Обзорная статья
УДК 004.5
ББК 32.972
Н 34
DOI: 10.53598/2410-3225-2023-4-331-33-43

Обзор и сравнение способов реализации интерфейса iOS-приложений

(*Рецензирована*)

Александр Евгеньевич Науменко

ООО «Дирион», Ростовская область, Россия, naumenko10@yandex.ru

Аннотация. *Рассмотрены основные способы верстки интерфейса в мобильных приложениях для системы iOS, выбор критериев для их сравнения. Проведен сравнительный анализ существующих способов верстки по выбранным критериям. На основе практического опыта работы в работе также дано краткое описание каждого рассмотренного способа создания интерфейса для приложения iOS.*

Ключевые слова: *iOS, мобильные приложения, интерфейс, Auto Layout, Autoresizing Mask, Swift, SwiftUI, Objective-C, Interface Builder*

Review Article

Overview and comparison of ways to implement the interface of iOS applications

Aleksandr E. Naumenko

Limited Liability Company "Dirion", Rostov Region, Russia, naumenko10@yandex.ru

Abstract. *The main methods of interface layout in mobile applications for the iOS system, selection of criteria for their comparison are considered. A comparative analysis of the existing layout methods was carried out according to the selected criteria. Based on practical experience, a brief description of each discussed method of creating an interface for the iOS application is also given.*

Keywords: *iOS, mobile applications, interface, Auto Layout in Swift, Autoresizing Mask, SwiftUI, Objective-C, Interface Builder*

Введение

Задача реализации интерфейса мобильного приложения стоит перед каждым мобильным разработчиком в любом проекте. Важно выбрать тот подход, который будет удовлетворять требованиям разрабатываемого приложения, а также соответствовать выбранному способу реализации архитектуры мобильного приложения [1], планам дальнейшего развития проекта, планируемой частоте редизайна как интерфейсной части, так и основного функционала.

Выбор критериев для оценки способов разработки интерфейса

Будем рассматривать в сравнительном аспекте возможные способы реализации интерфейса, описав все плюсы и минусы при выборе каждого из них.

Сначала определимся с критериями, по которым выбирается тот или иной способ реализации интерфейса. Их можно сформулировать достаточно много. Выделим среди них наиболее важные в практическом плане:

1) простота использования;
2) гибкость при доработке;
3) производительность;
4) возможность визуального моделирования;
5) сообщество и поддержка.

Возьмем их как основные для нашего сравнительного анализа. Помимо этих критериев можно выделить менее значительные, которые однако в зависимости от особенности проекта могут также использоваться в качестве существенных конкретно для данной области:

1) *Поддержка разных устройств и ориентаций.* Для приложений, которые поддерживают как портретную, так и ландшафтную ориентацию. Некоторые приложения для этого могут иметь собственную логику, и поддержка со стороны выбранного метода создания интерфейса будет критичным критерием при его выборе.

2) *Управление состоянием и анимациями* для приложений, нагруженных анимациями. Различные способы верстки предлагают разные процедуры обработки анимаций.

3) *Интеграция с другими технологиями.* Вокруг каждого способа верстки может существовать множество технологий для работы с ним, которые в свою очередь могут иметь интеграцию с важными не интерфейсными модулями приложения.

4) *Безопасность и соблюдение стандартов*. Тут важны как простота следования Apple Guidelines [2], так и отказоустойчивость при работе.

Специфика этих критериев требует отдельного изучения. Поэтому отметим наиболее важные для оценки методов создания интерфейсов критерии и остановимся отдельно на каждом из них, аргументируя их выбор для нашего последующего анализа.

Простота использования

Когда разработчик смотрит на структуру хорошо написанного приложения, то он должен в первую очередь видеть задачи, которые оно решает. Тогда не возникает проблем с его поддержкой и модификацией. Это же относится к используемым библиотекам и фреймворкам. Чем меньше времени разработчик тратит на технические детали при реализации задачи, тем эффективнее его работа. Именно критерий простоты использования является наиболее приоритетным при выборе технологии, с которой предстоит работать на проекте много месяцев, а может, и лет.

