JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 20 (2021) 170–174
Received: 14 May 2021
Accepted: 26 May 2021

Comparative analysis of UIKit and SwiftUI frameworks in iOS system

Analiza porównawcza szkieletów programistycznych UIKit i SwiftUI w systemie iOS

Piotr Wiertel*, Maria Skublewska-Paszkowska

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract
The paper concerns a comparison of SwiftUI and UIKit frameworks, used in iOS application development. SwiftUI was introduced in 2019 as a successor to UIKit as a main tool for creating application views. The purpose of this article is to compare the time performance of these two frameworks. Four testing applications have been implemented for the research, 2 in each framework. The software was complementary. The defined thesis "SwiftUI is more time efficient for applications with data collection and many filled text fields" was proved.

Keywords: SwiftUI; UIKit; time performance

Streszczenie
Artykuł dotyczy porównania szkieletów programistycznych SwiftUI i UIKit, wykorzystywanych przy tworzeniu aplikacji na system iOS. SwiftUI został zaprezentowany w 2019 jako następca UIKit dla tworzenia widoków aplikacji. Celem artykułu jest wykonanie analizy porównawczej tych szkieletów programistycznych w celu określenia ich wydajności czasowej. Na potrzeby pracy zostały wykonane cztery aplikacje testowe w obydwu badanych technologiach. Opracowane programy są komplementarne. Postawiona teza: „SwiftUI jest bardziej wydajny czasowo dla aplikacji obsługujących kolekcję danych i wiele pól tekstowych" została udowodniona.

Słowa kluczowe: SwiftUI; UIKit; wydajność

*Corresponding author
Email address: **piotr.wiertel1@pollub.edu.pl** (P. Wiertel)

1. Wstęp

Wraz z popularyzacją urządzeń mobilnych na świecie zapotrzebowanie na aplikacje rośnie i jednocześnie proces ich efektywnego wytwarzania jest stale usprawniany. Jednym z dwóch czołowych systemów operacyjnych na urządzenia mobilne jest system iOS, którego udział w rynku jest mniejszy niż konkurencyjnego Androida, aczkolwiek cieszącym się większymi dochodami ze sprzedawanych aplikacji [1]. Firma Apple prezentując pierwszego iPhona w 2007 roku przedstawiła również pierwszy system operacyjny iOS (początkowo nazwanym iPhone OS) wzorowany na macOS, stosowanym w komputerach stacjonarnych. Kolejne aktualizacje systemu wydawane każdego roku dodają nowe funkcjonalności i zwiększają użyteczność urządzeń mobilnych, umożliwiając twórcom aplikacji usprawnianie tworzonych produktów.

Razem z rozwojem aplikacji mobilnych jakość narzędzi oferowanych programistom wytwarzającym aplikacje również stale ewoluuje, jest możliwość wyboru pomiędzy różnymi zintegrowanymi środowiskami programistycznymi. Powstają także nowe języki programowania. W przypadku systemu iOS pierwszy język platformy Objective-C został w znacznej mierze zastąpiony nowszym - językiem – Swift [2]. Tworzone przez Apple środowisko XCode również jest wzbogacone o nowe dodatki, a błędy, które często utrudniały tworzenie aplikacji są stale eliminowane.

W ostatnim czasie istotną zmianą w kontekście programowania na iOS jest prezentacja nowego szkieletu programistycznego SwiftUI, deklaratywnego rozwiązania promowanego przez firmę Apple jako nowoczesne narzędzie do tworzenia aplikacji [2]. Ma ono zostać następcą dotychczas stosowanego narzędzia UIKit, który jako dojrzały produkt na przestrzeni lat stał się standardem każdego programisty działającego w obrębie technologii Apple. Jako nowe rozwiązanie SwiftUI jest obecnie w niewielkim stopniu wykorzystywany w celu wytwarzania oprogramowania w celach komercyjnych, w szczególności w aplikacjach o większym stopniu złożoności [3]. Z biegiem czasu jednak sytuacja może ulec zmianie, dlatego w niniejszej pracy przedstawiono badanie porównawcze obydwu technik.

