JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 33 (2024) 264–268
Received: 12 June 2024
Accepted: 4 September 2024

Comparative analysis of frameworks for creating user interfaces in iOS Systems

Analiza porównawcza szkieletów programistycznych do tworzenia interfejsu w systemie iOS

Sebastian Krzysztof Słupny*, Edyta Łukasik

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

Abstract
The paper presents a detailed comparative analysis of two key frameworks used for creating user interfaces in iOS: SwiftUI, which is based on a declarative approach, and UIKit with Storyboard, which uses an imperative approach. The aim of this analysis is to assess the impact of each environment on the design process, development, and final quality of mobile applications. For comparison, the RandomPhotos application was created, displaying random photos. The application was developed in two versions: one using SwiftUI and the other using Storyboard and UIKit, allowing for a direct comparison of performance, flexibility, and usability of both frameworks. The thesis "SwiftUI is more efficient during the startup of applications handling a data collection" was confirmed.

Keywords: UIKit; SwiftUIl; Storyboard; time performance

Streszczenie
Artykuł ten przedstawia szczegółową analizę porównawczą dwóch kluczowych frameworków używanych do tworzenia interfejsów użytkownika w systemie iOS: SwiftUI, który opiera się na podejściu deklaratywnym, oraz UIKit z użyciem Storyboard, który stosuje podejście imperatywne. Celem tej analizy jest ocena wpływu każdego z tych środowisk na proces projektowania, rozwój oraz jakość końcową aplikacji mobilnych. Dla porównania, stworzono aplikację RandomPhotos, która wyświetla losowe zdjęcia. Aplikacja została opracowana w dwóch wersjach: jedna za pomocą SwiftUI, a druga z wykorzystaniem Storyboard i UIKit, co umożliwiło bezpośrednie porównanie wydajności, elastyczności i użyteczności obu frameworków. Teza „SwiftUI jest bardziej wydajny podczas uruchamiania aplikacji obsługujących kolekcję danych" została potwierdzona.

Słowa kluczowe: UIKit; SwiftUIl; Storyboard; wydajność

*Corresponding author
Email address: sebastian.slupny@pollub.edu.pl (S. K. Słupny)

1. Wstęp

Wzrost popularności urządzeń mobilnych zwiększa zapotrzebowanie na aplikacje. Jednym z kluczowych systemów operacyjnych na urządzaniach mobilnych jest iOS, gdzie Objective-C został w dużej mierze zastąpiony przez Swift [1]. Kluczowym czynnikiem w rozwoju aplikacji mobilnych jest jakość narzędzi programistycznych, takich jak frameworki. Framework to struktura do tworzenia aplikacji, definiująca jej mechanizm działania i strukturę oraz dostarczająca biblioteki i komponenty [2]. Zaletami stosowania frameworków są sprawdzone rozwiązania, większe bezpieczeństwo, mniej błędów w kodzie, lepsza kontrola nad projektem, spójność kodu oraz łatwiejsze testowanie i debugowanie napisanego kodu. Aplikacje z ich użyciem są bardziej niezawodne, a kod może być wielokrotnie używany. W kontekście programowania na iOS ważne są dwa szkielety programistyczne UlKit i SwiftUl.

Celem artykułu jest dostarczenie kompleksowej analizy porównawczej UIKit i SwiftUI oraz pomoc programistom i projektantom w wyborze odpowiedniego narzędzia do tworzenia aplikacji mobilnych. Postawiono tezę, iż „SwiftUI jest bardziej wydajny niż UIKit podczas uruchamiania aplikacji obsługujących kolekcję danych". W obydwu porównywanych technologiach wykonano możliwie podobne aplikacje testowe. Badania polegały na zmierzeniu czasu tworzenia widoków.

