
}
```

Capture list y Task

```
class ViewController: UIViewController {
    let service = Service()

    var task: Task<Void, Never>?

    deinit {
        task?.cancel()
    }

    override func viewDidAppear(_ animated: Bool) {
        super.viewDidAppear(animated)

        task = Task {
            await service.someWork()
        }
    }
}
```

Capture list y Task

```
class ViewController: UIViewController {
    let service = Service()

    var task: Task<Void, Never>?

    deinit {
        task?.cancel()
    }

    override func viewDidAppear(_ animated: Bool) {
        super.viewDidAppear(animated)

        task = Task {
            await service.someWork()
        }
    }
}
```

Capture list y Task

```
class ViewController: UIViewController {
    let service = Service()

    var task: Task<Void, Never>?

    deinit {
        task?.cancel()
    }

    override func viewDidAppear(_ animated: Bool) {
        super.viewDidAppear(animated)

        task = Task { [weak self] in
            await self?.service.someWork()
        }
    }
}
```

Capture list y Task

```swift
class ViewController: UIViewController {
    let service = Service()

    var task: Task<Void, Never>?

    deinit {
        task?.cancel()
    }

    override func viewDidAppear(_ animated: Bool) {
        super.viewDidAppear(animated)

        task = Task { [weak self] in
            await self?.service.someWork()
            await service.someWork() // Внимание
        }
    }
}
```

Отмена задач

```
class ViewController: UIViewController {
    let service = Service()

    var task: Task<Void, Never>? {
        willSet {
            task?.cancel()
        }
    }

    deinit {
        task?.cancel()
    }

    override func viewDidAppear(_ animated: Bool) {
        super.viewDidAppear(animated)

        task = Task { [weak self] in
            await self?.service.someWork()
        }
    }
}
```

Отмена задач

```
class ViewController: UIViewController {
    let service = Service()

    override func viewDidLoad() {
        super.viewDidLoad()

        let task = Task {
            print(Task.isCancelled) // true
            await service.someWork()
        }
        task.cancel()
    }
}

class Service {
    func someWork() async {
        print(Task.isCancelled) // true

        Task {
            print(Task.isCancelled) // false - создали новую таску, поломали structured concurrency
                                    // флаг отмены не прокинулся
        }
    }
}
```

Отмена задач

```
class ViewController: UIViewController {
    let service = Service()

    override func viewDidLoad() {
        super.viewDidLoad()

        let task = Task {
            print(Task.isCancelled) // true
            await service.someWork()
        }
        task.cancel()
    }
}

class Service {
    func someWork() async {
        print(Task.isCancelled) // true

        Task {
            print(Task.isCancelled) // false - создали новую таску, поломали structured concurrency
                                    // флаг отмены не прокинулся
        }
    }
}
```

Отмена задач

```
class ViewController: UIViewController {
    let service = Service()

    override func viewDidLoad() {
        super.viewDidLoad()

        let task = Task {
            print(Task.isCancelled) // true
            await service.someWork()
        }
        task.cancel()
    }
}

class Service {
    func someWork() async {
        print(Task.isCancelled) // true

        Task {
            print(Task.isCancelled) // false - создали новую таску, поломали structured concurrency
                                    // флаг отмены не прокинулся
        }
    }
}
```

Отмена задач

```
class ViewController: UIViewController {
    let service = Service()

    override func viewDidLoad() {
        super.viewDidLoad()

        let task = Task {
            print(Task.isCancelled) // true
            await service.someWork()
        }
        task.cancel()
    }
}

class Service {
    func someWork() async {
        print(Task.isCancelled) // true

        Task {
            print(Task.isCancelled) // false - создали новую таску, поломали structured concurrency
                                    // флаг отмены не прокинулся
        }
    }
}
```

Отмена задач

```
class Service {
    var task: Task<Void, Never>?

    func someWork() async {

        task = Task {
            await withTaskCancellationHandler {
                print(Task.isCancelled) // true
            } onCancel: { [task] in
                task?.cancel()
            }
        }
    }
}
```

Отмена задач

- Кооперативная отмена дает простой API
- Отмену нужно поддержать - Task.isCanceled или try Task.checkCancellation
- Старайтесь использовать try Task.checkCancellation
- Отмену необходимо закладывать в API методов (nil, error, partial result)

Отмена задач

- Объединение в кооперативную отмену средства Swift Concurrency и другие инструменты
- Единая отмена группы независимых задач

Нюансы @MainActor и других globalActor

```
class Coordinator: PresenterOutput {

    func handle() {
        print(Thread.isMainThread)
    }
}

protocol PresenterOutput: AnyObject {
    func handle()
}

class Presenter {
    weak var output: PresenterOutput?

    func go() async {
        output?.handle()
    }
}
```

Нюансы @MainActor и других globalActor

```
class Coordinator: PresenterOutput {

    func handle() {
        print(Thread.isMainThread)
    }
}

protocol PresenterOutput: AnyObject {
    func handle()
}

class Presenter {
    weak var output: PresenterOutput?

    func go() async {
        output?.handle()
    }
}
```

Нюансы @MainActor и других globalActor

```
class Coordinator: PresenterOutput {

    func handle() {
        print(Thread.isMainThread)
    }
}

protocol PresenterOutput: AnyObject {
    func handle()
}

class Presenter {
    weak var output: PresenterOutput?

    func go() async {
        output?.handle()
    }
}
```

Нюансы @MainActor и других globalActor

```
class Coordinator: PresenterOutput {

    func handle() {
        print(Thread.isMainThread)
    }
}

protocol PresenterOutput: AnyObject {
    func handle()
}

class Presenter {
    @MainActor weak var output: PresenterOutput?

    func go() async {
        output?.handle()
    }
}
```

Нюансы @MainActor и других globalActor

```swift
@MainActor class Coordinator: PresenterOutput {

    func handle() {
        print(Thread.isMainThread)
    }
}

protocol PresenterOutput: AnyObject {
    func handle()
}

class Presenter {
    weak var output: PresenterOutput?

