**День 2.**

**2.2.2 Выполнение работ по Модулю Б. Верстка приложения – 2 часа, практическая работа**

1. *Навигация в приложении*
2. *Всплывающие окна Alert*
3. *Создание навигации в соответствии с заданием чемпионата*

**Навигация**

## NavigationView

— это контейнер, который представляет собой стек компонентов в иерархической навигации.

### Пример 1:

В примере создается форма с двумя кнопками навигации.

Заголовок окна задается с помощью .navigationTitle("Настройки")

**struct** SecondView: View {

**var** body: **some** View{

VStack{

Text("Второй экран")

}.navigationBarTitle("Второй экран")

}

}

**struct** ContentView: View {

**var** body: **some** View {

NavigationView{

NavigationLink("Второй экран", destination: SecondView())

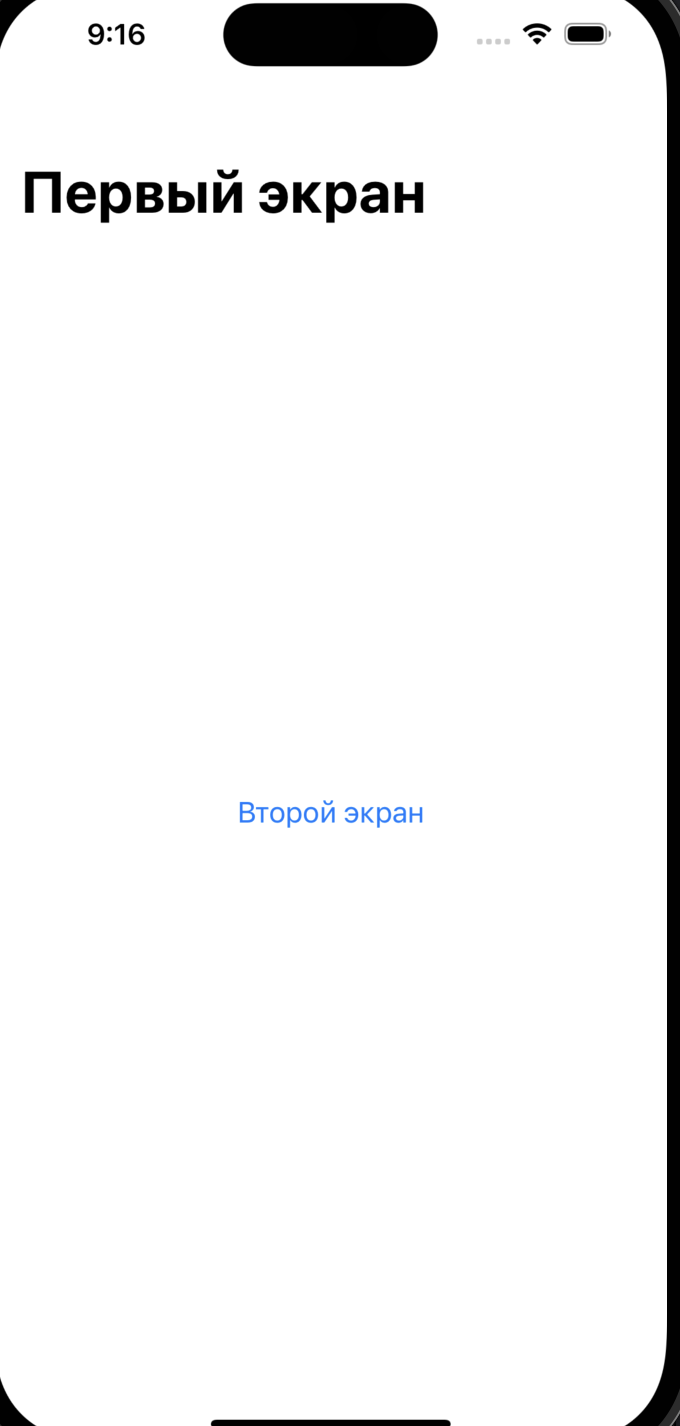
}.navigationTitle("Настройки")

}

}

}

При нажатии на соответствующий элемент формы вызывается то или иное окно (рис.ниже)

### Ссылка с измененным внешним видом

NavigationLink(destination: SecondView()){

Text("Второй экран")

.foregroundColor(.red)

.padding()

.background(Color.blue)

}

### Пример 2. Параметр isActive

NavigationLink имеет второй инициализатор с параметром isActive, который позволяет определить, активна ли навигационная ссылка в данный момент. То есть мы можем программно инициировать активацию навигационной ссылки, установив для любого состояния, которое она отслеживает, значение true.

Например, создаем пустую ссылку навигации и связываем ее со свойством isShowingDetailView.

**struct** ContentView: View {

@State **private** **var** isShowingDetailView = **false**

**var** body: **some** View {

NavigationView {

VStack {

NavigationLink(destination: SecondView(),

isActive: $isShowingDetailView)

{ EmptyView() }

Button("Tap to show detail") {

**self**.isShowingDetailView = **true**

}

}

.navigationTitle("Navigation")

}

}

}

### Управление отображением панели навигации

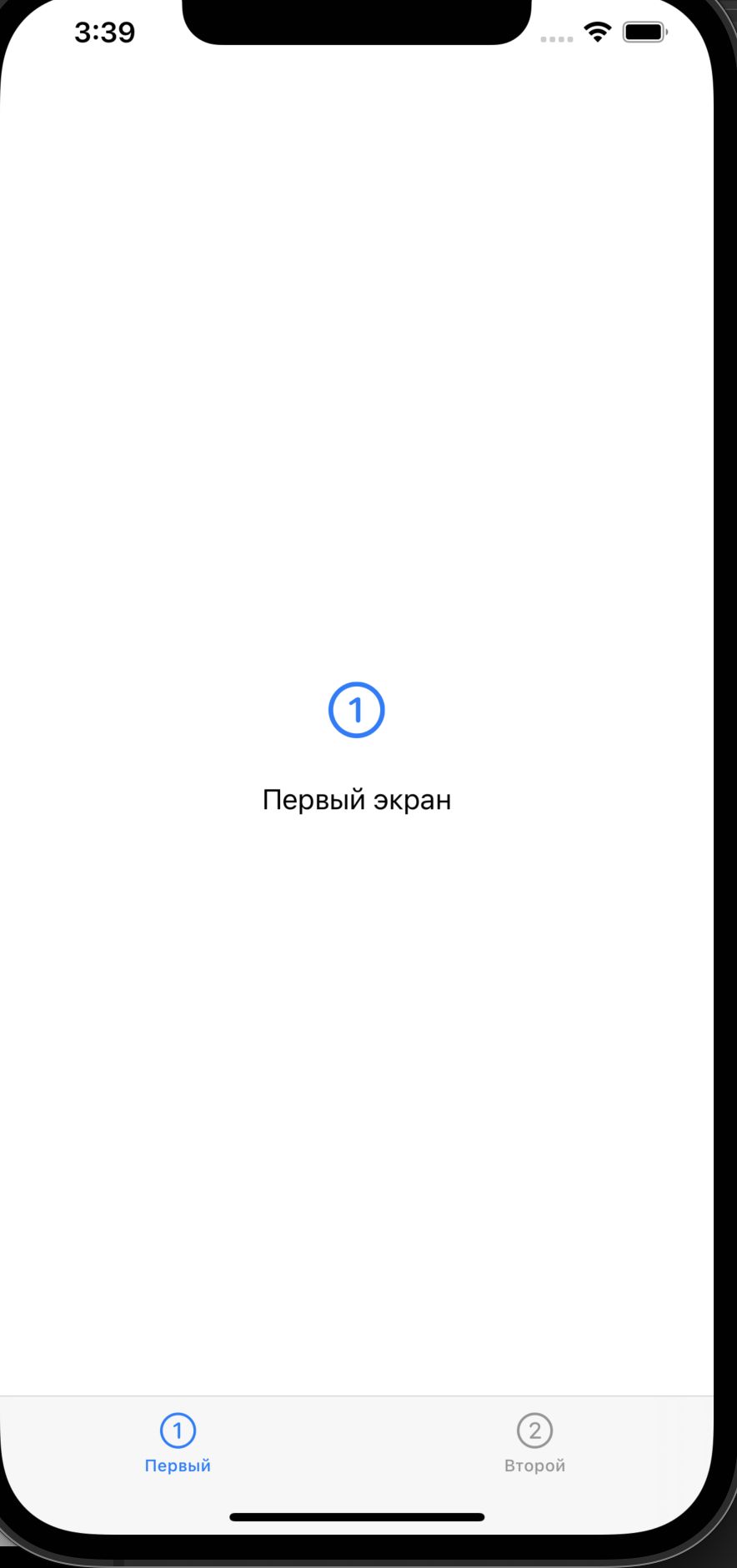
Модификаторы:

**.navigationBarHidden()** позволяет нам контролировать, будет ли вся панель видимой (false) или скрытой (true).

**.navigationBarBackButtonHidden(**) позволяет контролировать, является ли кнопка «Назад» скрытой или видимой, что полезно в тех случаях, когда вы хотите, чтобы пользователь активно сделал выбор, прежде чем двигаться назад.

Как и navigationTitle (), оба эти модификатора прикрепляются к View внутри вашего NavigationView, а не к самому NavigationView.

## TabView. Нижняя навигация между экранами



**struct** FirstView: View {

**var** body: **some** View{

VStack{

Image(systemName: "1.circle")

.padding()

.font(.largeTitle)

.foregroundColor(.blue)

Text("Первый экран")

}

}

}

**struct** SecondView: View {

**var** body: **some** View{

VStack{

Image(systemName: "2.circle")

.padding()

.font(.largeTitle)

.foregroundColor(.blue)

Text("Второй экран")

}

}

}

**struct** ContentView: View {

**var** body: **some** View {

TabView{

FirstView()

.tabItem {

Image(systemName: "1.circle")

Text("Первый")

}

SecondView()

.tabItem {

Image(systemName: "2.circle")

Text("Второй")

}

Text("Третий экран")

.tabItem {

Image(systemName: "3.circle")

Text("Третий")

}

}

}

}

# Всплывающие окна Alert

Оповещения привлекают внимание пользователя к чему-то важному, например, предупреждают о проблеме или запрос на подтверждение действия, которое может иметь серьезные последствия.

**Пример** . Окно сообщения с одной кнопкой, заголовком и текстом сообщения

**struct** ContentView: View {

@State **private** **var** showingAlert = **false**

**var** body: **some** View {

Button("Показать сообщение"){

**self**.showingAlert = **true**

}

.alert(isPresented: $showingAlert) {

Alert(title: Text("Заголовок сообщения"),

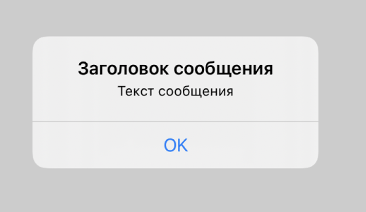
message: Text("Текст сообщения"),

dismissButton: .default(Text("OK")))

}

}

}



## Пример: проверка на пустоту перед переходом на второй экран

G**struct** SecondView: View {

**var** body: **some** View {

Text("Показать сообщение")

}

}

**struct** ContentView: View {

@State **private** **var** isShowingDetailView = **false**

@State **var** isShowAlert = **false**

@State **var** login = ""

**var** body: **some** View {

NavigationView {

VStack {

NavigationLink(destination: SecondView(),

isActive: $isShowingDetailView)

{ EmptyView() }

TextField("login", text: $login)

.textFieldStyle(RoundedBorderTextFieldStyle())

.padding()

.alert(isPresented: $isShowAlert) {

Alert(title: Text("Не заполнили логин"),

dismissButton: .default(Text("OK")))

}

Button(action: {

**if** login != "" {

isShowingDetailView.toggle()

}

**else** {

isShowAlert.toggle()

}

})

{

Text("SignUp")

.padding()

.background(.blue)

.foregroundColor(.white)

}

}

.navigationTitle("Navigation")

}

}

}

# Задание

1. Создайте переход на второй экран LogIn в соответствии с заданием. Создайте второй экран в соответствии с макетом.
2. Добавьте проверку перед переходом проверку, что все поля заполнены. Если они не заполнены, то выдавать предупреждающее окно сообщения.