1.1. Przegląd literatury

Frameworki interfejsu użytkownika odgrywają kluczową rolę we współczesnym tworzeniu oprogramowania, umożliwiając tworzenie wizualnie atrakcyjnych i intuicyjnych aplikacji na różne platformy. Jak twierdzą D. Białkowski i J. Smołka w publikacji "Evaluation of Flutter framework time efficiently in context of user interface tasks" [3], poprzez dostarczanie różnorodnych narzędzi, komponentów i abstrakcji, frameworki te ułatwiają proces projektowania i rozwijania elementów interfejsu. Większość frameworków interfejsu użytkownika opiera się na architekturze komponentowej, gdzie elementy interfejsu traktowane są jako wielokrotnego użytku komponenty, co zwiększa modularność, możliwość utrzymania i ponownego wykorzystania. Programiści mogą łatwo łączyć i dostosowywać te komponenty, aby efektywnie tworzyć złożone interfejsy użytkownika.

Zarówno UIKit, jak i SwiftUI mają swoje zalety i są odpowiednie dla różnych przypadków użycia oraz preferencji programistów. Wybór między tymi dwoma

frameworkami zależy od czynników takich, jak wymagania projektu, wiedza zespołu oraz cele długoterminowe. Niniejszy artykuł ma na celu porównanie dwóch frameworków programistycznych, SwiftUI i UIKit, używanych do tworzenia aplikacji na system iOS. SwiftUI został zaprezentowany w 2019 roku jako nowoczesna alternatywa dla UIKit w zakresie tworzenia interfejsów aplikacji.

Wraz z rozwojem aplikacji mobilnych, jakość narzędzi oferowanych programistom stale się poprawia, dając im możliwość wyboru spośród różnych zintegrowanych środowisk programistycznych oraz nowych języków programowania. Odnosząc się do zapisów dostępnych w publikacjach, m.in. K. Gut, M. Skublewskiej-Paszkowskiej, E. Łukasik i J. Smołki "Comparison of programming languages on the iOS platform in terms of performance" [4] oraz K. Banach i M. Skublewskiej-Paszkowskiej "Comparison of Objective-C and Swift on the example of a mobile game" [5], pierwotny język platformy iOS, Objective-C, został w dużej mierze zastąpiony nowszym językiem Swift. Środowisko programistyczne Xcode [9], stworzone przez Apple, jest również stale wzbogacane o nowe dodatki, a błędy, które mogły utrudniać proces tworzenia aplikacji, są regularnie eliminowane.

Odnosząc się do książki N. Smyth "SwiftUI Essentials – iOS 14 Edition" [6], można stwierdzić, że SwiftUI, wprowadzony przez Apple na konferencji WWDC 2019, jest nowoczesnym, deklaratywnym frameworkiem interfejsu użytkownika opartym na języku Swift. Język Swift został stworzony przez Apple w 2014 roku jako nowoczesna alternatywa dla języka Objective-C, wykorzystywanego do tworzenia aplikacji na platformy takie jak iOS, macOS, watchOS i tvOS. Jak pisze J. Hunt w artykule "Value Classes" [7], Swift to język wysokiego poziomu, który skupia się na trzech głównych zasadach: bezpieczeństwie, wydajności i czytelności kodu. Charakteryzuje się on czytelną składnią oraz zawiera wiele innowacyjnych funkcji, które ułatwiają pisanie kodu. Bezpieczeństwo typów jest jedną z głównych cech Swifta. Każda zmienna na etapie kompilacji musi mieć określony typ danych, co pomaga uniknąć błędów wynikających z niezgodności typów. Dodatkowo, Swift obsługuje również opcjonalne typy, które umożliwiają obsługę wartości null. Jako język obiektowy, Swift wspiera dziedziczenie, polimorfizm i enkapsulację, co umożliwia tworzenie modułowego i skalowalnego kodu. Oprócz tego, Swift oferuje zaawansowane funkcje, takie jak funkcje wyższego rzędu i domknięcia. Pozwala to na pisanie bardziej elastycznego i ekspresywnego kodu.