Гибкость при доработке

Простота использования должна также сочетаться с гибкостью при необходимости доработок. Это по большей части зависит от мощности фреймворка в плане предоставляемого функционала и грамотности его упаковки в предоставляемые разработчику интерфейсы.

Производительность

Зачастую борьба за простоту использования приводит к проблемам с производительностью. Это вполне логично, учитывая тот факт, что часть задач снимается с разработчика и возлагается на соответствующую библиотеку. Поэтому, выбирая что-то наиболее удобное в разработке, необходимо также учитывать задачи проекта. Вполне возможно, что именно под них придется сделать выбор в пользу менее удобного, но более производительного фреймворка.

Возможность визуального моделирования

Здесь имеется в виду не только возможность видеть результат до запуска проекта, но и работа с визуальным редактором, который используется как конструктор, получая затем скомпилированный результат в работающем проекте. Данное свойство называется WYSIWYG [3], и его наличие существенно снижает порог входа в технологию для новичков.

Сообщество и поддержка

Когда речь идет о выборе используемой технологии, наличие активного сообщества и поддержки играет решающую роль. Это особенно важно в iOS-разработке на языке программирования Swift. Активное и вовлеченное сообщество предоставляет доступ к качественным обучающим материалам, руководствам и курсам, что облегчает процесс обучения и адаптации к новым технологиям. Кроме того, даже опытнейшие разработчики сталкиваются с проблемами и ошибками, и здесь сообщество становится неоценимым ресурсом для обмена опытом и поиска решений. Это обогащает экосистему полезными инструментами, библиотеками и ресурсами, ускоряя разработку и повышая качество продукта. В целом наличие крепкого сообщества и поддержки делает технологию более устойчивой, динамичной и привлекательной для разработчиков.

Способы реализации интерфейса для iOS-приложений

Существует несколько основных способов реализации пользовательского интерфейса:

1) *XIB* (*или NIB*) *файлы* [4]. Эти файлы представляют собой отдельные файлы интерфейса для каждого представления или контроллера представления. Они визуально редактируются в Xcode и позволяют создавать пользовательские интерфейсы с помощью графического дизайна.

2) *Storyboard*. Визуальный инструмент в Xcode, позволяющий разработчикам создавать интерфейс приложения с помощью drag-and-drop. С его помощью можно легко создавать переходы между экранами и настраивать элементы управления.

3) *Auto Layout и Autoresizing Mark*. Эти технологии выделены специально, так как они всегда используются при работе с созданием интерфейса с помощью кода, XIB или Storyboards. Причем их работа происходит независимо от выбранного способа создания интерфейса и касается методов создания «резиновой» верстки.

4) *Programmatic UI*. Здесь интерфейс создается напрямую с помощью кода. Это может быть сложнее для новичков, но предоставляет максимальный контроль над поведением и внешним видом элементов интерфейса. Swift и Objective-C предоставляют API для создания и управления UI элементами в коде.

5) *SwiftUI.* Это современный декларативный способ создания интерфейсов для всех продуктов Apple. С помощью SwiftUI разработчики могут создавать интерфейсы с меньшим количеством кода и с лучшей поддержкой адаптивного дизайна для различных устройств.

Остановимся более подробно на перечисленных способах.

XIB (*или NIB*) *файлы*

Уже достаточно устаревший, но все еще используемый в iOS-проектах способ реализации интерфейса. Когда еще не было Storyboards, XIB-файлы использовались повсеместно. Каждый экран верстался в отдельном XIB-файле. Это можно делать и сейчас, поддержка данного подхода продолжается и в самом последнем XCode (на момент написания статьи XCode 15).