    func go() async {
        output?.handle()
    }
}
```

Нюансы @MainActor и других globalActor

```swift
@MainActor class Coordinator: PresenterOutput {
    // Main actor-isolated instance method 'handle()' ⚠️
    // cannot be used to satisfy nonisolated protocol requirement
    func handle() {
        print(Thread.isMainThread)
    }
}

protocol PresenterOutput: AnyObject {
    func handle()
}

class Presenter {
    weak var output: PresenterOutput?

    func go() async {
        output?.handle()
    }
}
```

Нюансы @MainActor и других globalActor

```
class Coordinator: PresenterOutput {

    @MainActor func handle() { ⚠️
        print(Thread.isMainThread)
    }
}

protocol PresenterOutput: AnyObject {
    func handle()
}

class Presenter {
    weak var output: PresenterOutput?

    func go() async {
        output?.handle()
    }
}
```

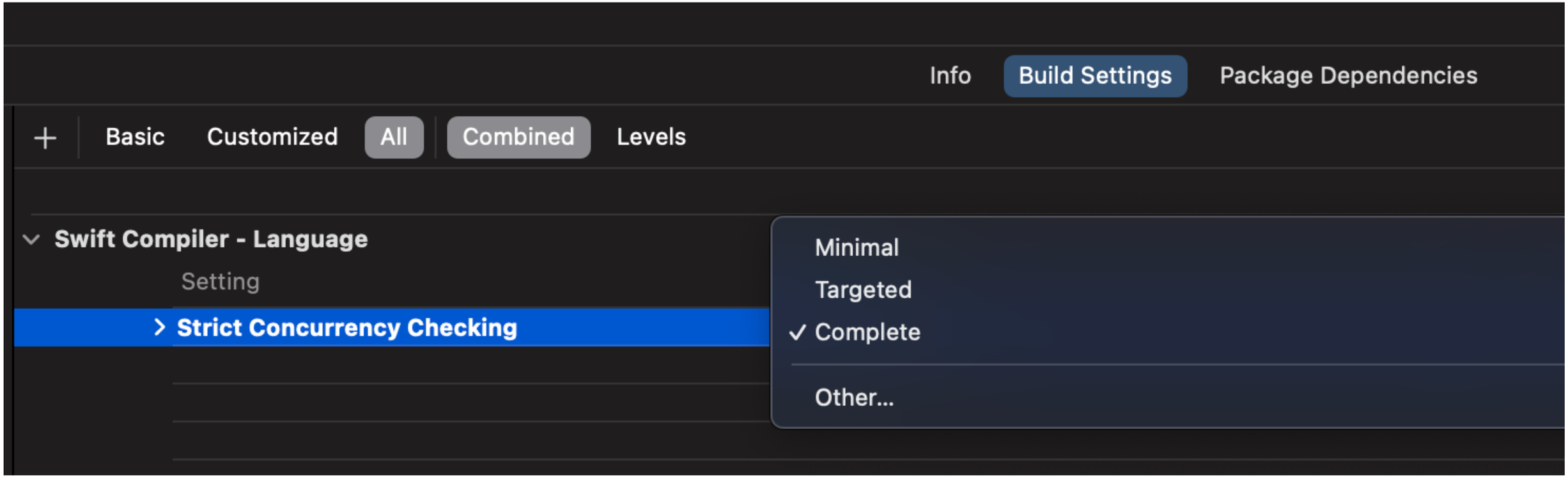
Нюансы @MainActor и других globalActor

```
@MainActor class Coordinator: PresenterOutput {

    func handle() {
        print(Thread.isMainThread)
    }
}

protocol PresenterOutput: AnyObject {
    func handle() async
}

class Presenter {
    weak var output: PresenterOutput?

    func go() async {
        await output?.handle()
    }
}
```

Нюансы @MainActor и других globalActor

```
class Coordinator: PresenterOutput {

    func handle() {
        print(Thread.isMainThread)
    }
}

@MainActor protocol PresenterOutput: AnyObject {
    func handle()
}

class Presenter {
    weak var output: PresenterOutput?

    func go() async {
        await output?.handle()
    }
}
```

Swift Concurrency Build Setting



Sendable

```
class SomeClass {
    var value = 1
}

class Service {
    func foo() async {
        let some = SomeClass()
        Task {
            some.value = 2
        }
    }
}
```

Sendable

```
class SomeClass {
    var value = 1
}

class Service {
    func foo() async {
        let some = SomeClass()
        Task {
            some.value = 2 // ⚠️ Capture of 'some' with non-sendable type 'SomeClass'
                           // in a `@Sendable` closure
        }
    }
}
```

Sendable

```
actor SomeClass {
    var value = 1

    func update(_ value: Int) {
        self.value = value
    }
}

class Service {
    func foo() async {
        let some = SomeClass()
        Task {
            await some.update(2)
        }
    }
}
```

Sendable

```
final class SomeClass: Sendable {
    var value = 1
}

class Service {
    func foo() async {
        let some = SomeClass()
        Task {
            some.value = 2
        }
    }
}
```

Sendable

```
final class SomeClass: Sendable {
    ⚠️ // Stored property 'value' of 'Sendable'-conforming class 'SomeClass' is mutable
    var value = 1
}

class Service {
    func foo() async {
        let some = SomeClass()
        Task {
            some.value = 2
        }
    }
}
```

Sendable

```swift
public final class Atomic<T>: @unchecked Sendable {
    private let lock = NSLock()
    private var _value: T

    public init(_ value: T) {
        self._value = value
    }

    public var value: T {
        lock.lock()
        defer { lock.unlock() }
        return _value
    }

    public func mutate<R>(_ transform: (inout T) throws -> R) rethrows -> R {
        lock.lock()
        defer { lock.unlock() }
        return try transform(&_value)
    }
}
```

Sendable

```
class SomeClass {
    var value = 1
}

class Service {
    func foo() async {
        let some = Atomic(1)
        Task {
            some.mutate { $0 = 2 }
        }
    }
}
```

Sendable