Jak czytamy w "UIKit Apprentice" F. Farooka i M. Hollemansa [8], UIKit jest solidnym i niezastąpionym frameworkiem interfejsu użytkownika od premiery iPhone'a. Wykorzystuje języki programowania Objective-C i Swift, oferując szeroką gamę komponentów interfejsu użytkownika oraz narzędzi do tworzenia zaawansowanych aplikacji na iOS. Opiera się na podejściu imperatywnym, gdzie kod opisuje krok po kroku tworzenie i zarządzanie komponentami interfejsu użytkownika oraz ich interakcję z systemem bazowym.

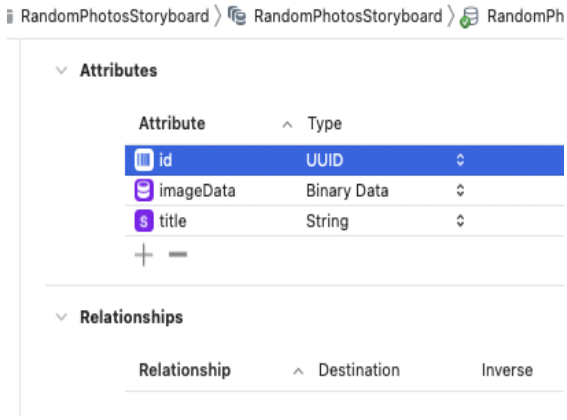
2. Implementacja aplikacji testowej

Poniżej przedstawiono implementacje aplikacji Random Photos, której celem jest wyświetlanie użytkownikom losowych zdjęć z różnych kategorii. Aplikacja jest dostępna w dwóch wersjach technologicznych: jedna korzysta z nowoczesnego podejścia SwiftUI, a druga opiera się na tradycyjnych technologiach UIKit z wykorzystaniem Storyboard. Obie wersje wykorzystują Core Data jako framework do zarządzania obiektami i przechowywania danych. Aplikacje te mają zapewnić podobną funkcjonalność i interakcję z użytkownikiem, co umożliwia praktyczne porównanie obu podejść technologicznych.

2.1. Implementacja Storyboard

Tworzenie aplikacji za pomocą Storyboard w Xcode jest jednym z najbardziej intuicyjnych i wizualnych sposobów projektowania interfejsów użytkownika. Poniżej przedstawiono proces konfiguracji aplikacji Random Photos w środowisku Storyboard, skonfigurowanej z odpowiednimi opcjami dla urządzeń docelowych.

Pierwszym krokiem jest skonfigurowanie modelu danych. W tym celu należy otworzyć plik z rozszerzeniem .xcdatamodeld, który domyślnie jest nazywany RandomPhotosStoryboard.xcdatamodeld. Następnie należy dodać nową jednostkę danych (Entity) o nazwie "Photo". Dla tej jednostki danych należy skonfigurować odpowiednie atrybuty. Konfiguracje modelu przedstawiono na Rysunku 1.



Rysunek 1: Model danych Photo.

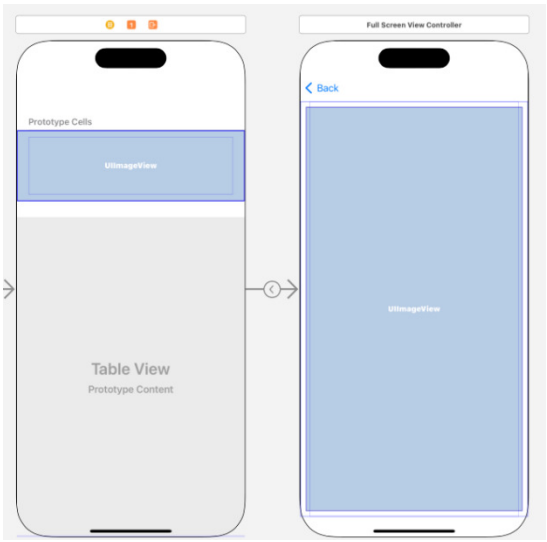
Po skonfigurowaniu modelu danych, należy przystąpić do konfiguracji Storyboard, który będzie odpowiadał za interfejs użytkownika aplikacji. W tym celu utworzono Main.storyboard w Xcode i dodano do niego odpowiedni widok. Kolejnym krokiem jest utworzenie nowego *UIViewController*, który będzie odpowiedzialny za pełnoekranowe wyświetlanie zdjęć. W Main.storyboard należy dodać nowy *UIViewController* i przypisać mu klasę *FullScreenViewController*.