Главные отличия между XIB и NIB:

- *NIB:* Этот формат был первоначально разработан для NeXTSTEP OS, предшественника современного macOS;
- *XIB:* С появлением Xcode 3 Apple ввела новый формат – XIB, который, по сути, является XML-версией NIB.

Когда создается интерфейс в Interface Builder, разработчик, как правило, работает с XIB-файлом. При компиляции проекта XIB-файлы преобразуются в NIB-файлы, которые затем включаются в приложение. При запуске приложения NIB-файлы загружаются, и их содержимое становится активным, создавая элементы интерфейса.

Опишем, как это происходит на практике. Создадим представление в XIB-файле,

на котором будет текстовая метка и кнопка. Это делается в Interface Builder (рис. 1).

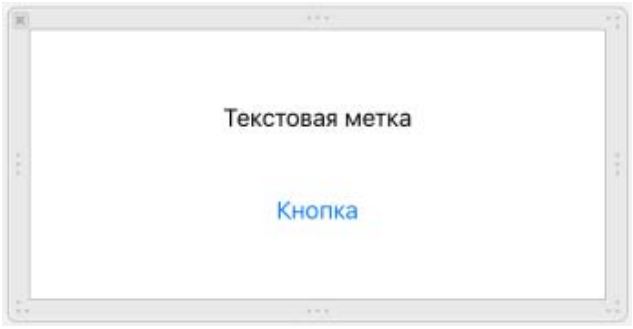

Рис. 1. Создание простого XIB-файла
Fig. 1. Creating a simple XIB-file

Теперь пишем класс для этого представления:

```
import UIKit
class CustomView: UIView {
    @IBOutlet weak var label: UILabel!
    @IBOutlet weak var button: UIButton!
    override init(frame: CGRect) {
        super.init(frame: frame)
        commonInit()
    }
    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
        commonInit()
    }
    private func commonInit() {
        let nib = UINib(nibName: "CustomView", bundle: nil)
        let view = nib.instantiate(withOwner: self, options: nil).first as! UIView
        view.frame = self.bounds
        addSubview(view)
    }
    @IBAction func buttonPressed(_ sender: UIButton) {
        // обработка нажатия на кнопку
    }
}
```

Якори @IBOutlet и @IBAction позволяют в Interface Builder легко привязать свойства и методы к элементам. Вернувшись в Interface Builder, назначаем представлению класс CustomView, что дает возможность затем настроить мышкой связи между классом и визуальными элементами.

Пройдемся по критериям оценки:

1) *Простота*. Данный способ верстки достаточно прост в использовании даже новичками. Работа с редактором Interface Builder интуитивно понятна;

2) *Гибкость*. У данного способа достаточно много ограничений. При значительном усложнении интерфейса, а также при вложенности компонентов становится очень тяжело, практически невозможно работать с версткой в XIB;

3) *Производительность*. Сильно зависит от использования технологии Auto Layout [5]. Но даже без нее производительность работы с XIB хуже, чем аналогичная работа в коде;

4) *Визуальное моделирование*. У данного способа есть возможность сразу видеть результат. Причем прямо из интерфейса можно создавать необходимые свойства и обработчики событий;

5) *Поддержка*. Несмотря на то, что данный способ верстки выходит из моды, Apple пока не спешит хоть как-то ограничивать его поддержку. Что же касается сообщества разработчиков, то за годы существования XIB-файлов вряд ли остался хоть один вопрос, на который нельзя найти ответ или статью на профильных ресурсах.

Storyboards

С выходом iOS 5 в 2011 году Apple предоставила разработчикам новый инструмент проектирования интерфейса – Storyboards. Storyboard – это XML-файл, благодаря которому разработчики могут визуализировать экраны iOS-приложений и связи между ними. Это мощный механизм*, который использует в своей основе систему работы с XIB, но предлагает также новые методы проектирования переходов между экранами. Если раньше разработчики использовали XIB-файлы, в которых обычно работали с одним представлением либо контроллером, то теперь появилась возможность добавлять в редакторе несколько контроллеров для разных экранов, настраивая всевозможные связи и переходы между ними, при этом практически без необходимости писать для этого код!