```
// Внутри модуля SomeFeature
public class SomeClass {
    public var value = 1

    public init() {
        //...
    }
}

import SomeFeature

func foo() async {
    let some = SomeClass()
    Task {
        some.value = 2
    }
}
```

Sendable

```
// Внутри модуля SomeFeature
public class SomeClass {
    public var value = 1

    public init() {
        //...
    }
}
```

```
import SomeFeature

func foo() async {
    let some = SomeClass()
    Task {
        some.value = 2
    }
}
```

Sendable

```
// Внутри модуля SomeFeature
public class SomeClass {
    public var value = 1

    public init() {
        //...
    }
}
```

```
import SomeFeature

func foo() async {
    let some = SomeClass()
    Task {
        some.value = 2
    }
}
```

Sendable

```
// Внутри модуля SomeFeature
public class SomeClass {
    public var value = 1

    public init() {
        //...
    }
}

import SomeFeature

func foo() async {
    let some = SomeClass()
    Task {
        some.value = 2 ⚠️
    }
}
```

Sendable

```
// Внутри модуля SomeFeature
public class SomeClass {
    public var value = 1

    public init() {
        //...
    }
}

@preconcurrency import SomeFeature

func foo() async {
    let some = SomeClass()
    Task {
        some.value = 2
    }
}
```

Sendable

```
class SomeClass {
    var value = 1
}

func foo() async {
    let some = SomeClass()
    DispatchQueue.global().async { @Sendable in
        some.value = 2 ⚠️
    }
}
```

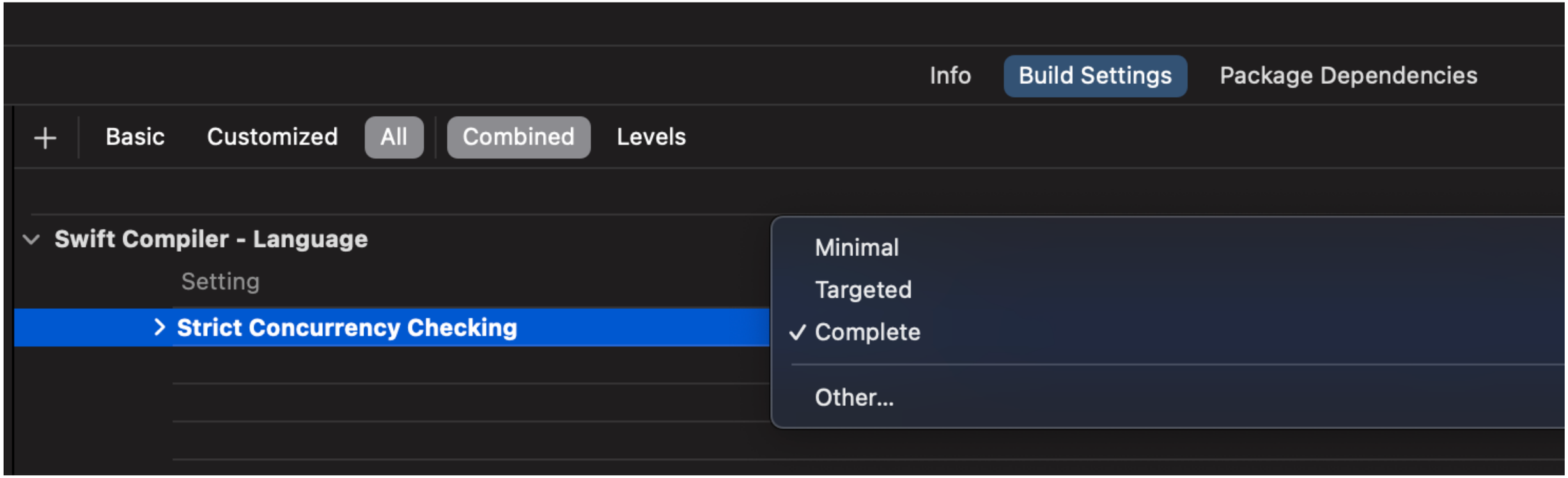
Swift Concurrency Build Setting

```swift
class Coordinator {
    private let navigationController: UINavigationController
    private let moduleAssembly: IModuleAssembly

    init(navigationController: UINavigationController, moduleAssembly: IModuleAssembly) {
        self.navigationController = navigationController
        self.moduleAssembly = moduleAssembly
    }

    func pushVC() {
        let viewController = moduleAssembly.assemble()
        navigationController.pushViewController(viewController, animated: true)
    }
}
```

Swift Concurrency Build Setting

```swift
class Coordinator {
    private let navigationController: UINavigationController
    private let moduleAssembly: IModuleAssembly

    init(navigationController: UINavigationController, moduleAssembly: IModuleAssembly) {
        self.navigationController = navigationController
        self.moduleAssembly = moduleAssembly
    }

    func pushVC() {
        let viewController = moduleAssembly.assemble()
        navigationController.pushViewController(viewController, animated: true)
    }
}
```

Включаем complete



Swift Concurrency Build Setting

```
class Coordinator {
    private let navigationController: UINavigationController
    private let moduleAssembly: IModuleAssembly

    init(navigationController: UINavigationController, moduleAssembly: IModuleAssembly) {
        self.navigationController = navigationController
        self.moduleAssembly = moduleAssembly
    }

    func pushVC() {
        let viewController = moduleAssembly.assemble()
        navigationController.pushViewController(viewController, animated: true) ❌
        // Call to main actor-isolated instance method 'pushViewController(_:animated:)'
        // in a synchronous nonisolated context
    }
}
```

Swift Concurrency Build Setting

```swift
@MainActor class Coordinator {
    private let navigationController: UINavigationController
    private let moduleAssembly: IModuleAssembly

    init(navigationController: UINavigationController, moduleAssembly: IModuleAssembly) {
        self.navigationController = navigationController
        self.moduleAssembly = moduleAssembly
    }

    func pushVC() {
        let viewController = moduleAssembly.assemble()
        navigationController.pushViewController(viewController, animated: true)
    }
}
```

План

1. Мотивация
2. Разработка
3. **Тестирование**
4. Reactive
5. Стратегия перехода на Swift Concurrency

Тестирование