**Выполнение работ по Модулю А. Архитектура приложения**

**3 часа, практическая работа**

*Разработка модульной структуры приложения, определение слоев приложения: слой представления, слой бизнес-логики, слой данных, а также способов взаимодействия между ними. Разработка стратегии управления состоянием приложения, в том числе выбор подходов и инструментов для хранения данных, обработки пользовательских взаимодействий и обновления интерфейса.*

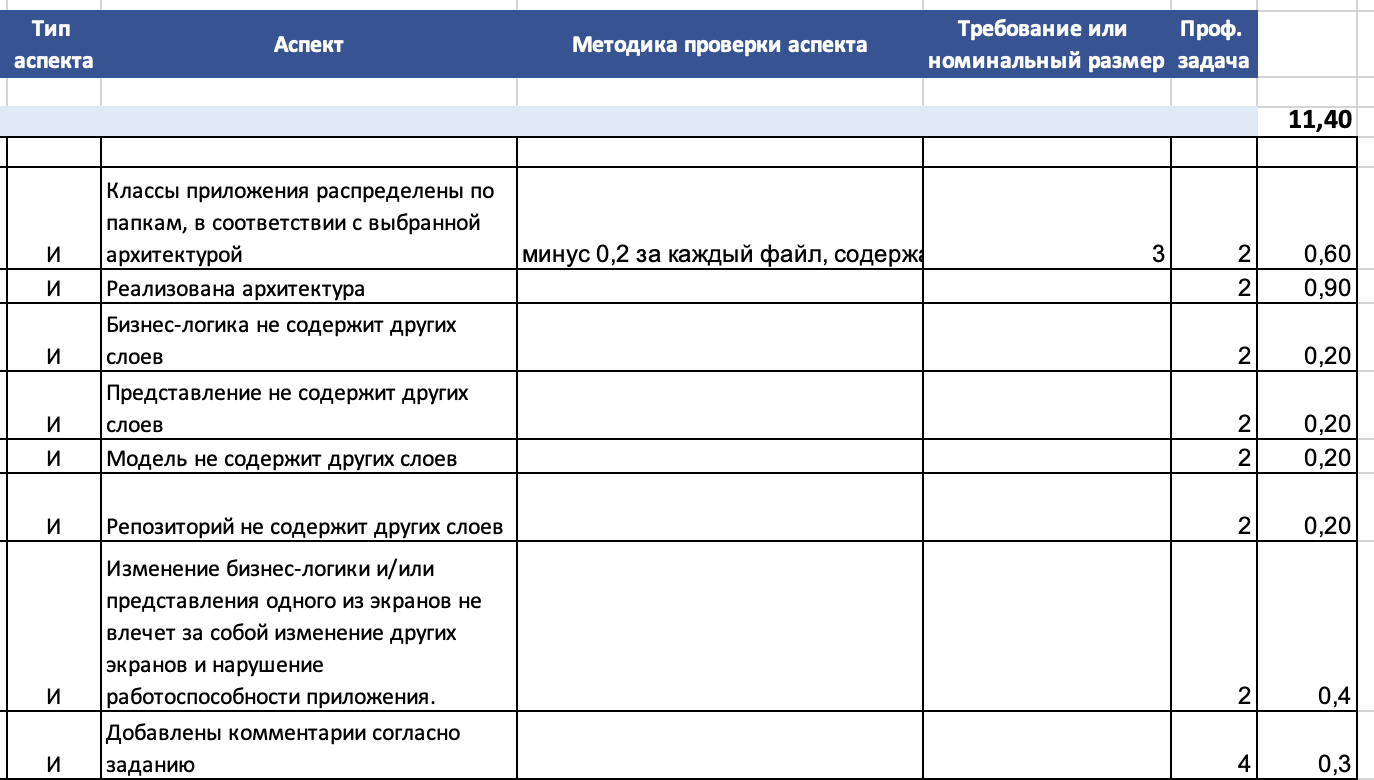
**Задание чемпионата и критерии оценивания**

## Модуль А - Архитектура приложения (инвариант)

(все сессии) Необходимо осуществлять комментирование кода в созданных классах. Обязательны следующие комментарии:

* Описание назначения класса
* Дата создания
* Автор создания
* Описание назначения вложенных элементов программного кода

(все сессии) При разработке проекта приложения вам необходимо использовать архитектуру, в которой будут разделены слои бизнес-логики, представлений и домена. Изменение бизнес-логики и/или представления одного из экранов не должно повлечь за собой изменение других экранов и нарушение работоспособности приложения, за исключением переходов.

**

# 1. Знакомство с понятием архитектуры мобильного приложения

## Что такое архитектура мобильного приложения?

Архитектура мобильного приложения — это совокупность правил, методов и шаблонов разработки мобильных приложений. Этот набор позволяет создавать логичные и хорошо структурированные приложения, отвечающие требованиям клиентов и мировым стандартам отрасли. Качественная архитектура позволяет создавать надежные, масштабируемые и поддерживаемые приложения, основанные на современных принципах разработки.

Существует множество аспектов, влияющих на разработку хорошей мобильной архитектуры, таких как тип мобильной платформы, тип устройства, скорость интернет-соединения, потенциальные характеристики мобильного гаджета — нагрузка на процессор, размер и разрешение экрана, и многое другое.

Прежде чем приступить к созданию мобильного приложения, необходимо четко определить требования заказчика и разделить их на небольшие логические задачи, которые будут выполнять команда разработчиков.

Как только команда получает представление о том, какие функции необходимо встроить в приложение, она готовит архитектуру приложения — скелет, который связывает эти функции в единое целое. Производительность приложения, его масштабируемость, удобство использования и другие факторы всегда зависят от того, насколько тщательно продумана архитектура приложения.

## Что определяет хорошую архитектуру мобильного приложения?

При создании архитектуры мобильного приложения разработчики программного обеспечения должны следовать ряду принципов, таких как:

* SOLID — 5 принципов объектно-ориентированного программирования для построения простых в обслуживании и масштабируемых приложений.
* KISS — принцип сохранения простоты системы и кода для минимизации количества ошибок.
* DRY — принцип сокращения повторений в паттернах программного обеспечения для избежания избыточности.

Кроме того, при создании мобильного приложения разработчики должны ориентироваться на архитектуру **CLEAN (Чистая архитектура**). Этот тип архитектуры означает, что каждый слой приложения не зависит от каких-либо внешних программ или других слоев. Для соединения независимых слоев разработчики используют правило зависимости, когда переходы между слоями осуществляются с помощью границ. Границы — это порты ввода и вывода, которые позволяют передавать данные между слоями.

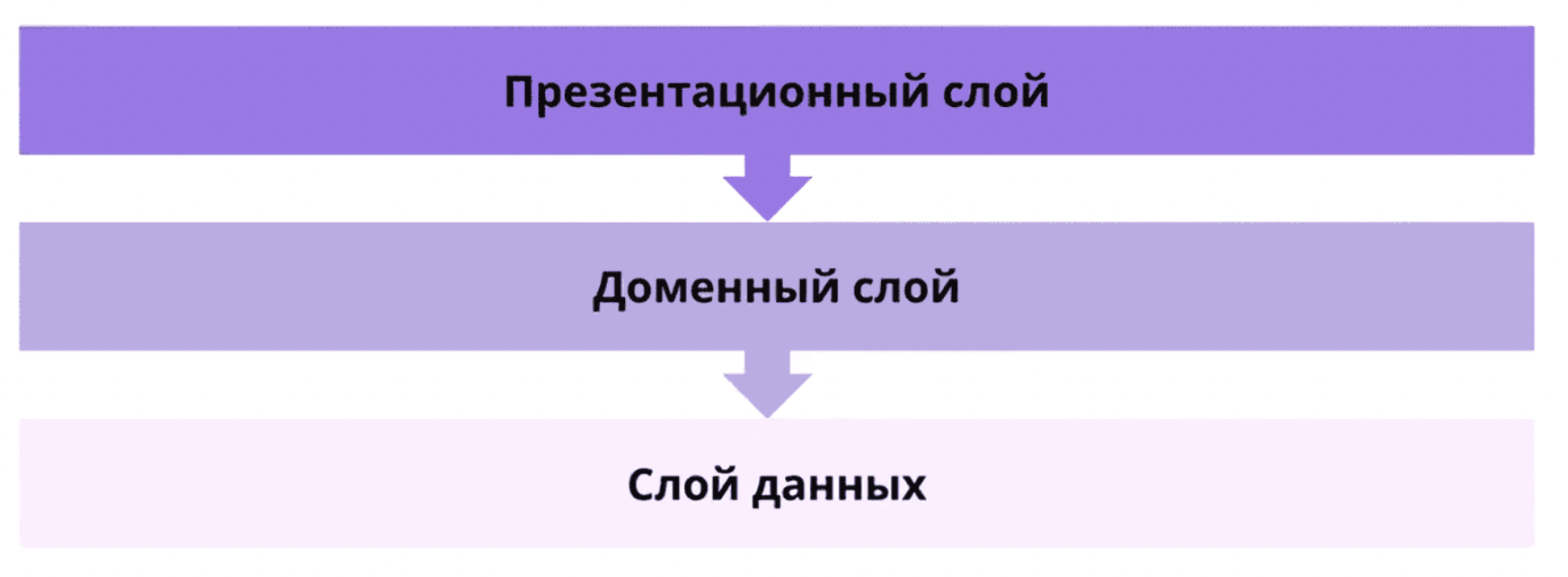
Архитектура мобильных приложений CLEAN является универсальной, с ее помощью разработчики могут легко добавлять в приложение множество плагинов, быстро устранять неполадки и проводить юнит-тесты, а также легко масштабировать приложение. В результате это ускоряет разработку мобильных приложений и значительно экономит бюджет.

Еще один момент, который необходимо учитывать при создании архитектуры приложения, — это мобильная платформа приложения (Android, iOS), поскольку для этих платформ потребуются разные технологии разработки.

## Многоуровневая архитектура мобильных приложений

Каждое мобильное приложение имеет свои слои. Наиболее популярной моделью слоев является трехслойная архитектура. Она состоит из: презентационного слоя, бизнес-слоя и слоя данных.

* Презентационный слой
* Слой бизнес-логики (доменный слой)
* Слой доступа к данным (слой данных)

**

### Презентационный слой (Уровень представления и интерфейс пользователя)

Верхний слой архитектуры, который формирует дизайн интерфейса и пользовательский опыт. По сути, это то, что видит и что воспринимается пользователем при работе с приложением. Благодаря ему на экране отображаются кнопки, иконки, текстовые поля и другие элементы, с которыми взаимодействуют юзеры. При его создании особое внимание уделяется UX/UI-дизайну: интуитивности навигации приложения, подбору шрифтов и цветовой гаммы, настройке анимации, тому какие периферийные устройства может подключать ваше приложение и другим.

### Слой бизнес-логики

Бизнес-слой отвечает за обмен данными и обработку операций. Он располагается между UI-слоем и слоем данных и координирует работу сервиса. На этом уровне ваше приложение выполняет различные задачи, такие как: проверка данных, кэширование, логирование, управление исключениями и другие. Кроме того, бизнес-слой устанавливает бизнес-правила, выполняет сложные бизнес-процедуры и регулирует рабочий процесс.

Этот уровень может быть развернут на внутреннем сервере или на устройстве пользователя, в зависимости от количества операций, выполняемых вашим приложением, и количества ресурсов, доступных на устройстве пользователя.

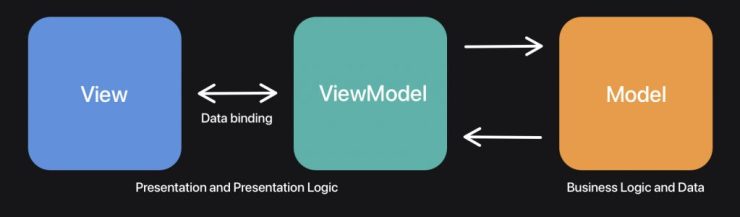
### Слой доступа к данным

Этот слой отвечает за сохранность и обслуживание данных. Он состоит из компонентов доступа к данным, сервисных инструментов и утилит. При создании этого слоя разработчики программного обеспечения должны помнить о том, что он может масштабироваться с изменением требований бизнеса в будущем. Также важно выбрать правильную технологию для доступа к данным и их проверки, чтобы этот слой был защищен от ввода недействительных данных и хорошо функционировал.

## Что такое архитектурный паттерн MVVM?

MVVM — это аббревиатура от Model-View-ViewModel, и изначально он был придуман архитекторами Microsoft. Разработчики используют его для разделения задач, гарантируя, что представление не привязано к определенной модели данных. Это приводит к созданию многократно используемого кода, поскольку вы можете использовать представление с любой моделью, если между ними есть модель представления, обеспечивающая коммуникационный слой.

Визуально паттерн выглядит следующим образом:



MVVM — распространенный архитектурный паттерн, используемый разработчиками в SwiftUI.

Паттерн состоит из трех слоев:

* **Представление (View).** Это определение представления. В SwiftUI это будет декларативное определение представления.
* **Модель представления (ViewModel).** Представление напрямую биндится со свойствами модели представления для отправки и получения обновлений. Поскольку модель представления не имеет ссылок на представление, она становится многоразовой для использования с несколькими представлениями.
* **Модель (Model).** Модель относится к домену модели. Например, у ContactView будет модель ContactViewModel, которая действует как уровень связи с доменной моделью Contact.

Для работы с MVVM [не всегда](https://apptractor.ru/info/articles/ubiraem-m-iz-mvvm-v-swiftui.html) нужно использовать модель. Модель также может быть уровнем доступа к данным, например, кэш контента на удаленном сервере. Чтобы эффективно использовать паттерн, применяйте несколько важных правил:

* Представление взаимодействует только с ViewModel.
* Представление ничего не знает о (доменной) модели, лежащей за ViewModel.
* ViewModel не имеет ссылок на View и становится многоразовой для использования с любым View.