Обратимся к практике. Например, поставлена максимально простая задача: сделать два контроллера, на первом добавить кнопку, по нажатию на которую будет происходить переход на второй контроллер. Без Storyboard пришлось бы создавать эти экраны в XIB, для каждого свой файл. Затем для кнопки первого экрана писать код. В частности, для разнообразия его можно представить на Objective-C:

```
SecondViewController* secondController = [SecondViewController new];
[self.navigationController pushViewController:secondController animated:YES];
[secondController release].
```

С использованием Storyboards это делается значительно проще и быстрее (рис. 2):

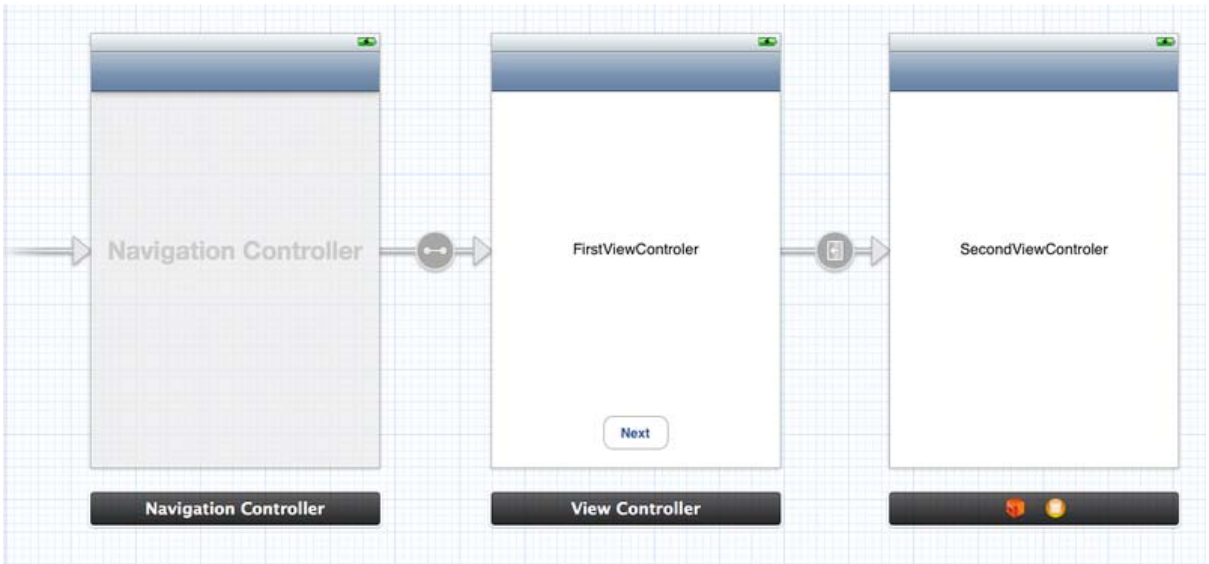

Рис. 2. Создание экранов в Storyboards

Fig. 2. Creating screens in Storyboards

В данном случае просто привязали в редакторе второй контроллер к событию нажатием кнопки первого.

* *Заметим, что механизм Storyboard имеет недостатки, см., например, статью «iOS Storyboards: анализ плюсов и минусов, best practices», URL: https://habr.com/ru/companies/mobileup/articles/456086/*

Однако такое удобство породило много проектов, в которых разработчики добавляли все экраны приложения в один Storyboard-файл, несмотря на возможность создания нескольких файлов в проекте. Это стало одной из самых распространенных ошибок работы с этим инструментом. Во-первых, это увеличивает число конфликтов при совместной работе над интерфейсом. Так как формат файлов Storyboard, как и файлов XIB, не особо читаемый, и разработчики никогда не правят из XML-код вручную, любые конфликты при редактировании становятся большой проблемой. Во-вторых, даже мощные компьютеры заметно подтормаживают в случае, если один файл содержит сотню экранов, которые тут же загружаются в память при нажатии на этот файл в проекте.