```swift
@MainActor protocol IView: AnyObject {
    func setupTitle(_ title: String)
}

class ViewController: UIViewController, IView {
    var presenter: IPresenter

    // ...

    override func viewDidLoad() {
        super.viewDidLoad()
        presenter.viewDidLoad()
    }

    func setupTitle(_ title: String) {
        // ...
    }
}
```

Тестирование

```
protocol ITitleFactory {
    func obtainTitle() async -> String
}

class TitleFactory: ITitleFactory {
    func obtainTitle() async -> String {
        return "Title"
    }
}
```

Тестирование

```
class Presenter: IPresenter {

    weak var view: IView?
    private let titleFactory: ITitleFactory

    init(titleFactory: ITitleFactory) {
        self.titleFactory = titleFactory
    }

    func viewDidLoad() {
        setupTitle()
    }

    func setupTitle() {
        Task {
            let title = await titleFactory.obtainTitle()
            await view?.setupTitle(title)
        }
    }
}
```

Тестирование

```
class Presenter: IPresenter {

    weak var view: IView?
    private let titleFactory: ITitleFactory

    init(titleFactory: ITitleFactory) {
        self.titleFactory = titleFactory
    }

    func viewDidLoad() {
        setupTitle()
    }

    func setupTitle() {
        Task {
            let title = await titleFactory.obtainTitle()
            await view?.setupTitle(title)
        }
    }
}
```

Тестирование

```swift
func testPresenter() {
    // given
    titleFactoryMock.stubbedTitle = "Test"

    // when
    presenter.viewDidLoad()

    // then
    let invokedTitle = view.invokedTitle
    XCTAssertEqual(titleFactoryMock.stubbedTitle, invokedTitle)
}
```

Тестирование

```
func testPresenter() {
    // given
    titleFactoryMock.stubbedTitle = "Test"

    // when
    presenter.viewDidLoad()

    // then
    let invokedTitle = view.invokedTitle
    XCTAssertEqual(titleFactoryMock.stubbedTitle, invokedTitle)
}
```

Тестирование

```
func testPresenter() {
    // given
    titleFactoryMock.stubbedTitle = "Test"

    // when
    presenter.viewDidLoad()

    // then
    let invokedTitle = view.invokedTitle
    XCTAssertEqual(titleFactoryMock.stubbedTitle, invokedTitle)
}
```

Тестирование

```
func testPresenter() {
    // given
    titleFactoryMock.stubbedTitle = "Test"

    // when
    presenter.viewDidLoad()

    // then
    let invokedTitle = view.invokedTitle
    XCTAssertEqual(titleFactoryMock.stubbedTitle, invokedTitle)
}
```

Тестирование

```
func testPresenter() {
    // given
    titleFactoryMock.stubbedTitle = "Test"

    // when
    presenter.viewDidLoad()

    // then
    let invokedTitle = view.invokedTitle
    XCTAssertEqual(titleFactoryMock.stubbedTitle, invokedTitle) ❌❌❌
}
```

Task.yield

```
func testPresenter() async {
    // given
    titleFactoryMock.stubbedTitle = "Test"

    // when
    presenter.viewDidLoad()
    await Task.yield()

    // then
    let invokedTitle = view.invokedTitle
    XCTAssertEqual(titleFactoryMock.stubbedTitle, invokedTitle)
}
```

Тестирование

```
public protocol ITask {
    var finished: Any { get async throws }
}

extension Task: ITask {
    public var finished: Any {
        get async throws {
            try await value
        }
    }
}
```

Тестирование

```
protocol ITaskFactory {
    func task<Success>(
        priority: TaskPriority?,
        @_inheritActorContext operation: @Sendable @escaping () async throws -> Success
    ) -> Task<Success, Error>
}

struct TaskFactory: ITaskFactory {

    func task<Success>(
        priority: TaskPriority?,
        @_inheritActorContext operation: @Sendable @escaping () async throws -> Success
    ) -> Task<Success, Error> {

        Task(priority: priority) {
            try await operation()
        }
    }
}
```

Тестирование

```
protocol ITaskFactory {
    func task<Success>(
        priority: TaskPriority?,
        @_inheritActorContext operation: @Sendable @escaping () async throws -> Success
    ) -> Task<Success, Error>
}

struct TaskFactory: ITaskFactory {

    func task<Success>(
        priority: TaskPriority?,
        @_inheritActorContext operation: @Sendable @escaping () async throws -> Success
    ) -> Task<Success, Error> {

        Task(priority: priority) {
            try await operation()
        }
    }
}
```

Тестирование

```
class Presenter: IPresenter {

    weak var view: IView?
    private let titleFactory: ITitleFactory
    private let taskFactory: ITaskFactory

    // ..

    func setupTitle() {
        taskFactory.task { [weak self] in
            guard let self = self else { return }
            let title = await self.titleFactory.obtainTitle()
            await self.view?.setupTitle(title)
        }
    }
}
```

```swift
class TestTaskFactory: ITaskFactory {
    private let lock = NSLock()
    private var tasks: [ITask] = []

    private var firstTask: ITask? {
        lock.lock(); defer { lock.unlock() }
        return tasks.first
    }

    func task<Success>(priority: TaskPriority?, operation: @escaping @Sendable () async throws -> Success) -> Task<Success, Error> {
        lock.lock()
        let task = Task(priority: priority) {
            try await operation()
        }
        tasks.append(task)
        lock.unlock()
        return task
    }

    func runUntilIdle() async throws {
        while let task = firstTask {
            _ = try await task.finished
            lock.lock(); defer { lock.unlock() }
            tasks.removeFirst()
        }
    }
}
```