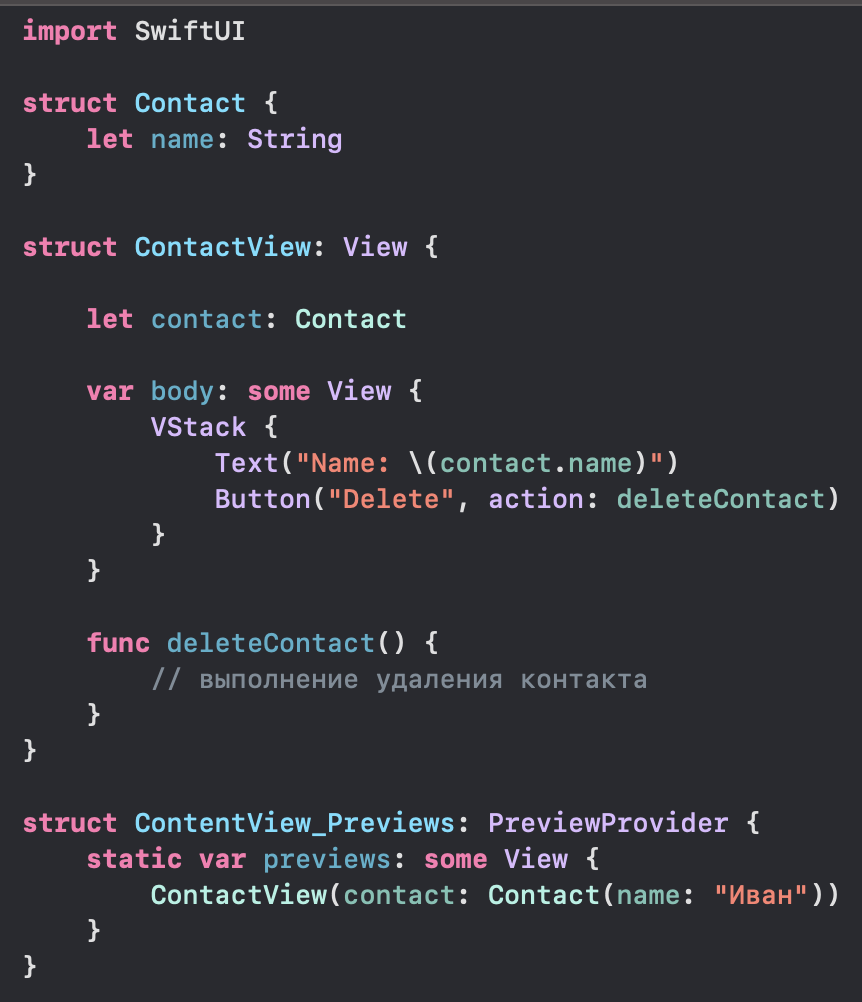
Обеспечение соблюдения этих правил в вашем коде автоматически приведет к созданию многократно используемого, тестируемого и изолированного кода.

# 2. Знакомство с реализацией архитектуры в iOS

## Как использовать MVVM в SwiftUI

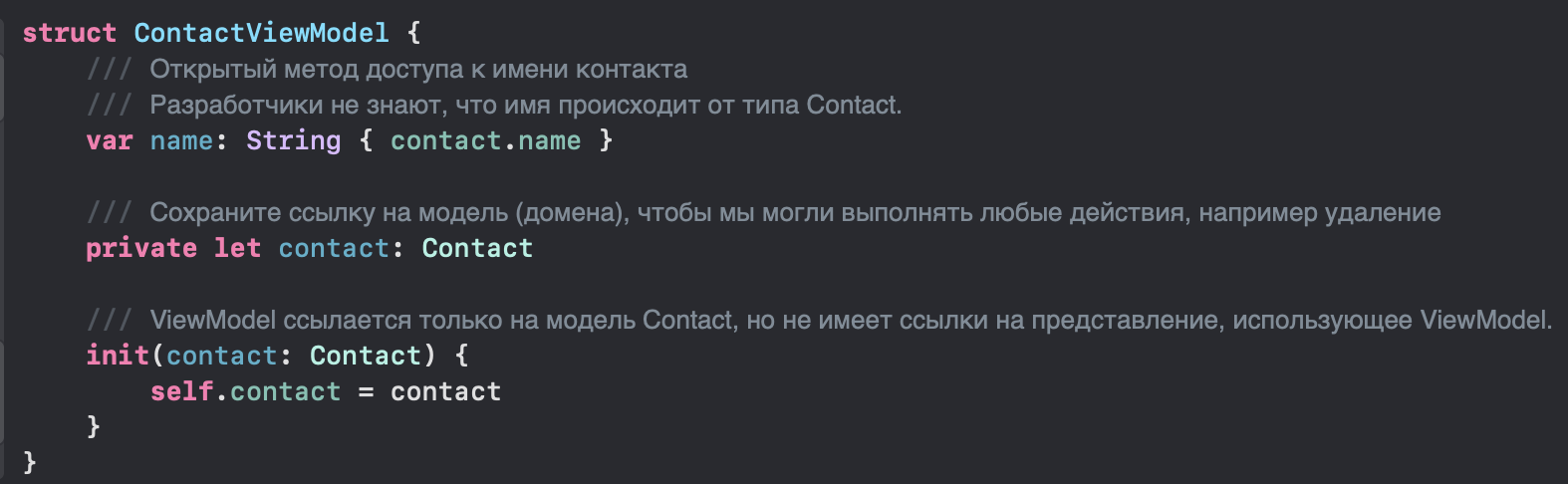
Рассмотрим на примере представления контактов.

Без использования MVVM ваш ContactView мог бы выглядеть следующим образом:



На этом этапе код может быть в полном порядке. Пока его еще не так много, и бизнес-логики тоже не слишком много. Однако ContactView строго связан с моделью Contact и содержит бизнес-логику для удаления контактов. Ее также нелегко протестировать или повторно использовать с другими подобными типами.

Мы можем переписать эту логику, используя архитектурный паттерн MVVM. Для изящной миграции мы начнем с того, что подключим метаданные только к новому типу ContactViewModel:



Теперь мы можем обновить наш ContactView, чтобы использовать эту новую ViewModel:

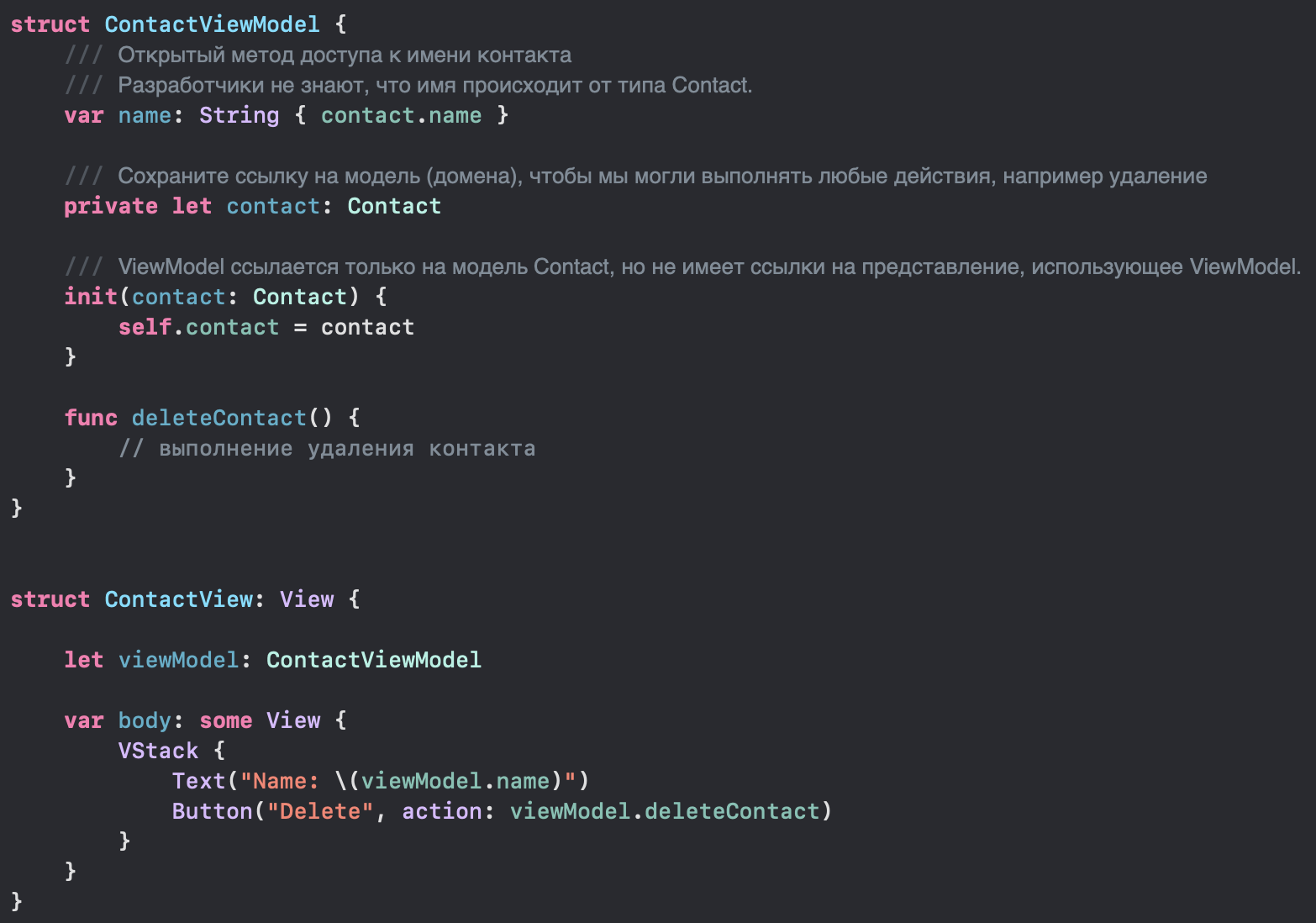


Это уже большое улучшение, поскольку мы убрали прямую связь между моделью и представлением.

## Извлечение бизнес-логики из представления

Следующий шаг — извлечение бизнес-логики из представления. Многие разработчики используют для этого разные реализации. Я предпочитаю писать логику удаления внутри ViewModel, но паттерн изначально описывает реализацию удаления внутри модели Contact. Еще лучше было бы использовать здесь паттерн Репозиторий, чтобы вынести бизнес-логику в один ответственный тип, в результате чего ViewModel станет лишь коммуникационным слоем.

В этом примере перенесем бизнес-логику удаления в ViewModel. Это уже большое улучшение, поскольку оно удаляет бизнес-логику из представления.



Окончательная реализация ContactView больше не связана с конкретной моделью или бизнес-логикой. Она взаимодействует только с нашей ViewModel, которая выступает в качестве коммуникационного слоя между представлением и моделью.

## Использование протоколов для повышения возможности повторного использования представлений в MVVM

Реализация MVVM сделала наши представления более простыми и менее зависимыми, но мы можем сделать еще один шаг вперед, используя протоколы. До сих пор нам приходилось работать только со структурой Contact, но может оказаться, что в вашем проекте потребуется другой тип контакта, например RemoteContact.

Используя протоколы, вы позволите своим представлениям стать более гибкими. Для этого мы начнем с создания протокола ContactViewModel:

protocol ContactViewModel {

var name: String { get }

func deleteContact()

}

Нам не нужно обновлять наше представление, поскольку оно уже взаимодействует с тем же именем типа, но нам нужно переименовать нашу оригинальную модель ContactViewModel в LocalContactViewModel. Она также должна соответствовать протоколу ContactViewModel:

struct LocalContactViewModel: ContactViewModel {

var name: String { contact.name }

private let contact: Contact

init(contact: Contact) {

self.contact = contact

}

func deleteContact() {

// Выполните удаление контакта \*\*локально\*\*

}

}

Наконец, мы можем приступить к определению новой модели RemoteContactViewModel, которая будет выступать в качестве коммуникационного слоя между ContactView и RemoteContact:

struct RemoteContact {

let name: String

}

​

struct RemoteContactViewModel: ContactViewModel {

var name: String { contact.name }

private let contact: RemoteContact

init(contact: RemoteContact) {

self.contact = contact

}

func deleteContact() {

// Выполняем удаление контакта \*\*удаленно\*\*

// Возможно использование сетевого запроса

}

}

Преимущество использования подобных протоколов заключается в том, что они позволяют инстанцировать один и тот же ContactView с несколькими типами ViewModel, которые взаимодействуют с различными (доменными) моделями:

let localViewModel = LocalContactViewModel(contact: Contact(name: "Иван"))

let remoteViewModel = RemoteContactViewModel(contact: RemoteContact(name: "Иван"))

/// Мы используем один и тот же `ContactView`, но различные типы `ContactViewModel`.

let localContactView = ContactView(viewModel: localViewModel)

let remoteContactView = ContactView(viewModel: remoteViewModel)

Мы создали многократно используемый и тестируемый код и настроились на масштабируемость в будущем.

**Итоговый код для проверки:**

**import** SwiftUI

**struct** Contact {

**let** name: String

}

**struct** RemoteContact {

**let** name: String

}

**struct** RemoteContactViewModel: ContactViewModel {

**var** name: String { contact.name }

**private** **let** contact: RemoteContact

**init**(contact: RemoteContact) {

**self**.contact = contact

}

**func** deleteContact() {

// Выполняем удаление контакта \*\*удаленно\*\*

// Возможно использование сетевого запроса

}

}

**struct** LocalContactViewModel: ContactViewModel {

/// Открытый метод доступа к имени контакта

/// Разработчики не знают, что имя происходит от типа Contact.

**var** name: String { contact.name }

/// Сохраните ссылку на модель (домена), чтобы мы могли выполнять любые действия, например удаление

**private** **let** contact: Contact

/// ViewModel ссылается только на модель Contact, но не имеет ссылки на представление, использующее ViewModel.

**init**(contact: Contact) {

**self**.contact = contact

}

**func** deleteContact() {

// выполненяем удаление контакта \*\*локально\*\*

}

}

**protocol** ContactViewModel {

**var** name: String { **get** }

**func** deleteContact()

}

**struct** ContactView: View {

**let** localViewModel = LocalContactViewModel(contact: Contact(name: "Иван"))

**let** remoteViewModel = RemoteContactViewModel(contact: RemoteContact(name: "Иван"))

**var** body: **some** View {

VStack {

Text("Name: \(localViewModel.name)")

Button("Delete", action: localViewModel.deleteContact)

}

}

}

# 3. Разработка стратегии управления состоянием приложения

*Разработка стратегии управления состоянием приложения, в том числе выбор подходов и инструментов для хранения данных, обработки пользовательских взаимодействий и обновления интерфейса.*

Эффективное управление состояниями — ключ к созданию стабильных и отзывчивых приложений в SwiftUI.