В результате большое число проектов стало требовать декомпозиции существующих Storyboard-файлов. А это оказывается порой достаточно сложной задачей, требующей аккуратного удаления Storyboard-связей, и реализацию их в коде. Поэтому, несмотря на удобство этого инструмента, пользоваться им следует аккуратно. Хорошим приемом стало добавление не более 5–10 экранов в один Storyboard-файл.

Рассмотрим этот способ создания интерфейса по выбранным критериям оценки.

1) *Простота*. Данный способ верстки также достаточно прост в использовании. Работа с редактором Interface Builder интуитивно понятна. Однако нужна аккуратность в проектировании.

2) *Гибкость*. У данного способа тоже достаточно много ограничений. Что вполне объяснимо работой все с тем же XIB-форматом. При значительном усложнении интерфейса, а также при вложенности компонентов становится весьма сложно и порой практически невозможно работать с верстской.

3) *Производительность*. Также сильно зависит от использования технологии Auto Layout. Но даже без нее производительность работы со Storyboard хуже, чем аналогичная работа в коде.

4) *Визуальное моделирование*. У данного способа есть возможность сразу видеть результат. Причем прямо из интерфейса можно создавать необходимые свойства и обработчики событий.

5) *Поддержка*. Данный способ верстки все еще весьма популярен, в том числе благодаря огромному сообществу разработчиков, использующих его. Практически любой вопрос можно найти на форумах разработчиков iOS.

Auto Layout и Autoresizing Mask

Рассмотрим вопрос создания, так называемой «резиновой» верстки элементов интерфейса. Обычно так называют способ верстки, который предполагает изменение расстояний и размеров некоторых элементов в зависимости от размера представления. Действительно, во многих случаях разработчику неизвестно, какой размер будет, к примеру, у экрана приложения. Поэтому следует создавать элементы, которым можно задать размер программно, и их верстка должны уметь адаптироваться под новый размер.

Существует 3 способа решения этой задачи:

1) *В коде*. Каждый раз при изменении размера элемента разработчик может высчитать размеры его дочерних элементов сам, а также рассчитать их позицию;

2) *Используя Autoresizing Mask*. Это очень простой способ, которым можно создавать «резиновую» верстку без «ручных» расчетов размеров и позиций элементов;

3) *Используя Auto Layout*. Более продвинутый способ расчета размеров и позиций элементов.

Рассмотрим эти способы более подробно.

В коде

У каждого представления в жизненном цикле есть метод layoutSubviews. Его задача как раз реагировать на любые изменения фрейма. Переопределяя этот метод, разработчик может задать любую логику расчета фреймов дочерних элементов. Каждый из

этих элементов в свою очередь также может иметь собственную логику в собственном layoutSubviews. Таким образом, рекурсивно перерисовывается все дерево элементов.

Autoresizing Mask

Apple предоставила разработчикам очень простой способ «резиновой» верстки. В силу своей простоты он решает только самые примитивные задачи, и при работе он редко бывает достаточным в сложных интерфейсах. Однако он прекрасно подходит для создания простых интерфейсов, а также экономит кучу времени при ручном расчете в коде, забирая на себя большу́ю часть задач.

Принцип данного способа очень простой. Для каждого представления рассматриваются 6 его свойств:

1) расстояние от левого края представления до левого края его родителя;
2) расстояние от правого края представления до правого края его родителя;
3) расстояние от верхнего края представления до верхнего края его родителя;
4) расстояние от нижнего края представления до нижнего края его родителя;
5) высота;
6) ширина.