Celem niniejszego artykułu jest porównanie szkieletów programistycznych UIKit i SwiftUI. Postawiono następującą tezę badawczą: „SwiftUI jest bardziej wydajny czasowo dla aplikacji obsługujących kolekcję danych i wiele pól tekstowych". W obydwu porównywanych technologiach wykonano możliwie podobne aplikacje testowe. Badania polegały na zmierzeniu czasu tworzenia widoków, dla każdego z testowanych urządzeń.

Na potrzeby badań wykonany został przegląd literatury. W artykułach [4, 5] porównywane były języki Swift oraz Objective-C pod względem wydajności. W pierwszym przypadku badana była szybkość wykonywania algorytmów sortowania i struktur danych. W drugiej pracy poddano analizie m.in. czas przejścia

pomiędzy widokami. W kolejnym artykule [6] poddano analizie antywzorce wydajności dotyczące aplikacji na system iOS, bez wyróżnienia zastosowanych szkieletów programistycznych. W artykule [7] dokonano przeglądu literatury, dotyczącego testowania aplikacji iOS, gdzie wyróżniono różne aspekty wpływające na wydajność testowanych urządzeń takie jak animacje, obsługa gestów czy zarządzanie danymi.

W obecnym czasie nie znaleziono artykułów naukowych poruszających tematykę porównawczą przedmiotowych szkieletów programistycznych, natomiast pomimo krótkiego czasu istnienia SwiftUI napisanych zostało wiele książek na jego temat.

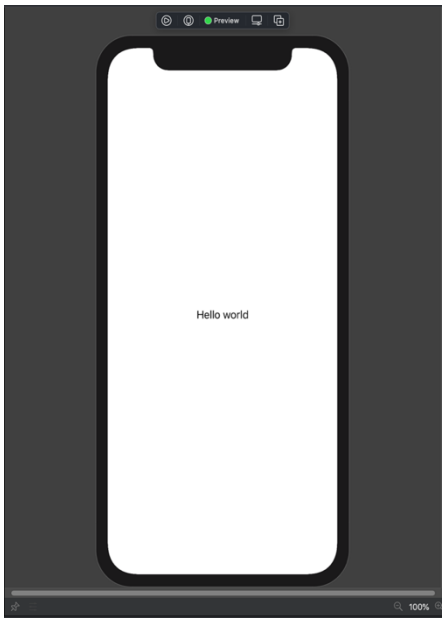
2. Implementacja aplikacji testowych

Na potrzeby badań wydajnościowych porównywanych szkieletów programistycznych zostały wykonane po dwie aplikacje o tej samej funkcjonalności, dla każdego z testowanych scenariuszy badawczych. Pierwsza z nich wykorzystuje UIKit w celach przedstawiania widoków aplikacji, a druga SwiftUI.

Projekt w SwiftUI wykorzystuje dostępny od wersji iOS 14.0 nowy cykl życia aplikacji składający się z protokołu App. Projekt zawiera 4 zaimplementowane widoki oraz początkowy, dodany przy utworzeniu projektu. Kod aplikacji w SwiftUI nie zawiera plików XML, interfejs jest definiowany tylko i wyłącznie w kodzie. Widoki są modyfikowalne w celu przeprowadzenia testów na konfigurowalnej liczbie elementów.

Projekt w UIKit wykorzystuje standardowy cykl życia aplikacji wykorzystujący AppDelegate, widoki definiowane są w większości przypadków w graficznym interfejsie. Komórki tabel i widoków kolekcji zawarte są w plikach o rozszerzeniu .xib. Graficzne reprezentacje posiadają swoje odpowiedniki w postaci klas dziedziczących po standardowych komponentach UIKit [8].

Pierwszym z testowanych widoków (rys. 1) jest pojedyncze nieedytowalne pole tekstowe na środku ekranu, jest to stan domyślnie utworzonego projektu w obydwu badanych szkieletach programistycznych.