## Понимание состояния и привязки в SwiftUI

**Состояние** (state) и **привязка** (binding) — две фундаментальные концепции в SwiftUI, которые позволяют создавать динамичные и интерактивные интерфейсы. Их понимание — ключ к эффективному управлению состоянием пользовательского интерфейса.

## Что такое состояние

В SwiftUI состояние — это [обертка свойства](https://apptractor.ru/info/articles/property-wrapper.html), которая используется для хранения простых значений, принадлежащих определенному представлению и изменяющихся с течением времени. Когда значение State изменяется, представление аннулирует свой внешний вид и пересчитывает свойство body. Это означает, что SwiftUI автоматически перерисовывает представление при изменении состояния.

Рассмотрим следующий пример, который управляет состоянием представления Toggle:

struct ContentView: View {

@State private var isSwitchedOn = false

​

var body: some View {

VStack {

Toggle(isOn: $isSwitchedOn) {

Text("Turn me on or off")

}

if isSwitchedOn {

Text("The switch is on!")

}

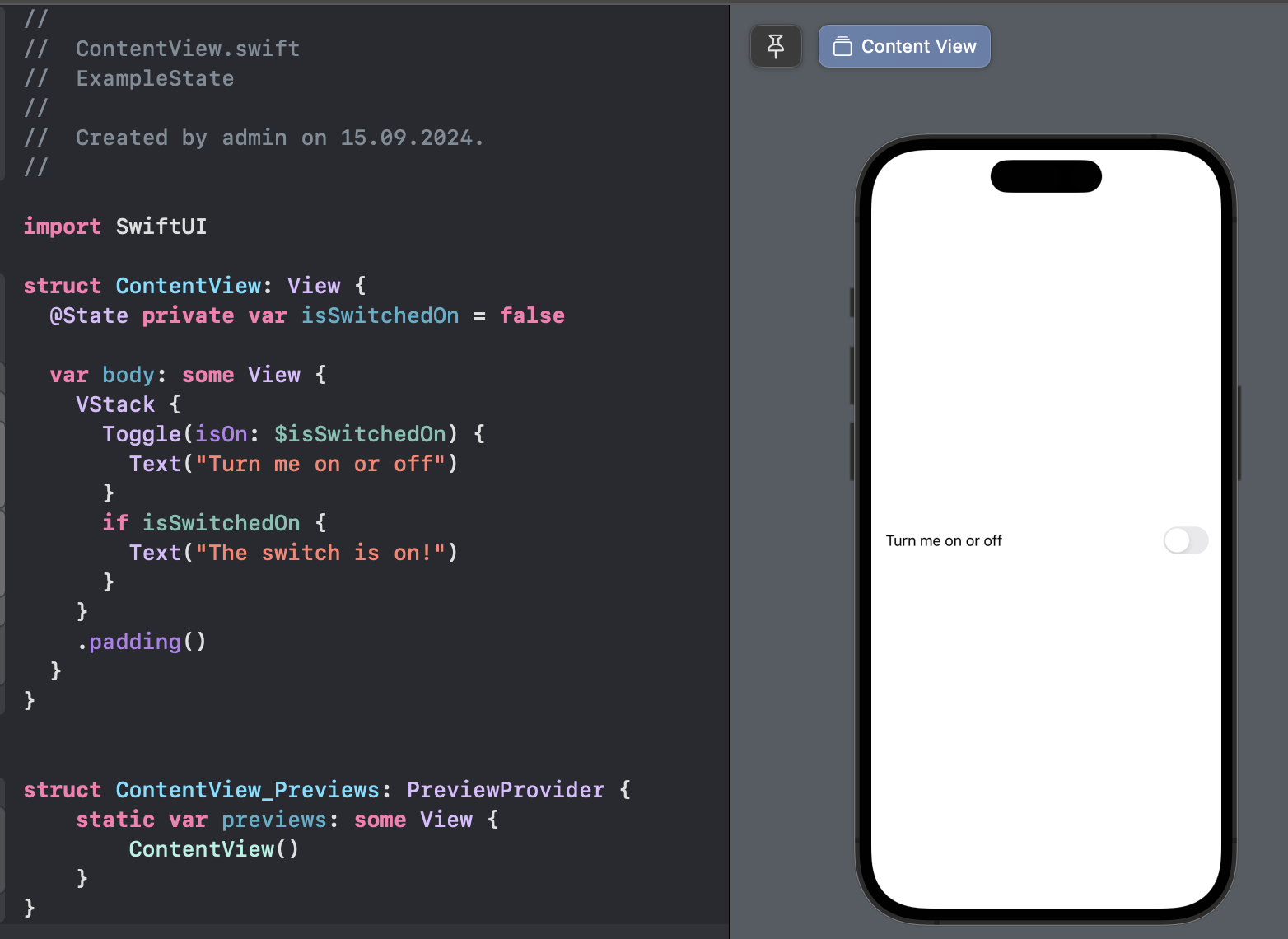
}

.padding()

}

}

Ваш предварительный просмотр должен выглядеть следующим образом:



Используйте обертку свойства State для управления данными, локальными для одного представления.

В приведенном выше примере isSwitchedOn — это State свойство, которое отслеживает, включен или выключен тумблер. Когда тумблер переключается, isSwitchedOn изменяется, заставляя представление перерисовываться и обновлять отображаемый текст.

## Понимание привязки

Привязка в SwiftUI создает двустороннюю связь между свойством, хранящим данные, и представлением, которое отображает и изменяет эти данные. Она позволяет изменять данные, так что SwiftUI может изменять значение внутри структуры, которая обычно является неизменяемой.

Рассмотрим следующий пример:

struct ContentView: View {

@State private var selectedColor = Color.red

​

var body: some View {

VStack {

Rectangle()

.fill(selectedColor)

.frame(width: 100, height: 100, alignment: .center)

​

ColorPickerView(selectedColor: $selectedColor)

}

.padding()

}

}

​

struct ColorPickerView: View {

@Binding var selectedColor: Color

​

let colors: [Color] = [.red, .green, .blue, .yellow, .orange]

​

var body: some View {

HStack {

ForEach(colors, id: \.self) { color in

Rectangle()

.fill(color)

.frame(width: 50, height: 50)

.onTapGesture {

selectedColor = color

}

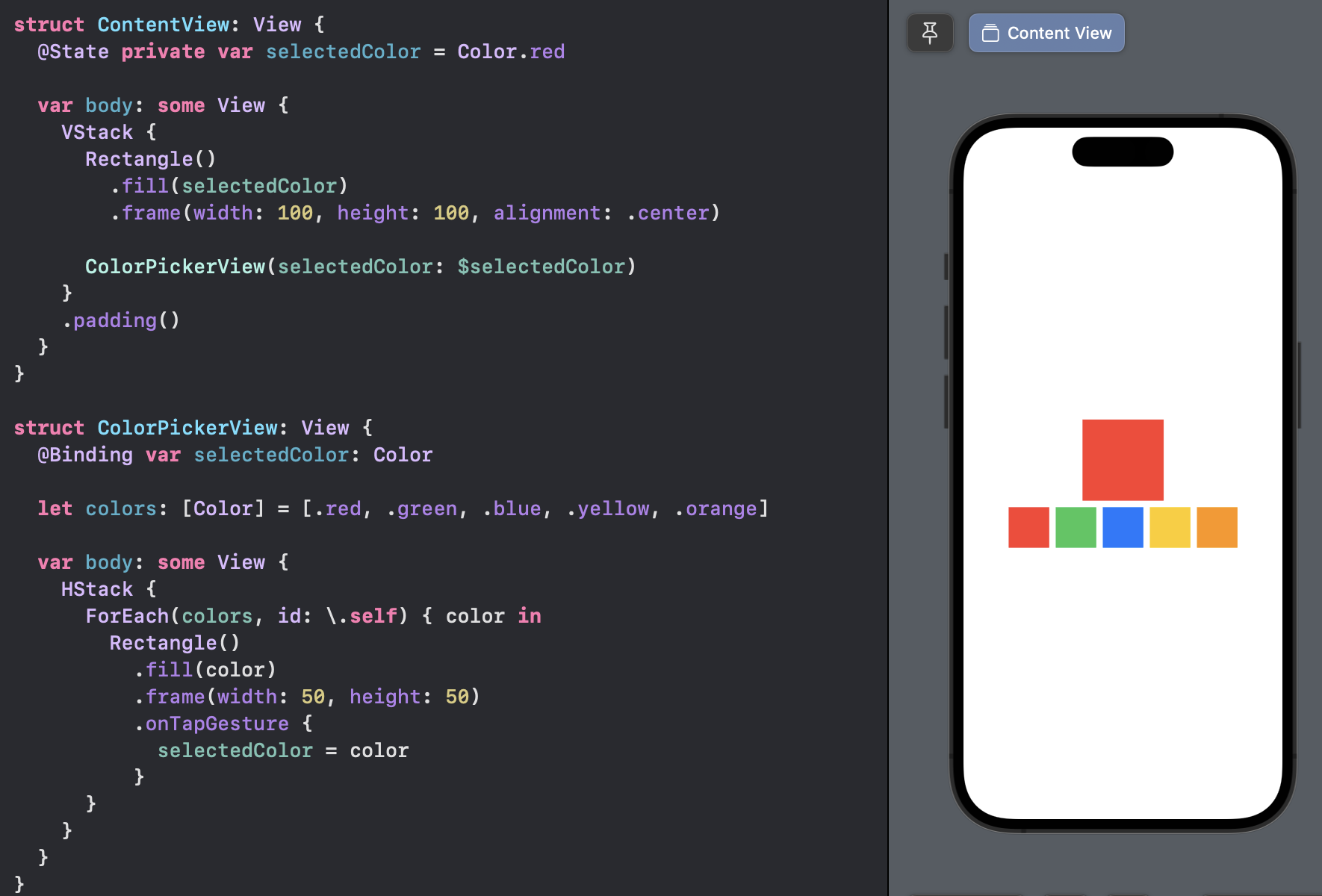
}

}

}

}

Ваш предварительный просмотр должен выглядеть следующим образом:



Привязки обеспечивают бесшовное взаимодействие между представлениями.

В этом примере у вас есть ContentView, содержащий прямоугольник, цвет заливки которого определяется состоянием selectedColor. У вас также есть ColorPickerView, который принимает привязку к selectedColor. Когда цвет в ColorPickerView нажимается, он обновляет selectedColor, который затем изменяет цвет заливки прямоугольника в ContentView. Это возможно благодаря двусторонней связи, созданной привязкой — ColorPickerView не только считывает значение selectedColor, но и имеет возможность изменять его.

В заключение отметим, что состояние и привязка являются важнейшими инструментами в SwiftUI для управления состоянием представлений и создания интерактивных пользовательских интерфейсов. Понимая и используя эти концепции, вы сможете создавать динамичные и отзывчивые пользовательские интерфейсы, которые автоматически обновляются в ответ на взаимодействие с пользователем.

## Понимание ObservableObject и ObservedObject

В SwiftUI управление состоянием, разделяемым между несколькими представлениями, требует инструментов, выходящих за рамки обертки свойства State. Именно здесь в игру вступают ObservableObject и ObservedObject.

**ObservableObject** — это протокол, который SwiftUI предоставляет для объектов, за которыми можно наблюдать на предмет изменений. Когда вы помечаете класс как соответствующий ObservableObject, вы сигнализируете SwiftUI, что свойства этого объекта при изменении должны вызывать обновление всех представлений, которые от них зависят. Любые свойства, помеченные **@Published** в ObservableObject, будут автоматически уведомлять представление о необходимости обновления при их изменении.

Давайте посмотрим на это в действии:

class UserSettings: ObservableObject {

@Published var username = "Anonymous"

}

​

struct ContentView: View {

@ObservedObject var settings = UserSettings()

​

var body: some View {

VStack {

Text("Hello, \(settings.username)!")

Button("Change Username") {

settings.username = "John Doe"

}

}

}

}

Ваш предварительный просмотр должен выглядеть следующим образом:



Использование оберток свойств ObservableObject и Published.

В этом примере UserSettings является объектом ObservableObject. У него есть свойство username, помеченное @Published, что означает, что любые изменения в имени пользователя будут уведомлять наблюдателей.

ContentView наблюдает за UserSettings через обертку свойства @ObservedObject. Когда нажимается кнопка, она изменяет свойство username в настройках, что, в свою очередь, заставляет ContentView обновляться и отображать новое имя пользователя.

Такое взаимодействие между ObservableObject и ObservedObject является основополагающим для управления общим состоянием различных представлений в SwiftUI. Оно позволяет вам создать источник истины для ваших данных, которые затем могут быть замечены и использованы любой частью вашего приложения.

Эффективное управление состояниями — ключ к созданию стабильных и отзывчивых приложений в SwiftUI.

Ниже приведены некоторые лучшие практики управления состоянием в ваших приложениях SwiftUI:

1. **Используйте State и Binding для простых локальных состояний**. State и Binding идеально подходят для управления простым состоянием, которое локально для представления или может быть передано от родительского представления к дочернему. Помните, что эти [обертки свойств](https://apptractor.ru/info/articles/property-wrapper.html) предназначены для работы с типами значений.
2. **Для сложных состояний используйте ObservedObject и Published**. Если у вас есть более сложное состояние, которое может быть общим для нескольких представлений, рассмотрите возможность использования ObservedObject и Published в сочетании с отдельным классом управления состоянием.
3. **Используйте EnvironmentObject для совместного использования состояния в несвязанных представлениях**. Если вам нужно разделить состояние между несколькими представлениями, которые не связаны напрямую через отношения родитель-ребенок, EnvironmentObject может быть хорошим выбором.
4. **Избегайте больших State переменных**. Хранение большого количества данных в State переменных может привести к проблемам с производительностью, поскольку SwiftUI пересоздает представление при каждом изменении состояния.
5. **Отложите сложные вычисления и побочные эффекты**. Избегайте выполнения сложных вычислений или побочных эффектов, например сетевых запросов, непосредственно в структурах представления.