Na Rysunku 2 przedstawiono dodany *UIViewController*, który będzie służył do wyświetlania zdjęć w trybie pełnoekranowym.

Następnie należy utworzyć segue, czyli przejście między widokami, między *PhotoListViewController* a *FullScreenViewController*. W tym celu należy

dodać segue typu "show", a następnie ustawić identyfikator tego segue na "*showFullScreenImage*". Na rysunku 2 pokazano przykład storyboardu z utworzonymi widokami i segue.

Na Rysunku 2 widać *UITableViewController* oraz *UIViewController* połączone ze sobą segue typu "show", który umożliwia przejście do pełnoekranowego widoku zdjęcia po wybraniu zdjęcia z listy.



Rysunek 2: UIViewController.

Na Listingu 1 przedstawiono implementację klasy FullScreenViewController, która odpowiada za wyświetlanie obrazu na pełnym ekranie. Klasa ta dziedziczy po UIViewController i zarządza ustawieniami obrazu w pełnym ekranie. W metodzie viewDidLoad(), po załadowaniu widoku, obraz jest przypisywany do UIImageView, jeśli jest dostępny. Zastosowano tryb scaleAspectFit, aby obraz zachował swoje proporcje i był odpowiednio skalowany do rozmiaru ekranu. W konsoli wypisywana jest informacja o tym, czy obraz został ustawiony, czy też nie.

Listing 1: Klasa *FullScreenViewController*

```
9
10 class FullScreenViewController: UIViewController {
11     @IBOutlet weak var imageView: UIImageView!
12     var image: UIImage?
13
14     override func viewDidLoad() {
15         super.viewDidLoad()
16         if let image = image {
17             imageView.image = image
18             imageView.contentMode = .scaleAspectFit
19             print("Image set in FullScreenViewController")
20         } else {
21             print("Image is nil in FullScreenViewController")
22         }
23     }
24 }
25
```

Główny ekran aplikacji wyświetla listę zdjęć pobranych z bazy danych, gdzie każde zdjęcie jest przedstawione jako osobny wiersz na liście. Dotknięcie wybranego zdjęcia przenosi użytkownika do pełnoekranowego widoku zdjęcia. Zdjęcia są ładowane i wyświetlane zgodnie z wcześniej opisanym kodem pokazanym na Listingu 1.

Rysunek 3: Aplikacja RandomPhotos – widok główny.

Na Rysunku 3 widoczny jest główny widok aplikacji, prezentujący listę zdjęć załadowanych z Core Data. Każde zdjęcie jest konwertowane z danych binarnych na obraz (UIImage) i wyświetlane w widoku listy.

Rysunek 4: Aplikacja RandomPhotos – pełnoekranowy widok zdjęcia.

Rysunek 4 pokazuje pełnoekranowy widok zdjęcia. Po dotknięciu zdjęcia na głównym ekranie, użytkownik jest przenoszony do pełnoekranowego widoku, gdzie wybrane zdjęcie zajmuje cały ekran. Ten widok wykorzystuje komponent *Image* z ustawieniami *resizable()* oraz *aspectRatio(contentMode: .fit)*, aby obraz zachował swoje proporcje i wypełniał cały ekran, ignorując bezpieczne strefy.