Rysunek 1: Widok ekranu początkowego w teście aplikacji z pojedynczym widokiem.

Drugim z badanych widoków jest lista składająca się z komórek z pojedynczym polem tekstowym. Na rysunku 2 po lewej stronie znajduje się widok utworzony w UIKit, a po prawej SwiftUI. Różnice pomiędzy nimi są znikome.

Rysunek 2: Widok aplikacji wyświetlającej listę.

Kolejnym z zaimplementowanych widoków był widok pola tekstowego, który podlegał częstym zmianom w celu weryfikacji optymalnego mechanizmu rysowania widoków. Widok zaimplementowanych aplikacji przedstawiono na rysunku 3. Po lewej stronie znajduję się aplikacji wykorzystująca UIKit, a po prawej SwiftUI.

Rysunek 3: Widok aplikacji z jednym zmiennym polem tekstowym.

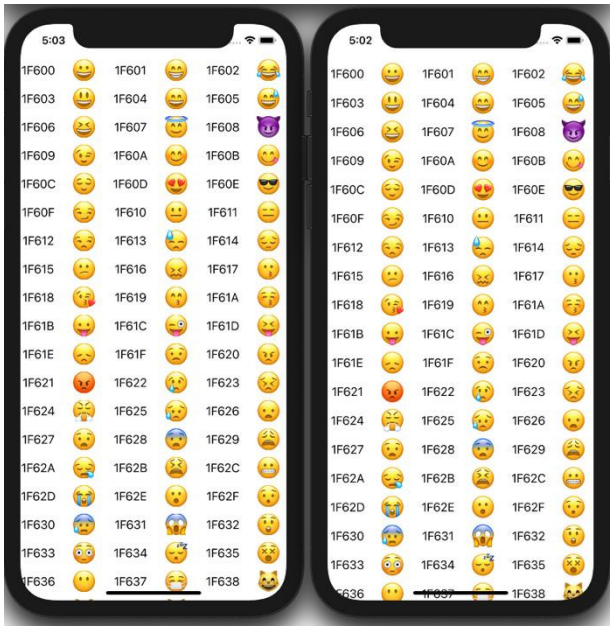
W ramach testów przygotowany został również wariant posiadający 40 pól tekstowych ułożonych jeden po drugim, przedstawiony na rysunku 4. Pola są również aktualizowane z częstotliwością wynoszącą 100Hz, więc zwiększona liczba obiektów na ekranie generuje więcej zasobów.

Rysunek 4: Widok z wieloma zmiennymi polami tekstowymi.

Następnie zaimplementowany został przykładowy widok pokazujący aktualnie wybrane miejsce na mapie. Składa się on z MapView, obrazka i pól tekstowych opisujących daną lokalizację. Wygląd został przedstawiony na rysunku 5, po lewej stronie znajduje się aplikacja wykorzystująca UIKit, a po prawej SwiftUI.

Rysunek 5: Widok z mapą.

Ostatnim z zaimplementowanych widoków jest kolekcja. W przypadku UIKit wykorzystywany jest gotowy komponent UICollectionView, który zarządza wyświetlaniem obiektów w różnorodnych układach kolumn i wierszy [9]. SwiftUI nie posiadał w czasie tworzenia pracy takiej funkcjonalności, szczególnie biorąc pod uwagę możliwości tworzenia zaawansowanych widoków. Zamiast tego, do utworzenia kolekcji zastosowano komponent ScrollView. Utworzone aplikacje przedstawiono na rysunku 6, po lewej stronie UIKit, a po prawej SwiftUI.



Rysunek 6: Widok kolekcji.

3. Metody wykonania pomiarów

Badania wydajnościowe porównywanych szkieletów programistycznych polegały na zmierzeniu czasu wygenerowania widoków i przejść między nimi. Czas został zmierzony wykorzystując wbudowaną w zintegrowane środowisko programistyczne Xcode funkcję measure [10], pozwalającą na określenie czasu wykonania bloku kodu. Metoda wykonywania pomiaru czasu startu aplikacji, wykorzystana przy przeprowadzaniu testów jest przedstawiona na listingu 1. Rezultatem uruchomienia funkcji są wartości czasów w sekundach, dla poszczególnych uruchomień testowych z zawartym błędem pomiaru.