**Пример**: Давайте проиллюстрируем эти лучшие практики на примере простого приложения менеджера задач:

class TaskManager: ObservableObject {

@Published var tasks = [String]()

​

func addTask(\_ task: String) {

tasks.append(task)

}

}

​

struct TaskListView: View {

@EnvironmentObject var taskManager: TaskManager

@State private var newTask = ""

​

var body: some View {

NavigationStack {

VStack {

TextField("New task", text: $newTask)

.onSubmit {

if !newTask.isEmpty {

taskManager.addTask(newTask)

newTask = ""

}

}

.padding()

List(taskManager.tasks, id: \.self) { task in

Text(task)

}

}

.navigationTitle("Task List")

}

}

}

​

struct ContentView: View {

@StateObject var taskManager = TaskManager()

var body: some View {

TaskListView()

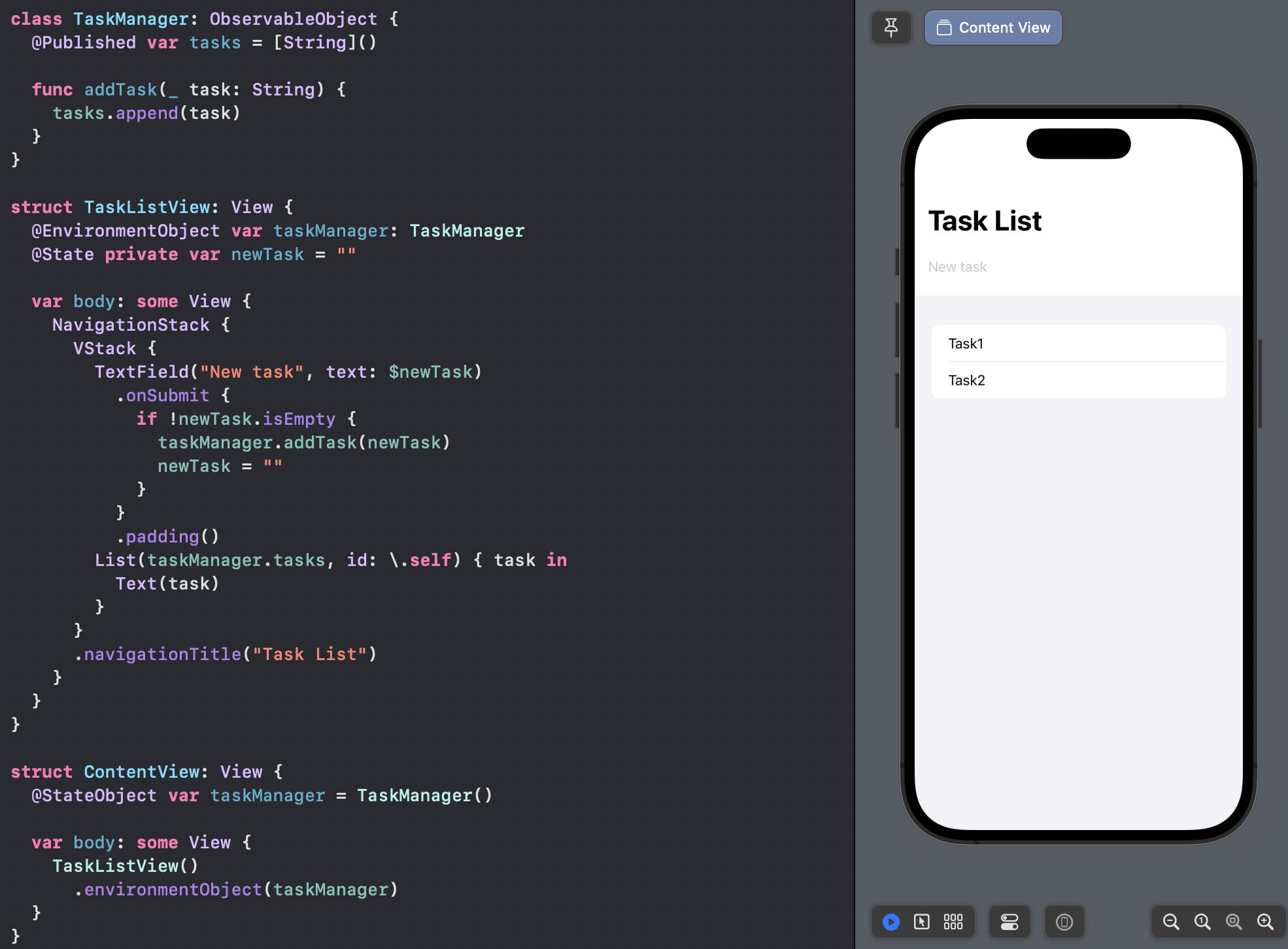
.environmentObject(taskManager)

}

}

​

Ваш предварительный просмотр должен выглядеть следующим образом:



Этот простой менеджер задач иллюстрирует несколько техник управления состоянием SwiftUI.

В этом примере вы создаете класс TaskManager, который управляет списком задач. Это ObservableObject, что означает, что SwiftUI будет следить за изменениями его свойств Published и обновлять все представления, которые зависят от этих свойств. TaskListView использует EnvironmentObject для доступа к общему TaskManager и State для свойства newTask, которое является локальным для этого представления. Когда в текстовом поле фиксируется новая задача, она добавляется в диспетчер задач, а текстовое поле очищается.

# 4. Создание приложения ToDoList в архитектуре MVVM

Разберем на примере списка задач модель MVVM

Исходный пример:

Создайте новый проект и назовите его **MVVM\_TODOList**

Внесите следующий код

**import** SwiftUI

**struct** Item: Identifiable{

**var** id = UUID()

**var** name: String

}

**struct** ContentView: View {

@State **var** items = [

Item(name: "First"),

Item(name: "second"),

Item(name: "third")

]

**var** body: **some** View {

NavigationView{

List{

ForEach(items){item **in**

NavigationLink(

destination: Text("Destination \(item.name)")){

Text(item.name)

}

}//End ForEach

//удаление элемента

.onDelete(perform: { indexSet **in**

**for** index **in** indexSet {

items.remove(at: index)

}

})

//перемещение элементов

.onMove(perform: { indices, newOffset **in**

items.move(fromOffsets: indices, toOffset: newOffset)

})

}//End List

.navigationBarTitle(Text("TODO"), displayMode: .large)

.toolbar(content: {

ToolbarItemGroup(placement: .navigationBarTrailing){

//кнопка Edit - переводит в режим редактирования

EditButton()

//кнопка + - добавляет элемент в список

Button(action: {

items.append(Item(name: "NewItem"))

}) {

Image(systemName: "plus")

}

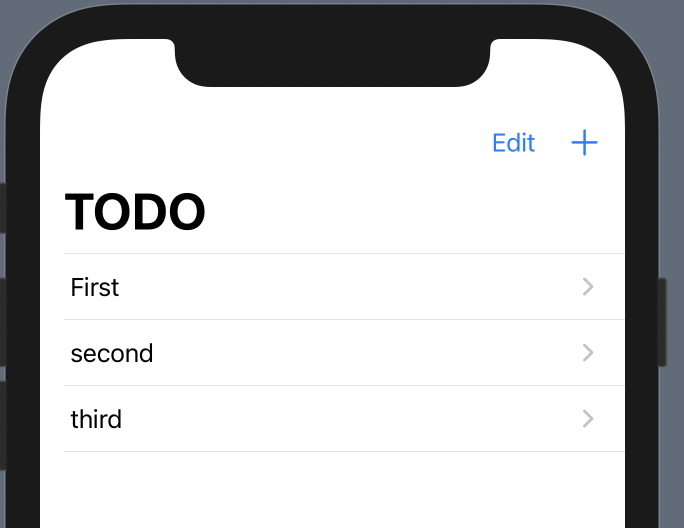
}

})

}//End navigationView

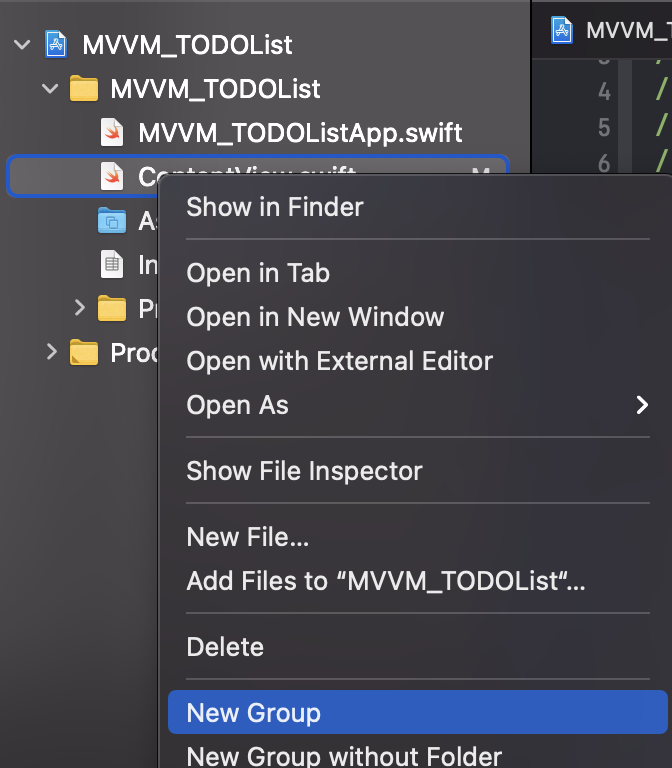
}

}

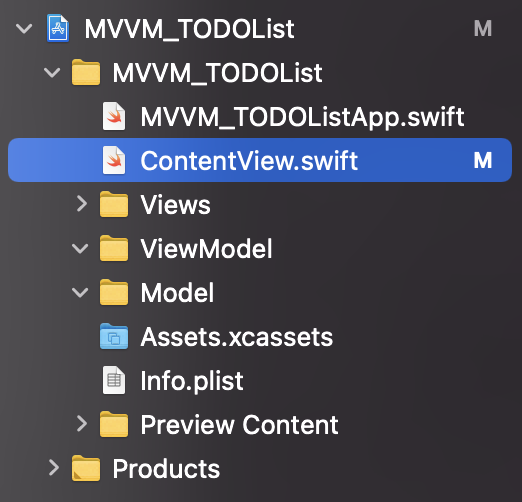


Реализуем в этом примере MVVM-архитектуру

Создадим новые группы (папки) в структуре проекта – правой кнопкой мыши по ContentView – New Group – Название группы.

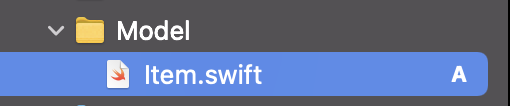


Создадим их 3 – Model, Views, ViewModel.

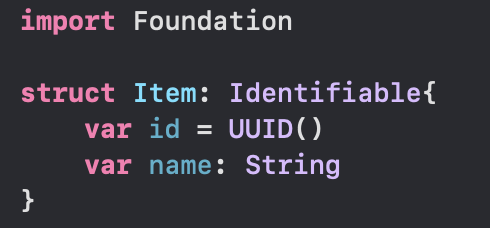


Начнем с отделения модели.

В папке Model создадим новый файл – из контексного меню папки New File – Swift file – название Item

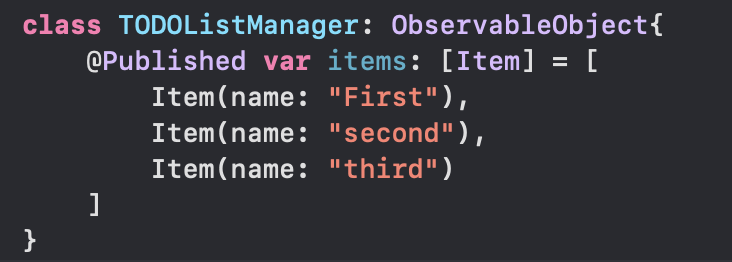


**Перенесем** в него нашу структуру.

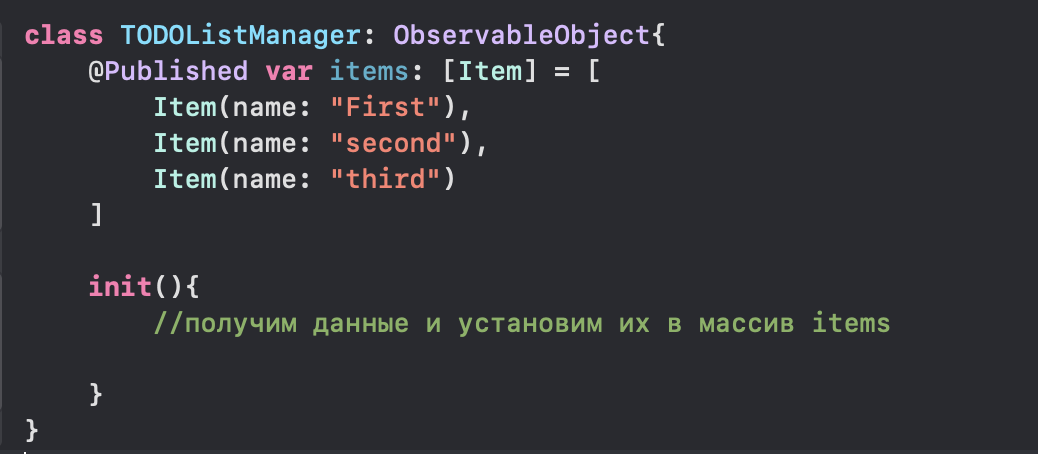


Теперь создадим новый файл в папке ViewModel новый swift файл и назовем его **TODOListManager**

Создадим в нем класс TODOListManager, который подпишем протоколом ObservableObject. В классе разместим массив объектов Item, который обернем в обертку свойства @Published. Значения массива скопируем из нашего исходного примера.