```swift
class TestTaskFactory: ITaskFactory {
    private let lock = NSLock()
    private var tasks: [ITask] = []

    private var firstTask: ITask? {
        lock.lock(); defer { lock.unlock() }
        return tasks.first
    }

    func task<Success>(priority: TaskPriority?, operation: @escaping @Sendable () async throws -> Success) -> Task<Success, Error> {
        lock.lock()
        let task = Task(priority: priority) {
            try await operation()
        }
        tasks.append(task)
        lock.unlock()
        return task
    }

    func runUntilIdle() async throws {
        while let task = firstTask {
            _ = try await task.finished
            lock.lock(); defer { lock.unlock() }
            tasks.removeFirst()
        }
    }
}
```

```swift
class TestTaskFactory: ITaskFactory {
    private let lock = NSLock()
    private var tasks: [ITask] = []

    private var firstTask: ITask? {
        lock.lock(); defer { lock.unlock() }
        return tasks.first
    }

    func task<Success>(priority: TaskPriority?, operation: @escaping @Sendable () async throws -> Success) -> Task<Success, Error> {
        lock.lock()
        let task = Task(priority: priority) {
            try await operation()
        }
        tasks.append(task)
        lock.unlock()
        return task
    }

    func runUntilIdle() async throws {
        while let task = firstTask {
            _ = try await task.finished
            lock.lock(); defer { lock.unlock() }
            tasks.removeFirst()
        }
    }
}
```

```swift
class TestTaskFactory: ITaskFactory {
    private let lock = NSLock()
    private var tasks: [ITask] = []

    private var firstTask: ITask? {
        lock.lock(); defer { lock.unlock() }
        return tasks.first
    }

    func task<Success>(priority: TaskPriority?, operation: @escaping @Sendable () async throws -> Success) -> Task<Success, Error> {
        lock.lock()
        let task = Task(priority: priority) {
            try await operation()
        }
        tasks.append(task)
        lock.unlock()
        return task
    }

    func runUntilIdle() async throws {
        while let task = firstTask {
            _ = try await task.finished
            lock.lock(); defer { lock.unlock() }
            tasks.removeFirst()
        }
    }
}
```

Тестирование

```swift
func testPresenter() async throws {
    // given
    titleFactoryMock.stubbedTitle = "Test"

    // when
    presenter.viewDidLoad()

    try await taskFactory.runUntilIdle() // ждем выполнение всех задач

    // then
    let invokedTitle = view.invokedTitle
    XCTAssertEqual(titleFactoryMock.stubbedTitle, invokedTitle)
}
```

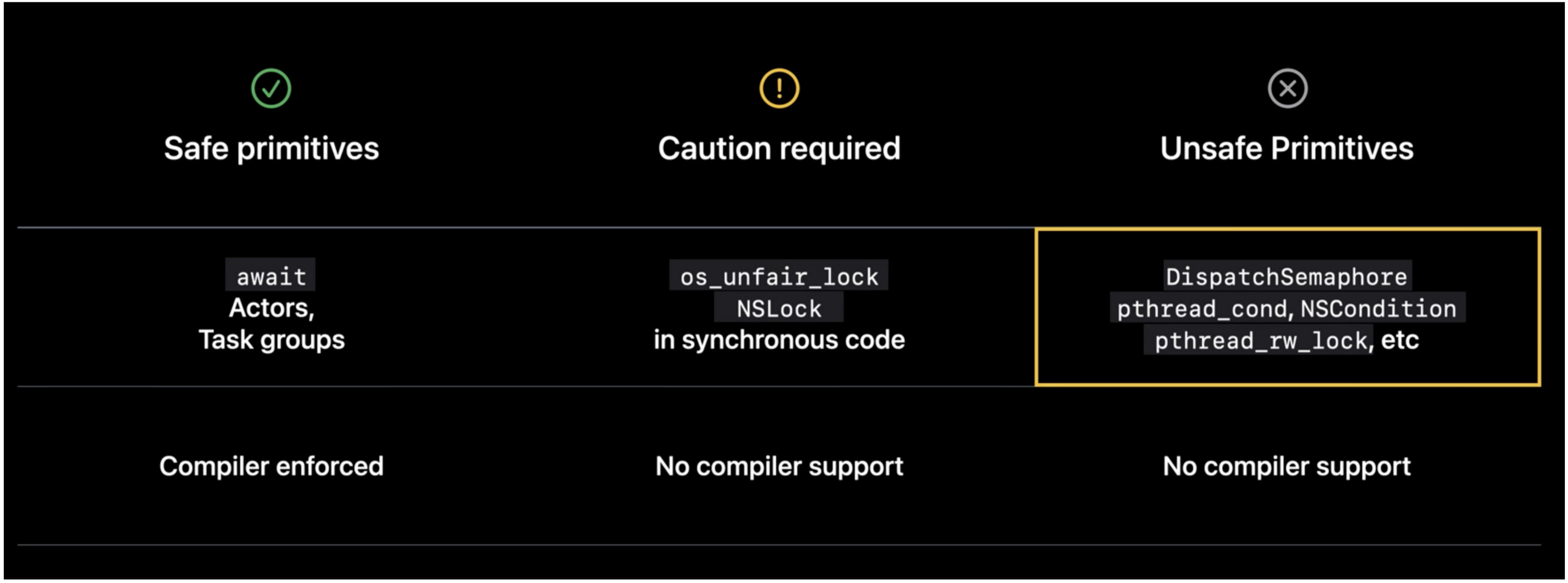
Тестирование

```
@MainActor func testSnapshot() async {
    // ..
}
```

Средства синхронизации

```
class TestTaskFactory: ITaskFactory {
    // ..

    func task<Success>(
        priority: TaskPriority?,
        operation: @escaping @Sendable () async throws -> Success
    ) -> Task<Success, Error> {

        lock.lock()
        let task = Task(priority: priority) {
            try await operation()
        }
        tasks.append(task)
        lock.unlock()
        return task
    }