Listing 1: Metoda wykorzystywana przy wykonywaniu pomiarów czasu start aplikacji

```
func testLaunchPerformance() throws {
    if #available(macOS 10.15, iOS 13.0, tvOS 13.0, *) {
        let options = XCTMeasureOptions()
        options.iterationCount = 20
        measure(metrics: [XCTApplicationLaunchMetric(
            waitUntilResponsive: true)], options: options) {
            XCUIApplication().launch()
        }
    }
}
```

W celu przeprowadzenia badań wydajnościowych wykorzystane zostało również wbudowane w Xcode narzędzie Instruments pozwalające na monitorowanie stanu aplikacji, umożliwiając podgląd użycia procesora czy czasu wykonania poszczególnych metod [10].

Testy przeprowadzono na następujących urządzeniach:

- iPhone 12 Mini (wersja systemu iOS 14.5),
- iPad Pro 11 2020 (wersja systemu iPadOS 14.5).

Przed rozpoczęciem testów urządzenia były uruchamiane ponownie w stanie pełnego naładowania baterii, bez aplikacji działających w tle, z wyłączonym Internetem. Odstęp czasowy pomiędzy każdą z serii testów wynosił 15 minut.

Testy uruchomieniowe przeprowadzone zostały dla pięciu różnych widoków dla każdego z testowanych urządzeń:

- widok początkowy,
- widok listy,
- widok wielu pól tekstowych,
- widok mapy, obrazka i listy,
- widok kolekcji.

W badaniach wykonane zostało po 20 prób dla każdego testowanego wariantu.

4. Wyniki badań

Badania wydajnościowe przeprowadzone dla pięciu różnych widoków, na dwóch testowanych urządzeniach przedstawione zostały w tabelach 1 - 10.

Widok początkowy

Pierwszym z badanych przypadków był widok początkowy.

Tabela 1: Średni czas uruchomienia aplikacji z widokiem początkowym na urządzeniu iPad Pro 11 2020

Liczba prób: 20	Średni czas uruchomienia [s] ± odchylenie standardowe
SwiftUI	**1,348 ± 2,67**
UIKit	**1,304 ± 2,12**

Tabela 2: Średni czas uruchomienia aplikacji z widokiem początkowym na urządzeniu iPhone 12 mini

Liczba prób: 20	Średni czas uruchomienia [s] ± odchylenie standardowe
SwiftUI	**0.965 ± 2,01**
UIKit	**0,816 ± 1,91**

Widok listy

Następnym spośród wykonywanych testów było sprawdzenie widoku listy.

Tabela 3: Średni czas uruchomienia aplikacji z widokiem listy na urządzeniu iPad Pro 11 2020

Liczba prób: 20	Średni czas uruchomienia [s] ± odchylenie standardowe
SwiftUI	**1,358 ± 2,67**
UIKit	**1,338 ± 2,07**

Tabela 4: Średni czas uruchomienia aplikacji z widokiem listy na urządzeniu iPhone 12 mini

Liczba prób: 20	Średni czas uruchomienia [s] ± odchylenie standardowe
SwiftUI	**0,986 ± 0,31**
UIKit	**1,006 ± 1,71**

Widok wielu pól tekstowych

Kolejnym spośród wykonywanych testów było sprawdzenie przypadku, w którym obiekt widoku jest aktualizowany często w krótkim czasie. W celu zmierzenia parametrów wykorzystane zostało narzędzie Profiler, które przedstawia wiele parametrów związanych z uruchamianą aplikacją.