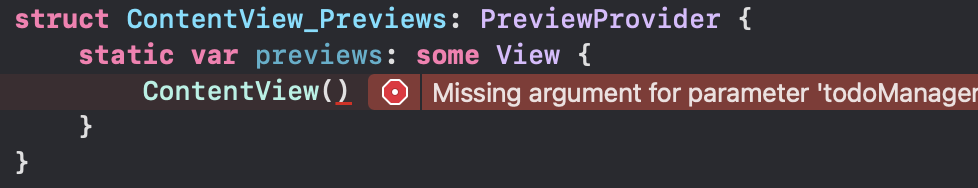
Создадим инициализатор класса, в котором будем получать данные и устанавливать их в массив. Пока оставим его пустым.



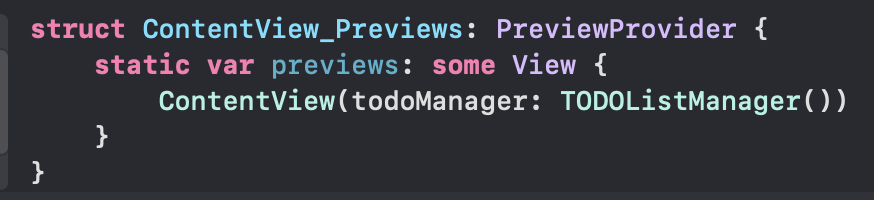
Вернемся в наш ContentView удалим в нем массив и добавим переменную, которая будет отслеживать изменения в ViewModel.



Внизу отобразится ошибка, при попытке отрисовать в Preview экрана.

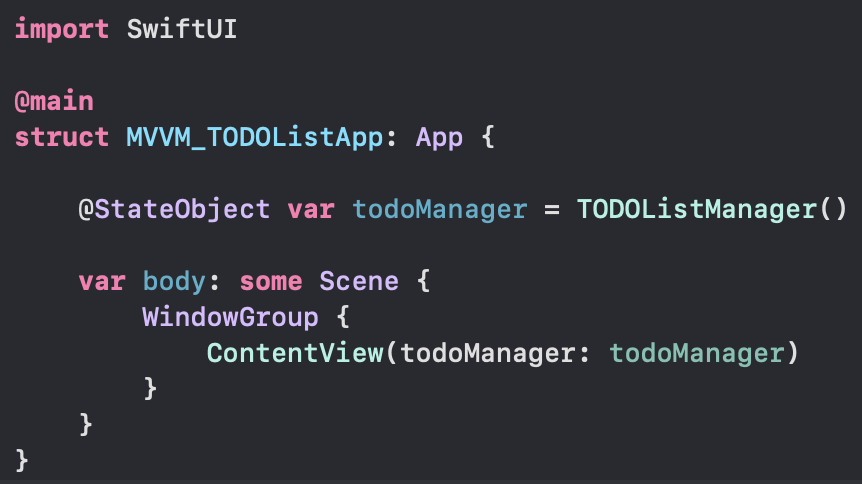


Исправим ее, добавив инициализацию переменной

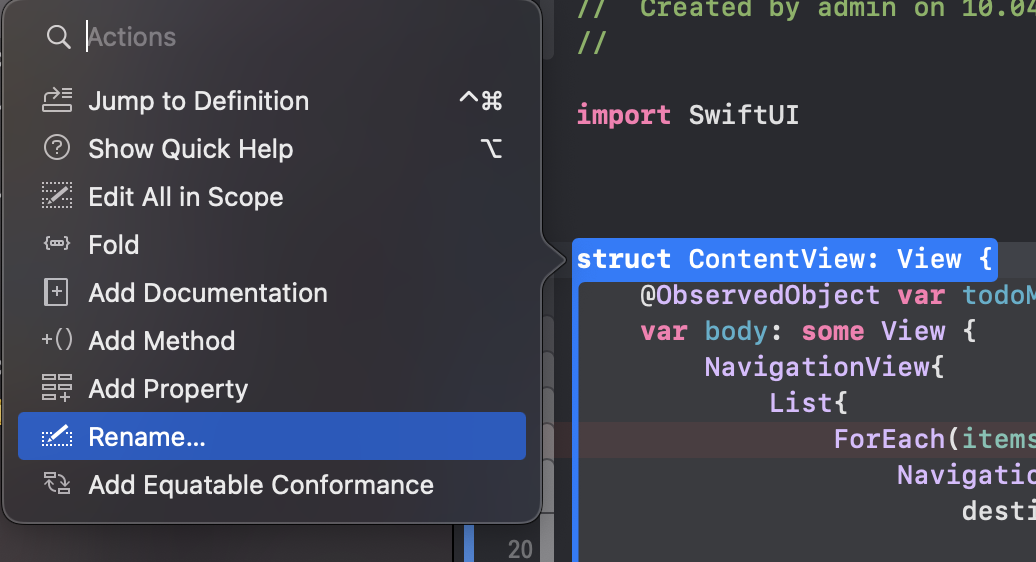


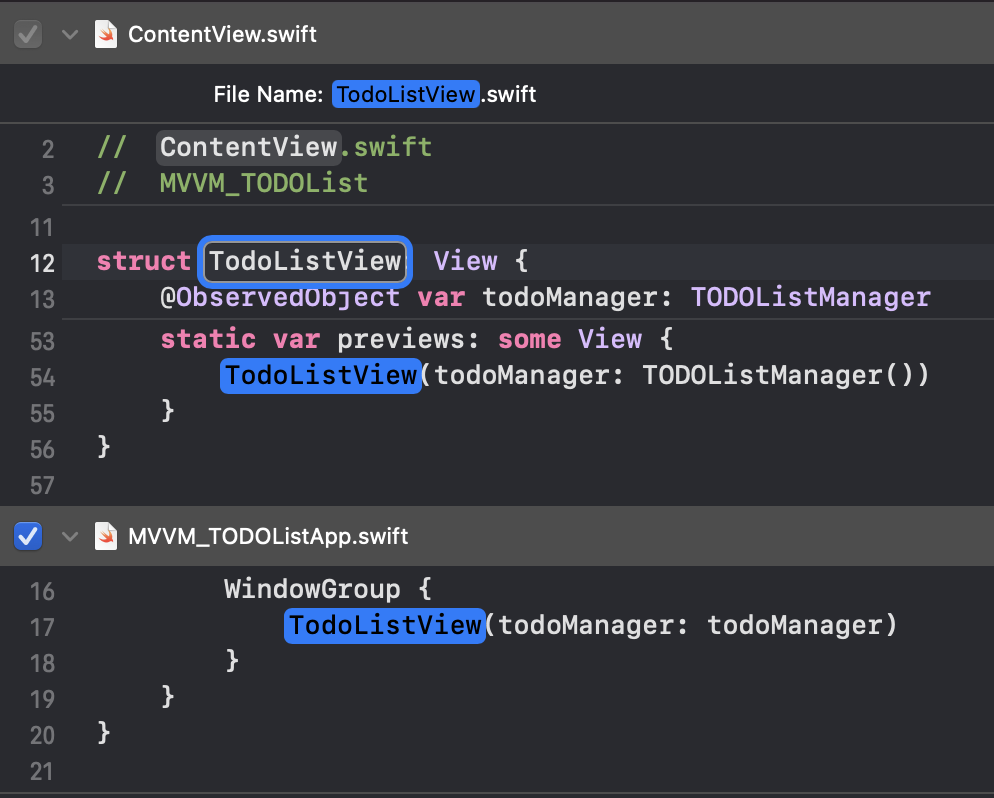
Теперь перейдем в файл **MVVM\_TODOListApp.swift**

И исправим старт программы. При запуске должен создаваться экземпляр созданного класса, поэтому добавим объект todoManager и используем его при запуске ContentView

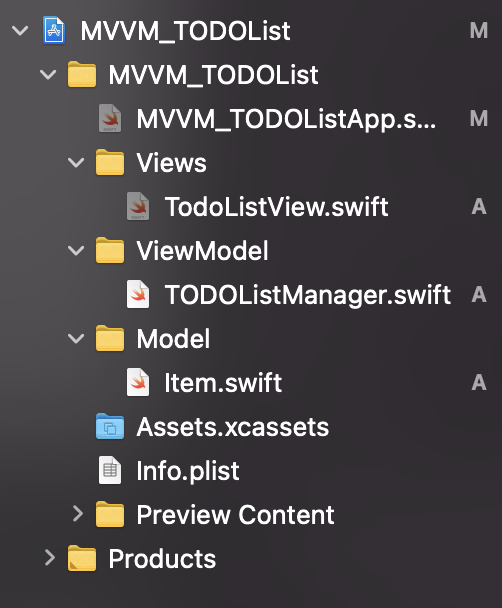


Переименуем ContentView, дадим ему более информативное название. Щелкните левой кнопкой мыши по названию с нажатой клавишей Command и выберите Rename. Введите новое название **TodoListView** и нажмите Enter, автоматически переименуются все вхождения прежнего названия, кроме комментариев.

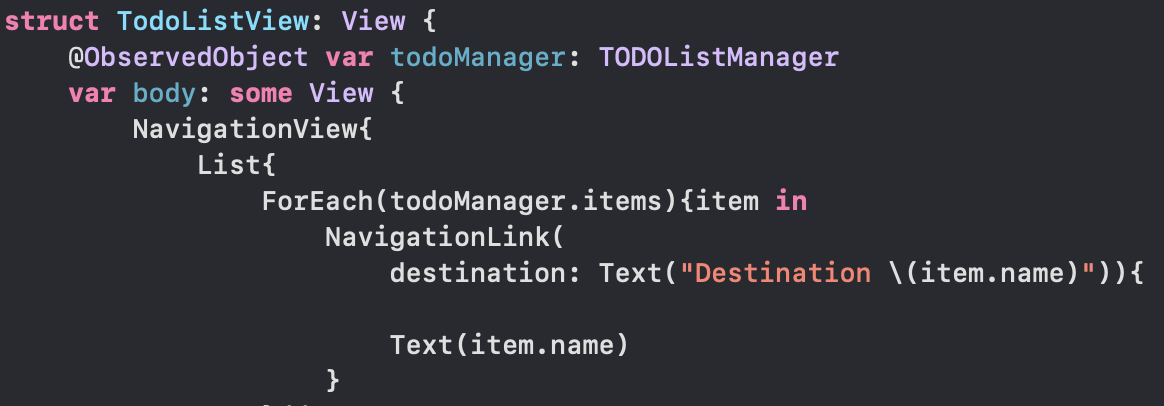




Далее переместим этот файл в папку Views



Теперь исправим код нашего View. Удалим старый массив items. Будем обращаться теперь не к массиву, а к todoManager для получения информации.

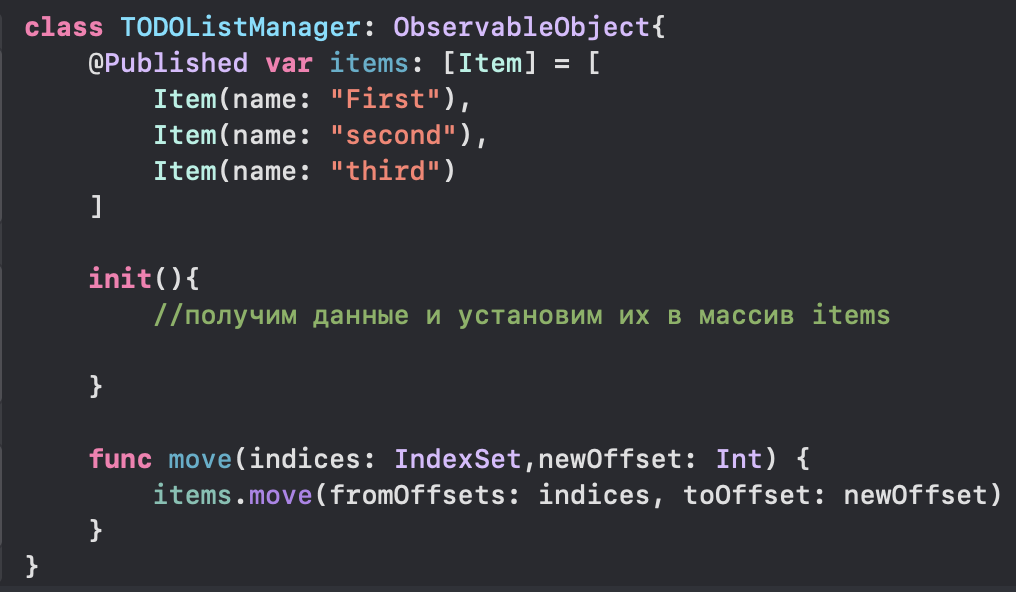


У нас ошибка – Swift жалуется на модификаторы onMove. Так как обращение к данным идет напрямую.