При работе с представлением разработчик может зафиксировать каждый из 6 элементов. Он сам решает, какие именно свойства будут фиксированы. К примеру, фиксируя расстояния от краев, но не фиксируя ширину и высоту, можно получить поведение, при котором размеры дочернего элемента увеличиваются вместе с размерами родителя.

В коде это задать весьма просто, используя свойство Autoresizing Mask у представления UIView:

```
view.autoresizingMask = flexibleWidth | flexibleHeight
```

Здесь указано, что ширина и высота теперь будут плавающими, а остальные элементы из списка выше фиксируются.

Решим теперь другую задачу. Зафиксируем левый и верхний отступы, а также ширину и высоту. Сделаем теперь это из Interface Builder.

Рис. 3. Редактирование Autoresizing Mask
Fig. 3. Editing Autoresizing Mask

В редакторе выбраны верхняя и левая линии. Это означает, что они зафиксированы. Для внутренних линий все ровно наоборот – они зафиксированы в случае, когда они не выбраны.

Та же задача в коде будет решена так:

```
view.autoresizingMask = flexibleBottomMargin | flexibleTopMargin
```

Как видим, инструмент простой, поэтому и задачи он может решать только простые.

Рассмотрим теперь более продвинутый и мощный способ создания «резиновой» верстки - Auto Layout.

Auto Layout

Начиная с iOS 6, разработчики получили совершенно новый инструмент, позволяющий верстать сложные интерфейсы, сохраняя при этом контроль изменения размеров и позиций элементов при изменении размера контейнера.

Технология Auto Layout основана на так называемых ограничениях (constraints), которые добавляются для определения необходимого отношения одного свойства элемента к другому его свойству, либо к свойству соседнего элемента. Множество таких ограничений в итоге и определяют необходимое поведение представления и всех его элементов.

Примеры таких ограничений:

1) Расстояние от верхнего края объекта к верхнему краю родителя равно 20;
2) Расстояние от левого края до края родителя равно 20;
3) Расстояние от нижнего края до верхнего края соседнего элемента равно 8;
4) Высота элемента равна высоте соседнего элемента;
5) Расстояние от нижнего края к нижнему краю родителя больше или равно 100.

На картинке, взятой с официальной страницы документации Apple, продемонстрированы некоторые из описанных выше ограничений (пункты с 1-го по 4-й).

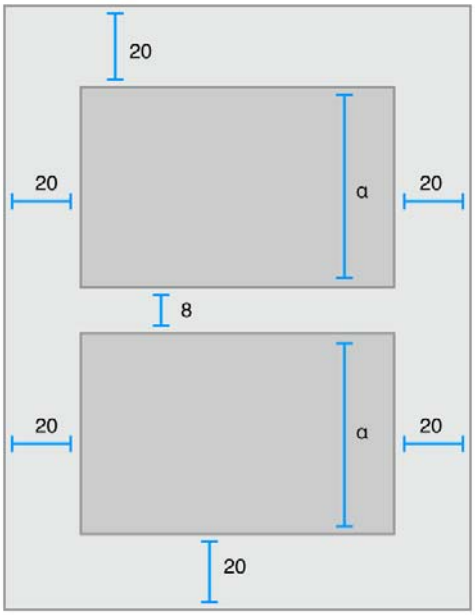

Рис. 4. Ограничения в Auto Layout
Fig. 4. Constraints in Auto Layout

Из описания ограничений можно сделать вывод, что этот инструмент гораздо более мощный, чем Autoresizing Mask. При этом можно использовать всевозможные комбинации ограничений, а также выставлять приоритеты, чтобы в случае, когда удовлетворение сразу двум ограничениям не представляется возможным, применялось то, у которого выше приоритет.