Tabela 5: Średni czas uruchomienia aplikacji z widokiem wielu pól tekstowych na urządzeniu iPad Pro 11 2020

Liczba prób: 20	Średni czas uruchomienia [s] ± odchylenie standardowe
SwiftUI	**1,347 ± 4,34**
UIKit	**1,281 ± 0,66**

Tabela 6: Średni czas uruchomienia aplikacji z widokiem wielu pól tekstowych na urządzeniu iPhone 12 mini

Liczba prób: 20	Średni czas uruchomienia [s] ± odchylenie standardowe
SwiftUI	**1.031 ± 1,98**
UIKit	**1,021 ± 3,33**

Widok mapy, obrazka i listy

Kolejnym z przeprowadzonych testów było utworzenie przykładowego widoku składającego się z widoku mapy, obrazka i listy.

Tabela 7: Średni czas uruchomienia aplikacji z widokiem mapy na urządzeniu iPad Pro 11 2020

Liczba prób: 20	Średni czas uruchomienia [s] ± odchylenie standardowe
SwiftUI	**2,015 ± 2,71**
UIKit	**1,983 ± 2,01**

Tabela 8: Średni czas uruchomienia aplikacji z widokiem mapy na urządzeniu iPhone 12 mini

Liczba prób: 20	Średni czas uruchomienia [s] ± odchylenie standardowe
SwiftUI	**1,206 ± 4,33**
UIKit	**1,118 ± 0,42**

Widok kolekcji

Następnie przeprowadzony został test widoku kolekcji, składającego się z układu sześciu kolumn z emotikonami i odpowiadającym im kodem Unicode.

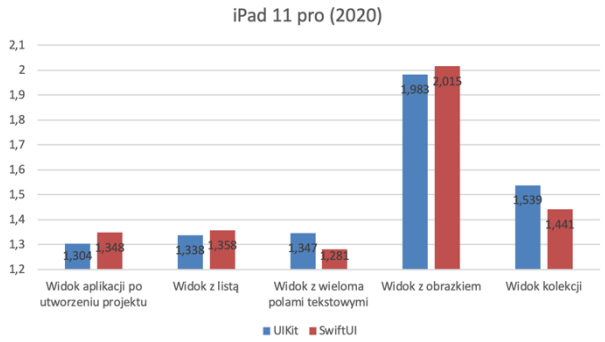
Tabela 9: Średni czas uruchomienia aplikacji z widokiem kolekcji na urządzeniu iPad Pro 11 2020

Liczba prób: 20	Średni czas uruchomienia [s] ± odchylenie standardowe
SwiftUI	**1,441 ± 2,24**
UIKit	**1,539 ± 2,03**

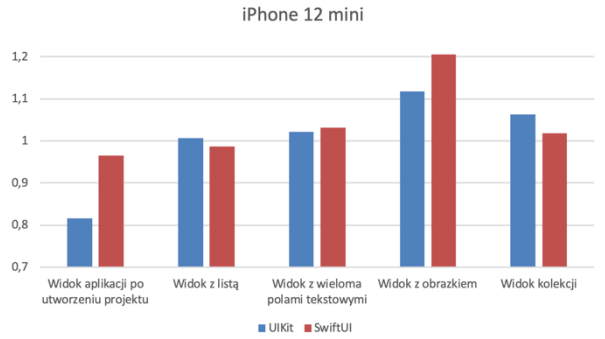
Tabela 10: Średni czas uruchomienia aplikacji z widokiem kolekcji na urządzeniu iPhone 12 mini

Liczba prób: 20	Średni czas uruchomienia [s] ± odchylenie standardowe
SwiftUI	**1,018 ± 1,28**
UIKit	**1,063 ± 0,80**

Zestawienie wyników badań wydajnościowych testowanych szkieletów programistycznych dla poszczególnych widoków zostało przedstawione na rysunkach 7 i 8.



Rysunek 7: Testy wydajnościowe badanych widoków dla urządzenia iPad 11 Pro 2020.



Rysunek 8: Testy wydajnościowe badanych widoków dla urządzenia iPhone 12 mini.