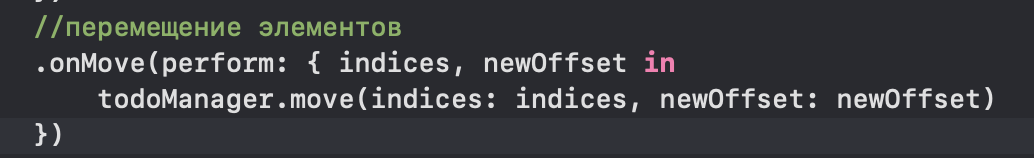


Вынесем во ViewModel функционал.

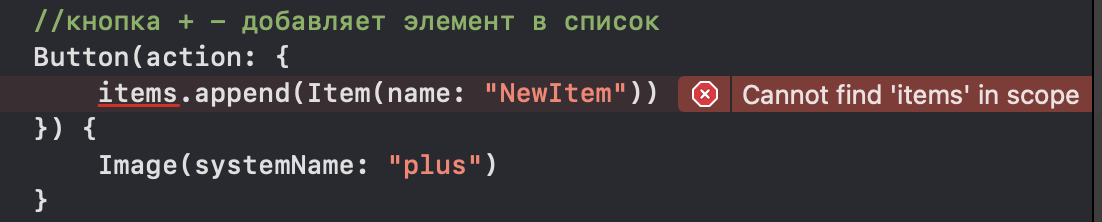
Создадим в классе функцию и добавим строку в нее. У функции добавим два параметра



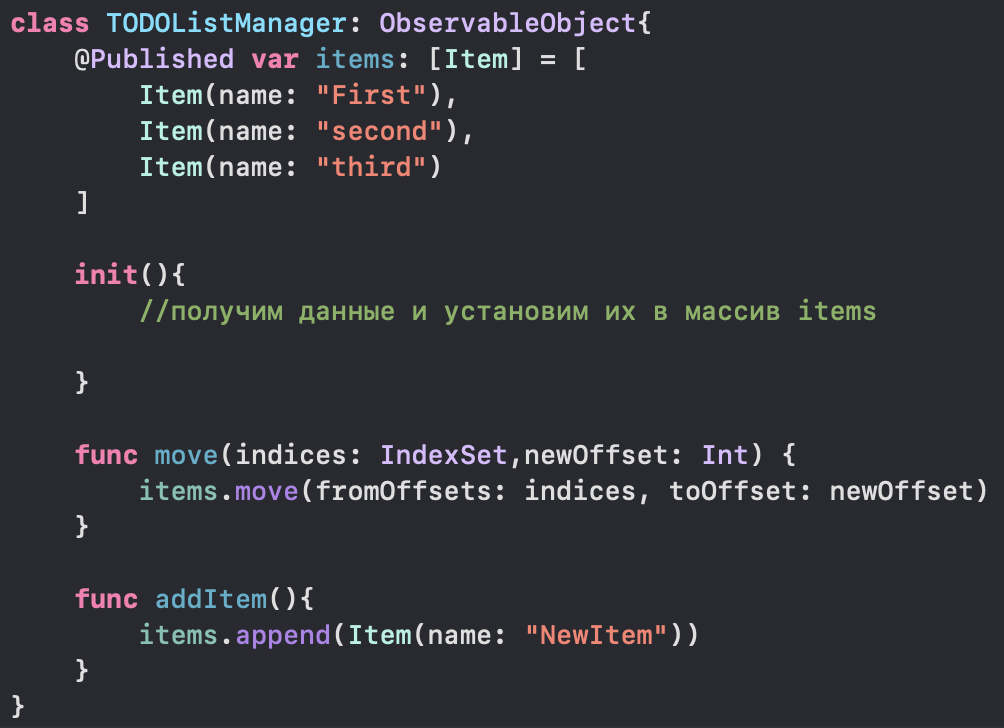
Вернемся в View и добавим вызов этой функции



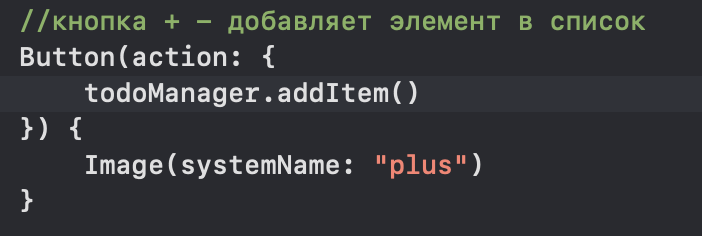
Следующая ошибка при добавлении элемента.



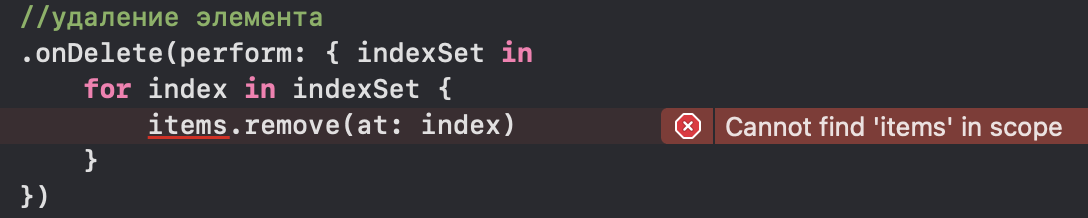
Её тоже вынесем в класс менеджера и назовем addItem



В кнопке в нашем View вызовем эту функцию с помощью менеджера.



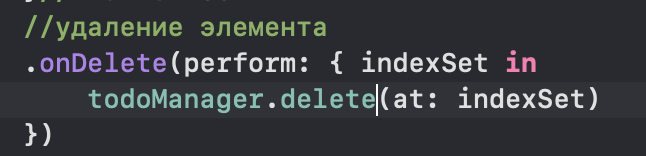
Осталось исправить ошибку удаления элемента



Вынесем цикл в функцию delete класса менеджера



В View вызовем эту функцию с помощью менеджера



Запустите проект и проверьте работу приложения.

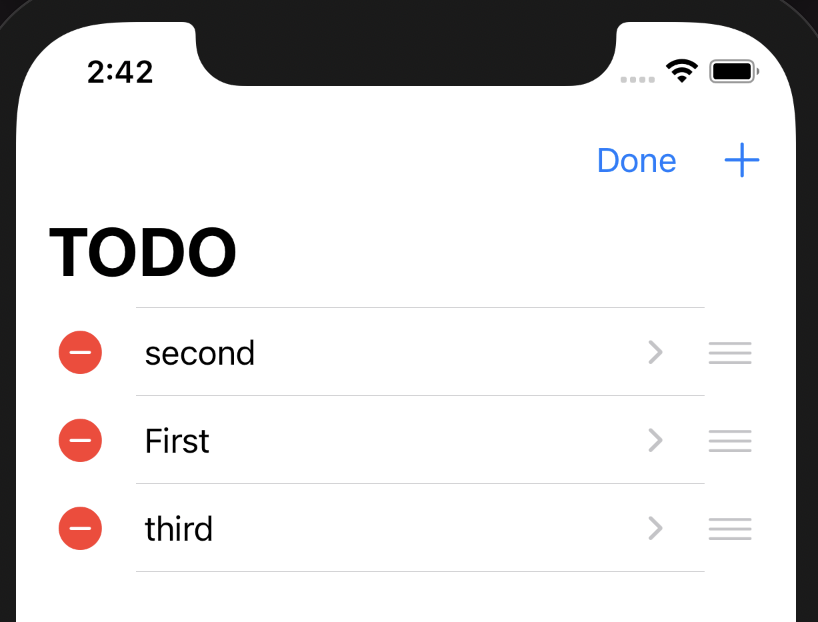
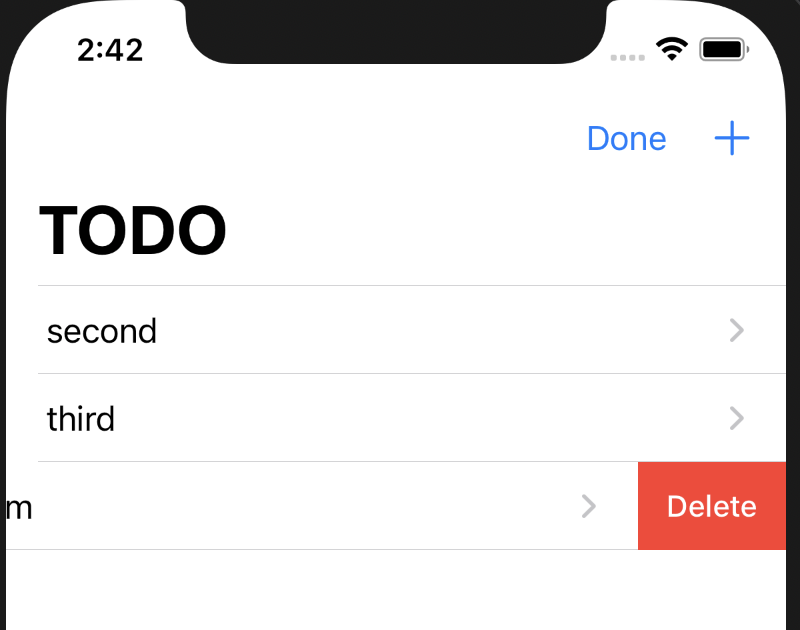
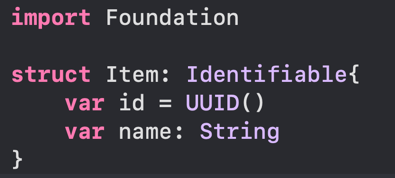
 

Схема взаимосвязи Model – View – ViewModel в данном примере



# 5. Разработка модульной структуры приложения (по заданию чемпионата)

*Определение слоев приложения: слой представления, слой бизнес-логики, слой данных, а также способов взаимодействия между ними.*

**Задание**: продумайте архитектуру мобильного приложения задания регионального чемпионата. За основу возьмите одну из сессий. Продумайте как будут хранится данные. Какие будут представления и как будет организована взаимосвязь.

# Ссылки:

1. <https://scand.com/ru/company/blog/the-ultimate-guide-to-mobile-application-architecture/?ysclid=m0zh9uytpj504352681>
2. <https://www.appypie.com/mvvm-swiftui-how-to>
3. <https://apptractor.ru/info/articles/mvvm-arhitekturnyy-shablon-dlya-strukturirovaniya-predstavleniy-swiftui.html>
4. <https://studfile.net/preview/6413815/page:107/>
5. <https://apptractor.ru/info/articles/povarennaya-kniga-swiftui-luchshie-praktiki-upravleniya-sostoyaniyami-v-swiftui.html>
6. <https://apptractor.ru/info/articles/povarennaya-kniga-swiftui-sostoyanie-privyazka-observableobject-i-observedobject.html>
7. SOLID в картинках - <https://habr.com/ru/companies/productivity_inside/articles/505430/>
8. KISS - <https://habr.com/ru/articles/249639/>
9. Clean архитектура + MVVM в iOS - <https://habr.com/ru/articles/808923/>
10. <https://mobileup.ru/blog/arhitektura-mobilnogo-prilozheniya>

[*https://habr.com/ru/articles/500158/*](https://habr.com/ru/articles/500158/)

# Дополнительно

В iOS существует несколько популярных архитектурных подходов, таких как MVC (Model-View-Controller), MVVM (Model-View-ViewModel) и VIPER (View-Interactor-Presenter-Entity-Router). Каждый из этих подходов имеет свои особенности и преимущества, а выбор подходящей архитектуры зависит от конкретных целей и требований проекта.

Основной целью хорошей архитектуры iOS является разделение ответственности между компонентами приложения, что позволяет упростить разработку, тестирование и поддержку кода. Хорошая архитектура также способствует повышению читаемости и понятности кода, что упрощает работу для команды разработчиков.

| **Название** | **Описание** | **Примеры** |
| --- | --- | --- |
| MVC | Model-View-Controller. Архитектурный шаблон, разделяющий приложение на три основных компонента: модель данных, представление и контроллер, чтобы обеспечить разделение логики и упростить обслуживание приложения. | Contacts, Calendar |
| MVVM | Model-View-ViewModel. Архитектурный шаблон, основанный на паттерне MVVM, упрощающий разработку пользовательского интерфейса, разделяя его на три компонента: модель данных, представление и модель представления. | News Feed, Weather |
| VIPER | View-Interactor-Presenter-Entity-Router. Архитектурный шаблон, целью которого является повышение расширяемости и тестируемости приложения за счет разделения его на высокоуровневые модули. | Chat, ToDo List |
| Clean Architecture | Чистая архитектура. Архитектурный подход, основанный на разделении приложения на независимые уровни, чтобы сохранить его независимость от фреймворков и библиотек. | Payment Gateway, E-commerce |
| Redux | Redux - библиотека управления состоянием, ориентированная на предсказуемость, однонаправленные потоки данных и неизменяемое состояние. | Todo App, Shopping Cart |
| Coordinator | Паттерн координатора. Целью является управление навигацией и координация переходов между экранами приложения. | E-commerce, Social Media |

Для реализации клиентского приложения на платформе iOS Apple советует использовать MVC. На рисунке 1 представим архитектуру MVC.