После того, как все ограничения выставлены, создается система линейных уравнений, которую теперь предстоит решить. При большом количестве элементов на одном представлении, а также при большом количестве ограничений сложность вычисления системы уравнений возрастает квадратично. При верстке экранного представления это вряд ли когда-то будет заметно, но при работе со списком элементов таблицы, каждая ячейка которой имеет 10–20 элементов со сложными взаимосвязями между ними, вполне возможны существенные подтормаживания интерфейса. Для этого можно предложить в основном следующие решения:

1) упрощать структуру элементов;
2) декомпозировать структуру элементов;
3) отказываться от Auto Layout в пользу ручного расчета.

Стоит отметить, что выставление ограничений возможно как в коде, так и в Sto-

ryboards. Однако в Storyboards, в силу ограниченности возможностей редактора, доступны не все возможности. Например, нет возможности выставлять отношение высоты одного элемента к высоте другого.

Programmatic UI

Кратко остановимся на программном создании интерфейса. В силу того, что возможности Interface Builder при работе с Auto Layout ограничены, а также из-за недостатков при совместном редактировании XIB и Storyboards, разработчики все чаще выбирают создание интерфейса в коде.

Рассмотрим небольшой пример для технологии Auto Layout. Допустим, разработчик желает реализовать в коде все ограничения, которые были приведены выше:

```
NSLayoutConstraint.activate([view1.topAnchor.constraint(equalTo: view.topAnchor, constant: 20),
view1.leadingAnchor.constraint(equalTo: view.leadingAnchor, constant: 20),
view1.bottomAnchor.constraint(equalTo: view1.topAnchor, constant: -20),
view1.heightAnchor.constraint(equalTo: view2.heightAnchor, multiplier: 1.0),
view2.bottomAnchor.constraint(lessThanOrEqualTo: view.bottomAnchor, constant: -100)])
```

Обратим внимание на порядок в коде на ограничения. Если «идем» справа налево или снизу вверх, то значения положительные. Если наоборот, то значения отрицательные, и логические выражения инвертированы.

Рассмотрим этот способ создания интерфейса по выбранным критериям оценки.

1) *Простота*. Этот способ создания интерфейса сложнее, чем его создание в визуальном редакторе. Но аккуратное оформление кода, а также декомпозиция задачи значительно облегчает как разработку, так и читаемость кода.

2) *Гибкость*. У данного способа практически отсутствуют ограничения. В коде можно создавать структуру UI практически любой сложности, что еще раз доказывает важность умения создавать верстку в коде, не упираясь в ограничения XIB и Storyboards.

3) *Производительность.* Также сильно зависит от использования технологии Auto Layout. Однако грамотная комбинация Auto Layout и Autoresizing Mask, при должной декомпозиции задачи и структуры UI, позволяет выжать максимальную производительность, недоступную при использовании XIB и Storyboards.

4) *Визуальное моделирование*. У данного способа нет возможности сразу видеть результат до запуска приложения. Это может быть проблемой для начинающих разработчиков, но обычно более опытные не испытывают с этим трудности.

5) *Поддержка*. В последние годы данный способ создания интерфейса стал очень популярным, что привело к значительному развитию поддержки со стороны сообщества разработчиков. В связи с этим появилось много библиотек, позволяющих облегчить работу и предлагающих более простой синтаксис для работы.

SwiftUI

Данный способ создания интерфейса является самым передовым и активно продвигается компанией Apple в последние годы. С выходом iOS 13 возможности этой технологии стали доступными широкому слою разработчиков. Несмотря на это, даже через несколько лет после ее появления многие компании продолжают работать с UIKit. Однако новые проекты все чаще создаются именно на SwiftUI.

Данная технология разительно отличается от UIKit. SwiftUI использует декларативный синтаксис, что позволяет разработчикам описывать UI-элементы и их состояния с помощью простого и понятного кода. Стоит отметить, что метод декларативного описания синтаксиса сейчас используется практически в таком же виде во Flutter, а

также в Android-библиотеке Compose.