Funkcje aplikacji, takie jak wyświetlanie listy zdjęć oraz pełnoekranowy widok zdjęcia, zostały zaimplementowane za pomocą kodu zawartego w plikach PhotoListView.swift i FullScreenImageView.swift. Dzięki tym rozwiązaniom użytkownicy mogą przeglądać i wyświetlać zdjęcia w sposób wygodny i intuicyjny.

2.2. Implementacja SwiftUI

RandomPhotosSwiftUIApp to główna klasa aplikacji odpowiedzialna za jej uruchamianie i inicjalizację. Używa ona SwiftUI do tworzenia interfejsu użytkownika i Core Data do zarządzania danymi. Aplikacja uruchamia PhotoListView jako główny widok i konfiguruje kontekst zarządzany (ang. Managed Object Context) dla integracji z Core Data. Na Listingu 2 przedstawiono kod tej klasy.

Listing 2: RandomPhotosSwiftUIApp

```
import SwiftUI

@main
struct RandomPhotosSwiftUIApp: App {
    let persistenceController = PersistenceController.shared

    var body: some Scene {
        WindowGroup {
            PhotoListView()
                .environment(\.managedObjectContext, persistenceController.container.viewContext)
        }
    }
}
```

PhotoListView to główny widok, który jest wyświetlany po uruchomieniu aplikacji. Widok ten jest osadzony w kontekście zarządzanym przez Core Data (managedObjectContext), który jest pobierany z persistenceController. Główna część widoku składa się z NavigationView, który otacza widok listy, umożliwiając nawigację. Lista (komponent List) wyświetla każde zdjęcie w formie wiersza. Zdjęcia są ładowane z Core Data, Image(uiImage: UIImage) konwertuje dane obrazu na UIImage i wyświetla je. Gdy zdjęcie zostanie dotknięte, onTapGesture ustawia wybrane zdjęcie i prezentuje pełnoekranowy widok. Tytuł nawigacji jest ustawiany za pomocą .navigationTitle("Photos"). Widok pełnoekranowy zdjęcia jest prezentowany za pomocą sheet, który wyświetla widok zdjęcia, jeśli zmienna isFullScreenPresented jest ustawiona na true. Funkcja onAppear dodaje losowe zdjęcia, jeśli lista jest pusta.

FullScreenImageView.swift zarządza wyświetlaniem obrazu na pełnym ekranie. Najpierw importowany jest moduł SwiftUI, który jest niezbędny do budowania interfejsu użytkownika. Struktura FullScreenImageView definiuje widok pełnoekranowy obrazu i przyjmuje jako argument UIImage, który jest obrazem do wyświetlenia na pełnym ekranie. Na Listingu 3 pokazano implementacje.

Listing 3: FullScreenImageView

```
import SwiftUI

struct FullScreenImageView: View {
    var image: UIImage

    var body: some View {
        Image(uiImage: image)
            .resizable()
            .aspectRatio(contentMode: .fit)
            .edgesIgnoringSafeArea(.all)
    }
}
```

Aplikacja RandomPhotos, oparta na SwiftUI i Core Data, zapewnia nowoczesne i wydajne przeglądanie zdjęć. Dzięki dobrej organizacji projektu, zarządzanie kodem i jego rozwój są uproszczone, co pozwala na łatwiejsze utrzymanie i rozszerzanie funkcjonalności aplikacji. Implementacja SwiftUI umożliwia tworzenie intuicyjnych i responsywnych interfejsów użytkownika, które są spójne na różnych platformach Apple. Dzięki wykorzystaniu nowoczesnych technologii, aplikacja zapewnia płynne i przyjemne doświadczenie użytkownika.