    // ..
}
```

Средства синхронизации



Средства синхронизации

```
class Service {
    let lock = NSLock()

    func foo() async {
        lock.lock()
        await bar()
        lock.unlock()
    }

    func bar() async {
        // ..
    }
}
```

Средства синхронизации

```
class Service {
    let lock = NSLock()

    func foo() async {
        lock.lock()
        bar()
        lock.unlock()
    }

    func bar() {
        // ..
    }
}
```

NSLock + access

```swift
extension NSLock {
    func access<R>(_ body: () throws -> R) rethrows -> R {
        defer { self.unlock() }
        self.lock()
        return try body()
    }
}
```

План

1. Мотивация
2. Разработка
3. Тестирование
4. **Reactive**
5. Стратегия перехода на Swift Concurrency

AsyncStream

```
class StreamService {

    var stream: AsyncStream<Int> {
        AsyncStream { continuation in
            for i in 0 ... 100000 {
                continuation.yield(i)
            }
            continuation.finish()
        }
    }
}
```

AsyncStream

```
class Presenter {
    let service: StreamService

    func viewDidLoad() async {
        Task { [service] in
            for try await value in service.stream {
                // обработка
            }
        }
    }
}
```

AsyncStream

```
class Presenter {
    let service: StreamService

    func viewDidLoad() async {
        Task { [service] in
            for try await value in service.stream { // ⚠️ Поток может быть бесконечным
                // обработка
            }
        }
    }
}
```

AsyncStream

```
class Presenter {
    let service: Service
    var task: Task<Void, Error>?

    deinit { task?.cancel() }

    func viewDidLoad() async {
        task = Task { [service]
            for try await value in service.stream {
                // обработка
            }
        }
    }
}
```

Подписка на AsyncStream

```
let service: StreamService

Task {
    for try await value in service.stream {
        print(value)
    }
}

Task {
    for try await value in service.stream {
        print(value)
    }
}
```

Подписка на AsyncStream

```
let service: StreamService

Task {
    for try await value in service.stream {
        print(value) // 0, 2, 4, 6
    }
}

Task {
    for try await value in service.stream {
        print(value) // 1, 3, 5, 7
    }
}
```

Swift async algorithms

swift-async-algorithms

Swift Async Algorithms is an open-source package of asynchronous sequence and advanced algorithms that involve concurrency, along with their related types.

This package has three main goals:

- First-class integration with `async/await`
- Provide a home for time-based algorithms
- Be cross-platform and open source

Motivation

AsyncAlgorithms is a package for algorithms that work with *values over time*. That includes those primarily about *time*, like `debounce` and `throttle`, but also algorithms about *order* like `combineLatest` and `merge`. Operations that work with multiple inputs (like `zip` does on `Sequence`) can be surprisingly complex to implement, with subtle behaviors and many edge cases to consider. A shared package can get these details correct, with extensive testing and documentation, for the benefit of all Swift apps.

Swift async algorithms

Combine

```
class Service {
    lazy var stream: AnyPublisher<Int, Never> = {
        // ..
    }()
}

Task {
    for await value in service.stream.values {
        //
    }
}
```

Combine

```
class Service {
    lazy var stream: AnyPublisher<Int, Never> = {
        // ..
    }()
}

Task {
    for await value in service.stream.values {
        //
    }
}
```

Combine

```
for await value in service.stream.removeDuplicates().map { 0 * 2 }.values {
    //
}
```

Combine

```swift
extension Publisher where Self.Failure == Never {

    @available(macOS 12.0, iOS 15.0, tvOS 15.0, watchOS 8.0, *)
    public var values: AsyncPublisher<Self> { get }
}
```

AsyncSequence и операторы

```
let sequence = [1, 2, 3, 4]

for await value in sequence {
    print(value)
}
```

AsyncSequence и операторы

```
let sequence = [1, 2, 3, 4]

for await value in sequence { // ❌ Это не AsyncSequence
    print(value)
}
```

AsyncSequence и операторы

```
let sequence = [1, 2, 3, 4]

for await value in sequence.async {
    print(value)
}
```

AsyncLazySequence

```
struct AsyncLazySequence: AsyncSequence {

}
```

AsyncLazySequence

```
struct AsyncLazySequence: AsyncSequence {

    typealias AsyncIterator = ...
    typealias Element = ...

}
```

AsyncLazySequence

```
struct AsyncLazySequence<Base: Sequence>: AsyncSequence {

    typealias AsyncIterator = ...
    typealias Element = Base.Element

    private var base: Base

    init(_ base: Base) {
        self.base = base
    }
}
```

AsyncLazySequence

```
struct AsyncLazySequence<Base: Sequence>: AsyncSequence {

    typealias AsyncIterator = ...
    typealias Element = Base.Element

    private var base: Base

    init(_ base: Base) {
        self.base = base
    }
}
```

AsyncLazySequence

```
struct AsyncLazySequence<Base: Sequence>: AsyncSequence {

    typealias AsyncIterator = ...
    typealias Element = Base.Element

    private var base: Base

    init(_ base: Base) {
        self.base = base
    }

    struct Iterator: AsyncIteratorProtocol {
    }
}
```

AsyncLazySequence

```
struct AsyncLazySequence<Base: Sequence>: AsyncSequence {

    typealias AsyncIterator = ...
    typealias Element = Base.Element

    private var base: Base

    init(_ base: Base) {
        self.base = base
    }

    struct Iterator: AsyncIteratorProtocol {

        typealias Element = ...

        mutating func next() async throws -> Element? {
            ...
        }
    }
}
```

AsyncLazySequence

```
struct AsyncLazySequence<Base: Sequence>: AsyncSequence {

    typealias AsyncIterator = ...
    typealias Element = Base.Element

    private var base: Base

    init(_ base: Base) {
        self.base = base
    }

    struct Iterator: AsyncIteratorProtocol {
        typealias Element = Base.Element
        var base: Base.Iterator

        mutating func next() async throws -> Element? {
            guard !Task.isCancelled else { return nil }
            return base.next()
        }
    }
}
```

AsyncLazySequence