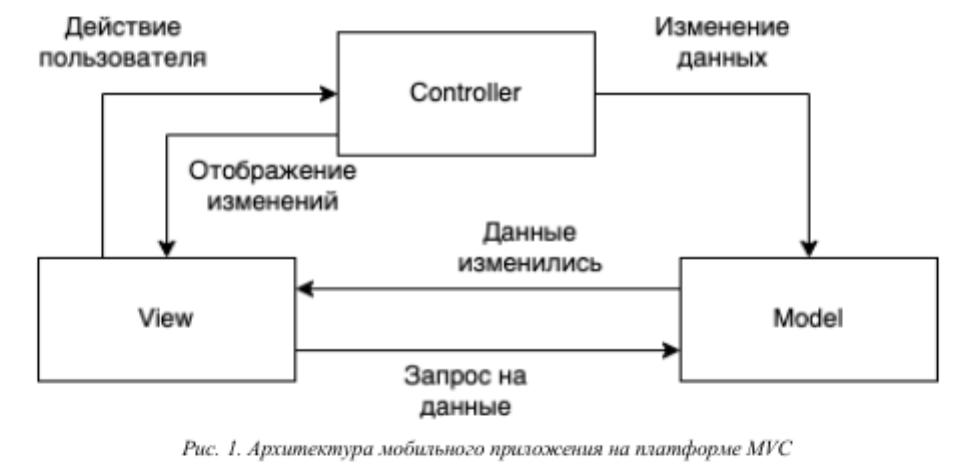


Рис. 1. Архитектура мобильного приложения на платформе MVC

На VIPER базируются большие проекты с зависимостями, MVVM является одним из лучших архитектур для средних проектов. MVVM облегчает отделение разработки графического интерфейса от разработки бизнес логики (бэк-энд логики), известной как модель (можно также сказать, что это отделение представления от модели). Модель представления является частью, которая отвечает за преобразование данных для их дальнейшей поддержки и использования. С этой точки зрения модель представления больше похожа на модель, чем на представление и обрабатывает большинство, если не всю, логику отображения данных. Модель представления может также реализовывать паттерн медиатор, организуя доступ к бэк-энд логике вокруг множества правил использования, поддерживаемых представлением [4].

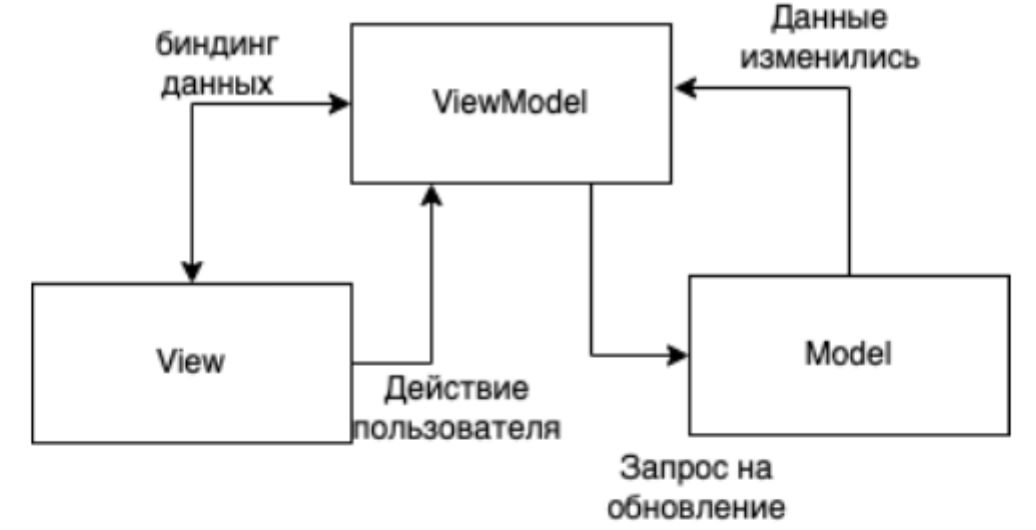


Рис. 3. Архитектура мобильного приложения на платформе MVVM

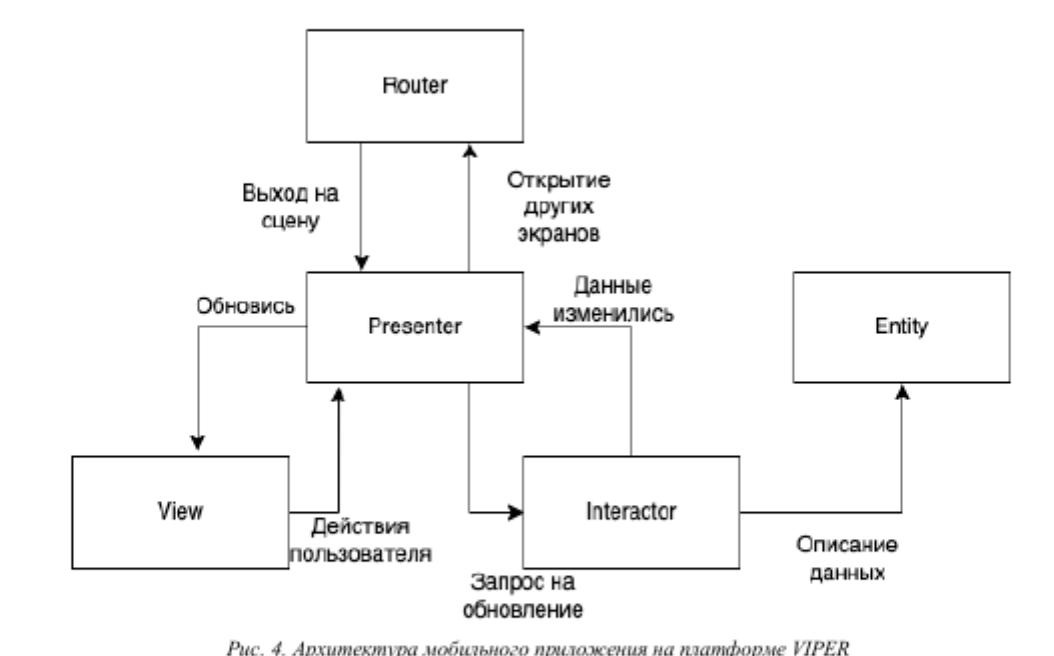


Рис. 4. Архитектура мобильного приложения на платформе VIPER

MVVM имеет два основных недостатка:

- Логика переходов содержится во View (ViewController), что нарушает принцип Single responsibility SOLID и ухудшает тестируемость.

- ViewModel по своему назначению должен содержать не только бизнес логику, но логику создания модулей (View-ViewModel).

Для избавления от этих недостатков необходимо выделить дополнительные слои программы, отвечающие за переходы и создание модулей. Например, создать дополнительный класс Router, содержащий логику переходов между модулями и статический способ для создания текущего модуля.

[MVVM](https://apptractor.ru/info/articles/chto-takoe-mvvm-arhitektura.html) (Model-View-ViewModel) — это архитектурный паттерн для структурирования представлений. Цель паттерна — отделить определение представления от бизнес-логики, лежащей в его основе. Если все сделано правильно, ваши представления не будут зависеть от конкретного типа модели.

Хотя MVVM в основном использовался во времена [UIKit/AppKit](https://apptractor.ru/info/articles/osvoenie-mvvm-na-ios.html), он по-прежнему часто применяется в SwiftUI. По моему опыту, его использование не всегда последовательно и приводит скорее к шаблону View-ViewModel. Однако это может быть оправдано, если рассматривать его как «чрезмерную инженерию». Давайте посмотрим!

**ДОПОЛНИТЕЛЬНО:**

**Навигация для iOS16 и старше**

[**https://www.youtube.com/watch?v=1tu1vLd7zH4**](https://www.youtube.com/watch?v=1tu1vLd7zH4)

<https://habr.com/ru/companies/surfstudio/articles/682156/>

<https://www.appcoda.com/navigationstack/>

### Проблемы, которые были раньше до изменения

**Возможности навигации были ограничены.** Навигация была сильно связана с UI-слоем: отделить их друг от друга было практически невозможно. На всех экранах, на которых требовалось сделать переходы, нужно было добавить нужные NavigationLink. Во всех NavigationLink —  обязательно прописать destination, при этом поменять их программно не могли. Из-за этого невозможно было полностью вынести навигацию, например, в роутер.

**Не получалось легко переходить на несколько уровней вперёд или назад**: не было прямого доступа к стеку навигации. Да, мы могли пользоваться параметрами isActive и selection у NavigationLink и управлять их изменениями программно. Например, для простого перехода на root-экран нужно было создать Binding-свойство isActive, привязать его к NavigationLink в первом экране, передать её в нужный View и в нужный момент выставить isActive в false.

Но если надо было сделать много экранов и много переходов с экрана на экран, количество таких Binding-переменных бы сильно разрослось: управлять ими стало очень сложно.

### Что изменилось с появлением iOS 16

**NavigationView стал deprecated:** вместо него теперь нужно использовать NavigationStack, который позволяет помещать View в стек навигации и убирать из него.

**В NavigationLink добавились несколько новых конструкторов**: теперь можно передавать параметр value вместо параметра destination. Value — это некий идентификатор того, куда будем переходить. На Value есть ограничение — его тип должен быть Hashable.

При этом можно использовать старый NavigationLink вместе с NavigationStack

NavigationStack {

NavigationLink (destination: Text("Destination"))

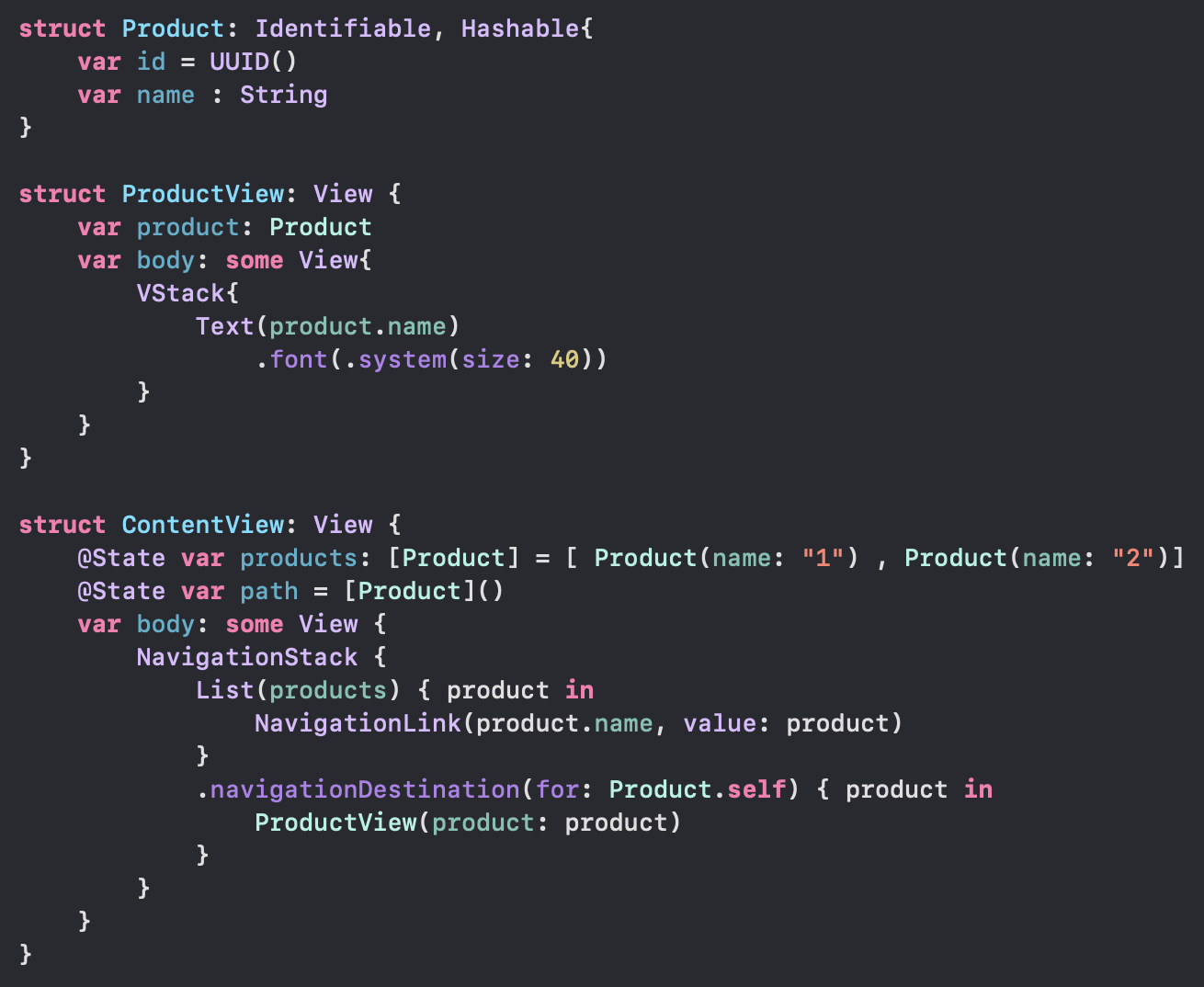
{

Text("Tap me")

}

}

Чтобы определить, куда же переходить, используется модификатор navigationDestination. Его нужно указать для всех value, которые хотим использовать. Пример:

****

**struct** Product: Identifiable, Hashable{

**var** id = UUID()

**var** name : String

}

**struct** ProductView: View {

**var** product: Product

**var** body: **some** View{

VStack{

Text(product.name)

.font(.system(size: 40))

}

}

}

**struct** ContentView: View {

@State **var** products: [Product] = [ Product(name: "1") , Product(name: "2")]

@State **var** path = [Product]()

**var** body: **some** View {

NavigationStack {

List(products) { product **in**

NavigationLink(product.name, value: product)

}

.navigationDestination(for: Product.**self**) { product **in**

ProductView(product: product)

}

}

}

}

**Еще примеры навигации в ios16**

<https://habr.com/ru/companies/surfstudio/articles/682156/>

[**https://www.appcoda.com/navigationstack/**](https://www.appcoda.com/navigationstack/)