3. Metodyka wykonania badań

Do badań użyto zestawu danych, który obejmuje różnorodne zdjęcia przechowywane w lokalnej bazie danych Core Data, pobrane z Internetu przy pierwszym uruchomieniu aplikacji. Zbiór danych zawiera obrazy o różnych rozmiarach i formatach, co ma na celu symulację realistycznych warunków użytkowania aplikacji. Dla zapewnienia spójności, każde zdjęcie jest ładowane z tej samej lokalizacji pamięci, co minimalizuje wpływ zewnętrznych czynników na wyniki eksperymentu. Eksperyment przeprowadzono na urządzeniu iPhone 14 Pro. Przed każdym testem aplikacja była restartowana, aby uniknąć wpływu danych pozostałych w pamięci cache na wyniki. Użytkownik ręcznie inicjował operację ładowania zdjęć poprzez wybranie konkretnej kategorii w aplikacji. Czas ładowania mierzony był od momentu zainicjowania żądania do momentu pełnego załadowania i wyświetlenia zdjęć na ekranie. Za pomocą narzędzi deweloperskich Xcode Instruments, dokładnie rejestrowano czas i zasoby wykorzystywane przez aplikację podczas każdego testu. Każdy test został wykonany 40 razy.

4. Wyniki badań

Tabela 1 przedstawia wyniki dla 40 prób ładowania zdjęć z bazy danych w aplikacji Random Photos, wykorzystując technologie SwiftUI i UIKit. Dane te obejmują średnie wartości czasów ładowania dla obu aplikacji wytworzonych w omawianych technologiach.

Tabela 1: Czas ładowania zdjęć z bazy danych

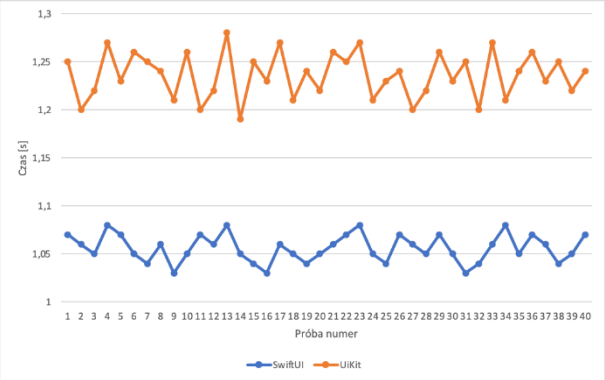
Liczba prób: 40	Średni czas załadowania zdjęcia [s]	Odchylenie standardowe
SwiftUI	1,057	0,014
UIKit	1,236	0,022

Średnia dla SwiftUI wynosi 1,057 sekundy, co wskazuje, że SwiftUI szybciej wykonuje proces ładowania zdjęć z bazy danych. Niska średnia sugeruje, że technologia ta jest optymalizowana pod kątem szybkiego i efektywnego przetwarzania danych, co jest istotne w aplikacjach wymagających płynnego dostępu do dużej liczby danych.

Średnia dla UIKit wynosi 1,236 sekundy, co jest wyższą wartością w porównaniu do SwiftUI, wskazującą, że UIKit może potrzebować więcej czasu na przetwarzanie tej samej liczby danych. Mimo, że różnica może wydawać się niewielka, w kontekście dużych aplikacji, gdzie czas ładowania ma krytyczne znaczenie, może to wpływać na ogólną wydajność i doświadczenie użytkownika.

Na rysunku 5 przedstawiono wykres liniowy porównujący czasy ładowania zdjęć z bazy danych dla obu technologii na przestrzeni 40 prób. Linia niebieska reprezentuje wyniki dla SwiftUI, a linia pomarańczowa dla UIKit. Jak można zauważyć, SwiftUI konsekwentnie osiąga lepsze czasy ładowania, które utrzymują się na niższym poziomie w całym zakresie prób, z

minimalnymi wahnięciami. UIKit wykazuje wyższe czasy ładowania, co sugeruje mniejszą spójność w porównaniu do SwiftUI.



Rysunek 5: Wyniki czasu ładowania obrazu.