```
struct AsyncLazySequence<Base: Sequence>: AsyncSequence {
    typealias AsyncIterator = Iterator
    typealias Element = Base.Element

    private var base: Base

    init(_ base: Base) {
        self.base = base
    }

    func makeAsyncIterator() -> Iterator {
        Iterator(base: base.makeIterator())
    }

    struct Iterator: AsyncIteratorProtocol {
        typealias Element = Base.Element
        var base: Base.Iterator

        mutating func next() async throws -> Element? {
            guard !Task.isCancelled else { return nil }
            return base.next()
        }
    }
}
```

AsyncLazySequence

```swift
extension Sequence {
    var async: AsyncLazySequence<Self> {
        AsyncLazySequence(self)
    }
}

let sequence = [1, 2, 3, 4].async

for await value in sequence {
    print(value)
}
```

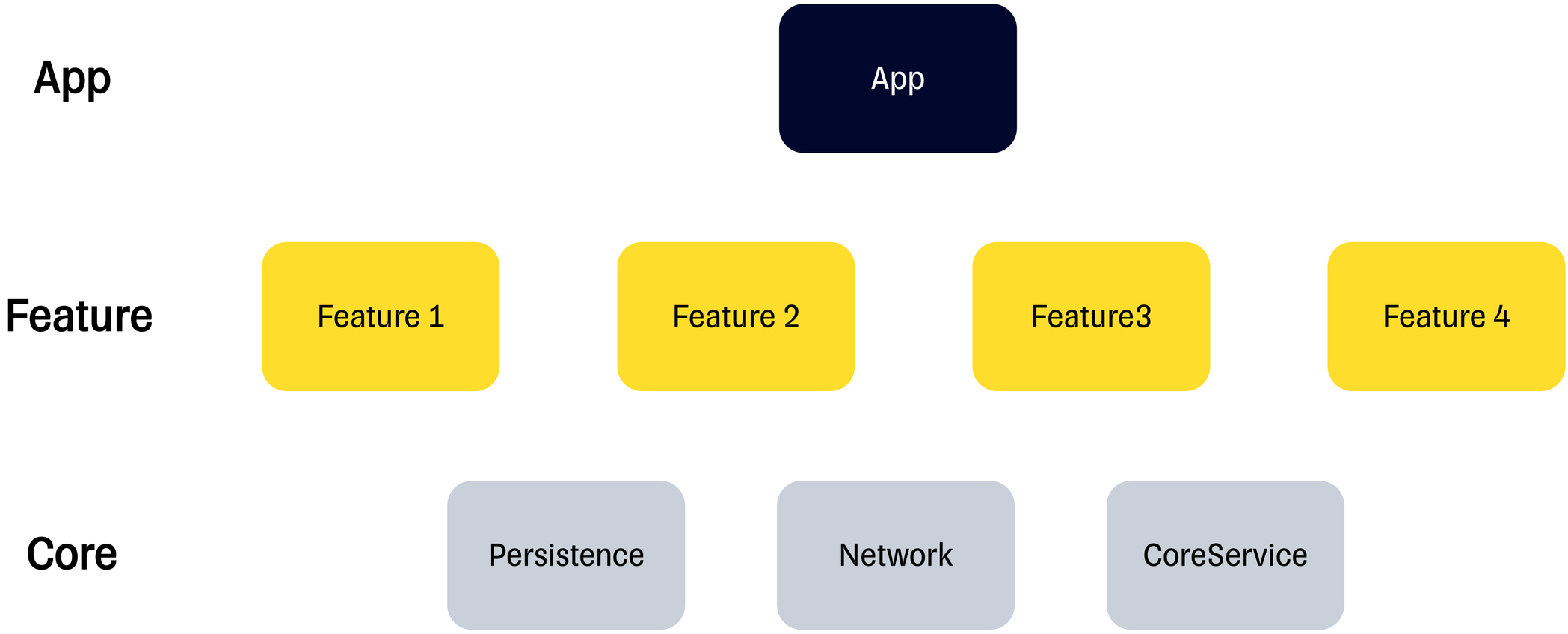
AsyncLazySequence

```
extension Sequence {
    var async: AsyncLazySequence<Self> {
        AsyncLazySequence(self)
    }
}

let sequence = [1, 2, 3, 4].async

for await value in sequence {
    print(value)
}
```

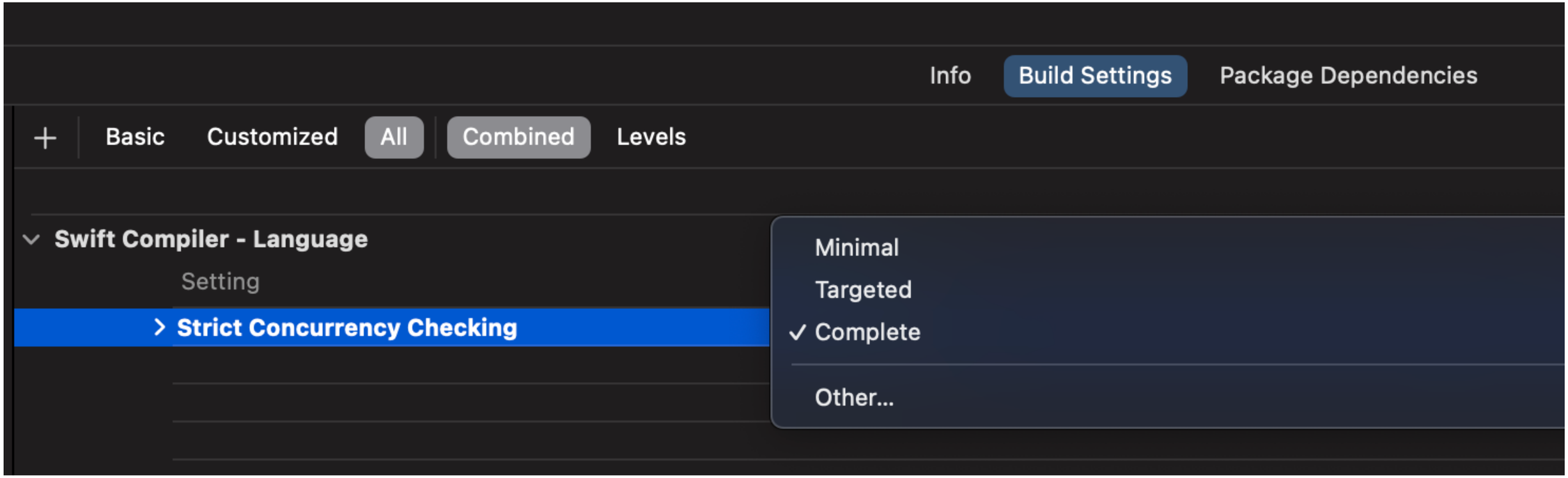
План

- Мотивация
- Разработка приложения
- Тестирование
- Reactive
- **Стратегия перехода на Swift Concurrency**

Иерархия кода



Swift Concurrency Checking



Небезопасный код