Разработчикам практически нужно заново изучать все методы верстки, так как отличие от UIKit колоссальное. Однако простота и интуитивная понятность синтаксиса очень сильно облегчают переход на эту технологию. Приведем наглядный пример. Допустим, нам нужно сделать простую View, на которой кнопка и текст под ней. Отметим простоту реализации задачи:

```
struct CustomView: View {
   var body: some View {
      VStack {
         Button(label: Text(“Кнопка”))
         Text(“Текст”)
      }
   }
}
```

SwiftUI очень гибкий и мощный, обладает огромным количеством инструментов, помогающих разработчику сделать UI любой сложности. К тому он же очень тесно и органично сочетается с технологией Combine, позволяющей легко писать реактивную логику, оформляя архитектуру в стиле MVVM.

Рассмотрим этот способ создания интерфейса по выбранным критериям оценки.

1) *Простота*. Данный способ создания интерфейса невероятно прост в использовании. Интуитивность и простота синтаксиса позволяют перейти на него буквально за месяц.

2) *Гибкость*. Эта технология очень мощная и гибкая. Сама структура предложенных инструментов подталкивает к легкой декомпозиции элементов, что облегчает верстку и ее поддержку.

3) *Производительность*. Apple постоянно работает над увеличением производительности данной технологии. Благодаря своей декларативной природе, а также изменению UI только для тех элементов, для которых изменилось состояние модели, данный способ более производителен по сравнению с теми же Auto Layout.

4) *Визуальное моделирование*. У данного способа есть инструменты для визуального моделирования, причем более мощные, чем Interface Builder для Storyboards и XIB.

5) *Поддержка*. За последние несколько лет этот способ верстки успел стать очень популярным. И хоть у него нет такой наработанной базы знаний, как для UIKit, сообщество постоянно растет, а значит, растет и уровень его поддержки.

Заключение

Таким образом, рассмотрены основные методы создания интерфейса на iOS, дана сравнительная оценка работы с каждым из них. Проведенная работа предлагает опираться на данный сравнительный анализ при выборе средств создания интерфейса в новых iOS-приложениях.

Примечания

1. Архитектурные паттерны в iOS: страх и ненависть в диаграммах. MV(X). URL: https://habr.com/ru/companies/croc/articles/549590 (дата обращения: 09.11.2023).

2. App Store Review Guidelines. URL: https://developer.apple.com/app-store/review/guidelines/#design (дата обращения: 09.11.2023).

3. WYSIWYG (Материал из Википедии). URL: https://ru.wikipedia.org/wiki/WYSIWYG (дата обращения: 09.11.2023).

4. What are. XIB Files. URL:

https://docs.elementscompiler.com/Platforms/Cocoa/XIB/WhatareXIBFiles/

5. Understanding Auto Layout. URL: https://developer.apple.com/library/archive/documentation/UserExperience/Conceptual/AutolayoutPG/index.html (дата обращения: 09.11.2023).

References

1. Architecture Patterns in iOS: Fear and Hate in Diagrams. MV(X). URL: https://habr.com/ru/companies/croc/articles/549590 (access date: 09/11/2023).

2. App Store Review Guidelines. URL: https://developer.apple.com/app-store/review/guidelines/#design (access date: 09/11/2023).

3. WYSIWYG (Material from Wikipedia). URL: https://ru.wikipedia.org/wiki/WYSIWYG (access date: 09/11/2023).

4. What are. XIB Files. URL: https://docs.elementscompiler.com/Platforms/Cocoa/XIB/WhatareXIBFiles/

5. Understanding Auto Layout. URL: https://developer.apple.com/library/archive/documentation (access date: 09/11/2023).

Статья поступила в редакцию 13.11.2023; *одобрена после рецензирования 25.11.2023*; *принята к публикации 26.11.2023*.

The article was submitted 13.11.2023; *approved after reviewing 25.11.2023*; *accepted for publication 26.11.2023*.