5. Wnioski

W artykule została wykonana analiza porównawcza dwóch technologii. Na potrzeby badań stworzono aplikację Random Photos, która została zaimplementowana zarówno z użyciem SwiftUI, jak i UIKit (Storyboard). Celem była szczegółowa ocena tych dwóch szkieletów programistycznych pod kątem architektury, funkcjonalności, wydajności oraz wpływu na proces projektowania i implementacji aplikacji mobilnych. Kluczowym celem było również dostarczenie wyczerpujących informacji, które pomogą programistom i projektantom w wyborze odpowiedniego frameworka do konkretnych potrzeb projektowych oraz zrozumienie ewolucji narzędzi programistycznych w kontekście iOS.

Szczególną uwagę poświęcono roli Storyboard w procesie tworzenia aplikacji Random Photos. Storyboard, jako integralna część środowiska deweloperskiego Xcode, umożliwia wizualne projektowanie interfejsu użytkownika, co znacznie przyspiesza i upraszcza pracę nad aplikacją. Korzystanie ze Storyboard pozwala na łatwe mapowanie przepływów użytkownika i wizualizację całej struktury aplikacji, co jest szczególnie korzystne w początkowych etapach projektowania. Dzięki interaktywnemu interfejsowi drag-and-drop, deweloperzy mogą efektywnie tworzyć i modyfikować interfejsy użytkownika bez bezpośredniego pisania kodu, co redukuje czas potrzebny na implementację i testowanie różnych rozwiązań interfejsowych.

Testy wykazały, że aplikacja zbudowana w SwiftUI uruchamia się szybciej w porównaniu do tej zbudowanej przy użyciu UIKit i Storyboard. Szybszy czas uruchamiania aplikacji przy użyciu SwiftUI potwierdza jego efektywność i korzyści w kontekście wydajności. Wyniki pokazują, że SwiftUI zapewnia szybsze i bardziej spójne czasy ładowania zdjęć w porównaniu do UIKit. Mniejsze odchylenie standardowe i węższy zakres wskazują na większą niezawodność i przewidywalność w scenariuszach wymagających intensywnego dostępu do danych. Dla deweloperów i projektantów taka analiza może sugerować, że SwiftUI będzie lepszym wyborem dla aplikacji korzystających z intensywnego ładowania danych, takich jak galerie zdjęć czy aplikacje z dużą ilością dynamicznie generowanych treści.

Mimo, że UIKit nadal prezentuje solidną wydajność, deweloperzy mogą rozważyć, czy potencjalne korzyści płynące z szybszego ładowania i większej spójności wyników w aplikacjach napisanych w SwiftUI przeważają nad tradycyjnymi metodami oferowanymi w UIKit, które mogą oferować więcej kontroli i elastyczności w niektórych aspektach projektowania aplikacji. Na podstawie otrzymanych wyników, można stwierdzić, że postawiona teza: „SwiftUI jest bardziej wydajny podczas uruchamiania aplikacji obsługujących kolekcję danych” została udowodniona.

Literatura

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

[2] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku, Helion, 2010.

[3] D. Białkowski, J. Smołka, Evaluation of Flutter framework time efficiency in context of user interface tasks, Journal of Computer Sciences Institute 25 (2022) 309–314, https://doi.org/10.35784/jcsi.3007.

[4] K. Gut, M. Skublewska-Paszkowska, E. Łukasik, J. Smołka, Comparison of programming languages on the iOS platform in terms of performance, Informatyka, Automatyka, Pomiary W Gospodarce I Ochronie Środowiska, 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] N. Smyth, SwiftUI Essentials - iOS 14 Edition, Payload Media, Inc., 2020.

[7] J. Hunt, Value Classes, In A Beginner’s Guide to Scala, Object Orientation and Functional Programming, Springer International Publishing, 2018.

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

[9] Xcode, https://developer.apple.com/xcode/, [23.05.2024].