```
protocol IService {
    @_unavailableFromAsync func load(completion: @escaping (Int) -> Void)
}

func viewDidLoad() async {
    service.load { [weak self] result in
        // Instance method 'load' is unavailable from asynchronous contexts;
        // this is an error in Swift 6
        self?.value = result
    }
}
```

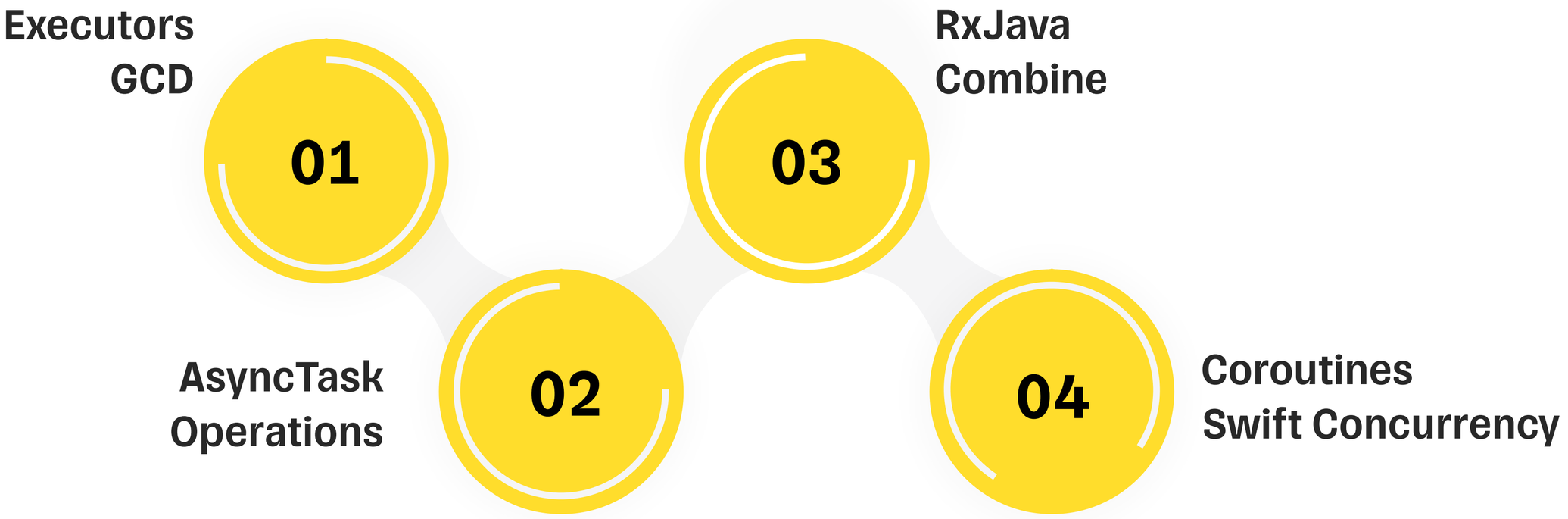
Core код

```swift
@available(macOS 12.0, iOS 15.0, tvOS 15.0, watchOS 8.0, *)
extension NSManagedObjectContext {

    func perform<T>(
        schedule: NSManagedObjectContext.ScheduledTaskType = .immediate,
        _ block: @escaping () throws -> T
    ) async rethrows -> T
}
```

Итоги

1. Мотивация
2. Разработка приложения
3. Тестирование
4. Reactive
5. Стратегия перехода на Swift Concurrency

История и будущее



- Basics
- Concepts
- Multiplatform development
- Platforms
- Standard library
- Official libraries
  - Coroutines (kotlinx.coroutines)
    - Coroutines guide
    - Coroutines basics
    - Coroutines and channels - tutorial
    - Cancellation and timeouts
    - Composing suspending functions
    - Coroutine context and dispatchers
    - Asynchronous Flow
    - Channels
    - Coroutine exceptions handling
    - Shared mutable state and concurrency
    - Select expression (experimental)
    - Debug coroutines using IntelliJ IDEA – tutorial

Flows

Using the `List<Int>` result type, means we can only return all the values at once. To represent the stream of values that are being computed asynchronously, we can use a `Flow<Int>` ↗ type just like we would use a `Sequence<Int>` type for synchronously computed values:

```
fun simple(): Flow<Int> = flow { // flow builder
    for (i in 1..3) {
        delay(100) // pretend we are doing something useful here
        emit(i) // emit next value
    }
}

fun main() = runBlocking<Unit> {
    // Launch a concurrent coroutine to check if the main thread is block
    launch {
        for (k in 1..3) {
            println("I'm not blocked $k")
            delay(100)
        }
    }
    // Collect the flow
    simple().collect { value -> println(value) }
}
```

Open in Playground → Target: JVM Running on v.1.8.21

Ссылки

- https://www.youtube.com/watch?v=g0R0dbxRMxY (Рассказ про универсальную отмену)
- https://github.com/tinkoff-mobile-tech/TinkoffConcurrency (Наш репозиторий с туллингом)
- https://developer.apple.com/videos/play/wwdc2021/10254/ (Swift Concurrency под капотом)
- https://developer.apple.com/videos/play/wwdc2022/110355 (Реактивная составляющая Swift Concurrency)

Спасибо за внимание

IT's TINKOFF