5. Wnioski

W artykule wykonana została analiza porównawcza przedmiotowych technologii: UIKit i SwiftUI. Na potrzeby badań stworzono aplikacje testowe dla obydwu szkieletów programistycznych. Wykonane badania wydajnościowe obejmowały zmierzenie czasów uruchomienia aplikacji składających się z różnych ekranów. Przetestowany również został przypadek skrajny, w którym obydwa szkielety programistyczne mogą powodować inne, znacznie różniące się wyniki spowodowane różnicami w mechanizmie rysowania widoków i ich reagowania na zmiany przy częstym przeładowywaniu widoku.

W przypadku badań na urządzeniu iPhone 12 mini największą różnicę czasu uruchomienia aplikacji widać w przypadku widoku początkowego na korzyść UIKit. Natomiast w przypadku testów na urządzeniu iPad 11 Pro, największe różnice w czasach uruchomienia widać pomiędzy widokami z wieloma polami tekstowymi oraz widokiem kolekcji, przy czym w pierwszym przypadku dla SwiftUI, pomiary są bardziej rozproszone względem średniej.

Przeprowadzone testy wydajnościowe nie wykazały znaczących różnic pomiędzy badanymi szkieletami programistycznymi. Na podstawie badań można stwierdzić, że są one równie wydajne w przypadku czasu generowania widoków. W porównaniu do UIKit, SwiftUI jest szybszy i bardziej przystępny co do możliwości generowania widoków w przypadku podstawowych aplikacji (tabela 1 - 10, rysunek 7 - 8). Na podstawie otrzymanych wyników, można stwierdzić, że postawiona teza: „SwiftUI jest bardziej wydajny czasowo dla aplikacji obsługujących kolekcję danych i wiele pól tekstowych" została udowodniona.

Pomimo wielu korzyści SwiftUI nie jest na obecny moment alternatywą dla UIKit, zwłaszcza przy bardziej rozbudowanych programach. Jest on narzędziem, które wymaga dalszego udoskonalania i musi zostać wzbogacony o wiele dodatkowych komponentów. Oprócz tego, SwiftUI nie jest jeszcze tak obszernie udokumentowany jak UIKit. Jego gotowość na ten moment jest jeszcze niewystarczająco atrakcyjna w przypadku programów produkcyjnych, aczkolwiek do zadań hobbystycznych doskonale sprawdzi się w aplikacjach o mniejszym stopniu złożoności. Do osiągnięcia statusu aplikacji gotowej do użytku SwiftUI musi zostać jeszcze zaktualizowany i ulepszony w wielu aspektach.

Literatura

[1] App Revenue Data (2021), https://www.businessofapps.com/data/app-revenues/, [02.05.2021].

[2] B. Cahill, UI Design for iOS App Development: Using SwiftUI, Apress, 2021.

[3] J. deVilla, E. Ganim, M. Hollemans, iOS Apprentice (Eighth Edition): Beginning iOS Development with Swift and UIKit, Razeware LLC, 2019.

[4] K. Gut, M. Skublewska-Paszkowska, E. Łukasik, J. Smołka, Comparison of programming languages on the iOS platform in terms of performance, IAPGOŚ 7(3) (2017) 33-36, https://doi.org/10.5604/01.3001.0010.5211.

[5] K. Banach, M. Skublewska-Paszkowska, Comparison of Objective-C and Swift on the example of a mobile game, Journal of Computer Sciences Institute 16 (2020) 305-308, https://doi.org/10.35784/jcsi.2058.

[6] S. S. Afjehei, T. P. Chen, N. Tsantalis, iPerfDetector: Characterizing and detecting performance anti-patterns in iOS applications, Empirical Software Engineering 24 (2019) 3484-3513, https://doi.org/10.1007/s10664-019-09703-y.

[7] I. Kulesovs, iOS Applications Testing, Environment. Technology. Resources. Proceedings of the International Scientific and Practical Conference 3 (2015) 138-150, https://doi.org/10.17770/etr2015vol3.187.

[8] UIKit, https://developer.apple.com/documentation/uikit, [02.05.2021].

[9] F. Farook, M. Hollemans, UIKit Apprentice, Razeware LLC, 2020.

[10] Xcode, https://developer.apple.com/xcode/, [02.05.2021].